
FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

**Savidu Dias**

# INTEGRATION OF MLOPS WITH IOT EDGE

Master's Thesis
Degree Programme in Computer Science and Engineering
May 2023

**Dias S. (2023) Integration of MLOps with IoT Edge.** University of Oulu, Degree Programme in Computer Science and Engineering, 56 p.

# ABSTRACT

**Edge Computing and Machine Learning have become increasingly vital in today's digital landscape. Edge computing brings computational power closer to the data source enabling reduced latency and bandwith, increased privacy, and real-time decision-making. Running Machine Learning models on edge devices further enhances these advantages by reducing the reliance on cloud. This empowers industries such as transport, healthcare, manufacturing, to harness the full potential of Machine Learning.**

**MLOps, or Machine Learning Operations play a major role streamlining the deployment, monitoring, and management of Machine Learning models in production. With MLOps, organisations can achieve faster model iteration, reduced deployment time, improved collaboration with developers, optimised performance, and ultimately meaningful business outcomes.**

**Integrating MLOps with edge devices poses unique challenges. Overcoming these challenges requires careful planning, customised deployment strategies, and efficient model optimization techniques.**

**This thesis project introduces a set of tools that enable the integration of MLOps practices with edge devices. The solution consists of two sets of tools: one for setting up infrastructure within edge devices to be able to receive, monitor, and run inference on Machine Learning models, and another for MLOps pipelines to package models to be compatible with the inference and monitoring components of the respective edge devices.**

**This platform was evaluated by obtaining a public dataset used for predicting the breakdown of Air Pressure Systems in trucks, which is an ideal use-case for running ML inference on the edge, and connecting MLOps pipelines with edge devices.. A simulation was created using the data in order to control the volume of data flow into edge devices. Thereafter, the performance of the platform was tested against the scenario created by the simulation script. Response time and CPU usage in different components were the metrics that were tested. Additionally, the platform was evaluated against a set of commercial and open source tools and services that serve similar purposes.**

**The overall performance of this solution matches that of already existing tools and services, while allowing end users setting up Edge-MLOps infrastructure the complete freedom to set up their system without completely relying on third party licensed software.**

**Keywords: Edge computing, Machine Learning, MLOps, IoT**

**Dias S. (2023) MLOps-integraatio reunalaskennan tarpeisiin.** Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 56 s.

# TIIVISTELMÄ

**Reunalaskennasta (Edge Computing) ja koneoppimisesta on tullut yhä tärkeämpiä nykypäivän digitaalisessa ympäristössä. Reunalaskenta tuo laskentatehon lähemmäs datalähdettä, mikä mahdollistaa reaaliaikaisen päätöksenteon ja pienemmän viiveen. Koneoppimismallien suorittaminen reunalaitteissa parantaa näitä etuja entisestään vähentämällä riippuvuutta pilvipalveluista. Näin esimerkiksi liikenne-, terveydenhuolto- ja valmistusteollisuus voivat hyödyntää koneoppimisen koko potentiaalin.**

**MLOps eli Machine Learning Operations on merkittävässä asemassa tehostettaessa ML -mallien käyttöönottoa, seurantaa ja hallintaa tuotannossa. MLOpsin avulla organisaatiot voivat nopeuttaa mallien iterointia, lyhentää käyttöönottoaikaa, parantaa yhteistyötä kehittäjien kesken, optimoida laskennan suorituskykyä ja lopulta saavuttaa merkityksellisiä liiketoimintatuloksia.**

**MLOpsin integroiminen reunalaitteisiin asettaa ainutlaatuisia haasteita. Näiden haasteiden voittaminen edellyttää huolellista suunnittelua, räätälöityjä käyttöönottostrategioita ja tehokkaita mallien optimointitekniikoita.**

**Tässä opinnäytetyöhankkeessa esitellään joukko työkaluja, jotka mahdollistavat MLOps-käytäntöjen integroinnin reunalaitteisiin. Ratkaisu koostuu kahdesta työkalukokonaisuudesta: toinen infrastruktuurin perustamisesta reunalaitteisiin, jotta ne voivat vastaanottaa, valvoa ja suorittaa päätelmiä koneoppimismalleista, ja toinen MLOps "prosesseista", joilla mallit paketoidaan yhteensopiviksi vastaavien reunalaitteiden komponenttien kanssa.**

