Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Edge AI Integration for Anomaly Detection in Assembly using Delta Robot

Bachelor thesis

Vojtěch Hanzlík

Programme: Software Engineering and Technology
Branch of study: Internet of Things
Supervisor: Ing. Martin Macaš, Ph.D.

Prague, May 23, 2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hanzlík Vojtěch** Personal ID number: **503243**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Software Engineering and Technology**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Edge AI Integration for Anomaly Detection in Assembly using Delta Robot

Bachelor’s thesis title in Czech:

Edge AI integrace detekce anomálií v procesu montáže pomocí delta robota

Guidelines:

1. Deploy the provided model for anomaly detection on an edge device in the Industry 4.0 Testbed, specifically the delta robot performing assembly. The edge device may be a laptop or any other suitable device. Verify the functionality by running an experimental trial on-site and evaluating performance indicators like detection performance, computational efficiency, robustness, or impact on the robotic assembly process.
2. Implement data access from the delta robot in Python using the OPC UA protocol.
3. Explore the MLOps paradigm and analyze its applicability to anomaly detection in delta robots. Emphasize aspects such as engineering, testing, validation, deployment, and monitoring of the machine learning model.
4. Investigate the concept of Industry 5.0 and devise a strategy to incorporate human or machine feedback into the anomaly detection process for a deployed machine learning model.
5. Provide the machine learning algorithm as-a-service by leveraging the AI-on-demand platform available at https://aiexp.ai4europe.eu/.

Bibliography / sources:

[1] Rožanec, J. M., Novalija, I., Zajec, P., Kenda, K., Tavakoli Ghinani, H., Suh, S., ... & Soldatos, J. (2023). Human-centric artificial intelligence architecture for industry 5.0 applications. International Journal of Production Research, 61(20), 6847-6872.
[2] Lindner, F., & Reiner, G. (2023, May). Industry 5.0 and Operations Management—the Importance of Human Factors. In NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium (pp. 1-4). IEEE.
[3] Ordieres-Meré, J., Gutierrez, M., & Villalba-Díez, J. (2023). Toward the industry 5.0 paradigm: Increasing value creation through the robust integration of humans and machines. Computers in Industry, 150, 103947
[4] Loizaga, E., Eyam, A. T., Bastida, L., & Lastra, J. L. M. (2023). A Comprehensive study of human factors, sensory principles and commercial solutions for future human-centered working operations in Industry 5.0. IEEE Access.

Name and workplace of bachelor's thesis supervisor:

Ing. Martin Macaš, Ph.D. Intelligent Systems for Industry and Smart Distribution Networks CIIRC

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **19.02.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **15.02.2026**

Ing. Martin Macaš, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Declaration

I hereby declare I have written this bachelor thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis. Moreover, I state that this thesis has neither been submitted nor accepted for any other degree.

In Prague, May 23, 2024

.....
Vojtěch Hanzlík

Abstract

This thesis presents the development and deployment of an anomaly detection system for a multi-axis delta robot located at the Testbed for Industry 4.0 at the Czech Institute of Informatics, Robotics, and Cybernetics at the Czech Technical University in Prague. By integrating machine learning algorithms, the project analyzes multidimensional time-series data from torque and force sensors to find anomalies in the process of wheel assembly for remote-controlled vehicles. The Open Platform Communications United Architecture protocol (OPC UA) is used in the system architecture to transfer data from the delta robot. Protocol Buffers are used for structured data serialization in the general-purpose Remote Procedure Calls (gRPC) client-server architecture, which offers real-time data processing and anomaly detection. The gRPC server integrates the anomaly detection model for time series data analysis, which is packaged as a Python module for simple distribution.

With the help of a web-based user interface created with Flask and WebSockets, users can communicate with the system, see real-time results, and provide feedback. A MongoDB database is used to store historical data and human labels for inaccurate predictions, allowing continuous model development. Metrics like processing time and latency were used to assess the system's performance.

To ease the deployment and monitoring process, Machine Learning Operation Procedures have been used. Dockerization ensures a consistent deployment across all platforms. By showing time series data and anomaly detection results, integration with Grafana offers a number of monitoring options.

Keywords: Industry 4.0, Machine learning deployment, Communication protocols, Real-time systems

Acknowledgements

I would like to express my gratitude to my supervisor, Martin Macaš, for his guidance, support, and encouragement throughout this project. His expertise and insights were essential in shaping this thesis.

I would like to thank Aleš Trna for his patience and understanding during the testing and data collection sessions. The support during those times when things didn't go as intended was very much appreciated.

I am also grateful to the testbed team for their technical support and for providing the necessary working space for this project. The assistance and resources were crucial for the successful execution of this project.

List of Tables

List of Figures

List of Acronyms

gRPC general-purpose Remote Procedure Calls. x, 5–7, 9–12, 14, 15, 17, 20–24, 27

IoT Internet of Things. 14

LAN Local Area Network. vii, 18, 19

MLOps Machine Learning Operations. 1, 16, 20

MQTT Message Queuing Telemetry Transport. 14, 15

OPC UA Open Platform Communications United Architecture. x, 1, 3–5, 10, 12, 19, 27

PLC programmable logic controller. 4, 19

PyPI Python Package Index. 9, 10, 16, 27

RTT Round Trip Time. 17

UI User Interface. x, 2, 10–14, 25, 28

Contents

Chapter 1

Introduction

The project, which this thesis is based on, is part of a larger project conducted at the Testbed for Industry 4.0 at the Czech Institute of Informatics, Robotics, and Cybernetics at the Czech Technical University in Prague. This facility is equipped with various advanced robotic systems, including a multi-axis delta robot with a conveyor system, which is the core of this project. The robot is tasked with assembling small wheels for remote-controlled vehicles. The assembly process is monitored using force and torque sensors mounted on the robot. These sensors generate multidimensional time-series data, which is analyzed in real-time using machine learning algorithms to identify any anomalies within the wheel assembly process.

Incorporating Edge AI technology is a significant aspect of this project. Edge AI refers to the deployment of artificial intelligence algorithms directly on devices located at the edge of the network, instead of relying on cloud-based solutions. This approach enables real-time data processing at the source of data generation.

The main goal of this project is to develop an anomaly detection system with a human feedback, housing a machine learning model developed by Aleš Trna [1]. Deploy said solution both on-site within the Industry 4.0 Testbed, conducting testing and verification of the solution, and on the AI-on-demand platform[1], making the model accessible by wider audience. Another goal is to suggest Machine Learning Operations (MLOps) practises and their applicability in the project´s use case. Several steps have been made to achieve the desired outcomes. First task was to create a script using the OPC UA protocol to enable real-time data access from the delta robot. The project continued by developing a client-server based system and integrating such script into a client component within the client-server architecture, where it serves as a source of data that are further streamed to the server component, that houses the machine learning model providing anomaly detection. The said model was made into a Python package, making it easily distributable

[1] AI-on-demand platforml

and maintainable. To satisfy another desired goals of this thesis, a simple web-based UI was developed and integrated into the client component, making human feedback possible. For the same purpose a database component, based on MongoDB, was introduced into the system along with setting up a visualization tool, Grafana.

Chapter 2

System Architecture and Implementation

2.1 OPC UA Client and Delta Robot Communication

A key component of the communication architecture is the OPC UA client, which provides a standardized communication protocol for data exchange between machines [2]. A data collection script from MATLAB to Python needed to be converted for the project. Python's wide libraries support and smooth integration with several AI and machine learning frameworks served as the main motivation. The Python script provides a reliable and efficient data transmission process by communicating with the delta robot via the OPC UA client.

Using the OPC UA protocol, the Python script connects to the delta robot. In order to read sensor data, control data buffers for signal sampling, and trigger the robot, it specifies the required nodes. The Python script establishes a secure connection between the OPC UA client and the delta robot, which guarantees data integrity and reliability[3]. The following part will analyze the communication process, showing the steps taken by the script to manage data collection, maintain data buffers, and read data from and send to the delta robot.

The algorithm implemented in Python script creates a communication connection over the OPC UA protocol with the delta robot to allow a real-time data collection throughout an assembly process. A description of the algorithm's operating steps may be found below:

1. **Initialize OPC UA Client:** Create an instance of the OPC UA client and configure the server's URL, session timeout, user credentials (username and password), and then establish a connection to the server.

2. **Retrieve OPC UA Nodes:** Access specific nodes from the OPC UA server related to the delta robot's operational parameters, including nodes for triggering data collection, enabling the switch buffer, and identifying the number of traces. Information about specific nodes was gathered from the programmable logic controller (PLC)'s manual. [1]

3. **Preparation for Data Collection:**

   (a) Disable the switch buffer to prepare the system for a new data collection loop (assembly of wheels).

   (b) Configure signal usage and trace selector nodes for each trace.

4. **Enable Data Collection:** Activate the switch buffer to start the data collection process.

5. **Data Collection Loop:** One loop consists of the assembly of typically four wheels.

   (a) Wait for the "readyForTrigger" signal, indicating that the system is prepared for data recording.

   (b) As soon as the trigger is received, begin recording and collect data in packets.

   (c) For each packet:

   - Check and wait for the handshake bit to be set, which indicates that either of the edge interfaces (buffers, 0 or 1) is ready for data collection.
   - Retrieve and store data samples from the selected interface. Each buffer contains 500 samples.
   - Reset the handshake bit once data collection for the current packet is completed.

   (d) Repeat the process for a predefined number of packets.

   (e) Disable the switch buffer in order to finalize the data collection loop and proceed to next one.

