Design and Development of a Visualization System for Real-Time Dataflows and KPI Monitoring in automotive MES

MASTERS THESIS

Abdul Haseeb

Submitted on 10 April 2025

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Prof. Dr. Dirk Riehle, M.B.A.

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 10 April 2025

License

Erlangen, 10 April 2025

Abstract

Production systems in modern automotive manufacturing generate huge amounts of operational data in real time, but there are often complex IT processing systems behind the scenes that are opaque and do not provide good insights. The opaqueness affects decision making, hindering the process of obtaining operational knowledge. The thesis will focus on the complete flow of data through each stage of production. The research will introduce a visualization solution that enhances transparency and scaling. The data is processed in backend using Quarkus, while real-time data streaming is handled by Kafka. Finally, the visualization on the front-end is done with Angular. This allows operators and decision-makers to be aware of the inefficiencies and identify anomalies. The introduction of interactive visualizations improves usability, especially during real-time visualization as data is transmitted. The goal is to demonstrate data more intuitively and improve visibility allowing teams to take preventive actions towards possible disruptions in workflow. The objective is to develop a user-friendly tool that simplifies system monitoring, troubleshooting and encourages data-driven decision-making.

Contents

List of Figures

List of Tables

Listings

Acronyms

PLC Programmable Logic Controller

MES Manufacturing Execution System

IoT Internet of Things

Kafka Apache Kafka

UI User Interface

JSON JavaScript Object Notation

OT Operational Technology

IT Information Technology

EDA Event Driven Architecture

1 Introduction

1.1 Background

Bayerische Motoren Werke AG (BMW) is a multinational corporation rooted in Germany with a global recognition and a powerful international presence. Headquarters for the company are in Munich, Germany. The focus of the corporation is on the production of not just any automobiles, but also and especially a premier category of vehicles known as "luxury cars." Among these cars, however, in the traditions and specifications that unique brands maintain for themselves, BMW places a heavy emphasis on quality. This notion and its implications are best explained by recalling the engineering "excellence" that the company has consistently been known for.

Recent figures show BMW employs more than 150,000 people and it along with its many subsidiaries, produces over 2.5 million vehicles each year. BMW is leading the way towards adopting the principles of "Industry 4.0", digitizing production plants and working with experimental next-generation technologies in smart factories.

BMW Shopfloor digitization initiative and tryFACTORY

BMW Shopfloor digitization is an initiative to reshape production plants. Its aim is to transform the processes involved in production moving away from aged manual labor in favor of an optimized IT landscape. The data for all of the processes has been digitized allowing seamless real-time data access.

The tryFACTORY has an essential part in this process, creating an environment close to authentic, where teams can develop, test, and validate new services under more or less actual conditions of industry. It allows users to gain insights about what kinds of operations a factory might be performing in the future and about the kinds of data flows that might be happening. It puts those users in a position to enhance the processes of companies. This approach, in effect, connects all the theoretical concepts with practical, real-world applications.

Connector Connector, is a middleware that enables communication between the programmable logic controllers (PLCs) and the higher level IT systems. Its main job is to take the raw data produced by the PLC and transform it into a usable "structured" format before sending it up to the cloud. That makes Connector the first stage of the smart factory's data conveyor belt.

Connector Service Connector Service is the processing component that performs the essential tasks of decoding and transforming telegrams sent from Connector. It does the important work of cleaning up the telegrams and converting them into useful JSON structures using internal "rules" and conventions to identify the significant types of telegrams it handles and telegram types. It also sends back any responses to Connector when a request is made by PLC.

Programmable Logic Controller (PLC) A PLC is an industrial computer designed to monitor and control machinery in manufacturing. In the tryFACTORY context at BMW, the PLC is the first component along the production chain, its job is to build telegrams for the scanned parts and send out the orders that get the data flowing across the system. Once started, the data must be dealt with in real time, and the PLC works closely with the Connector to see that all the necessary production instructions are automated and enforced.

PLC Answer Services The PLC Answer Services is composed of multiple microservices. Upon getting a request from the PLC, it builds the response data and passes it on to Kafka so that it is sent back to the PLC.

1.2 Motivation And Problem Statement

The 4th industrial revolution, or the Industry 4.0, brings a significant shift in production systems, pivoting away from conventional systems in favor of advanced ones and adopting new technologies. These include the Internet of things (IoT), artificial intelligence, big data analytics, and so on. These technologies are used in production to realize the interoperability and interconnectedness of different units including the free flow of data, with the aim of flexible and intelligent manufacturing systems that are sustainable in the long run. Industry 4.0 is all about the conversion of raw data into information that can be utilized to good effect by managers. The data must be analyzed in real-time in order to help the industry find the best strategies for their operational success and at the same time keep the competitiveness in the market given that they are still on the path to grow in the digital economy (Peres et al., 2018; Wang & Wang, 2016).

Monitoring and visualizing information simultaneously throughout automation and production operations is a very important issue of Industry 4.0. In modern

production lines that have a huge number of machines and complex interactions, visualizing the data in real-time allows operators to quickly check the status of the machines and processes. This is an important function because it enables rapid responses to anomalies, which will reduce downtime and contribute to increased productivity (Buer et al., 2020; Okuyelu et al., 2024). The ability to visualize key performance indicators (KPIs) in real time allows decision-makers to extract actionable insights, ultimately improving operational efficiency (Adeleke, 2024).

FischerTechnik The FischerTechnik model offers a functional and module-based approach to develop, conceptualize, and provide a simplified model of a production line, used for training and testing in automotive contexts with reference to Manufacturing Execution Systems (MES). As a tangible realization of a smart factory, the Fischertechnik model (See Figure 1.1 comprises of components that can be systematically arranged and monitored in a predetermined manner. Furthermore, its modularity and modular connectivity allows for physical simulation of smart factory aspects of Industry 4.0 like; data monitoring, system behavior in real-time, horizontal and vertical integration of data. The model shows the context in which real physical systems can be seamlessly integrated with digital technologies making manufacturing processes more effective. With Industry 4.0 being programmatic in design, it brings about extensive changes that govern manufacturing industries. The necessity for integrating real-time data with operational flexibility has never been greater.

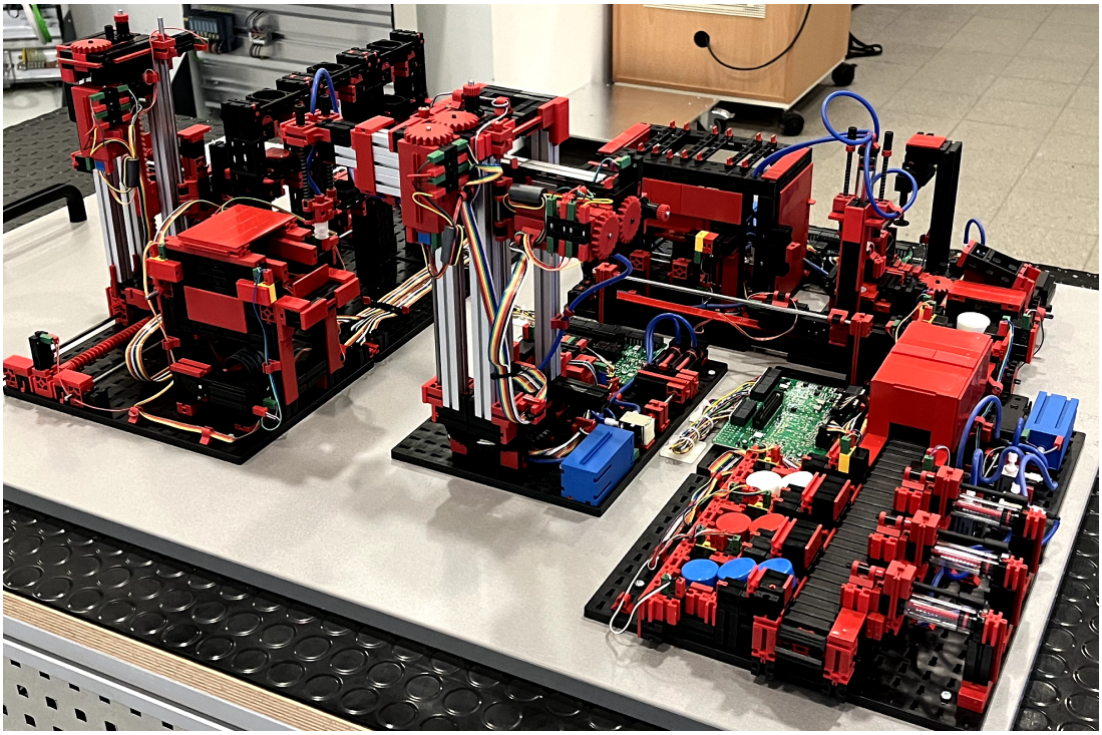

Figure 1.1: tryFACTORY Fischer Technik Model

Considering the possibilities of Industry 4.0 and the need for effective data visualization systems, this dissertation presents the design and development of a scalable visualization system to facilitate real-time monitoring of data flows and tracking of KPIs in an automotive manufacturing environment. The research addresses an important gap in a manufacturing operations context with a focus on linking multiple data sources from both the operational technology (OT) and information technology (IT) layers in the tryFACTORY, enabling improved understanding of system behavior and performance.

Today's automotive manufacturers are working in an environment of nearly continuous complexity and connectivity. This surely must make it all the more difficult to understand and maintain visibility of just what operational functions are doing. And the appearance of the 'black box' phenomenon is no help, either. With the seeming disappearance of the operational technology (OT), how are managers, engineers, and operators even to begin making the informed decisions that the interpretation of system behavior and the system's accessibility calls for? And even if system access is granted, are the concerns related to delays in detecting anomalies warranted? And with this appearance of the 'black box,' is there any cause for concern about the health of the operation? If these questions seem obvious, then a significant juncture in the black box phenomenon has been reached.

Organizations must install advanced data visualization systems to implement the transition to Industry 4.0. These systems bridge operational technology with information technology to improve situational awareness in "smart" manufacturing systems that are the hallmark of this era (Iftikhar et al., 2020).

A real-time data visualization system pulls significant information from complex data ecosystems that exist in. The purpose of this system is to take unrefined data and convert it into graphs and charts that are easily understood by users(Chen, 2019).

Through the adoption of state-of-the-art visualization technologies, including requested live graph visualizations and enhanced interfaces for users, the system allows users to interactively explore representations of dynamic data, increasing transparency and enabling informed decision making (X. Wei et al., 2018).

The current body of literature highlights the need for real-time analytics in smart manufacturing environments. For example, an incremental method for visualizing large volume data allows actors to easily identify patterns and irregularities in operational data, allowing for timely action (Fradejas-García et al., 2016). To confront these challenges, creating a data visualization system will necessitate refinement of processing architectures to handle the massive volume of real-time data produced by modern manufacturing processes while maintaining information relevance and timeliness (Zhang, 2023a).

1.3 Challenges and Limitations

There are various commercial tools and academic research that attempted to address the problem of data visualization in manufacturing environments. However, each approach has its limitations:

1. **Proprietary visualization platforms** such as (e.g., SCADA, MES tools.) can be very costly and require complex customization.
2. **Traditional dashboards** do not offer any interactivity and may not even support real-time updating capabilities.
3. **Lack of scalability** and flexibility typically does not allow for incorporation into new modern streaming architectures (like Kafka).
4. **High learning curve** makes many existing tools impractical for operational decision-making in real time on a manufacturing floor.

This thesis proposes a lightweight, scalable and interactive visualization system to counteract these limitations in the automotive production pipeline.

1.4 Thesis Objectives

The main objective of this thesis is to provide a scalable visualization system that increases data transparency in automotive production settings. To this end, the system is intended to:

- Present a glance of the live monitoring of the data flows across the production pipeline.
- provides proper visibility for operators and decision makers through a simple and intuitive experience.
- Using modern technologies like Kafka for real-time data streaming, Quarkus for backend, and Angular for front-end.
- Improve decision making by being able to provide useful insights from live data.

1.5 Scope of Work

This Thesis concentrates primarily on the design and development of an experimental visualization system that is specifically designed for real-time operational data in a manufacturing context. The project will involve the following:

- Designing a backend for real-time data streaming using Kafka
- Building a web-based visualization tool using Angular.
- Utilizing Quarkus as a framework for backend processing and data flow handling
- Verifying and testing the capabilities of the system in an automotive production setting.

Nonetheless, this thesis does not emphasize:

- Developing personalized hardware solutions for data gathering.
- Replacing existing Manufacturing Execution Systems (MES).
- Concerning non-automotive industrial sectors beyond automotive production.

1.6 Methodology Overview

The system is developed to utilize an Event-driven architecture to collect, process and visualize real-time data from production processes. This approach consists

of:

- **Backend:** Kafka is for real-time data ingestion and Quarkus which is a Java backend framework is for API requests in processing
- **Front-end:** Angular is a Javascript/Typescript front-end framework and here it is used for real-time dynamic visualization.
- **Data Processing:** Formatting the raw data into insights.