**Ratkaisun toimivuutta arvioitiin avoimeen dataan perustuvalla käyttötapauksella. Datan avulla luotiin simulaatio, jonka tarkoituksena oli mahdollistaa reunalaitteisiin suuntautuvan datatovirran kontrollonti. Tämän jälkeen suorituskykyä testattiin simuloinnin luoman skenaarion avulla. Testattaviin mittareihin kuuluivat muun muassa suorittimen käyttö. Lisäksi ratkaisua arvioitiin vertaamalla sitä olemassa oleviin kaupallisiin ja avoimen lähdekoodin alustoihin.**

**Tämän ratkaisun kokonaissuorituskyky vastaa jo markkinoilla olevien työkalujen ja palvelujen suorituskykyä. Ratkaisu antaa samalla loppukäyttäjille mahdollisuuden perustaa Edge-MLOps-infrastruktuuri ilman riippuvuutta kolmannen osapuolen lisensoiduista ohjelmistoista.**

**Avainsanat: Reunalaskenta, koneoppiminen, MLOps, Esineiden internet**

# TABLE OF CONTENTS

# ABBREVIATIONS

| | |
|---|---|
| API | Application Programming Interface |
| APS | Air Pressure System |
| AWS | Amazon Web Services |
| ACK | Acknowledge |
| CD | Continuous Delivery |
| CI | Continuous Integration |
| CT | Continuous Training |
| CLI | Commandline Interface |
| CPU | Central Processing Unit |
| DevOps | Development Operations |
| GPU | Graphics Processing Unit |
| HTTP | HyperText Transport Protocol |
| HTTPS | Secure HTTP |
| IoT | Internet of Things |
| JSON | JavaScript Object Notation |
| KPI | Key Performance Indicator |
| MICE | Multiple Imputation by Chained Equations |
| ML | Machine Learning |
| MLOps | Machine Learning Operations |
| MQTT | Message Queuing Telemetry Transport |
| REST | Representational State Transfer |
| RTSP | Real Time Streaming Protocol |
| S3 | Simple Storage Service |
| TPU | Tensor Processing Unit |

# 1. INTRODUCTION

With the rapid advancements in computing and capabilities of the cloud, the need to utilise Machine Learning (ML) functionalities with data collected from IoT devices is rapidly increasing. Advancements in cloud computing has paved the way for Internet of Things (IoT) and Artificial Intelligence (AI) to work together in starting a disruptive transformation in several domains [1]. The exponential growth in the amount of data generated by connected devices will eventually lead to a point where the cloud will not be able to sustain the requirements of the industry. For example, connected cars generate around 25GB per hour [1].

Edge computing enables generating insights and making decisions at the data source, reducing the amount of data sent to the cloud and central repository [2]. Additionally, performing ML inference at the edge has several benefits compared to traditional cloud-based ML [2]. For example, if there is a setup of 100,000 IoT devices processing 1 KB of data every second, running that processing on the cloud would amount to processing 100 MB per second, which could be computationally expensive, and if cloud services are being utilised for processing, it could be financially expensive as well. Additionally, factors such as network connectivity and its associated costs must be considered as well. Thus, running the processing on IoT devices themselves without having to connect to the cloud could save a large amount of these costs.

A real-world example that could be used to discuss the value provided by processing data closer to the source is Floyer [3] studying data management and processing costs of a remote wind-farm using a cloud-only system compared to a combined edge system. The edge-cloud system turned out to be 36% less expensive, costing only $28,927 as opposed to the cloud-only system that costs $80,531. Additionally, the volume of data required to be transferred was observed to be 96% less compared to the cloud-only system.

One of the major limitations of traditional approaches to running ML inference at the edge is that models are often trained centrally and deployed manually and individually to the edge nodes [4]. This thesis investigates, designs, and develops a solution that enables the automation of deploying ML models at the edge to run inference. A major practice that enables this automation is MLOps. At its core, MLOps is the standardisation and streamlining of ML lifecycle management [5]. The purpose of MLOps is to deploy and maintain ML models in production environments reliably and efficiently.