6. **Conclusion:** Disconnect the OPC UA client and prepare the system for the next data collection loop or for shutdown.

[1] PLC manual

The outcome of one loop is the following time series. Six different force and torque sensor readings and process identifiers. A non-zero identifier, coloured pink in the graph, signals a point in time where a wheel is being assembled, which means that in one loop there is much more unnecessary data than data relevant to the machine learning model. Specifically 12000 data points, out of which only around 4000 (1000 per wheel) are desired. Such fact will be important and mentioned later in the thesis 2.4. By reading first and last timestamps of the buffers, its was calculated that a buffer of 500 data points takes 2000 milliseconds, meaning the OPC UA server is ready to transfer data every 2 seconds.

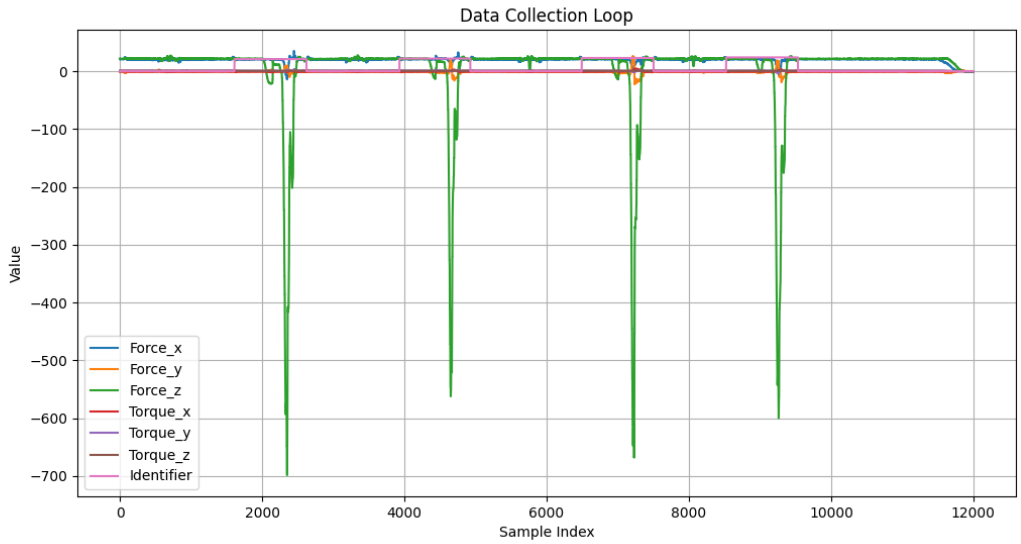

Figure 2.1: Sensor readings from the assembly of four wheels

2.2 gRPC Client-Server Architecture

The project's architecture consists of a gRPC server responsible for handling data processing and anomaly detection, and a gRPC client that communicates with the server to send data collected from the delta robot via the OPC UA client. gRPC is an open-source remote procedure call system initially developed by Google, which uses HTTP/2 for transport, Protocol Buffers as the interface description language, and provides features such as authentication, load balancing, and more [4]. The decision to utilize gRPC in this project was motivated by several factors, supported by the findings in the paper *"A Review of Application Layer Communication Protocols for the IoT Edge Cloud Continuum"* [5]:

- **Performance and Efficiency:** The efficiency of gRPC, thanks to its use of HTTP/2, enables a more efficient data transfer, which is crucial for the real-time

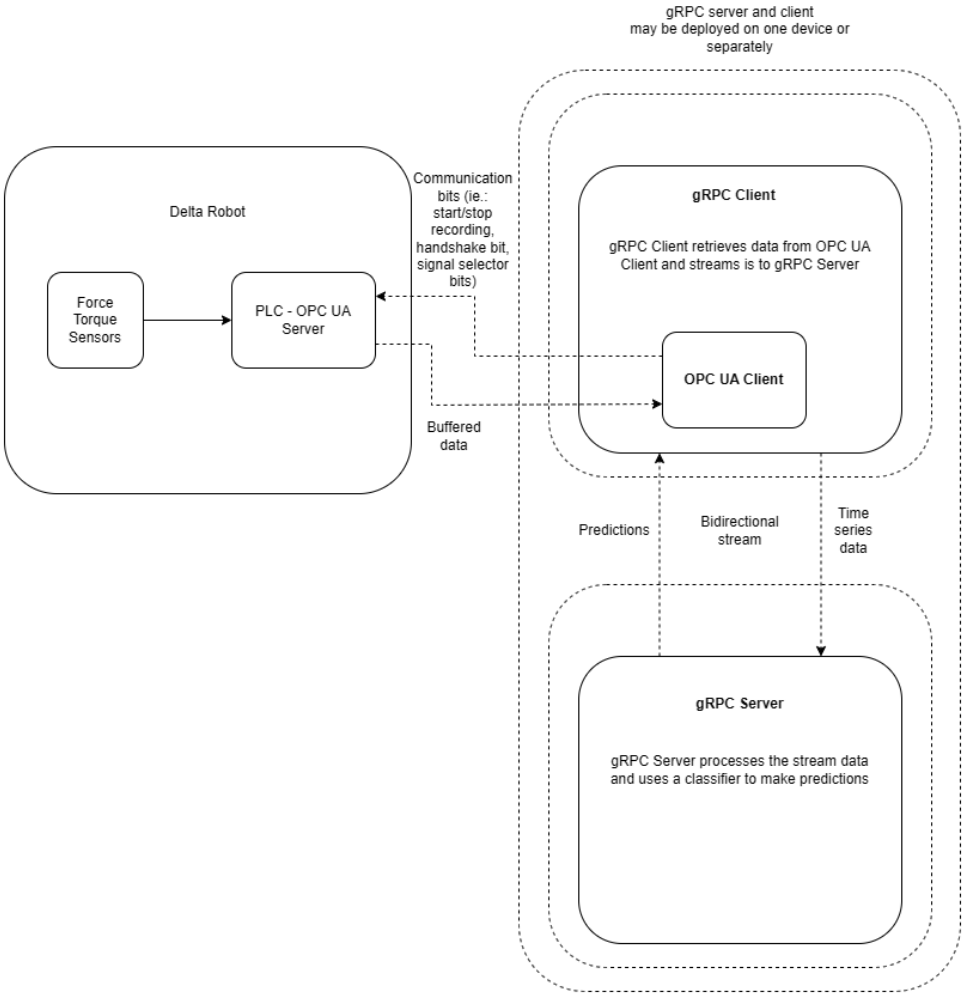

Figure 2.2: Client-Server Architecture Diagram

aspects of this project. This efficiency is particularly important when dealing with the high-throughput data generated by the delta robot.

- **Support for Bidirectional Streaming:** gRPC's native support for bidirectional streaming allows for continuous communication between the client and server, as seen in the diagram Figure 2.2, which enables real-time updates and responses. This feature is crucial for the project's use case, where the server needs to process streaming data from the client, apply the housed machine learning model, and simultaneously send feedback back to the client.

- **Cross-Language Support:** gRPC offers easy implementation across different programming languages, improving the adaptability and integration potential of the system, which might become useful later in the project´s future after it has been handed over and maintained by the team at the Testbed for industry 4.0.

- **Strongly Typed Interfaces:** Using Protocol Buffers, mechanisms for serializing

structured data[2], ensures predefined and strongly typed service interfaces, improving the communication protocol's maintainability, which is essential for accurately defining the inputs and outputs of the machine learning model. Protocol buffer for this project is shown below. One service, providing the bidirectional stream, and two types of messages are defined. The *NumpyArray* message is sent by the gRPC client, whereas the other message *AnomalyDetResponse* is returned by the gRPC server.

```
service AnomalyDetectionService {
    rpc StreamData(stream NumpyArray) returns (stream
       AnomalyDetResponse);
}

message AnomalyDetResponse {
    int32 id = 1;
    bool result = 2;
    int32 series_len = 3;
    int32 msg_id = 4;
}

message NumpyArray {
    repeated double values = 1;
    int32 rows = 2;
    int32 cols = 3;
    int32 msg_id = 4;
}
```

The gRPC client is designed to establish a connection with the gRPC server, transmit the collected data via a bidirectional stream, and receive the processed results. As seen in the defined *NumpyArray* message above, protocol buffers only support serialization of simple one dimensional arrays. However, since a seven dimensional array needs to be transferred, the client first flattens the array into a one dimensional one and sends it together with the original numbers of rows and columns, so that the array can be reconstructed after reaching its destination.

The gRPC server is implemented to provide several functionalities, including data processing and anomaly detection, utilizing the housed machine learning model. It listens for incoming stream requests from the gRPC client, processes the data, and sends back results of the machine learning model inference. Since the messages might not always

[2]Protocol Buffers Documentation

contain data relevant for the model (data with identifier equal to zero), which happens very often within the project´s use case, it keeps track of its own buffer, implemented as Python´s version of a hash table, a dictionary, where keys stand for the non-zero identifiers and values for their corresponding time series together with a prediction result. Using a dictionary instead of, for example, an array covers cases where one message contains data points corresponding to more than one non-zero identifier. The server first checks for any non-zero identifiers. If there are not any in the message and the buffer is empty, the message is discarded, effectively filtering machine learning model-irrelevant data. If the buffer is not empty and the current message contains identifiers equal to zero, that means that the buffer´s identifiers, which are not contained in the current message, have reached their end within the assembly process and can be cleared.