This method guarantees low-latency, highly scalable, and rich experience for the user.

1.7 Thesis Structure

The rest of the thesis is organized as follows:

- **Chapter 2: Literature Review** – Describes prior work on data visualization applied to industrial settings.
- **Chapter 3: Requirements** – Specifies the functional and non-functional system requirements.
- **Chapter 4: Architecture** – Describes the recommended system architecture and technology stack.
- **Chapter 5: Design & Implementation** – Illustrates the development and implementation process
- **Chapter 6: Evaluation** – Discusses the evaluation of the system performance and usability.
- **Chapter 7: Conclusions & Future Work** – Closes with a synthesis of the findings and suggestions for further research.

2 Literature Review

2.1 Introduction

The concept of Industry 4.0 transforms the way manufacturing is handled. Introducing smart factories that value interconnectedness and automation. It integrates technologies such as Internet of Things (IoT), Manufacturing Execution Systems (MES) and Programmable Logic Controllers (PLCs) to enhance production processes in the automotive industry resulting in the overall increase in productivity, quality and efficiency.

2.2 Smart Factories: The Integration of Industry 4.0

The Industry 4.0 paradigm is based on the concept of smart factories. These facilities utilize seamless connectivity between systems that allow equipment and machines to communicate with each other through centralized systems (Lampropoulos et al., 2019). The constant communication between the different systems allows control of manufacturing process and real-time monitoring. By integrating IoT devices across production, automotive manufacturers can collect vast amounts of data related to production rates and machine performance which can provide valuable insights (Karabegović et al., 2021).

PLCs serve as the central computers that monitor and control operations through constant communication with the machinery (Karabegović & Karabegović, 2019). By using advanced PLC systems enhanced with IoT capabilities, manufacturers improve machine-to-machine communication and facilitate quick response times (Jankovic-Zugic et al., 2023).

2.2.1 The Internet of Things (IoT)

One of the corner stones of Industry 4.0 in automotive production is the implementation of IoT. IoT technologies allow manufacturers to collect real-time data,

as a result they help automotive manufacturers move away from traditional reactive maintenance strategies and move towards predictive and condition-based maintenance models (Karabegović & Husak, 2018).

2.2.2 Manufacturing Execution Systems (MES)

In the case of Industry 4.0 Manufacturing Execution Systems (MES) plays a critical role. Think of MES platforms as the hub present at the intersection of an enterprise-level business, and it integrates data across all stages of production. By feeding data from PLCs and IoT devices to an MES manufacturers can achieve a higher level of operational transparency and can know what happens at each step of the way.

The goal of an MES is to improve communication between different areas of the organization and break down barriers that would otherwise hinder efficiency and responsiveness. All of these features are critical in automotive manufacturing where the sheer complexity of operations require real-time adjustments based on numerous factors (Parkouhi et al., 2023).

2.3 Real-time Data Processing and Messaging Systems in the Automotive Sector

The introduction of real-time data processing systems have significantly transformed the way various industries approach the problem of increasing efficiency, even more so in the automotive sector. There is a demand for immediate insights that allow rapid decision-making on the fly. Platforms such as RabbitMQ, MQTT and Apache Kafka have risen to address these needs.

2.3.1 Kafka Architecture

Apache Kafka is an event streaming platform that is capable of handling high throughput data streams quite efficiently. The main idea behind Kafka is the publish-subscribe model, according to which there are various data producers (publishers) that send messages to topics which are subsequently consumed by subscribers. Kafka's architecture comprises of several components such as producers, consumers, brokers, topics and partitions.

Brokers refer to the servers that receive messages from producers and redirect them towards consumers. A Kafka cluster can consist of multiple brokers working in parallel. Topics are categories where messages are published and partitions allow topics to be divided into smaller chunks (Bozkurt et al., 2023).

This architecture allows Kafka to deliver exceptional performance and as noted

by Bozkurt, Kafka's fault-tolerant framework ensures reliability. Kafka provides a powerful API that allows streaming capabilities where all the data can be processed in real time (Patil et al., 2022). These features make Kafka an attractive choice for sectors like automotive manufacturing, where rapid data retrieval, fault tolerance, and scalability are crucial for operational success.

2.3.2 Comparison with Other Messaging Systems

While Apache Kafka is the most commonly used messaging system, there are other messaging systems like MQTT and RabbitMQ that can also be used for real-time data processing applications and the choice depends on the requirements. RabbitMQ is based on the Advanced Message Queuing Protocol (AMQP). It generally offers a lower throughput as compared to Kafka making it the go to option for applications where complex message routing is a priority and data volume is lower (Fu et al., 2021) .

MQTT or Message Queuing Telemetry Transport is a publish-subscribe messaging protocol that works best for low-bandwidth and high-latency networks. This makes it ideal for IoT applications in automotive systems where there are small data packets unlike Kafka which focuses on high throughput and message retention (Fu et al., 2021).

It is commonly used in scenarios where devices have to communicate over unreliable networks. Overall RabbitMQ and MQTT have their strengths and weakness. However, when it comes to high throughput scenarios with large scale data integration, Kafka is the clear winner.

2.4 Real-time Visualization Techniques in Automotive Manufacturing

The automotive industry has begun relying on real-time data visualization to increase operational efficiency, responsiveness and enhance decision-making. The manufacturing environment generates large amounts of data and effective visualization techniques are crucial for decision-makers to understand the data quickly. This section discusses different visualization libraries, frameworks that can be used to make data easy to understand.

2.4.1 Overview of Visualization Libraries and Frameworks

Angular is one of the most popular front-end web frameworks right now. Developed by Google, Angular is credited for building single-page applications that can convert dynamic data into user-friendly and easy to understand interfaces. It uses a component-based architecture that allows for the creation of reusable

UI components. It comes with an extensive ecosystem that includes highly useful libraries such as ngx-charts, BMW DS-Icons and Angular Material that are used to facilitate visualizations such as dashboards, charts and graphs.

React is another well-established library used for building user interfaces. It is developed by Meta and similar to Angular it also opts for a component-based architecture, helping developers manage complex state and rendering updates efficiently. React is easy to integrate and provides various data visualization libraries such as D3.js and Chart.js for this purpose. All of these features allow React developers to create dynamic and interactive visualizations for the automotive manufacturing sector.

Other than React and Angular, the third tool that can be used for visualization is Grafana. It is an open source visualization platform that is commonly used to monitor real-time data. However, Grafana prefers working with time-series databases such as InfluxDB and Prometheus. In the case of automotive manufacturing, Grafana gives its users the option to create custom dashboards, this helps provide insights into production metrics, process efficiencies and machine health. Grafana can be integrated with platforms like Kafka which streamlines the visualization of real-time data.

2.4.2 Event-Driven Architecture

Event-driven architecture is an architectural style that promotes decoupling in an application by having the system components communicate asynchronously through event triggers. The approach is mostly used for large-scale enterprise applications as it allows for flexibility and scalability in the future. This architectural style allows loose coupling among different components of the application. Such characteristics make EDA ideal for applications within the automotive sector where reliability and real-time responsiveness is crucial.

Applications built in accordance with an event-drive architecture promote asynchronous communications. This means that each component can react to events independently. Each components ”listens” or waits for specific events to occur and then processes them autonomously. This approach allows for reduced dependencies among system components. Since each component is independent, it allows developers to easily integrate new components and services without disrupting operations.

2.5 Related Work: Visualizing Kafka Streams and Real-time Factory Data

There are studies that have explored various techniques for visualizing real-time data, particularly when using Kafka streams. These studies show the importance of stream processing and visualizations.

In his paper, Khan discusses the integration of Kafka in a push-based data streaming architecture for real-time visualization applications. By using a message broker architecture their framework is able to meet latency requirements also allowing concurrent visualization of multiple data streams. This shows the importance of Kafka in facilitating real-time insights (Khan & Alghazzawi, 2021).

Wei in their paper also explore the importance of Kafka streams. They use a web browser-based data visualization scheme and shed light on the demand for interactive tools that can easily display real-time data. The need for displaying real-time data has grown in smart factories where meaningful data visualization can help during operational decision-making (H. Wei, 2022).

Zhang in their paper presents a prototype of a real-time big data visualization platform. Their prototype also employs Apache Kafka for message handing and big data analysis. This shows that integrating Kafka can significantly increase the analytical power of visualization solutions. The prototype emphasizes on the benefits of designing good visualization tools that can handle the complexities of real-time data processing (Zhang, 2023b).

Bozkurt in their paper shows that combining Kafka with Flink technologies can enhance real-time data processing capabilities which can result in better visualization of factory data. Their case study discusses the importance of real-time visualization platforms and how they can offer better operational insights in smart manufacturing (Bozkurt et al., 2023).

2.6 Comparative Analysis of Related Research

The research surrounding real-time data handling and visualization within Manufacturing Execution Systems (MES) in the automotive sector has brought forth a number of studies that explore the area from different angles. This comparative analysis focuses on their findings, methodologies with an emphasis on identifying gaps.

2.6.1 Real-Time Data Processing Frameworks

In his paper titled "In-vehicle Distributed Time-critical Data Stream Management System for Advanced Driver Assistance" Yamaguchi discusses the idea of a distributed data stream management system that is designed for Automotive Advanced Driver Assistance Systems (ADAS). The paper highlights the management of real-time vehicle data for enhanced decision-making (Yamaguchi et al., 2017).

However, in their paper "Product Lifecycle Management in the Era of Internet of Things". D'Antonio talks about the integration of Product Lifecycle Management (PLM) systems and MES to provide real-time data feedback from the production floor. The paper discusses the importance of data flow management (D'Antonio et al., 2016).

In contrast, Simon Oman in his paper talks about the integration of MES and Enterprise Resource Planning (ERP) systems. This analysis focuses solely on the operational benefits that come with real-time data exchange and showing how it leads to better decision-making. Both D'Antonia and Oman talk about the integration of real-time data exchange to improve operational inefficiencies. However, D'Antonio focuses more on the product development aspect through PLM but Oman prioritizes supply change management through ERP (Oman et al., 2017).

Rho in their paper talk about the role of Apache Storm in providing a proper solution for spatio-temporal vehicle data access solution. The research highlights the ways data flow can be optimized through distributed processing, which in turn can improve the efficiency and reliability of data handing within an MES (Rho et al., 2016).

Bello on the other hand talks about the challenges of data stream management in automotive applications. They highlight the need for frameworks that can effectively manage increasing sensor data loads (Bello et al., 2019).

Liu & Lu in their 2023 paper explore visualization techniques used in lean manufacturing practices but focus on it's use in automotive assembly processes. In their paper, they discuss how better data visualization can help pinpoint inefficiencies in production workflows (Liu & Lu, 2023).

Manoj Kannan in their 2017 paper tries to identify gaps with current MES implementations. By performing model-based requirement analysis of different automotive MES, they were able to find that there is a need for predictive analytics capabilities so that manufacturers can anticipate operational issues early instead of reacting to them once the issue has arisen (Kannan et al., 2017).

2.6.2 Addressing the Need for Knowledge of Data Flow in Related Works

After going through numerous research papers and studies associated with automotive Manufacturing Execution Systems (MES) and real-time data handling, it is important to discuss the gaps that this research tries to fulfill. To do so, one must ask the question whether any of the previous studies address the crucial aspect of tracking data flow.

One important gap that still remains is the need to focus on cross-platform compatibility among visualization tools. With so many devices being used in modern manufacturing, challenges can arise when trying to access visual insights.

Visualizing real-time data is important and there are many ways to do so. However, most of these solutions overlook the usability criteria for non-technical staff. There is a need for better user-centric designs that prioritize ease of use in operational dashboards (Bansal, 2023).

There is a lack of research addressing the interoperability of new MES platforms with legacy systems. This is crucial for organizations when transitioning between systems. Studies in the past have typically overlooked this aspect. (Nurdiyanto, 2024).

Many of the studies talk about the need for visual representation of data, but that is as far as they go on the topic. They fail to propose a comprehensive approach to integrate advanced visualization frameworks. Developing better visualization tools that encompass diverse data streams could bridge this gap (Teucke et al., 2018).

While MES and improving efficiency in automotive manufacturing is talked about in many studies, a collective analysis makes it clear that there is a recurring deficiency in addressing data flow. How data traverses and is managed within interrelated systems like MES is important to know and that is exactly what this paper covers. While various studies talk about real-time data management and its important, they fall short at detailing data flow mechanics.

Something that all of these research papers lack is that they don't talk about the need to know how the data is traveling. In most of these studies, the focus is on visualizing the data once it has already arrived. However, it is important to know how the data is traveling. If the path of the data is known it can provide some valuable information in times of need.