Adhering to proper MLOps standards and practices [5] would allow the automation of the entire ML model lifecycle. That is, the collection of new data from edge devices, updating models with new data, and deploying the updated models to the edge can all be done without human intervention. The main advantage here is that engineers would not have to worry about manually updating the model to account for data drift [5].

Figure 1 shows the thought process in which this thesis investigates the practices followed in the processes of delivering and deploying models into the edge, and how these practices can be optimised such that similar results could be achieved with edge devices.

The research question posed by this thesis project is "How can a platform that enables the integration of MLOps practices with Edge devices be developed using state of the art tools and methods?".

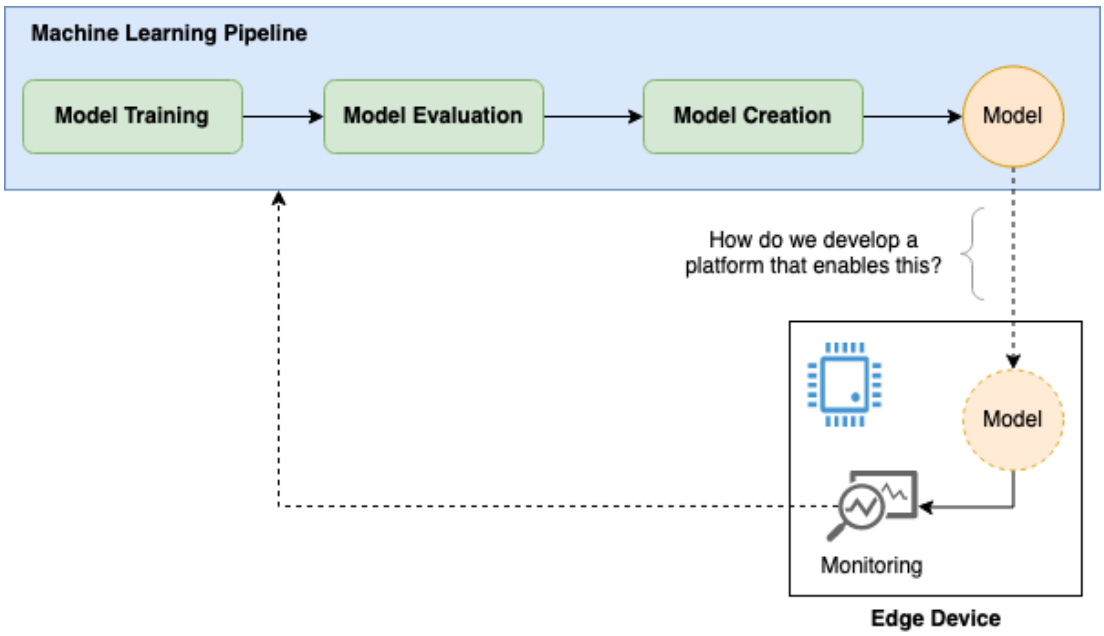


Figure 1. Project thought process

The main contribution is the proposal of a set of tools that enable the delivery of ML models to edge devices following best practices, while conducting an in-depth study of scientific literature and discussion from its findings. Additionally the tools provide functionalities to analyse and monitor IoT data and edge node operations such that the results could be used to further improve the models.

After the proposed set of tools were developed, a validation scenario was created using an existing dataset in order to evaluate the platform and its performance. The platform was evaluated by creating a simulation environment from the data and feeding them to the edge device at a controlled rate.

# 2. RELATED WORK

This chapter gives focus to existing research and literature, along with existing commercial tools and solutions that work towards integrating MLOps with edge devices. Before delving into existing research and solutions, we must first go over some basic concepts and terminology of the topic.