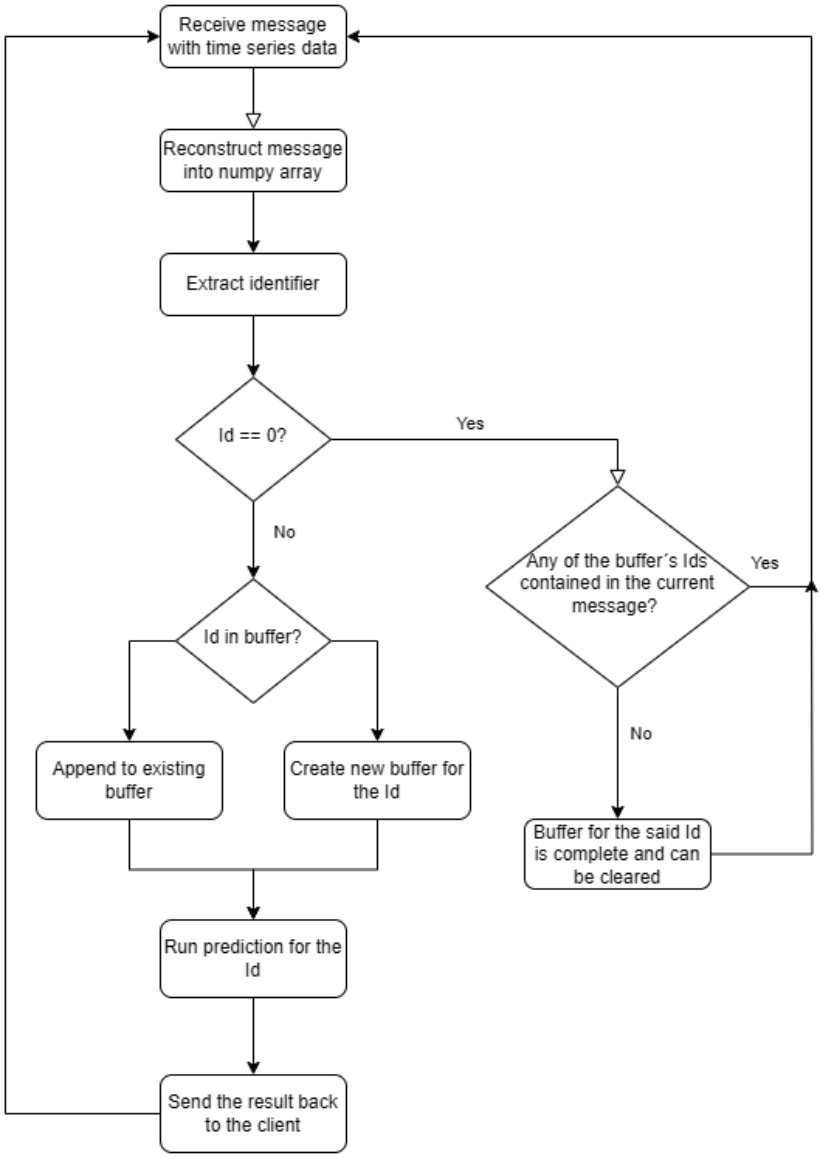

Figure 2.3: Server Flow Chart

Machine Learning Model Integration

The gRPC server houses a machine learning model developed by Aleš Trna [1], designed to analyze time-series data collected from the assembly line in real-time. An important advancement in the integration process is the transformation of the machine learning model into a Python package. This packaging simplified its distribution done via Python Package Index (PyPI), an official Python package repository[6]. The distribution process involved the use of *setuptools*, a tool, that creates a so called egg [7] and wheel, both a type of built distribution files, that can be then easily installed. The files are created with the help of a *setup.py* file that contains metadata and dependencies required by the model. Another tool *twine* [3] was then used to upload the package to PyPI.

The distribution of the machine learning model as an installable Python package offers several advantages. Users can easily install the model using *pip*, which significantly simplifies the deployment process and integration into various systems. Versioning is made possible by distributing the model via PyPI, which guarantees that users can access particular model versions and efficiently manage dependencies [8]. By utilizing PyPI, the model is made accessible to a wide audience. The model's packaging provides reusability by allowing the model to be applied in a variety of settings without requiring direct access to the initial development setup. The distribution of the machine learning model as a Python package was achieved in several steps described below.

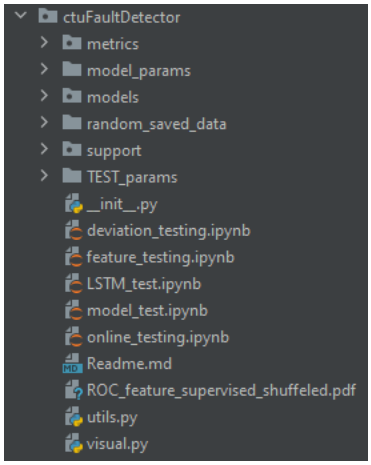

Figure 2.4: ctuFaultDetector Package Structure

1. **Package Preparation:** Following standard practices for Python packaging, the machine learning model and its corresponding modules were placed inside a defined directory structure [4]. The *setup.py* should be created in the root folder of the repository together with the desired package. That way it can reach the *ctuFaulDetector*

[3] twine introduction
[4] setuptools guide

folder, which is the root folder of the package, as shown in Figure 2.4, and is specified in the *setup.py* file. Every folder that should be packaged must contain a *__init__.py* file, that way *setup.py* knows it is a Python module which should be distributed together with the root folder. The Figure 2.4 shows several folders without the dot in the folder icon, which means they do not contain a *__init__.py* file and are therefore not modules and consequently will not be included in the distributed package.

2. **Package Building:** Using *setuptools*, the package was prepared for distribution. The command *python setup.py bdist_wheel sdist* was executed to generate distribution archives in the *dist* directory.

3. **Package Uploading:** The built distribution files were uploaded to PyPI using *twine* with the command *twine upload dist/**.

4. **Package Installation:** The Python package can now be easily installed using *pip install ctuFaultDetector*.

2.3 Architecture for Real-time Anomaly Detection Web UI

In order to satisfy one of the project‘s goals, the human-machine interaction within the concept of Industry 5.0, a web-based UI was developed, enhancing user interaction and providing real-time feedback on anomaly detection results. This UI is designed to display predictions streamed from the gRPC server, which processes data collected from industrial sensors. In order to make real-time interaction and display of anomaly detection results possible, several architecture options were evaluated. The decision on the architecture affects latency, which is critical for real-time operations. The Following is the thought process behind the selection of the potential architectures and the reasons behind the selected approach.

The first approach is to host the UI with the gRPC client. This approach integrates the UI directly within the gRPC client that interfaces with the OPC UA, making it a monolithic architecture [9]. Since the gRPC client is already connected to the gRPC server, it can receive the processed results and directly update the UI without any additional network hops. This setup also simplifies the overall system architecture by minimizing the number of components involved, which eases implementation and debugging, but on the other hand takes a toll on the scalability of the system.

Although hosting the UI on the same server that runs the ML model would provide immediate access to processed data, it introduces potential drawbacks such as increased

server load and potentially higher network latency if the server is not in close proximity to the client. The server would need to handle UI rendering and then send the entire rendered UI to the client.

Making the UI into a separate component would also be an option. It would make the system more scalable by separating tasks into several microservices [10], but it might also make the anomaly detection results delivery longer, because it would have to travel between more components, opening additional communication channels.

The architecture of hosting the UI together with the gRPC client was chosen because of its lower latency and simple nature of the UI required by this project, also supported by the findings in the paper [11]