3 Requirements

3.1 Requirement Derivation

The specifications for this system were obtained through stakeholder collaboration, technical workshops, and hands-on implementation. Early in the thesis, there were workshops and interviews conducted with developers and stakeholders at BMW to determine expectations and the context for operations within the TryFactory ecosystem. These discussions were critical in understanding high-level business needs, technical objectives, and architectural limitations.

Even though there was no direct interfacing with the connector, and TryFactory in terms of changes or integrations, the project did rely to a great extent on the data that they provided. The Kafka topics are the backbone of real-time information. The form and nature of these telegrams necessitated the development of several fundamental operational requirements, including message decryption and assignment of service.

Aside from stakeholder input, several requirements were either developed or improved upon in development and testing. Using WebSocket instead of Representational State Transfer Protocol (REST) polling mechanisms for front-end allows low latency communication. Performance bottlenecks, limited deployment contexts (e.g., Docker on Raspberry Pi), and complexity of message format also had considerable impact on both functional and non-functional system requirements

Thus, the eventual set of requirements represents a balance of stakeholder aspirations, operational objectives specific to the given domain, and pragmatic lessons learned during the process of prototyping and system installation.

3.2 Business and Operational Requirements at BMW

BMW's TryFactory is a highly digitized, modular production environment that mirrors a larger transition toward Industry 4.0 manufacturing concepts. In that

environment, massive amounts of data are routinely exchanged among different system layers, including physical devices, PLCs, and higher-level manufacturing services. The challenge is that, with all this data being produced, engineers, operators, and decision-makers are rarely in-the-know regarding the underlying IT and OT processes.

The discussions with BMW stakeholders during the workshops highlighted one of the main business needs was to optimize their visibility into how systems are communicating during execution and in real-time. While the systems typically work fine to do the things they are supposed to, for example, the Connector makes communication possible between FischerTechnik model and PLC to the MES. On the other hand connector Service does the job of transforming the telegram for the MES services and PLC so that they can understand and work with the telegrams. Another detail needed is the complete picture of how an initial request from the PLC to the MES moves through the execution pipeline and back. This limited visibility slows down the incident analysis process, makes it problematic from the standpoint of troubleshooting, and reduces the level of confidence in the system's trustworthiness.

The primary operational objective was to create a scalable visualization tool that bridges the visibility gap to gain a structured and intuitive perspective of the data flow. The system must track and visualize how the messages (telegrams) traverse among the various components of the system. It should also track how each of the system transforms the data being transferred and how long each step of execution takes. By visualizing this flow in a real-time process, stakeholders would be given more insight into the behavior and bottlenecks of the system as well as potential failure points.

Furthermore, there was a need for the visualization tool to be intuitive and usable for technical and non-technical users within the factory ecosystem. The visualization tool was expected to allow critical KPIs to be observed at a glance in a dashboard format, allow root causes to be derived and monitored by displaying observed execution details per service, and allow filtering or drilling based on identifiers like a part number, message type.

Therefore, the overall operational goal was to optimize the existing view of interaction between systems with a visual workflow-compatible interaction: enabling data driven decisions, provide lower response times during incidents, and supporting an ever-improving infrastructure across the factory environment.

3.3 Functional Requirements

3.3.1 Front-end Requirements

The front-end system is built using Angular and is tasked with displaying real-time execution data through a user-friendly structured visual interface. As a result of stakeholder feedback on usability, functionality requirements for the front-end system were defined and documented.

- **Real-time Visualization of Execution Data:** The system must be able to display execution telegrams from the backend dynamically. Such messages are displayed telegrams reflecting various system activities, for example, transport, ping, or loading and displayed in chronological and logical sequences as flow cards.
- **WebSocket-based Live Updates:** The front-end needs to establish a sustained WebSocket connection with the backend, to relay live data updates as soon as a new message is processed. This is done to ensure minimum latency compared to traditional polling methods and to provide continuous data updates on the execution progress.
- **Flow Visualization with Timestamps and Identifiers:** Each flow card must contain at least a part number, a timestamp, a service name, and a Kafka topic name so that users understand not only the contents of the message, but also the time context for each execution step.
- **Filtering by Service or Identifier:** It is important that users can filter visualized data based on criteria like service name (e.g., Connector Service, AnswerService) or part ID. This helps focus the flow towards entries that matter to the user and eliminates visual noise.
- **Clearing Execution Cards**: The UI should allow users to clear the previous execution data to make room for next executions and reduce clutter.

Overall, these requirements create a visualization interface that is not only technically sound but also user-oriented, and allows the user to see system performance in real-time without them having to engage in backend infrastructure or Kafka topics.

3.3.2 Backend Requirements

The backend part of the system is implemented with Quarkus and handles real-time ingestion of data, decoding of messages, running of business rules, and pushing changes to the front-end interface through WebSocket. The backend takes a central role in facilitating the proper processing of data received from Kafka

topics and its availability in a structured and timely format to the visualization interface.

- **Kafka Topic Consumption:** The backend needs to subscribe to a number of Kafka topics, which are 'FromPLCData', 'ConnectorServiceData', and 'ToPLCData'. Each of the topic broadcasts the transformed telegrams at different stages that correspond to system requests, responses, or status updates. The consumers are responsible for de-serializing the raw telegram payload and extracting important fields such as request types, service IDs, part numbers, and timestamps.
- **Data Processing:** Each Kafka topic needs to have its own processor which should be responsible for processing the data of each queue, the processing includes getting the information from the Kafka topic message, build the JSON object and sends it to the front-end through WebSocket, this should be the core functionality of the processors, although each processor should also have its own different functionality because of the transformation of the messages in each Kafka messages.
- **WebSocket Broadcasting:** After execution details are collected and paired up, the backend needs to format and transmit this information to the front-end. This method keeps the front-end updated in real time and removes the need for polling. The messages are packaged as 'ExecutionCard' objects containing all the data required for visualization.
- **Telegram Decoding and Service Identification:** The backend needs to realize parsing logic in order to decode telegrams, especially those using internal conventions at BMW. The backend attributes the message to its originating or processing service (e.g., Connector Service) based on this decoding.
- **Resilience and Error Handling:** The backend should recover from Kafka failures, network issues, or message decoding errors. Proper fallback options and retry mechanisms provide seamless data processing and logging of error messages without crashing the service.
- **Security Configuration for Kafka:** As Kafka Confluent Cloud is utilized by the system, it is done using secure connection via Simple Authentication and Security Layer (SASL) authentication Java Authentication and Authorization Service (JAAS) configuration and encrypted transport Secure Socket Layer (SSL). The credentials as well as connection parameters are stored securely and hidden from application logic.

These backend requirements allow the system to consume the high-throughput, asynchronous Kafka messages in real-time and push them towards the front-end in a structured, secure, and scalable way. Collectively, these requirements form

the underlying support of the real-time visualization tool and play an integral role in precise monitoring and transparency of the manufacturing data pipeline.

3.4 Non-functional Requirements

Apart from the functional requirements, the system has to fulfill a number of non-functional requirements for its smooth and effective operation within the scenario of a real-world automotive manufacturing setup. The requirements identified are concerned with the system's performance attributes and operational functionality that directly affect its usability, maintainability, and scalability within the scope of a real-world industrial environment.

- **Performance (Low Latency):** The system must minimize latency between message ingestion from Kafka and their display on the front-end. The system should be responsive as any noticeable delay can cause inaction against system failures. The use of reactive programming with Quarkus, combined with lightweight caching and the employment of WebSocket delivery mechanisms, ensures that latency is kept within acceptable limits.
- **Scalability:** The architecture must handle a growing number of messages from many Kafka topics and services. This way, the system can horizontally scale across many microservice instances, which is crucial for testing purposes. Allowing BMW to add new microservices and Fischertechnik models in the future.
- **Reliability and Fault Tolerance:** The system must function non-stop without having errors induced by bad format of messages, connection losses, or component failures. Bad telegrams must be logged and skipped in a way that allows the system to keep processing the next good telegram. The vital components to achieving this are: 1. Resilience of the Kafka consumer. 2. WebSocket reconnection logic. 3. Exception handling.
- **Maintainability and Modularity:** The system must conform to the principles of clean code and modular design. In the system, each Kafka topic is managed by a separate consumer module. The execution processing logic is abstracted well enough to allow simple updates and the easy integration of future new topics. The architecture of Angular is used by the front-end to allow independent updates to visual components, filters, and layouts.
- **Portability and Lightweight Deployment:** The system must be lightweight and resource-efficient so it can adapt to any deployment server. We find that using native Quarkus executables and minimal container images strikes a good balance between system performance. This allows us to take

the system across various types of environments for testing and small-scale demonstrations.

These non-functional specifications ensure that the system is not only capable in a functional sense but also good in terms of dependability, security, and being future-proof. They're particularly relevant in production environments where you have to have real-time responsiveness, resilience, and secure communication.

4 Architecture

4.1 Overview of System Architecture

The architecture of the event-driven system supports real-time data handling for the smart factory model of FischerTechnik, its event-driven nature is tied to Kafka topics. Events also arise from user interactions, such as scanning a part.

The main components include:

- **Kafka**: Functions as the messaging bus where events circulate through the different stages of the pipeline.
- **Backend Quarkus**: Lightning fast, prepared for containerized environments and reactive processing.
- **Redis**: Serves as a makeshift data storage center where the relevant information can be found for caching the identification numbers as per telegram if available.
- **Real-time push-based communication**: Occurs between the backend and front-end with the WebSocket server.
- **Angular Front-end**: Shows a card-based UI with real-time execution flows.

The complete system runs in either cloud-based or Raspberry Pi-based embedded deployments, showcasing its portability and efficiency.

4.2 Event Driven Pipeline

Decoupling of data producers and consumers is enabled by Kafka. Three main Kafka topics provide the pipeline's core.

- **FromPLCData** – Receives initial requests from PLC via Connector.

- **ConnectorServiceData** – Carries structured telegrams requests and responses from Connector Service.
- **ToPLCData** – Handles response telegrams ready for PLC consumption.

each kafka consumer has its own seprate processor **FromPLCDataProcessor**, **ConnectorServiceDataProcessor**, **ToPLCDataProcessor** responsible for processing the messages for its kafka topic.

Each topic stimulates a relevant Quarkus Kafka consumer service. The consumer takes and parses the payloads and send it down to its processor where processor extracts the fields needed from the data, use the fields to figure out the flow direction, and enrich the details of the execution before sending to Front-end. processors also make sure that every message gets served with an executionName, a status, a timestamp, and a map of executionData that includes the Kafka topic, the raw telegram (decoded) and other metadata.

The event-driven pipeline guarantees:

- Processing that does not block other processes from executing.
- The ingestion at high throughput.
- Immediate responsiveness
- Minimal coupling aids scalability.

4.3 Websocket Communication Layer

The system retains a real-time user interface by means of a WebSocket connection through the ExecutionWebSocketEndpoint class in the Quarkus backend. This class takes in client connections and permits the broadcasting of JSON messages to all clients who have subscribed.

Key features: When each Kafka consumer consumes a message successfully it passes the message to its processor for further processing. The processor takes the information that is required for visualization, converts it into a JSON (see Listing 4.1) and forwards it to the front-end through WebSocket. In the front-end the ExecutionCard view updates everytime a message is received in websocket to maintain the live view.

Advantages:

- Reduces the necessity of regular inspections.
- Permits sincere updates instantly.
- Tightly separates backend and front-end, strict adherence to reactive paradigms.

```
{
  "executionName": "PLC + Connector",
  "description":"PLC sends telegram via Connector to cloud."
  "decodedTelegram": "300SFD2_...",
  "kafkaMessage": "{
      "namespace": "bmw.Connector.Connector-from-plc-data",
      "payload": {
        "type": "BASE64ENCODED_TELEMETRY",
        "data": {
          "value": "MzAwU0ZEMl8zMDBTRkQyX1B...",
          "status": "Good"
        }
      }
    }"
  "metaData": {
  "telegramType":"INTERNAL_TELEGRAM_TYPE"
  "timestamp": "2025-03-21T10:15:30Z",
  "kafkaTopic": "FromPLCData"
  //metadeta can change depending on the telegram type
  }
}
```

Listing 4.1: Processed Message JSON

4.4 Redis as a Real-time Execution Store

Redis is an In-memory data store, that functions as a short-lived cache and real-time message tracker for the backend. The event-driven nature makes Redis effective for holding temporary execution metadata that helps the front-end stay up-to-date and visibly in real-time.

Key Responsibilities:

- **Temporary Execution Snapshot Storage**: Every time a telegram is processed, a snapshot of the execution data is taken and stored in Redis under a unique key. So, this means that right after a telegram is executed, the snapshot can be immediately sent to any WebSocket client.
- **Storing Identifiers**: Redis stores unique identifiers from recent telegrams. When a new telegram arrives, a quick search is performed on identifiers to check whether a previous telegram with the same identifier exists. This allows us to tag messages that belongs to same identifier.
- **WebSocket Push with State Awareness**: Backend modules can interact with Redis to determine the state of a certain execution.

Advantages:

- High-throughput systems need fast access.
- Separates real-time data flows from database ”Persistence”.

- Allows quick lookup for the data.

4.5 Detailed Data Flow

This section demonstrates the sequential data flow in the event-driven system, from the moment a telegram is emitted by the PLC to its final acknowledgment by PLC via Connector.

1. **PLC sends Request (FromPLCData)**
    - A Base64-encoded telegram arrives on the `FromPLCData` Kafka topic and is consumed by the Consumer.
    - The data is then passed to the designated processor for processing.
    - The Json is formed and sent to the front-end with `executionName = PLC+Connector and layer as Edge and Cloud`.
2. **Connector Service trims and forwards (ConnectorServiceData)**
    - The Connector Service trims the telegram and publishes it on `ConnectorServiceData` Kafka topic where it is consumed by the consumer.
    - The key in the message by Connector Service (e.g., `fromPLC`, `toPLC`) is used to determine direction (`PLC to MES` or `MES to PLC`).
    - The processor temporarily stores raw telegram in Redis under `TelegramType:<Identifier>`.
    - It is forwarded to the front-end with `executionName = Connector Service Request`.
3. **Response Arrives (ConnectorServiceData)**
    - Then the MES microservices consumes the data from ConnectorServiceData and gathers all the necessary information that the PLC requires to proceed further. Once the information is complete, the PLC Answer Service publishes the response data to the same topic `ConnectorServiceData` where it is consumed by the Connector Service.
    - The Connector Service identifies the direction of the message whether its fromPLC or toPLC, gathers the necessary data required for visualization..
    - Message is forwarded to front-end as `executionName = PLC Answer Service`.
4. **Connector Service prepares PLC-Compatible Response (ToPLCData)**

- The Connector Service converts the response into PLC-compatible format.
- It is published on `ToPLCData` topic where it is consumed by ToPLCData Consumer and passed to its processor.
- The message is forwarded to the front-end as `executionName = Connector Service Response`.

5. **Connector Confirms Success**
   - Lastly, final success message is published by Connector on `ToPLCData`.
   - This message confirms the completion of the execution cycle.
   - The message is tagged and visualized as a `Success Message` on the front-end as executionName = "Success Message"

The following figure shows the general flow of the system without incorporating the backend (see Figure 4.1).

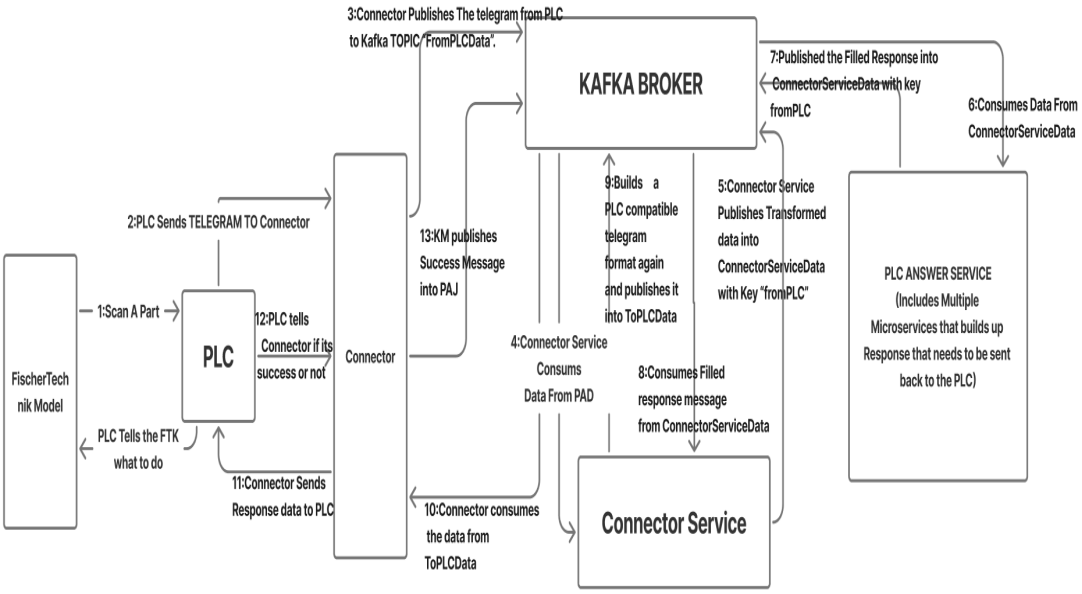

Figure 4.1: System Data Flow from PLC to Connector to Kafka and Back

4.6 Deployment Model

The system is designed for flexible deployment models that includes local development, edge based(Raspberry pi) and cloud environments.

4.6.1 Kafka Cloud Instance

- The Kafka broker is hosted on Cloud.
- Authentication via SASL/SSL with credentials defined in the application.properties file.

4.6.2 Backend (Quarkus)

- The backend is built into an ARM-compatible docker container through Dockerfile.
- Transferred to Raspberry pi through Dockerhub.
- Configured through environment variables and application.properties
- docker-compose is also used to pull and run the Redis Docker image from Docker hub to support the application in deployment.

5 Design and Implementation

5.1 Design Goals and Principles

The development of the system follows a group of design goals and software engineering principles. The aim was to ensure performance, flexibility, and maintainability in actual manufacturing environments. There were a lot of discussions regarding how the end users want the system to be, what do they want to see? This allowed for a clear understanding of the architecture of the system.

5.1.1 Design Goals

- **Real-time Performance:** The system is designed for low latency real-time data ingestion, transformation, and visualization. Ensuring immediate insight into ”the factory” was the main motivation when designing the system.
- **Modularity:** The components of the system are decoupled and consist of the following: Kafka consumers; processing services, which in turn have two kinds of components; storage systems; and the front-end UI. This modular approach provides maintenance, independent deployment, and responsibility separation.
- **Scalability:** The publish-subscribe model and distributed processing of Kafka allows it to scale horizontally. Kafka can be made to run on many servers as an application that serves many instances of the same service. More consumers or services can be added to it to handle data volumes.
- **Low Latency:** The system reduces latency from data intake to front-end presentation through reactive programming, optimized message flows, and WebSocket-based communication.

5.1.2 Design Principles

- **Separation of Concerns:** Each system module has its own dedicated responsibility. Parsing messages is the job of the Kafka consumers. While handling business logic is the job of the processors, therefore, each Kafka topic basically has its own separate processor that handles the business logic. The backend services then sends messages to the front-end through WebSocket.
- **Reactive Programming:** Using the abilities of Quarkus and SmallRye Reactive Messaging, the backend handles messages asynchronously. This is part of what allows it to deliver high throughput and responsiveness.
- **Event-Driven Design:** The architecture of the system is constructed from an event-driven framework. The incoming Kafka events serve as the fundamental impetus for the entire processing flow. From here, it can asynchronously chain together any number of transformations and enhancements before delivering the results.

5.2 Backend Implementation

The backend is built using Quarkus and is responsible for consuming messages from different Kafka topics. It decodes telegrams and forwards real-time updates to the front-end via WebSocket. The backend also performs error handling and ensures message reliability.

5.2.1 Kafka Consumer Design

The system architecture revolves around three primary Kafka consumers, each tailored to handle different stages of the data flow in the automotive Manufacturing Execution System (MES). These consumers operate in a decoupled and reactive manner, ensuring low-latency and real-time responsiveness.

5.2.2 FromPLCData Consumer

This consumer (see Listing 5.1) is responsible for consuming the initial telegrams sent by the Programmable Logic Controller (PLC). The telegrams are received in a Base64-encoded format through the `FromPLCData` Kafka topic. Upon consumption by FromPLCDataConsumer, the message is de-serialized . The payload is then passed into a dedicated processor in this case `FromPLCDataProcessor` class which handles:

- Transforming the Kafka message into a structured JSON and adds the necessary information which helps visualize the execution object on front-

end e.g telegramType, partNumber or identifier (if available), description and so on.

- stores the telegram identifier if available into the redis for matching the requests later.
- Then Broadcasts the processed JSON object to the front-end through WebSocket for visualization.

```
@Incoming("from-plc-data")
    @Blocking
    public Uni<Void> consume(IncomingKafkaRecord<String,
   String> record) {
        try{
        String key = record.getKey();
        String payloadStr = record.getPayload();
        Instant timestamp = record.getTimestamp();
        LOG.infof("Timestamp: %s", timestamp.toString());
        LOG.infof("message received: %s", record);

        Map<String, Object> payloadMap = objectMapper.
   readValue(payloadStr, new TypeReference<>() {});
        fromPLCDataProcessor.processMessage(payloadMap,
   timestamp);

        record.ack();
        }catch (Exception e){
            LOG.error("Failed to parse Kafka payload into JSON
   ", e);
        }

        return Uni.createFrom().voidItem();
    }
```

Listing 5.1: Kafka Consumer for FromPLCData kafka topic

5.2.3 ConnectorServiceData Consumer

The consumer is responsible for consuming transformed telegrams that are published by the Connector service. Response telegrams from MES are published by PLC Answer Service. Messages from this topic use Kafka record keys such as `fromPLC` and `toPLC` to distinguish between requests and responses. This Kafka topic also has a processor `ConnectorServiceDataProcessor` which handles:

- Building the JSON object from the messages that needs to be sent to the front-end for visualization.

- Deciding the name for the visualization card based on the direction key ("fromPLC", "toPLC")
- Attempting to associate the telegram with a stored identifiers in Redis to enable part tracking if the requested telegram has an identifier/PartNumber.
- Broadcasting both transformed requests and full responses to the front-end with different execution metadata depends on the telegram.

5.2.4 ToPLCData Consumer

The `ToPLCData` consumer consumes the response published by Connector Service and forwarded to the PLC. It also listens for success messages published by Connector Service confirming that the PLC has successfully processed the instruction. The processor `ToPLCDataProcessor` of this Kafka topic:

- Decodes the response telegram and fills the JSON with the metadata important for visualizing the Response data.
- Broadcasts messages to the WebSocket channel.
- Logs success messages for visualization as the final step of the cycle.

5.2.5 Topic-Specific Processors

Each Kafka consumer is responsible for consuming and passing the message to its associated processor (`FromPLCDataProcessor`, `toPLCDataProcessor`, `ConnectorServiceDataProcessor`). These processor classes includes:

- Abstract business logic and transformation pipelines.
- Extract metadata, decode telegrams, and construct an `ExecutionDetails` object.
- Push data to Redis (for tracing execution cycles) and broadcast via WebSocket.

5.2.6 WebSocket Integration

All three consumers broadcast structured execution messages to a shared WebSocket endpoint through their processors (`ExecutionWebSocketEndpoint`) (see Listing 5.2).

```java
@ServerEndpoint("/ws/executions")
public class ExecutionWebSocketEndpoint {

    private static final Logger LOGGER = Logger.getLogger(String.valueOf(ExecutionWebSocketEndpoint.class));

    // Store connected sessions
    private static Set<Session> sessions = Collections.newSetFromMap(new ConcurrentHashMap<>());

    @OnOpen
    public void onOpen(Session session) {
        sessions.add(session);
        LOGGER.info("WebSocket connection opened: " + session.getId());
    }

    @OnClose
    public void onClose(Session session) {
        sessions.remove(session);
        LOGGER.info("WebSocket connection closed: " + session.getId());
    }

    @OnMessage
    public void onMessage(String message, Session session) {
        LOGGER.info("Received message from client: " + message);
    }

    public void broadcast(Object message) {
        ObjectMapper mapper = new ObjectMapper().registerModule(new JavaTimeModule()).disable(SerializationFeature.WRITE_CHAR_ARRAYS_AS_JSON_ARRAYS);
        try {
            String jsonMessage = mapper.writeValueAsString(message);
            for (Session session : sessions) {
                session.getBasicRemote().sendText(jsonMessage);
            }
        } catch (IOException e) {
            LOGGER.info("Error sending WebSocket message: " + e.getMessage());
        }
```

```
36      }
37  }
```

Listing 5.2: Websocket Endpoint

The front-end receives these updates in real time and renders them as execution cards. Each message contains:

- `executionName` (e.g., PLC+Connector, Connector Service Request, Connector Service Response etc..)
- `description`
- `timestamp`
- `decodedTelegram`
- `kafkaTopic`
- `metaData`
- `kafkaMessage`

5.2.7 Redis-Based Temporary Storage

Redis is used as a fast-access temporary store saving the identifiers of the telegram if available throughout the cycle from the point when PLC request instructions. Specifically:

- Identifiers with Telegram Types are stored under keys like `TelegramType:<identifier>`.
- These values are later retrieved to provide end-to-end context during front-end visualization.