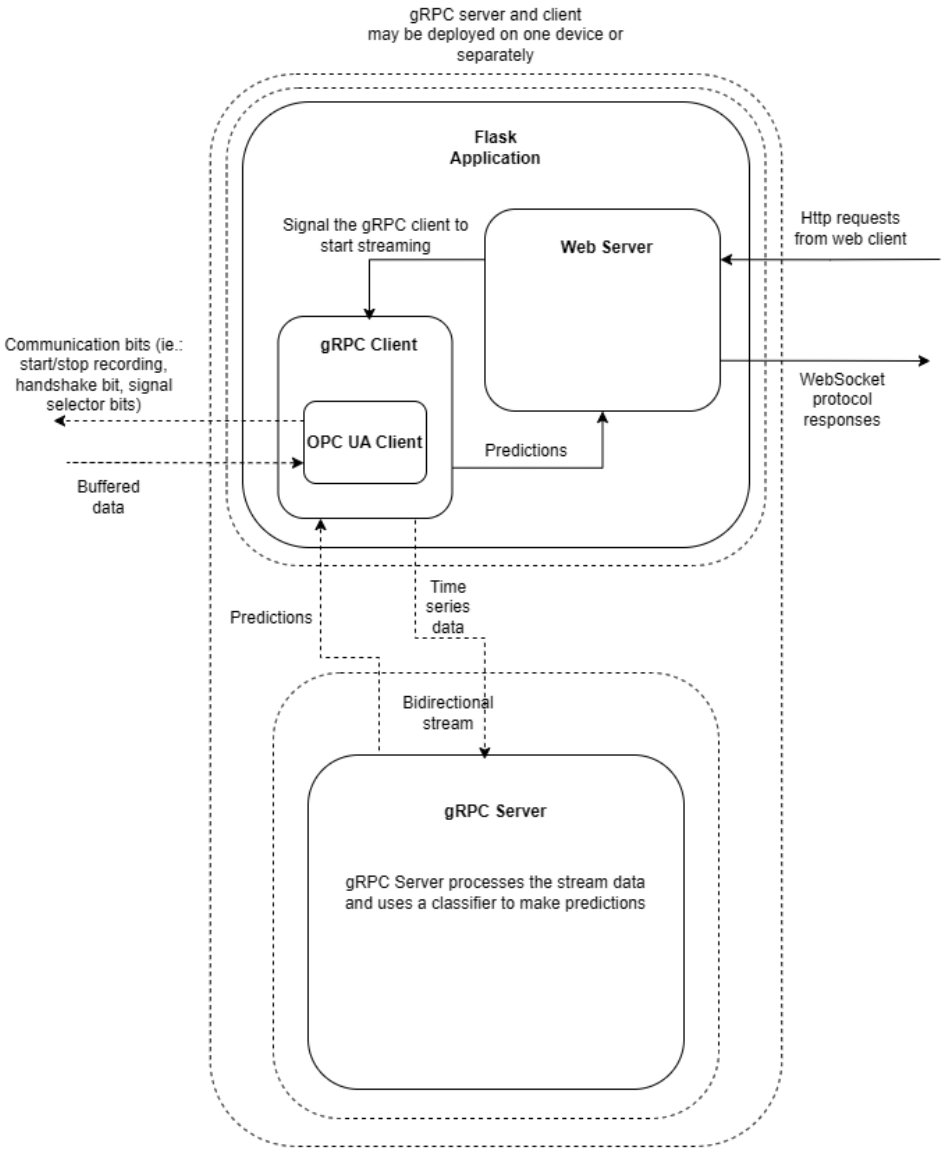

Figure 2.5: Updated Architecture After Flask Integration

WebSockets for real-time communication and Flask for the web development were used in the implementation. WebSockets provide real-time updates that can be viewed directly in the browser without having to constantly poll the server, allowing the server to push new data to the client's browser [12]. By doing this, it is guaranteed that the UI is updated quickly with the most recent anomaly detection results.

Flask is a lightweight web framework [13], that also supports a built-in development WebSockets server, which makes the development process easy by making local testing possible without having to install an additional server. Running a Flask based application is as easy as running a simple Python script.

The Flask server handles HTTP requests and WebSocket events. The server renders the HTML page to the client and establishes WebSocket connections for transmitting real-time data. Using Flask-SocketIO, the server can emit and listen for events on the WebSocket. This allows the server to receive commands from the UI [14] (e.g., start or stop streaming) and push prediction results to the UI as they are processed by the gRPC server.

Web User Interface Design and Features

The UI includes basic HTML elements such as buttons for starting and stopping data streaming and a designated area to display the prediction results along with buttons that can be used to mark a prediction result as incorrect. JavaScript, along with Socket.IO, is used to provide real-time communication between the client's browser and the Flask server. This setup enables dynamic updates without the need to refresh the web page.

The user can initiate the streaming of data by clicking the "Start Streaming" button. This sends a command to the Flask server via WebSockets, which instructs the gRPC client to begin the data collection via the OPC UA client and open the stream to the gRPC server. As predictions are generated by the gRPC server, they are sent to the Flask server, which then pushes these results to the web UI. The user can stop the streaming of data at any point by clicking the "Stop Streaming" button, which sends another command to the Flask server to stop the process. The prediction results are displayed in the designated area in the UI, with each new prediction dynamically added to the display without reloading the page. Users can mark incorrect predictions, which are then further processed by the Flask server. Each prediction record consists of a prediction boolean value, true signaling an anomaly and false a normal operating state, identifier and a corresponding time series length, which the prediction was calculated on. In order not to overload the memory with prediction results, they are kept in a circular queue, whose capacity is currently set to 40, but can be adjusted to specific needs, so the oldest

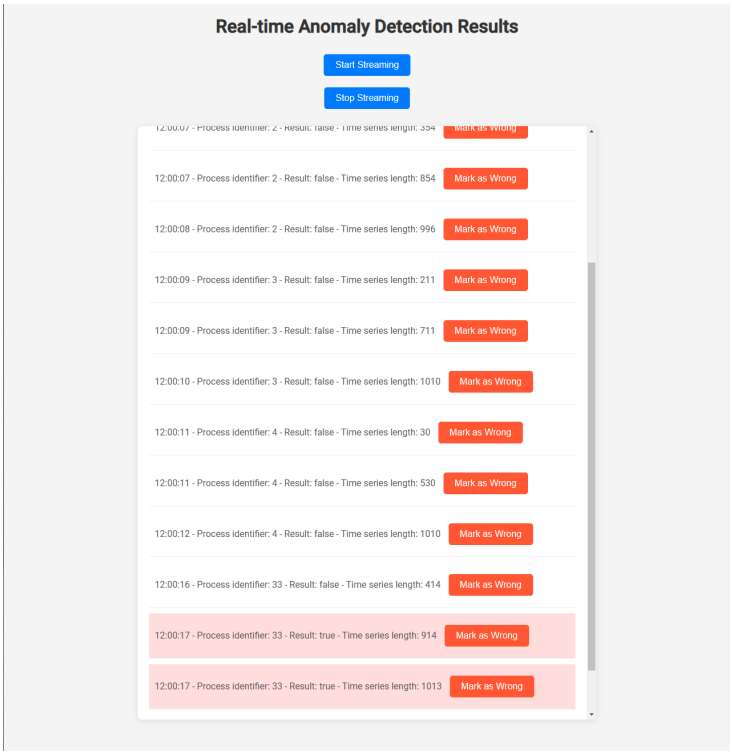

Figure 2.6: User Interface Design

records are overwritten by the new ones. In the delta robot´s use case, generally three predictions are made over the process of one wheel assembly, meaning over the period of one data collection loop, 12 predictions would be displayed in the UI, which makes the current queue capacity sufficient for several assembly loops.

2.4 Database Integration for Anomaly Detection Results Analysis

To once again support the human-machine interaction within the concept of Industry 5.0 goal, to store and manage the time series data along with prediction results and possible human labels, a database was integrated into the system. The choice between InfluxDB[5] and MongoDB[6] was considered. They are both NoSQL databases, meaning they do not store data within relational tables [15]. NoSQL databases can thus omit data joining from multiple tables, which is an operation that becomes increasingly time expensive with growing data[7]. NoSQL databases are for this reason more suitable for high volume time series data, that this project utilizes. Even though InfluxDB may have faster query and write times [16], such aspects are not as important for this project´s use cases. The

[5]InfluxDB
[6]MongoDB
[7]NoSQL Querying

intentions behind integrating a database are to provide a way to store human feedback on the predicted anomalies, be able to analyse the data and reuse them in other applications such as training, testing and validating of other machine learning models. InfluxDB was still used for a pilot implementation for its wide use within Internet of Things (IoT), however it was quickly found that it does not support one crucial feature necessary for the human feedback, which is updating existing records. Therefore the project migrated to MongoDB due to its capability to update existing records, and also due to the fact that MongoDB stores data within a JSON-like key-value format that is easy to use[8].

Database Service and Message Queuing Telemetry Transport (MQTT) Integration

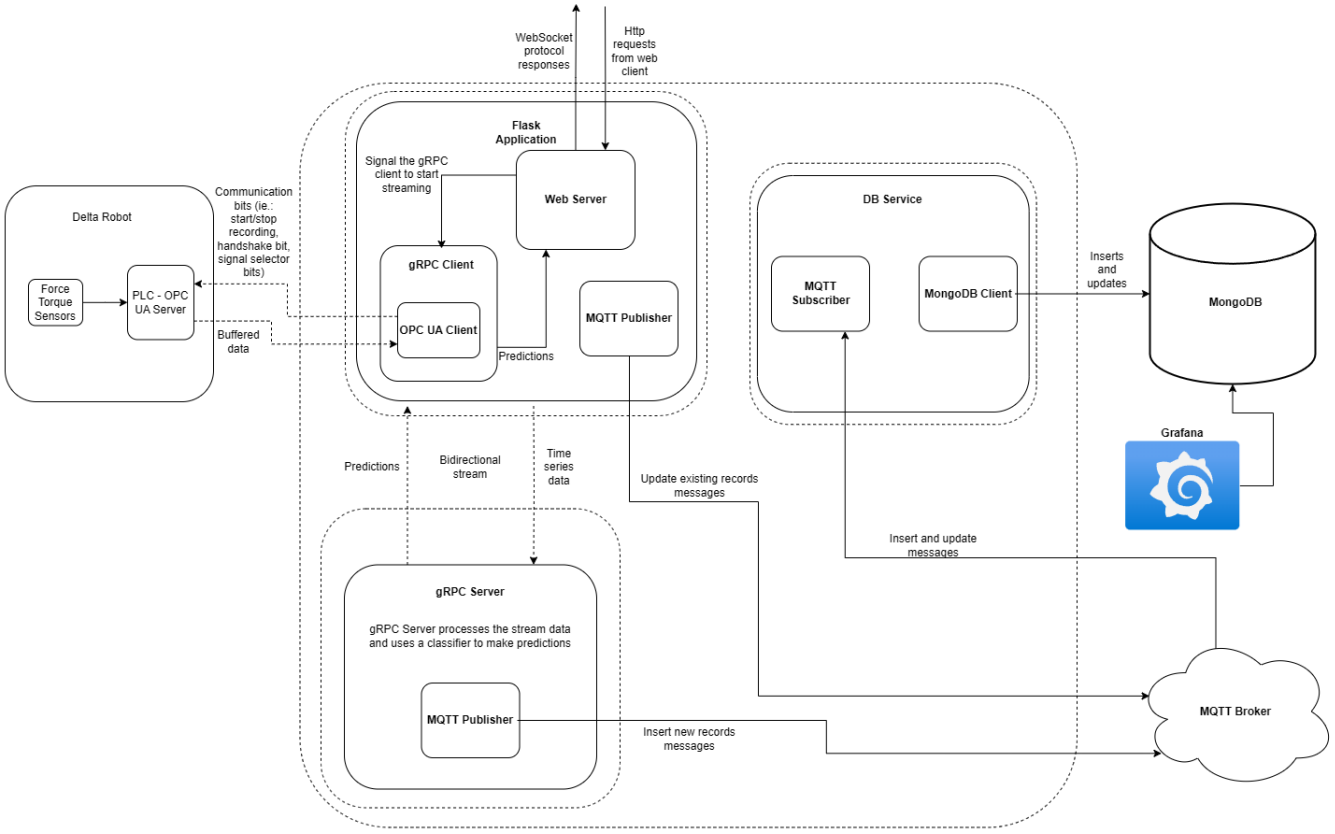

Figure 2.7: Final Architecture

A new service was created to handle communication with the MongoDB database. This service is subscribed to an MQTT topic, where two types of messages are sent, time series data with prediction results from the gRPC server, and incorrect prediction messages with human labels from the Flask web UI. Due to the nature of the gRPC server´s buffer 2.2, only time series relevant to the machine learning model is sent, while