5.2.8 Error Handling

Each consumer and processor implements exception handling to ensure resilience and fault isolation. Common scenarios include:

- Kafka de-serialization failures.
- JSON parsing errors.
- WebSocket broadcasting exceptions.
- Redis command timeouts or connection errors.

The consumer architecture (See Figure 5.1) enables the system to perform reliably in real-time environments while keeping extensibility for future enhancements.

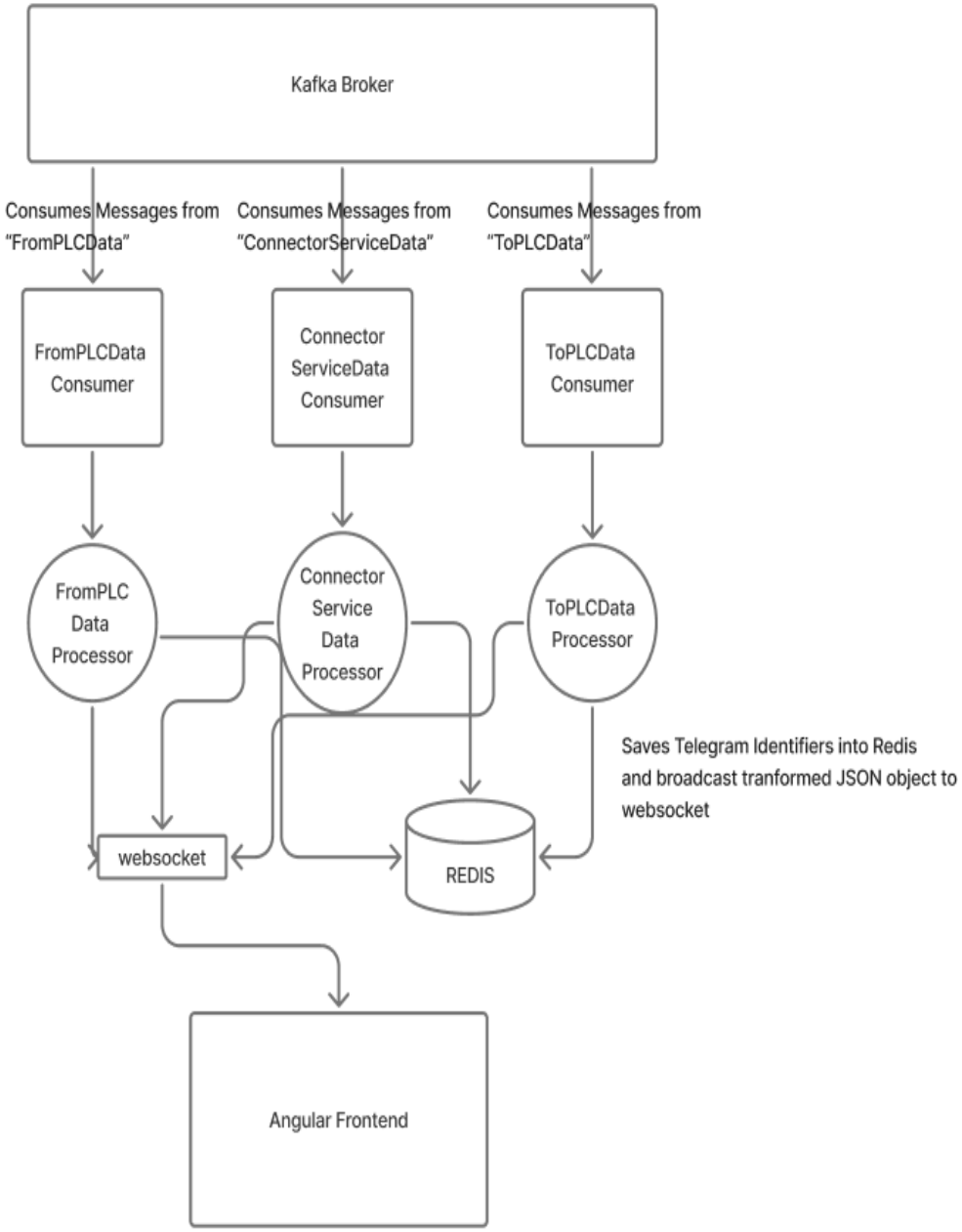

Figure 5.1: processing Flow of consumers and processors

5.3 Front-end Implementation

The system's front-end is designed and developed using Angular. The data is displayed in a card-based format that is easy to understand even for non-technical users. BMW DS icons are used to design elements ensuring a consistent and modern appearance.

5.3.1 Component Structure

There are two key components in an Angular front-end responsible for rendering and managing real-time data(See Figure 5.2) :

- **ExecutionLiveComponent** – This acts as the container for all incoming WebSocket messages. It gets the WebSocket updates from WebSocket service.
- **Websocket Service** - This WebSocket service (See Listing 5.4) listens to the WebSocket endpoint and distributes the messages to other components.
- **ExecutionInfoCardComponent** – Represents an individual execution. It receives structured input from the ExecutionLiveComponent and displays details such as telegrams, timestamps, Kafka topics, and decoded metadata. One thing to note here is that metadata can be different based on the type of the telegram the system processes.

Each execution card expects the following data structure:

```
@Input() execution!: {
  name: string,
  decodedTelegram: string,
  rawKafkaMessageJson: string,
  timestamp: string,
  kafkaTopic: string,
  description: string,
  metaData: any
};
```

Listing 5.3: Execution Card Input Model

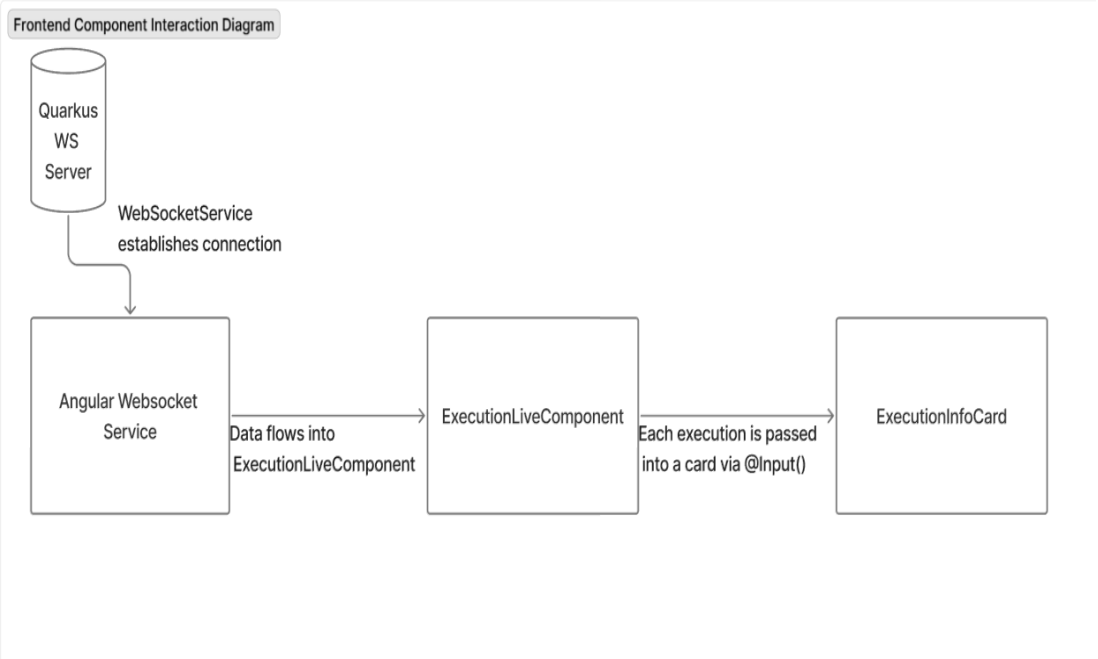

Figure 5.2: Front-end Component Interaction Diagram

5.3.2 Real-Time Updates via WebSocket

The Angular application establishes connection to the Quarkus WebSocket endpoint using RxJS's `webSocket` utility. This logic is encapsulated in the `WebsocketService` (See Listing 5.4).

```
@Injectable({ providedIn: 'root' })
export class WebsocketService {
  private readonly WS_ENDPOINT = 'ws://<raspberry-pi-ip>:9090/ws/executions';
  private socket$: WebSocketSubject<any>;

  constructor() {
    const config: WebSocketSubjectConfig<any> = {
      url: this.WS_ENDPOINT,
      deserializer: (msg: MessageEvent) => JSON.parse(msg.data),
      serializer: (value: any) => JSON.stringify(value)
    };
    this.socket$ = webSocket(config);
  }

  public onMessage(): Observable<any> {
    return this.socket$.asObservable();
  }

  public sendMessage(message: any): void {
    this.socket$.next(message);
  }

  public close(): void {
    this.socket$.complete();
  }
}
```

Listing 5.4: WebSocket.service.ts

Messages received by the WebSocket are formatted beforehand in the backend and rendered live on the UI as execution cards.

5.3.3 Filtering and Search

The UI offers multiple ways to filter the executions:

- **Name-Based Filtering** – Execution cards are filterable by name (e.g., "PLC + Connector", "Connector Service Request") through a filter chip component.

- **Part Number Search** – System allows users to search for specific executions based on `identifier` within metadata.

5.3.4 Styling and Layout

BMW DS and Angular Material components provide a responsive layout. Execution cards are vertically stacked and connected with lines to visually represent the chronological flow of data.

The live visualization of execution flows enables both technical and non-technical users to understand and trace the system behavior in real time.

5.4 Data Flow and Message Lifecycle

This section demonstrates how the data actually flows across the system, from a point when PLC requests something to when the PLC receives the response and tells the Connector to trigger a success message.

5.4.1 PLC Sends Request Telegram via Connector

The interaction between the machinery of the factory and our system starts when the request telegram is sent by the Programmable Logic Controller (PLC). This telegram is a structured message, one that contains either instructions or data, needed to either initiate or carry on with a production task.

Data is sent from the PLC to the Kafka broker, through a middle component called connector. The job of the Connector is to take the data from the PLC and format it into JSON before sending it onto the broker. Once the data is in json then the connector publishes that JSON data to the Kafka topic called **FromPLCData**.

This message encompasses significant metadata. It is a telegram that has been encoded in Base64. It also contains particulars regarding the source PLC and the Connector that published the message. The message comes enclosed with the data that must also be decoded by the Connector Service and sent elsewhere for further processing.

This is what the data looks like when it arrives in the FromPLCData Kafka topic and consumed by the consumer (see Figure 5.5) :

```
{
  "namespace": "bmw.connector.schemas.connector-from-plc-data",
  "payload": {
    "type": "BASE64ENCODED_TELEMETRY",
    "data": {
      "value":
    "MDIzMDBTRkRCTUVTU0ZEVFNQU0RBVEVOS0VJTjAxMDBGRzAwMDBDNFQyNTIxODIwMDAwMA==",
      "status": "Good"
    },
    "dataSource": {
      "globalIdentifier": "d7b2b184-6b0e-42a2-a770-fc8b75cc4946",
      "name": "10.174.9.213",
      "sourceType": "Industry-Components:PLC",
      "protocolType": "INTERNAL_PROTOCOL",
      "aliases": {
        "akzMasterstring": "NotAvailable",
        "itamAssetId": "2601908"
      }
    }
  },
  "id": "02e119ef-ebc1-4da1-85e3-0bccd65bd03d",
  "schemaVersion": "1.3.0",
  "metadata": {
    "timestamp": "2025-01-17T13:30:36.102Z",
    "senderIdentifier": {
      "id": "d7b2b184-6b0e-42a2-a770-fc8b75cc4946",
      "name": "Connector-KmVmId_INT.TRYFACTORY.1-Version_v2.4.1",
      "idType": "INTERNAL_PROTOCOL"
    }
  }
}
```

Listing 5.5: Kafka Message from PLC via Connector to FromPLCData Topic

The key fields include:

- `payload.data.value` — This field contains the telegram encoded in Base64 format.
- `dataSource` — This field includes metadata about the PLC (IP address, source type, connector version).
- `timestamp` — The exact time the message was sent.

After being published to Kafka, this message is consumed by the FromPLCData-Consumer, decoded, and passed to the processor for real-time visualization. This is where our first execution card arrives at the front-end with name ”PLC+Connector” (see Figure 1) This marks the beginning of a full message life cycle through our architecture.

5.4.2 Connector Service Transforms the Telegram

Now when the request message is published in the FromPLCData Kafka topic, Connector service consumes the data from there and trims the telegram down so that PLC Answer Service can easily builds up information object required by PLC to proceed further. When the Connector Service is done transforming the telegram, it publishes the telegram data as JSON into ConnectorServiceData Kafka topic with key "fromPLC". This is what the data looks like after being processed by the Connector Service (see Listing 5.6):

```
"Key":"fromPLC",
"message":
{
  "source": "300SFD2_",
  "sender": "300SFD2_",
  "identifier": "INTERNAL_TELEGRAM_TYPE",
  "acknowledgementInfo": "TRAN",
  "version": "0002",
  "plantIdentifierHeader": "02",
  "originalSource": "300SFD2_",
  "tgIdentifier": "INTERNAL_TELEGRAM_TYPE",
  "transactionIdentifier": "32956654142729688#",
  "statusCode": "OKAY",
  "normal_simulated": "N",
  "componentPartProduced": "T1",
  "partNumberPartProduced": "32956654",
  "timeComponentProduced": "20250313142729",
  "plantIdentifierPayload": "02",
  "checkpoint": "Z1000   ",
  "identNr": "956654",
  "transponderType": "T1",
  "variantIdComponentProduced": "S",
  "partNumberComponentProduced": "ORDERxxxx ",
  "IpsPartNumberRevisionIndex": "00",
  "NumberPartsPredecessorsOfNewComponent": "00",
  "predecessors": []
}
```

Listing 5.6: Kafka Message from Connector Service into ConnectorServiceData Topic

This is also where our ConnectorServiceData Consumer consumes the data from the Kafka topic and passes the data down to its Processor, the processors again take the required fields out from the JSON like **partNumberPartProduced**, **transactionId**, **identifier**, **checkpoint** and builds up a JSON for the visualization on the front-end. This is where our second execution card arrives at the front-end with name "Connector Service Request" (see Figure 2):

5.4.3 PLC Answer Service

When the data is available in the ConnectorServiceData topic, the PLC Answer Service consumes the data from that topic. It builds up the JSON object required by the PLC to proceed further, when the service completes building the data. The

data is then published on the same ConnectorServiceData kafka topic with key "toPLC", this is the response data in the ConnectorServiceData (see Listing 5.7

```
"Key":"toPLC",
"message":
{
  "source": "IPS#####",
  "sender": "IPS#####",
  "identifier": "INTERNAL_TELEGRAM_TYPE",
  "acknowledgementInfo": "TRAN",
  "version": "0002",
  "plantIdentifierHeader": "02",
  "originalSource": "300SFD2_",
  "tgIdentifier": "INTERNAL_TELEGRAM_TYPE",
  "transactionIdentifier": "32956654142729688#",
  "statusCode": "OKAY"
}
```

Listing 5.7: Kafka Message from PLC Answer Service into ConnectorServiceData Topic

Here Our ConnectorServiceData Consumer again consumes the data and passes it down to its processor, which again builds the Object for the visualization and sends the data to front-end Through WebSocket, here our third visualization card arrives at the front-end named "PLC Answer Service" (see Figure 3).

5.4.4 Connector Service Response

When the response data is available in ConnectorServiceData kafka topic, Connector Service consumes it and checks the key, if its "toPLC". Connector Service converts the data back to PLC format which is basically a telegram and puts the data on the ToPLCData Kafka topic. The data then looks like this (see Listing 5.8).

```
"message":
{
  "id": "6ef3d73a-1703-41c6-ad93-a469ac778914",
  "schemaVersion": "2.1.0",
  "metadata": {
    "senderIdentifier": {
      "id": "APPID-123456",
      "idType": "ConnectIT",
      "name": "Your Business Application Name"
    },
    "timestamp": "2025-01-17T13:25:21.639Z"
  },
  "namespace": "bmw.connector.schemas.connector-to-plc-data",
  "payload": {
    "jobId": "0d57f4d1-bdde-4245-9dfe-ad58add6846a",
    "status": "open",
    "connectionIds": [
      "96ca2317-f291-40be-9aa4-3c898e3c5de3"
    ],
    "jobOrigin": "APPID-123456",
    "contractSpecificMetadata": {
      "$schema": "http://your-business-defined-schema.json",
      "yourJobProperty": "001"
```

```
    },
    "requestType": "INTERNAL_PROTOCOL",
    "payloadRaw": {
      "requests": [
        {
          "value": "Qk1FU1NGRFRTRkRUQk1FU1NQU0ZSR0l
          PS0VJTjEyMzRGRzU1NTQ1Njc4OTAx
          MjEyMzQ1Nnh4eDEyMzRJT3h4eA=="
        }
      ],
      "responses": []
    },
    "timeToLiveMs": 60000
  }
}
```

Listing 5.8: Kafka Message from Connector Service into ToPLCData Topic

This is where our ToPLCDataConsumer comes in, it consumes the data from ToPLCData, builds a JSON object for visualization and sends it to front-end through WebSocket. This is where our fourth execution card appears at the front-end named ”Connector Service Response” (see Figure 4).

5.4.5 Connector Success Message

When the Response Data arrives in the ToPLCData, it is then consumed by the Connector and sent down to the PLC. When the PLC receives the data successfully, Connector sends a success message in the ToPLCData Kafka topic (see Listing 5.9).

```
"message":
{
  "namespace": "bmw.connector.schemas.connector-to-plc-data",
  "payload": {
    "jobId": "0d57f4d1-bdde-4245-9dfe-ad58add6846a",
    "jobOrigin": "APPID-123456",
    "connectionIds": [
      "96ca2317-f291-40be-9aa4-3c898e3c5de3"
    ],
    "status": "delivered",
    "requestType": "INTERNAL_PROTOCOL",
    "timeToLiveMs": 60000,
    "contractSpecificMetadata": {
      "$schema": "http://your-business-defined-schema.json",
      "yourJobProperty": "001"
    },
    "payloadRaw": {
      "requests": [
        {
          "value": "Qk1FU1NGRFRTRkRUQk1FU1NQU0Z
          SR0lPS0VJTjEyMzRGRzU1NTQ1Njc4
          OTAxMjEyMzQ1Nnh4eDEyMzRJT3h4eA=="
        }
      ],
      "responses": [
        {
          "status": {
            "isSuccessful": true,
            "message": "Success"
```

```
          }
        }
      ]
    }
  },
  "id": "1461cd2b-ef5f-4f05-a045-5bd33a4582c7",
  "schemaVersion": "2.1.0",
  "metadata": {
    "timestamp": "2025-01-17T13:25:22.0327438Z",
    "senderIdentifier": {
      "id": "96ca2317-f291-40be-9aa4-3c898e3c5de3",
      "name": "Connector",
      "idType": "ConnectionId"
    }
  }
}
```

Listing 5.9: Kafka Message from connector into ToPLCData Topic

Here again our ToPLCData Consumer consumes the data and builds a final success object and sends it to front-end. This marks the complete cycle of a single process workflow, it keeps going on until a part is fully produced, and the "success" message execution card is received at the front-end named "Connector Success" (see Figure 5). This means that a single step in part production process is complete. The PLC can again request for the next step and follows the same flow until the part is fully produced.

5.5 Deployment

The completed system emphasizes portability and easy deployment at the edges, ensuring that everything works across the many different settings from which the system can be run, ranging from local development setups to edge computing devices like the Raspberry Pi or other.

The backend was packed in a Docker container using dockerfile (See Listing 5.10) and compiled into an image compatible with ARM (which is the architecture of the Raspberry Pi). The environment variables and application properties used in the Quarkus application were configured in such a way as to allow it to run in both a staging and a production environment. Once the Docker image was built, it was pushed to Docker Hub and pulled down to the Raspberry Pi, where it was running using standard Docker commands. In addition, the supporting services that the Quarkus application required (such as Redis) were also containerized, reducing the installation complexity.

```
# syntax=docker/dockerfile:1
FROM --platform=$BUILDPLATFORM maven:3.8.8-eclipse-temurin-17 AS build
WORKDIR /app

COPY pom.xml .
RUN mvn dependency:go-offline -B

COPY src ./src
RUN mvn clean package -DskipTests

FROM --platform=$TARGETPLATFORM eclipse-temurin:17-jre
WORKDIR /app

COPY --from=build /app/target/quarkus-app/lib/ /app/lib/
COPY --from=build /app/target/quarkus-app/app/ /app/app/
COPY --from=build /app/target/quarkus-app/quarkus/ /app/quarkus/
COPY --from=build /app/target/quarkus-app/quarkus-run.jar /app/quarkus-run.jar

EXPOSE 9090
CMD ["java", "-jar", "/app/quarkus-run.jar"]
```

Listing 5.10: Dockerfile for the backend

following command was used to build the docker image:

```
docker buildx build --platform linux/arm64 -t {dockeruname}/
    reponame:tag --push .
```

Listing 5.11: Building Docker Image for ARM64

this is the docker-compose file:

```
    version: '3.8'

services:
  backend-app:
    image: <backend-image-from-registry>
    container_name: backend-app
    ports:
      - "9090:9090"
    environment:
      - QUARKUS_PROFILE=prod
      - KAFKA_BOOTSTRAP_SERVERS=${KAFKA_CLOUD_BROKER}
      - KAFKA_SASL_JAAS_CONFIG=${KAFKA_SASL_JAAS_CONFIG}
      - KAFKA_SECURITY_PROTOCOL=SASL_SSL
      - KAFKA_SASL_MECHANISM=PLAIN
      - REDIS_HOST=redis
    volumes:
      - ./config:/app/config
    depends_on:
      - redis

  redis:
    image: redis:latest
    container_name: redis
    ports:
      - "6379:6379"
    command: ["redis-server", "--appendonly", "yes"]
    volumes:
      - redis-data:/data

```

```
volumes:
  redis-data:
```

Listing 5.12: docker-compose used to run all the images

5.6 Summary

This chapter introduced the real-time data flow visualization system, with an emphasis on how the system was designed and implemented. A modular, event-driven architecture allows us to scale the system beyond the point where one server can handle all the work and requires adding many more servers to cope with the load. This architecture also enables us to reach high performance and makes the system react quickly to incoming data.

Most of the message processing happens in three main consumers, which are parts in the architecture that handle messages from one or more specific topics that are relevant to them. These consumers are coupled with processors that decode the messages and get the data ready for visualization on the front-end. After they finish, they use WebSocket to communicate with the front-end part of the system.

The UI is styled using the same design system that BMW uses for its products. The incoming execution flows from the PLC are displayed as a set of scroll-able cards, where each card contains messages that can be filtered so that only the cards relevant to the user are visible.

6 Evaluation

The implemented system brought significant improvements in transparency and remote access in TryFactory environment. Previously, when an item was scanned or if the process button was activated, observers and operators merely had to rely on physical movement of parts or equipment—without any understanding of what exactly was occurring within the inherent functions of the IT and OT layers. This transparency deficit rendered the troubleshooting and process analysis activities more complex. Following the installation of Real-time visualisation system developed in this thesis allows users to monitor the complete message cycle, with the way each request is handled in both different systems such as PLC, ConnectorService, and Connector. This clear and descriptive sequence Access to information allows stakeholders to deepen their understanding of system behavior at each stage. Besides, when enabling web-based access, the system allows users to monitor the production processes anywhere on the planet, thereby annihilating the functional constraint of being close to the central facility and maintaining cooperation among geographically distributed teams.

6.1 Functional Requirements Fulfillment

This section contains a detailed evaluation of the extent to which the system satisfies the functional requirements set out in Chapter 3. Each requirement is assessed with respect to three criteria: its implementation status, the completeness of that implementation, and the degree to which what is implemented corresponds to the original intent that the requirement should serve. The evaluation is presented in two parts, the first dealing with requirements for the front-end, the second with those for the backend.

6.1.1 Front-end Evaluation

The system's front end was developed with a strong focus on real-time visualization, user-friendliness, and adherence to BMW's design principles. Until now, the system has successfully implemented every significant functional requirement

that the user specified and that the system can verify. The requirements on the front-end for this project were identified and are presented in summarized form (see Table 6.1).

Table 6.1: Evaluation of Front-end Functional Requirements

Requirement	**Status**	**Details**
Real-time Visualization of Execution Data	Fulfilled	Each execution card displays data such as telegram, timestamp, service name, and Kafka topic in chronological order.
WebSocket-based Live Updates	Fulfilled	Live WebSocket connection ensures real-time updates without the need for polling.
Filtering by Service and Identifier	Fulfilled	Users can filter executions by service type and specific part number using a chip-based filter interface.
Execution Summary and Status Indicators	Fulfilled	Execution Success should appear if process is completed
BMW Design System Integration	Fulfilled	All UI components follow BMW DS standards for styling and branding.

6.1.2 Backend Evaluation

All key functional specifications for the ingestion, transformation, decoding, and real-time broadcasting of Kafka telegrams were met successfully by the backend. Moreover, essential backend processes were implemented effectively. (see Table 6.2 for an overview and evaluation of each backend feature).