[8]MongoDB Document Format

the other data, that do not correspond to any assembly process, is discarded. The service processes these messages and writes new data to the database or updates existing ones accordingly. MQTT is a publish/subscribe communication protocol, where both the publisher and subscriber connect to a message broker that distributes the messages into different topics [17]. The database service in this case becomes a subscriber to one topic, receiving messages containing new data or updated versions of existing data. The service then handles communication with the database. Both the gRPC client and gRPC server become MQTT publishers. The server sends messages containing new records to be inserted in the database, whereas the client publishes messages containing human labels with their corresponding assembly process identifiers. In order not to block the thread running the main tasks on the gRPC server (processing the stream, managing its buffers, running predictions) while publishing MQTT messages, a thread pool was used to assign the publishing tasks to existing threads in the pool. Utilizing existing threads is less time consuming than creating new threads for each task, as it eliminates thread creation and destruction [18]. The addition of Grafana, as seen in the Figure 2.7, is discussed later in the thesis 3.4.

Chapter 3

MLOps in Anomaly Detection for Delta Robots

MLOps is a set of practices aimed at automating the development and operations of machine learning systems.[1] Its goal is to speed up the process of continuous integration, delivery, and deployment of machine learning models in a production environment. The MLOps paradigm emphasizes the development of machine learning models, their deployment, monitoring, and maintenance in real-world settings [19].

3.1 Machine Learning Model Development

The base of any MLOps strategy is a machine learning model. In this project, several anomaly detection models were developed by Aleš Trna to analyze time-series data from industrial sensors and identify deviations from normal operating status. The methodologies of the models are explained in detail in Aleš Trna's thesis. [1] The model's development involved packaging the machine learning model into a Python package and distributing it via PyPI, we ensured that the model could be easily reused and integrated into various systems. This approach provides simple version control and simplifies the deployment process.

3.2 Testing and Validation

Testing and validation are important components of MLOps, they make sure that the model performs reliably before it is deployed into the real world. The models were tested using historical data from the delta robot's operation. Various validation techniques, including multiple types of cross-validation and performance metrics such as precision

[1] MLOps Introduction

and recall, were evaluated to assess the model's effectiveness. Detailed analysis of validation has been described by Aleš Trna in his thesis [1]. By integrating these testing and validation steps into the development, we can ensure that the model functions as intended.

3.2.1 gRPC Client-Server Latency

To evaluate the performance of the anomaly detection system, latency, also known as Round Trip Time (RTT) [20], was used as a metric, it tells the time elapsed from sending a data message from the gRPC client to receiving the prediction result from the server. This includes request and response transmission time, and data processing time. To measure latency, the following approach was used. Timestamps were recorded at the gRPC client when a data message is sent and when the prediction result is received [21]. Inputs to the models consisted of 200, 300, 500 - 1000 data point long time series, that are evenly distributed due to the nature of the delta robot´s buffers 2.1 and the nature of the gRPC server´s buffer 2.2. The server's processing time was logged to understand the time taken for the anomaly detection model to analyze the data and generate a prediction. Measurements took place while running the gRPC client and server on the same computer and running the client and server separately on two computers in the same network, while one being connected wirelessly, to see the impact of network communication on latency. The wireless connection presents the worse case scenario for latency, since wireless connection is generally slower that wired one. The analysis provides insights into where optimizations could be made. For instance, if transmission times were significantly higher in the networked setup, this might indicate a need for communication optimization or lead to deployment on a single device. If the processing time was a significant portion of the total latency, optimizing the anomaly detection model or server processing would be beneficial. The measurements were taken for two different models, the deviation classifier model and the feature classifier model[1]. Mean, standard deviation and median of the latency and processing time were calculated using the Python´s Pandas [2] data analysis library. The summaries and visualizations below present an analysis of the performance.

[2]Pandas

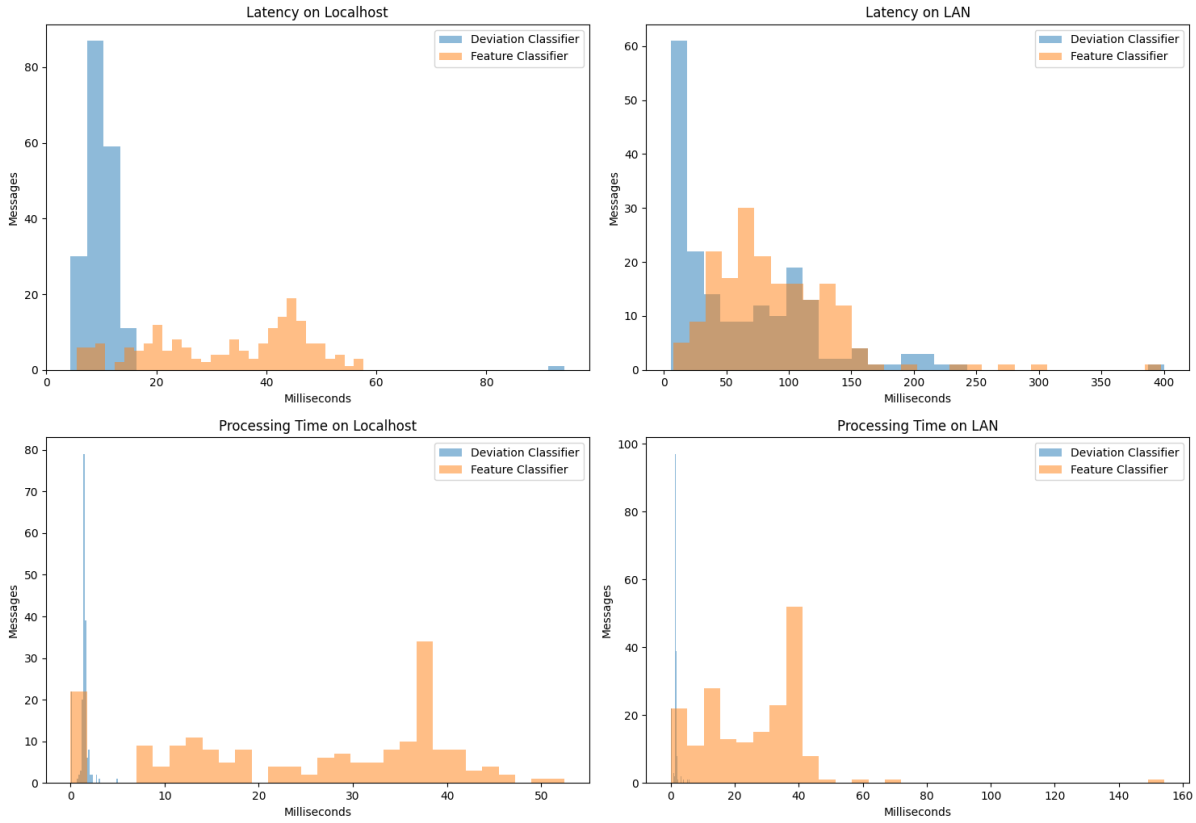

Figure 3.1: Latency Distribution for Feature and Deviation Classifier Models

Metric	Mean (ms)	Std (ms)	Median (ms)
Latency Deviation (localhost)	10.22	6.54	9.84
Latency Feature (localhost)	32.91	14.04	35.78
Latency Deviation (LAN)	62.21	58.97	40.38
Latency Feature (LAN)	88.32	51.56	78.75

Table 3.1: Latency metrics for localhost and LAN setups

Metric	Mean (ms)	Std (ms)	Median (ms)
Processing Deviation (localhost)	1.37	0.61	1.49
Processing Feature (localhost)	24.83	14.25	28.68
Processing Deviation (LAN)	1.45	0.73	1.54
Processing Feature (LAN)	25.48	17.23	28.62

Table 3.2: Processing time metrics for localhost and LAN setups

The average latency for the deviation model running on localhost is significantly lower (10.22 ms) compared to when it is running on the LAN (62.21 ms). This shows that the network transmission time increases the overall latency. Similarly, the feature model shows an increase in latency from localhost (32.91 ms) to LAN (88.32 ms). This suggests that the feature model, which is more complex, experiences even greater latency when

network transmission is involved. The standard deviation for latency is higher in the LAN setup for both models, showing more variability and less predictability in networked setup compared to localhost.

The processing time for the deviation model is low on both localhost (1.37 ms) and LAN (1.45 ms). This consistency suggests that the processing in the deviation model is minimal. The feature model, however, shows a much higher processing time on both localhost (24.83 ms) and LAN (25.48 ms). This shows that the complexity of the feature model is much higher than the deviation model´s. The standard deviation in processing time for the feature model is higher, indicating more variability in the time taken to process different data messages. This variability could be due to the complexity of the model and the nature of the input data. For both latency and processing times, the median values are close to the mean values, which shows that the data is symmetrically distributed without significant outliers. By reading first and last timestamps of the PLC´s buffers, it was calculated that one buffer takes 2000 ms to fill up. Based on findings in the OPC UA performance evaluation paper [22], data transfer times between the OPC UA server and client should be more than sufficient for the real-time requirement and should not affect this analysis too much. Even if considering the worst-case scenarios of the LAN setup shown in the Figure 3.1 for the feature classifier, the entire solution´s latency is still capable of real-time usage, as illustrated in the Figure 3.2

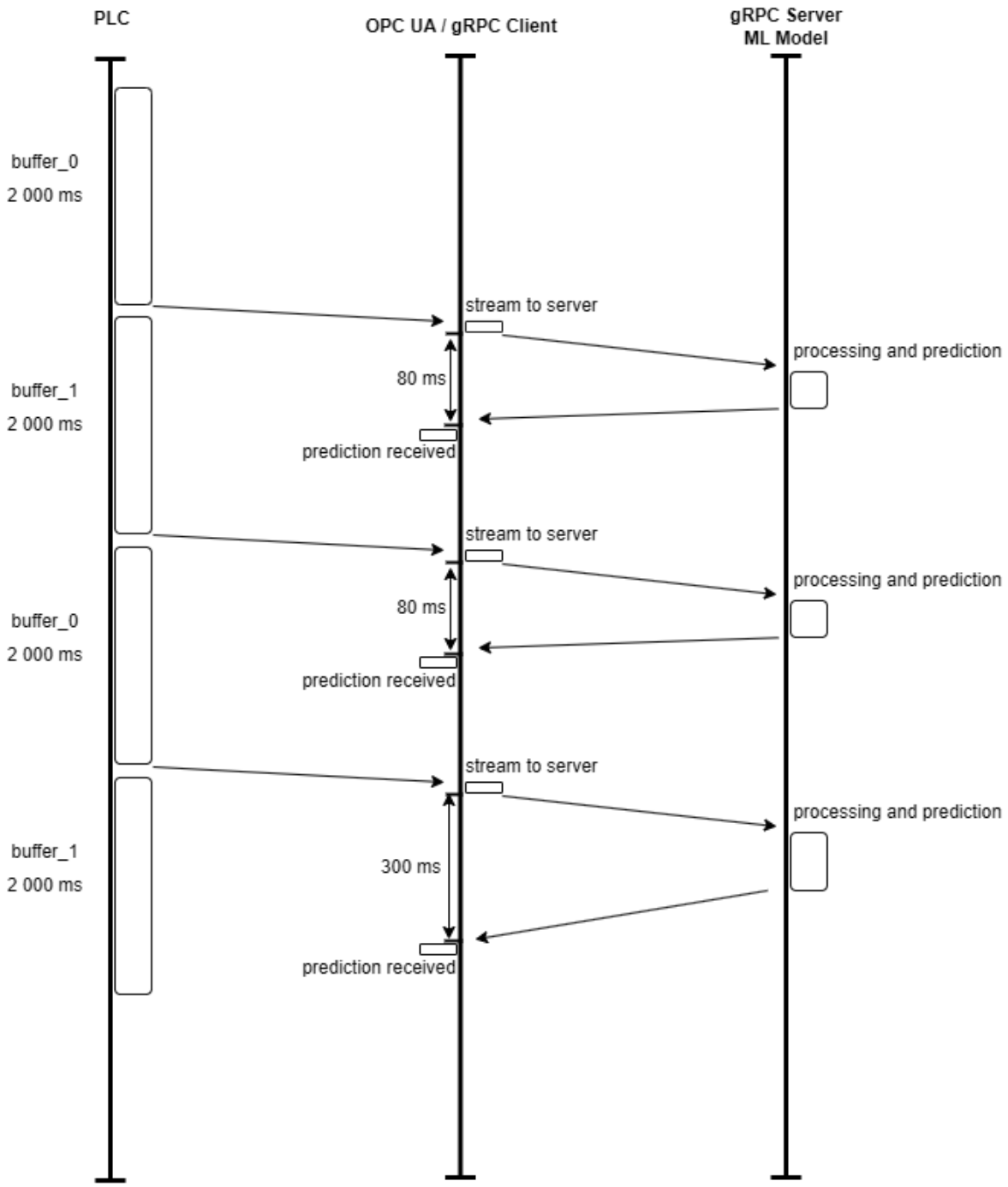

Figure 3.2: System Latency Diagram With Variable Latencies

3.3 Deployment and Integration

Deployment and integration are crucial stages in the MLOps life cycle. In this project, the model was deployed within a gRPC server, which provides data processing and anomaly detection in real-time. The server was designed to handle streamed data from the delta robot, apply the anomaly detection model, and send feedback to the gRPC client and the web-based user interface. This integration was achieved by utilizing Flask for the web

server and WebSockets for real-time communication, enabling dynamic updates and user feedback. The entire architecture can be seen in the Figure 2.7.

The gRPC server has been containerized using Docker to ensure its scalability and ease of deployment. This approach enables the system to be deployed in different environments without compatibility problems, including the Ai-Builder[3], which is one of the objectives of this thesis. The use of Docker makes it possible to maintain the consistency of dependencies and configurations, which reduces the risk of errors in deployment [23].

3.3.1 Dockerization of the gRPC Server

An isolated environment consistent in all stages of development, testing, and production is provided by the Docker container. In order to simplify the process of getting the server running, Docker containers can be deployed on any system that has Docker installed [24]. In order to make it easier to scale up or down as needed, multiple instances of the server can be run as separate containers, which may be useful in the case of deployment of the model to other robots in the Testbed for Industry 4.0.

Dockerization was achieved by creating a Dockerfile, which provides a blueprint for Docker to build an image of the gRPC server. The Dockerfile specifies how the environment is set up, which dependencies are installed, and what commands are run when the container starts. Below is an overview of the Dockerfile used for the gRPC server:

```
FROM python:3.11

WORKDIR /app

COPY ./requirements.txt /app
RUN pip install --upgrade -r requirements.txt

COPY . /app

EXPOSE 8061

CMD ["python", "./server_main.py"]
```

Listing 3.1: Dockerfile for Python Application

1. **Base Image:** The Dockerfile specifies a base image of Python, which makes the server operate within a consistent Python environment.

2. **Working Directory:** It sets a working directory inside the container for organizing application files and dependencies.

[3]Ai-Builder

3. **Dependencies Installation:** The Dockerfile copies the *requirements.txt* file into the container and runs *pip install* to install the necessary Python packages, including our packaged machine learning model.

4. **Application Files:** The server source code files are copied into the container's working directory.

5. **Exposing Ports:** The Dockerfile specifies which ports the server listens on, making the gRPC server accessible outside the Docker container.

6. **Startup Command:** Finally, it defines the command to run the gRPC server when the container starts, making the server available immediately upon running the Docker image.

Building the Docker Image

Using the following steps, a Docker image of the gRPC server was created.:

1. Navigate to the directory containing the Dockerfile.

2. Execute the command:

```
docker build -t ctu-fault-detector .
```

   This command builds a Docker image named *ctu-fault-detector* based on the instructions specified in the Dockerfile. The period *"."* signifies that the Dockerfile is located in the current directory.

Uploading the Docker Image

The next step was to upload it to the Docker container registry, such as the Docker Hub, for easy distribution after successfully creating the Docker image. The process includes:

1. Log in to the Docker Hub from the command line using:

```
docker login
```

2. Tag the Docker image with a Docker Hub username:

```
docker tag ctu-fault-detector  username/ctu-fault-detector
```

3. Push the Docker image to Docker Hub:

```
docker push username/ctu-fault-detector
```

This will make the gRPC server image publicly available, allowing anyone to download and run a server without having to go through building.

Pulling and Running the Docker Image

To pull and run the Docker image on any operating system, Docker needs to be installed. For users to run the gRPC server on their systems, they need to pull the Docker image from Docker Hub (or any other Docker container registry where the image has been uploaded) and run it. The following commands are used to pull the image of this project´s gRPC:

1. Pull the Docker image:

```
docker pull vojtavoj/ctu-fault-detector
```

2. Run the Docker container:

```
docker run -d -p 8061:8061 vojtavoj/ctu-fault-detector
```

This command starts the gRPC server container in detached mode, meaning it does not block the current terminal as it runs the server in the background, mapping port 8061 of the container to port 8061 on the host. This setup allows the gRPC server to be accessible at the specified port of the hosting device. Other useful commands can be found in the official Docker cheat sheet [4].

3.3.2 Deploying to AI-Builder

The AI4EU Experiments platform is an online tool for creating and sharing artificial intelligence projects. It's made for researchers, developers, and businesses to collaborate on AI solutions. The platform offers resources and tools for creating machine learning solutions and sharing their projects. A key feature is the design studio, which offers an easy, visual way to create workflows.[5]. A simpler demo version of our solution has been

[4]Docker Cheat sheet
[5]The Design Studio

uploaded to the platform. For a model to be able to be uploaded it has to be implemented within a gRPC server contained in a Docker image and expose port 8061. Together with the Docker image, a Protocol Buffer, used by the gRPC server, is uploaded. For the demo purposes, another gRPC server containing some of our data, using the same Protocol Buffer interface, was created. Its task is to feed data to the machine learning model [6].