Table 6.2: Evaluation of Backend Functional Requirements

Requirement	Status	Details
Kafka Topic Consumption	Fulfilled	Kafka consumers are configured for all relevant topics: `FromPLCData`, `ConnectorServiceData`, and `ToPLCData`.
WebSocket Broadcasting	Fulfilled	All processed telegrams are sent to the front-end in JSON format using a shared WebSocket endpoint.
Telegram Decoding and Flow Direction Mapping	Fulfilled	Telegrams are decoded and categorized using key (e.g., "fromPLC" or "toPLC") to determine flow direction.
Error Handling	Fulfilled	All consumers and processors implement try-catch mechanisms and error logging to ensure resilience.

6.2 Evaluation of Non-Functional Requirements

the system was also evaluated on non-functional requirements to ensure real-world applicability, performance reliability, and maintainability. And the results are discussed in the evaluation below.

Performance (Low Latency): A main aim was achieving low latency between the ingestion of Kafka telegrams and their visualization on the front-end. This was done to enable near real-time responsiveness for our users. How was this done? For one, Quarkus's reactive programming model is used. For another, efficiently de-serialized Kafka messages. and lastly, WebSocket broadcasting to UIs. Latency was measured by checking the timestamps of each execution when it arrived at front-end.

Scalability: The system can be easily scaled because it is designed around event-driven Architecture and is topic-based. Each topic is handled by a separate Kafka consumer, which is independently managed. New consumers can be added, or the message flow can be expanded, without any redesigning of the core system. Because it is modular, and can already adapt to future changes.

Reliability and Fault Tolerance: Resilience mechanisms are put in place all over the system. Each Kafka consumer has its try-catch blocks for de-serialization and processing. When it comes to WebSocket broadcasts, they are safely wrapped

up to ensure system safety. Any malformed telegrams or unexpected JSON structures are logged and ignored. If anything goes wrong in these limited system areas, the overall system continues working without interruption.

Security: Kafka connections are secured with SSL encryption and SASL authentication, allowing for a safe and sound method of getting data to and from cloud-hosted Kafka brokers. Sending and receiving messages is a fairly straightforward process; most of the complexity happens during the initial connection setup, mostly due to the need for secure transmission of data. Given that sensitive credentials like API keys and topic names that specify where the messages should go are involved.

Maintainability and Modularity: The architecture of the system maintains a clear separation of concerns. Every Kafka topic is processed by its own class of processors. Services contain the common logic that was needed in multiple places. Components in the front-end, like ExecutionLiveComponent and ExecutionInfoCardComponent, also follow Angular's modular design. This is beneficial for maintainability because it allows us to make future updates or add features, and do it in a way that is isolated and, therefore, safe.

Portability: Because of Quarkus's native image support and lightweight runtime, the backend runs nicely in constrained environments like Raspberry Pi. Raspberry Pi successfully run Docker containers that were built using multi-stage builds. This demonstrates not only the portability of the full stack but also the efficacy of using Docker with ARM architectures.

6.3 System Scalability

The system is built to scale and easily handle an expansion in data size and an increase in functionality. Kafka handles the job of registering new topics well and has great throughput. The scalable modular design of Kafka consumers ensures that new data streams can be easily registered with new service instances. Overall, the combination of Kafka and new services that needs to be added in future guarantees a smooth ingestion of expanded data size without any service interruptions.

As operations in factories become more complex, Kafka topics that represent new machines or services can also be integrated easily, accomplished by adding new processors and doing so without making any changes to the core backend operation.

In addition, the system is compatible with the cloud and containerized in Docker,

so it can be deployed in a variety of environments from local Raspberry Pi installations to cloud infrastructures at scale. Its lightweight resource requirements and environment-variable configurations make the system very amenable to horizontal scaling in orchestration platforms.

6.4 System Evaluation Summary

The system successfully met the core objectives defined during the requirement phase and demonstrated robust performance during integration with real-time Kafka data streams from BMW's TryFactory ecosystem. All functional components Kafka topic consumers, topic-specific processors, WebSocket-based broadcasting, and Angular-based frontend performed reliably in both edge-based deployments and test environments. The real-time visualization tool allowed end-to-end tracking of telegram flows throughout services like Connector Service and PLC Answer Service, providing clear view on the message life cycle for both technical and non-technical users. Live filtering, decoding of telegrams, and service identification were performed with minimal latency, validating the system's responsiveness and modularity. Together with the technical benefits, the system earned excellent reviews from both shop floor workers, no matter their level of IT know-how, and upper management. They praised the remarkable transparency and intuitiveness of the tool in showing how machines and IT systems exchanged data. Such clarity had never been achieved before. This made it even more apparent how vital the visualization approach is in creating a bridge between operational processes and the digital insights that underpin them.

Overall, the system not only delivered operational transparency and reduced debugging complexity but also laid a scalable foundation for future enhancements.

7 Conclusion

7.1 Overview Of This Thesis

This thesis investigated the design and execution of a real-time visualization tool for interpreting message flows in a smart factory context, specifically in BMW's TryFactory environment. The aim was to close the visibility gap between discrete layers in a manufacturing system from PLCs and Connector to MES services that conceals the real-time workings of the systems. All of this needed to be done using a tool that engineers and other stakeholders use.

The core components of the system include:

- A **Quarkus-based backend** that consumes and processes Kafka messages.
- A **WebSocket layer** for real-time front-end updates.
- An **Angular front-end** displaying execution flows using a user-friendly card-based UI.
- A **flexible deployment model**, tested on both local setups and Raspberry Pi.

An event-driven architecture was adopted for the solution, with a high focus on three key attributes: modularity, scalability, and low-latency updates. This system is what helps solve the factory's transparency problem and allows the shop floor to gain a clearer view of the interactions taking place within the factory.

7.2 Achievements And Key Contributions

This thesis reached several important and beneficial results. They show the utility, integrity, and actual applicability of the TryFactory system.

Real-Time Data flow Visualization A completely operational framework was created to monitor and see the interactions of messages in real time based

on Kafka. This meant tracing each message, whether a request or a response, around the many different services and different stages across the whole system.

End-to-End Message Life Cycle Implementation The complete life cycle of a telegram was captured and represented successfully by the system from its origin at the PLC through Connector. All the way up to response generation and delivery acknowledgment from the PLC. Each transformation and forwarding stage was visualized as a separate execution card. That made it easy to view the transaction step by step.

WebSocket-Based Reactive Architecture WebSocket was used to achieve real-time updates, pushing changes instantly from the backend to the front-end. This enabled seamless visualization of telegrams without polling or delays and with very low latency in message propagation.

Modular, Event-Driven Backend The backend was constructed using Quarkus, a reactive, event-driven Java framework. Each Kafka topic possessed its own consumer and processor pipeline, which enhanced modularity, maintainability, and traceability.

Filtering and Execution Analysis Capabilities The front-end had smart filtering features that let the user shrink the execution flows to just the parts they were interested in. The need was to deliver meaningful events to the user for them to even think about isolating, let alone analyzing, individual message paths.

Portable Deployment Model The system was deployed onto a Raspberry Pi, using Docker. The whole system ran well on the edge device. If you wanted to go in the opposite direction and scale the same system all the way up, you could easily deploy it to the cloud.

Robust Error Handling and Logging Every aspect of the work involved in parsing messages from Kafka, sending them through WebSocket, and processing them was wrapped in error-handling mechanisms. The aim was to protect the system from crashing, to keep the logs clear and well-ordered, and to benefit the users and for the enhancement of debugging.

this not just narrowed the visibility gaps in TryFactory's digital manufacturing pipelines but also provide a scalable, real-time operational monitoring solution for modern Industry 4.0 environments.

7.3 Limitations

The system achieved its main purposes of displaying real time data flows based on Kafka. However, it was constrained by several limitations that prevented it from achieving a truly optimal state. some of the limitations are:

- Insufficient Metadata Visibility: In some telegrams, detailed metadata such as part numbers or internal service identifiers were either missing or embedded in unstructured formats. As a result, not all flows could be fully filtered.
- Limited Factory Integration: Due to access limitations, the system was not directly integrated with live Connector, or TryFactory microservices. It depended on Kafka topics and data structures shared in advance, which could differ from real-time operational deployments.
- Basic UI Feedback: The fundamental user interface feedback provided by the system could be improved to give a clearer picture of the system's current activity. Right now, it only supports real-time updates and basic status indicators. These could be enhanced to give the end-user a more manageable view of the system's activities. As it stands, there are no advanced visualization features to speak of for the direct-user interface.

7.4 Future Work and Enhancements

There are several promising directions in which this project could be extended:

- Telegram Pairing via Machine Learning: An advanced pairing mechanism that uses pattern recognition, deep inspection, or even ML-based correlation models can render the request-response relationships redundant and recover them in an asynchronous setup.
- Advanced Filtering and Aggregation: Providing insight into factory workflows could be accomplished by extending the front-end to include several elements, such as: advanced filtering logic graphical KPIs and timelines.
- For the structure and data that is available at this time, there are some Key Performance Indicators (KPI's) that can be easily added for some more insights like Round Trip time from PLC to MES and back to PLC.