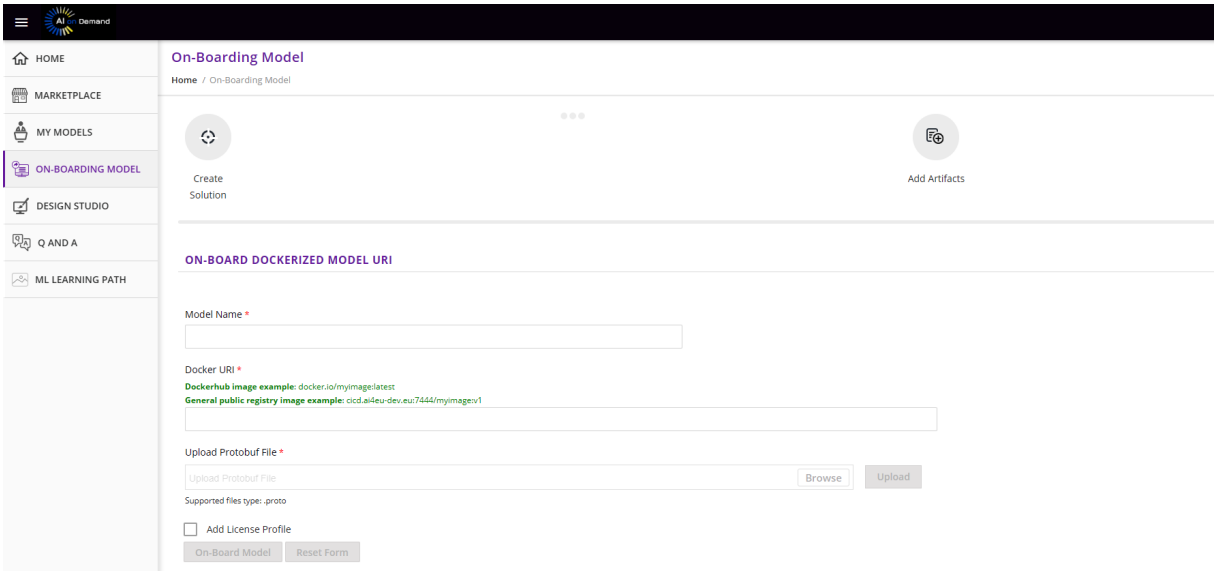

Figure 3.3: On-Boarding the Model Onto the Platform

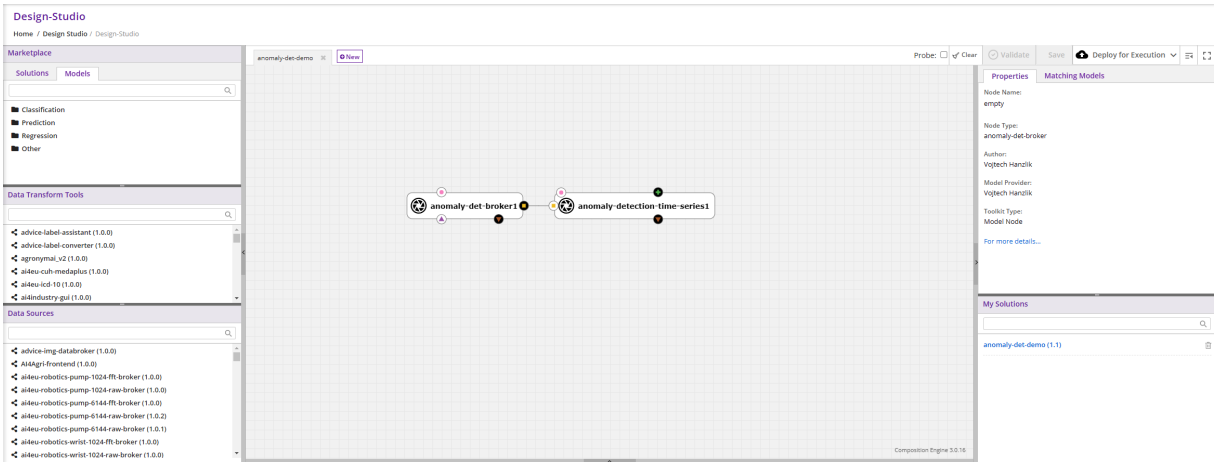

Figure 3.4: Connecting the Two Uploaded Images In the Design Studio

[6] AI-Builder Deployment Specifications

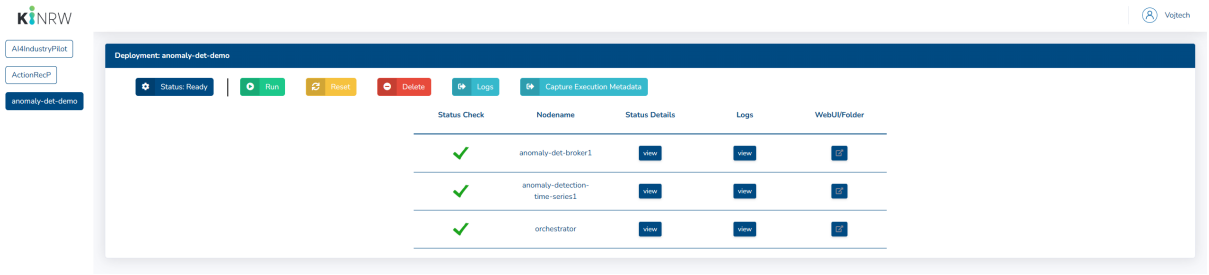

Figure 3.5: Deploying the Solution On-line

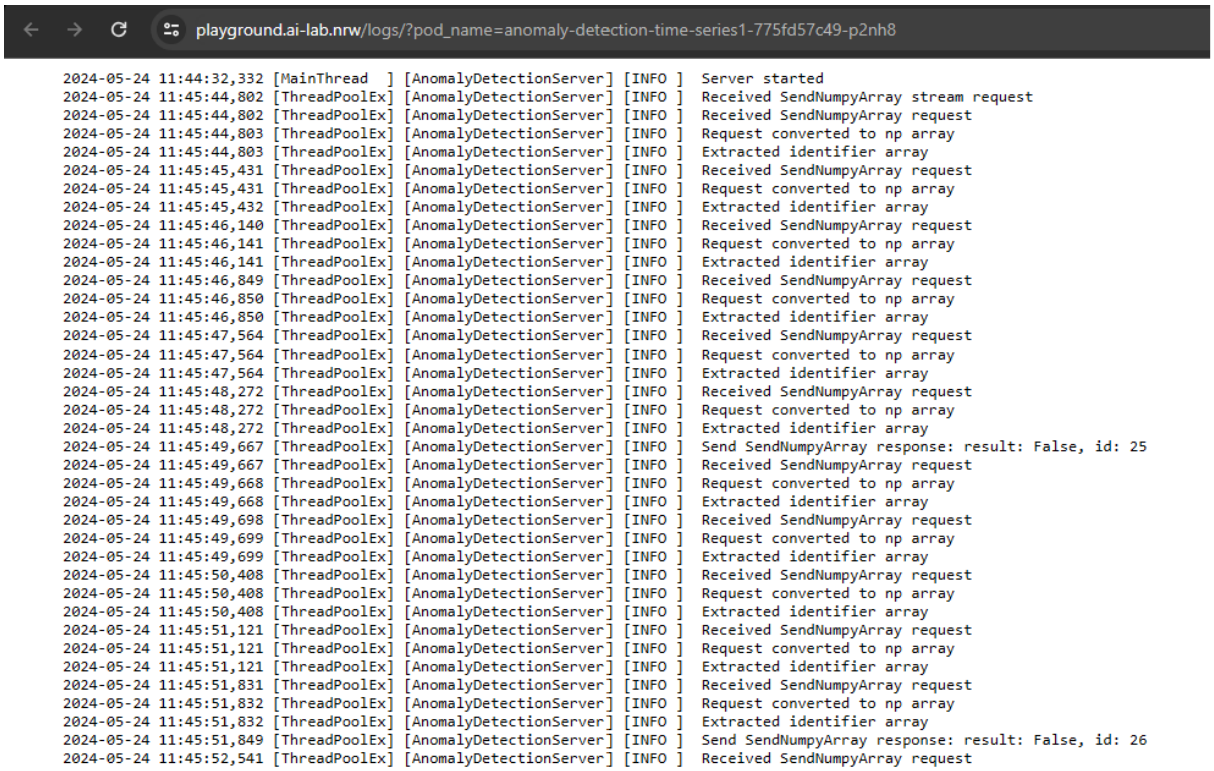

Figure 3.6: Deployed Model Logs

3.4 Monitoring and Maintenance

Continuous monitoring is important to keep the model functioning. Grafana [7] has been integrated into the anomaly detection system in this project, allowing it to monitor and display its results. By directly connecting to the database, Grafana offers a platform for viewing time series data together with prediction results. The integration of Grafana supports the human machine interaction aspect of Industry 5.0. Its usage for monitoring works very well with the feedback provided by users via the web-based UI, where users can label incorrect prediction results and provide ground truth. This feedback mechanism is important for refining the model over time and for developing other models, as the data together with its predictions can be reused and further analyzed.

[7] Grafana

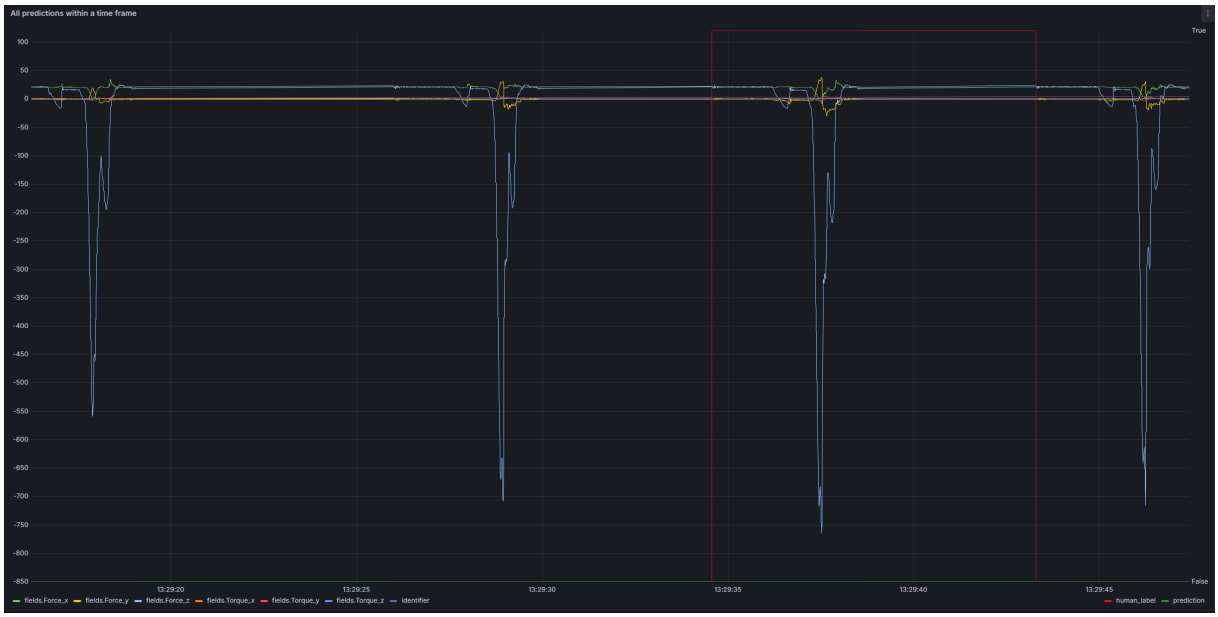

Figure 3.7: All predictions within a time frame viewed in Grafana

Figure 3.8: Prediction that has been labeled as a false negative viewed in Grafana

Chapter 4

Conclusion

This project made use of edge AI technology within an Industry 4.0 setting, specifically for real-time anomaly detection in a multi-axis delta robot assembling wheels for remote-controlled vehicles. By exploring the system architecture, communication protocols, machine learning model integration, and deployment strategies, several important accomplishments were achieved.

4.1 Accomplishments

Several important goals were accomplished in this thesis. First, a data collection script was successfully converted from MATLAB to Python using the OPC UA protocol. This change enabled real-time data exchange between the delta robot and the anomaly detection system, ensuring efficient data transmission. Second, a client-server architecture using gRPC for data processing and anomaly detection was developed. This setup allowed data streaming for real-time updates and support for multiple programming languages, which is important for its possible future reusability within the Testbed for industry 4.0.

The machine learning model was also transformed into a Python package and distributed via PyPI, making deployment and version control simpler. This ensured that the model could be easily integrated into different environments. Additionally, a web-based user interface was created using Flask and WebSockets. This interface allows users to see anomaly detection results in real-time and provide immediate feedback.

MongoDB was integrated to store and manage time series data and prediction results, which helps improve the anomaly detection model over time by storing historical data about the model´s runtime in a real setting. Docker was used to containerize the gRPC server, ensuring consistent deployment across different environments. Finally, Grafana was integrated for monitoring and visualization of anomaly detection results, which helps track model performance and also supports ongoing improvement.

An additional accomplishment was deploying a demo version of the anomaly detection solution on the AI-on-demand platform, showcasing the platform´s capabilities. This also makes the machine learning model accessible publicly.

4.2 Future Work

Several improvements could be worked on in the future. Improving the UI to include more advanced feedback features, implementing automated testing and deployment pipelines or increase the monitoring capabilities with more detailed analytics are all upgrades that the project would benefit from.

Future work could also focus on extending the solution to other robotic systems within the Industry 4.0 Testbed, making it applicable for different input formats, or even making it reusable in other applications in completely different industries and projects. Its presence within the AI-Builder platform will be improved as well.

Bibliography

[1] A. Trna, "Anomaly detection in robotic assembly process using force and torque sensors", B. Sc. thesis, Czech technical university in Prague, 2024.

[2] J. T. Da Silva, A. L. Dias, and I. N. Da Silva, "A survey on opc ua protocol: Overview, challenges and opportunities", in *2023 15th IEEE International Conference on Industry Applications (INDUSCON)*, 2023, pp. 1523–1530. DOI: 10.1109/INDUSCON58041.2023.10375053.

[3] N. Mühlbauer, E. Kirdan, M.-O. Pahl, and G. Carle, "Open-source opc ua security and scalability", in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2020, pp. 262–269. DOI: 10.1109/ETFA46521.2020.9212091.

[4] K. Indrasiri and D. Kuruppu, *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020.

[5] J. Kampars, D. Tropins, and R. Matisons, "A review of application layer communication protocols for the iot edge cloud continuum", in *2021 62nd International Scientific Conference on Information Technology and Management Science of Riga Technical University (ITMS)*, 2021, pp. 1–6. DOI: 10.1109/ITMS52826.2021.9615332.

[6] E. Bommarito and M. Bommarito, "An empirical analysis of the python package index (pypi)", *arXiv preprint arXiv:1907.11073*, 2019.

[7] R. L'Esteve, "Python wheels", in *The Azure Data Lakehouse Toolkit: Building and Scaling Data Lakehouses on Azure with Delta Lake, Apache Spark, Databricks, Synapse Analytics, and Snowflake*. Berkeley, CA: Apress, 2022, pp. 417–436, ISBN: 978-1-4842-8233-5. DOI: 10.1007/978-1-4842-8233-5_18. [Online]. Available: https://doi.org/10.1007/978-1-4842-8233-5_18.

[8] I. Rahman, N. Zahan, S. Magill, W. Enck, and L. Williams, "Characterizing dependency update practice of npm, pypi and cargo packages", *arXiv preprint arXiv:2403.17382*, 2024.

[9] M. Richards, *Software architecture patterns*. O'Reilly Media and Incorporated 1005 Gravenstein Highway North, Sebastopol, CA ..., 2015, vol. 4.

[10] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation", *IEEE Access*, vol. 10, pp. 20 357–20 374, 2022. DOI: 10.1109/ACCESS.2022.3152803.

[11] M. Ivanco, "Attractive effects for video processing", in *Proceedings of Excel@FIT 2018*, 2018. [Online]. Available: https://excel.fit.vutbr.cz/submissions/2018/014/14.pdf.

[12] V. Pimentel and B. G. Nickerson, "Communicating and displaying real-time data with websocket", *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, 2012. DOI: 10.1109/MIC.2012.64.

[13] M. Copperwaite and C. Leifer, *Learning flask framework*. Packt Publishing Ltd, 2015.

[14] R. Rai, *Socket. IO real-time web application development*. Packt Publishing, 2013.

[15] J. Han, H. E, G. Le, and J. Du, "Survey on nosql database", in *2011 6th International Conference on Pervasive Computing and Applications*, 2011, pp. 363–366. DOI: 10.1109/ICPCA.2011.6106531.

[16] S. N. Z. Naqvi, S. Yfantidou, and E. Zimányi, "Time series databases and influxdb", *Studienarbeit, Université Libre de Bruxelles*, vol. 12, pp. 1–44, 2017.

[17] D. B. Ansari, A.-U. Rehman, and R. Ali, "Internet of things (iot) protocols: A brief exploration of mqtt and coap", *International Journal of Computer Applications*, vol. 179, no. 27, pp. 9–14, 2018.

[18] M. D. Syer, B. Adams, and A. E. Hassan, "Identifying performance deviations in thread pools", in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, IEEE, 2011, pp. 83–92.

[19] S. Mäkinen, H. Skogström, E. Laaksonen, and T. Mikkonen, "Who needs mlops: What data scientists seek to accomplish and how can mlops help?", in *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, 2021, pp. 109–112. DOI: 10.1109/WAIN52551.2021.00024.

[20] N. Cardwell, S. Savage, and T. Anderson, "Modeling tcp latency", in *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, vol. 3, 2000, 1742–1751 vol.3. DOI: 10.1109/INFCOM.2000.832574.

[21] H. Kraft and R. Johansson, "Evaluating rpc for cloud-native 5g mobile network applications", 2020.

[22] S. Cavalieri and G. Cutuli, "Performance evaluation of opc ua", in *2010 IEEE 15th Conference on Emerging Technologies Factory Automation (ETFA 2010)*, 2010, pp. 1–8. DOI: 10.1109/ETFA.2010.5641184.

[23] M. Sollfrank, F. Loch, S. Denteneer, and B. Vogel-Heuser, "Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation", *IEEE Transactions on Industrial Informatics*, vol. 17, no. 5, pp. 3566–3576, 2021. DOI: 10.1109/TII.2020.3022843.

[24] J. Nickoloff and S. Kuenzli, *Docker in action*. Simon and Schuster, 2019.