Appendices

A Screenshots of Execution Cards

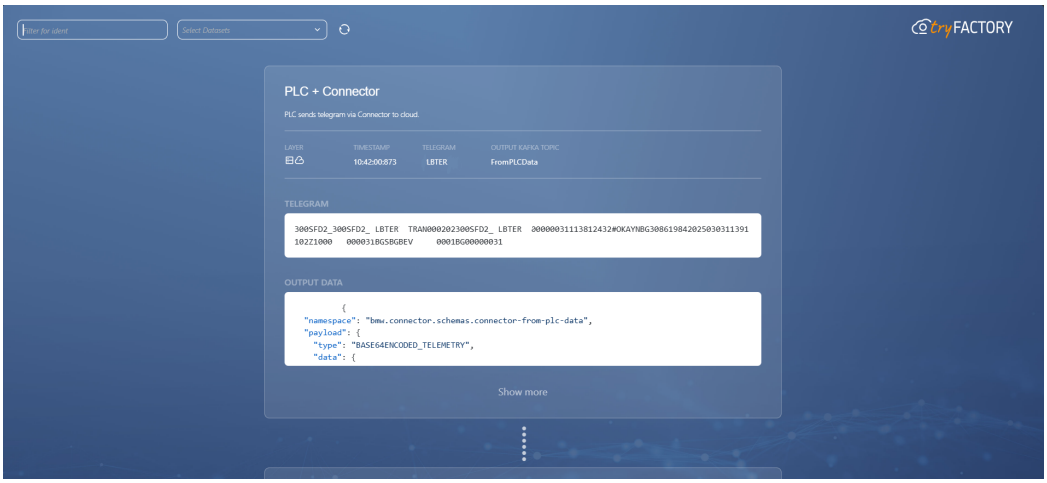

Figure 1: Execution card for PLC+Connector request

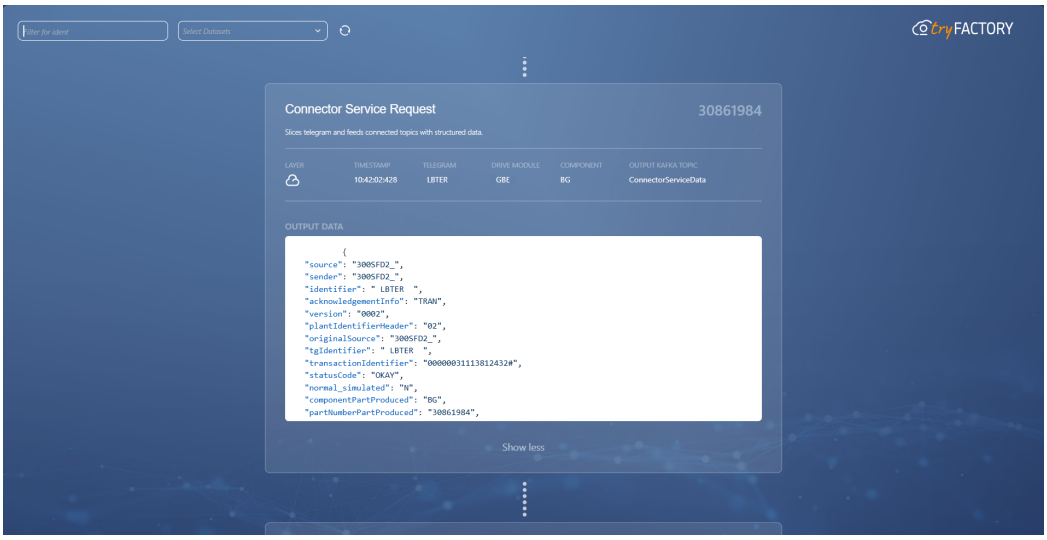

Figure 2: Execution card for connector service request

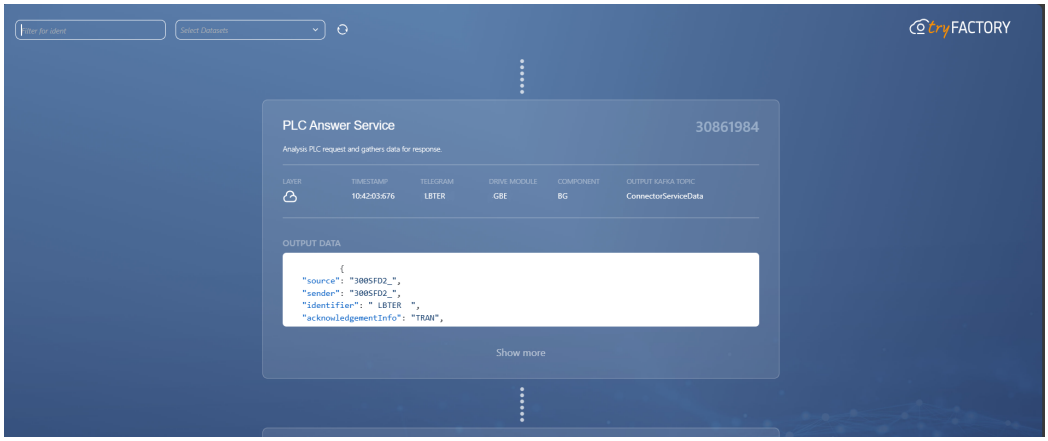

Figure 3: Execution card for PLC Answer Service

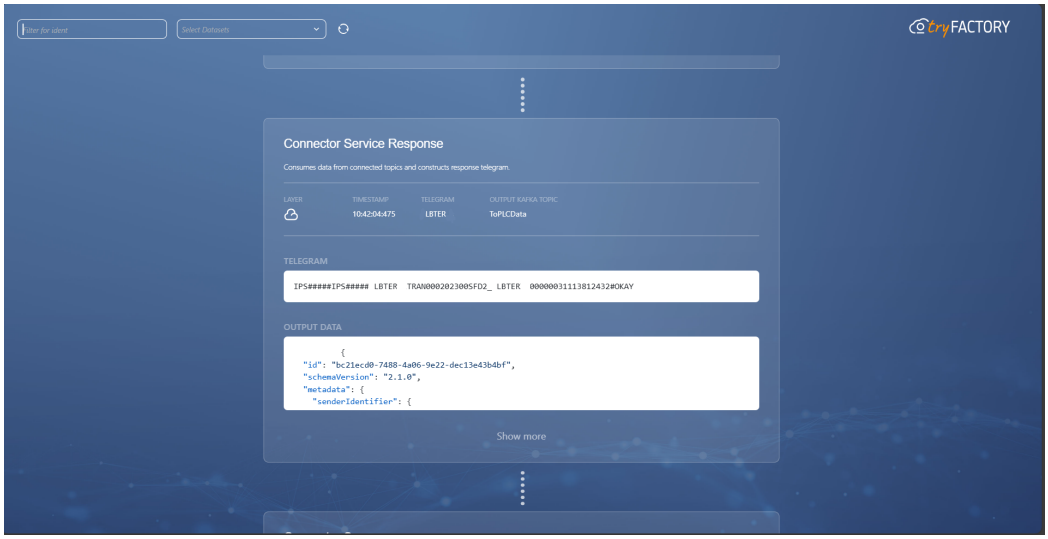

Figure 4: Execution card for connector service Response

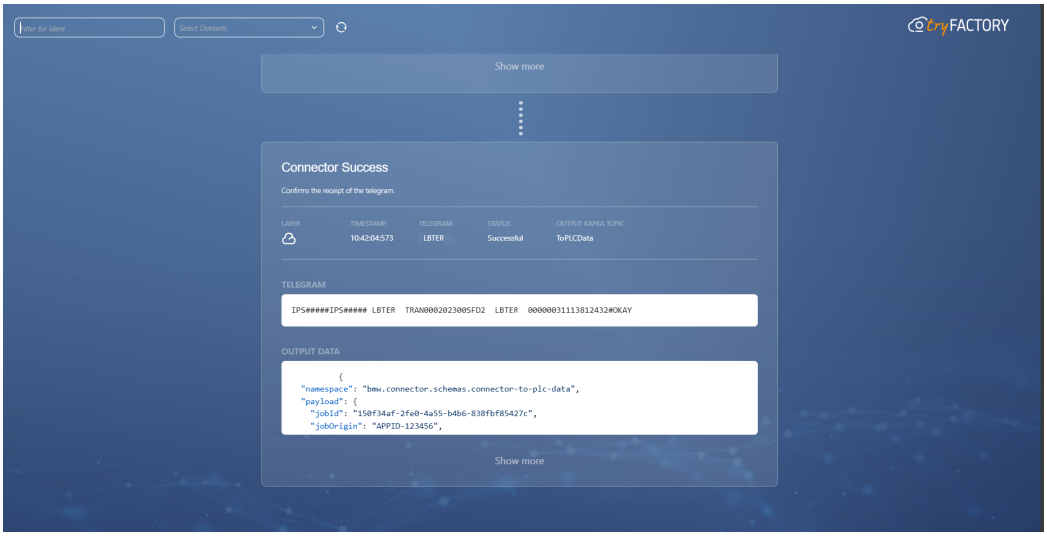

Figure 5: Execution card for connector Success Message

References

Adeleke, A. (2024). Intelligent monitoring system for real-time optimization of ultra-precision manufacturing processes. *Engineering Science & Technology Journal*, *5*(3), 803–810. https://doi.org/10.51594/estj.v5i3.904

Bansal, S. (2023). Kafka, a catalyst for real-time data innovation across diverse domains. *International Journal of Latest Engineering and Management Research (IJLEMR)*, *8*(10), 66–70. https://doi.org/10.56581/ijlemr.8.10.66-70

Bello, L., Mariani, R., Mubeen, S., & Saponara, S. (2019). Recent advances and trends in on-board embedded and networked automotive systems. *IEEE Transactions on Industrial Informatics*, *15*(2), 1038–1051. https://doi.org/10.1109/TII.2018.2879544

Bozkurt, A., Ekici, F., & Yetişkul, H. (2023). Utilizing flink and kafka technologies for real-time data processing: A case study. *The Eurasia Proceedings of Science Technology Engineering and Mathematics*, *24*, 177–183. https://doi.org/10.55549/epstem.1406274

Buer, S., Strandhagen, J., Semini, M., & Strandhagen, J. (2020). The digitalization of manufacturing: Investigating the impact of production environment and company size. *Journal of Manufacturing Technology Management*, *32*(3), 621–645. https://doi.org/10.1108/jmtm-05-2019-0174

Chen, P. (2019). Visualization of real-time monitoring datagraphic of urban environmental quality. *Eurasip Journal on Image and Video Processing*, *2019*(1). https://doi.org/10.1186/s13640-019-0443-6

D'Antonio, G., Bedolla, J., Genta, G., Ruffa, S., Barbato, G., Chiabert, P., & Pasquettaz, G. (2016). Plm-mes integration: A case-study in automotive manufacturing. https://doi.org/10.1007/978-3-319-33111-9_71

Fradejas-García, I., Casado, R., & Bouchachia, A. (2016). An incremental approach for real-time big data visual analytics. *Proceedings of the 4th IEEE International Conference on Future Internet of Things and Cloud (FiCloud)*, 177–182. https://doi.org/10.1109/w-ficloud.2016.46

Fu, G., Zhang, Y., & Yu, G. (2021). A fair comparison of message queuing systems. *IEEE Access*, *9*, 421–432. https://doi.org/10.1109/access.2020.3046503

Iftikhar, N., Lachowicz, B., Madarasz, A., Nordbjerg, F., Baattrup-Andersen, T., & Jeppesen, K. (2020). Real-time visualization of sensor data in smart manufacturing using lambda architecture. *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER)*, 215–222. https://doi.org/10.5220/0009826302150222

Jankovic-Zugic, A., Medić, N., Pavlović, M., Todorović, T., & Rakić, S. (2023). Servitization 4.0 as a trigger for sustainable business: Evidence from automotive digital supply chain. *Sustainability*, *15*(3), 2217. https://doi.org/10.3390/su15032217

Kannan, S., et al. (2017). Towards industry 4.0: Gap analysis between current automotive mes and industry standards using model-based requirement engineering. https://doi.org/10.1109/icsaw.2017.53

Karabegović, I., & Husak, E. (2018). Industry 4.0 based on industrial and service robots with application in china. *Mobility and Vehicle Mechanics*, *44*(2), 59–71. https://doi.org/10.24874/mvm.2018.44.04.04

Karabegović, I., & Karabegović, E. (2019). The role of collaborative service robots in the implementation of industry 4.0. *International Journal of Robotics and Automation Technology*, *6*. https://doi.org/10.31875/2409-9694.2019.06.5

Karabegović, I., Karabegović, E., Mahmić, M., & Husak, E. (2021). The application of industry 4.0 in production processes of the automotive industry. *Mobility and Vehicle Mechanics*, *47*(2), 35–44. https://doi.org/10.24874/mvm.2021.47.02.03

Khan, M. A., & Alghazzawi, D. (2021). A real-time data stream framework for visualizing kafka messages in smart manufacturing.

Lampropoulos, G., Siakas, K., & Anastasiadis, T. (2019). Internet of things in the context of industry 4.0: An overview. *International Journal of Entrepreneurial Knowledge*, *7*(1). https://doi.org/10.37335/ijek.v7i1.84

Liu, W., & Lu, S. (2023). Optimization of automotive wire harness production process based on lean manufacturing. https://doi.org/10.4108/eai.24-2-2023.2330620

Nurdiyanto, H. (2024). Critical role of manufacturing execution systems in digital transformation of manufacturing industry. *Journal of Engineering Science (JES)*, *20*(7S), 2432–2436. https://doi.org/10.52783/jes.4038

Okuyelu, O., Adaji, O., & Doskenov, B. (2024). Advancing manufacturing efficiency through real-time production monitoring and control systems. *Journal of Engineering Research and Reports*, *26*(4), 184–193. https://doi.org/10.9734/jerr/2024/v26i41125

Oman, S., Leskovar, R., Rosi, B., & , given=., giveni=. (2017). Integration of mes and erp in supply chains: Effect assessment in the case of the automotive industry. *Tehnicki Vjesnik - Technical Gazette*, *24*(6). https://doi.org/10.17559/tv-20160426094449

Parkouhi, S., Ghadikolaei, A., & Lajimi, H. (2023). Prioritizing the internet of manufacturing things (iomt) challenges in automotive industry using interpretive structural modeling (ism). https://doi.org/10.21203/rs.3.rs-1957288/v1

Patil, N., Krishna, C., & Kumar, K. (2022). Ks-ddos: Kafka streams-based classification approach for ddos attacks. *The Journal of Supercomputing*, *78*(6), 8946–8976. https://doi.org/10.1007/s11227-021-04241-1

Peres, R., Rocha, A., Leitão, P., & Barata, J. (2018). Idarts – towards intelligent data analysis and real-time supervision for industry 4.0. *Computers in Industry*, *101*, 138–146. https://doi.org/10.1016/j.compind.2018.07.004

Rho, J., Azumi, T., Yamaguchi, A., Sato, K., & Nishio, N. (2016). Reservation-based scheduling for automotive dsms under high overload condition. *Journal of Information Processing*, *24*(5), 751–761. https://doi.org/10.2197/ipsjjip.24.751

Teucke, M., Broda, E., Börold, A., & Freitag, M. (2018). Using sensor-based quality data in automotive supply chains. *Machines*, *6*(4), 53. https://doi.org/10.3390/machines6040053

Wang, L., & Wang, G. (2016). Big data in cyber-physical systems, digital manufacturing and industry 4.0. *International Journal of Engineering and Manufacturing*, *6*(4), 1–8. https://doi.org/10.5815/ijem.2016.04.01

Wei, H. (2022). Browser-based visualization of real-time kafka data streams.

Wei, X., Geng, L., & Zhang, X. (2018). Web browser based data visualization scheme for xbee wireless sensor network. *Transactions on Networks and Communications*, *6*(5). https://doi.org/10.14738/tnc.65.5261

Yamaguchi, A., et al. (2017). In-vehicle distributed time-critical data stream management system for advanced driver assistance. *Journal of Information Processing*, *25*, 107–120. https://doi.org/10.2197/ipsjjip.25.107

Zhang, H. (2023a). Research and design of real time big data visualization analysis platform based on flink. *Journal of Physics: Conference Series*, *2504*(1), 012053. https://doi.org/10.1088/1742-6596/2504/1/012053

Zhang, H. (2023b). Research and design of real-time big data visualization analysis platform based on flink. *Journal of Physics: Conference Series*, *2504*(1), 012053. https://doi.org/10.1088/1742-6596/2504/1/012053