## 2.1. Concepts and Terminology

### *2.1.1. Edge Computing and Machine Learning*

Edge Computing is a decentralised computing model where data processing occurs closer to the source of the data instead of a centralised data centre. The advantages of edge computing includes improved performance, reduced latency, and reduced network bandwidth usage. However the main reason edge computing is gaining in popularity is the fact that it can be used to process data in real-time and minimise the time delay between data collection and decision making, providing real-time analytics, predictive maintenance, and intelligent decision making. It is usually implemented in devices such as IoT sensors, mobile devices, and microcontrollers. The idea of edge devices can be used to refer either an end device or an edge-server, which is a computational entity with strong data processing capabilities that is situated closer to the edge [1]. These edge devices possess both computational and communication capabilities [6].

Machine Learning (ML) is a subset of Artificial Intelligence, where machines learn from data and improve their performance over time. Training ML models to learn from data traditionally requires a large amount of computational resources.

Running ML on the edge involves deploying ML algorithms on edge devices. The main advantage of running ML on the edge is the improved response times in real-time applications. Additionally, it can reduce bandwidth requirements and data transmission costs. Edge computing can make use of ML to make intelligent decisions and take autonomous actions without relying on a cloud connection. Murshed, MG Sarwar, et al. [1] discuss reasons and issues with running ML at the edge in their survery paper. Transferring raw data to cloud servers increases communication costs, causes delays in system response, and makes any private data vulnerable to compromise. In order to address these issues, the processing data closer to its source and only transmitting necessary data to the cloud is a sensible approach [2].

### *2.1.2. MLOps*

Machine Learning Operations (MLOps) is a set of practices and tools that aim to streamline and operationalise the deployment, management, and monitoring of ML models. It combines the principles of DevOps, data engineering, and ML to create a systematic and efficient process for managing ML workflows. MLOps involves automating the end-to-end ML lifecycle including data preparation, model training, deployment, monitoring, and maintenance.

G Symeonidis et al. [7] have conducted a survey studying the most important and influential work in MLOps. A ML model is not independent, but is a part of a wider software system and consists of not only code, but also of data. As the data is constantly changing, the model is constantly called upon to retrain from new data that emerges. While there are several attempts to capture and describe MLOps, the one that is best known is the proposal of ToughWorks [8]. The authors have defined MLOps as "a software engineering approach in which an interoperable team produces machine learning applications based on code, data, and models that can be replicated and delivered reliably at any time in short custom cycles".

Integrating MLOps with edge computing involves deploying and managing ML models directly on edge devices, closer to the data source. MLOps for the edge focuses on optimising model performance, resource usage, and latency to enable real-time inference on edge devices. Apart from deploying and managing models, MLOps for the edge also involves monitoring and updating models on edge devices, ensuring they remain up-to-date and accurate while operating in resource-constrained environments.

The surveys conducted by Murshed, MG Sarwar, et al. [1] and G Symeonidis et al. [7] highlight different approaches and tools used in MLOps and running ML on the edge. Additionally, the authors highlight the advantages and usefulness of systems that enable the deployment of models on edge devices. The authors of both papers discuss approaches that could be used to integrate MLOps with edge devices, which involves running inference in the form of a cloud service. This indicates that the research area of deploying ML models into edge devices still remains to be explored.

## 2.2. Previous Research and Literature

### *2.2.1. MLOps Processes and Paradigms*

M. Treveil et al. [9] and S. Alla et al. [5] follow an MLOps paradigm which includes a loop consisting of seven distinct phases: plan, create, verify, package, release, configure, and monitor. These phases describe the process that is traditionally followed when deploying ML models, all the way from the point in which data is collected and studied. The process is described as a set of steps that are performed iteratively, each iteration making improvements on the previous one.

In these designs, the phases in blue are mainly carried out by data scientists and domain experts, while the phases in green are operated by software and data engineers. The *plan* phase involves a data scientist that studies data sources. An important part of this phase is feature engineering, where the data scientist derives a set of processes and features that later need to be selected. The derived processes and features are used in the *create* phase, where a large set of ML models are trained, while adapting the feature set, with the objective of crafting effective models on the training data. The *verify* phase checks how the model behaves in a production environment.

Once the model is verified, it is stored in a repository for future use in the *package* phase. The model is packaged into an executable binary file. This can be followed in two alternative approaches: model containerization (i.e. Docker container), or model embedding, where the model is embedded directly inside the application code.

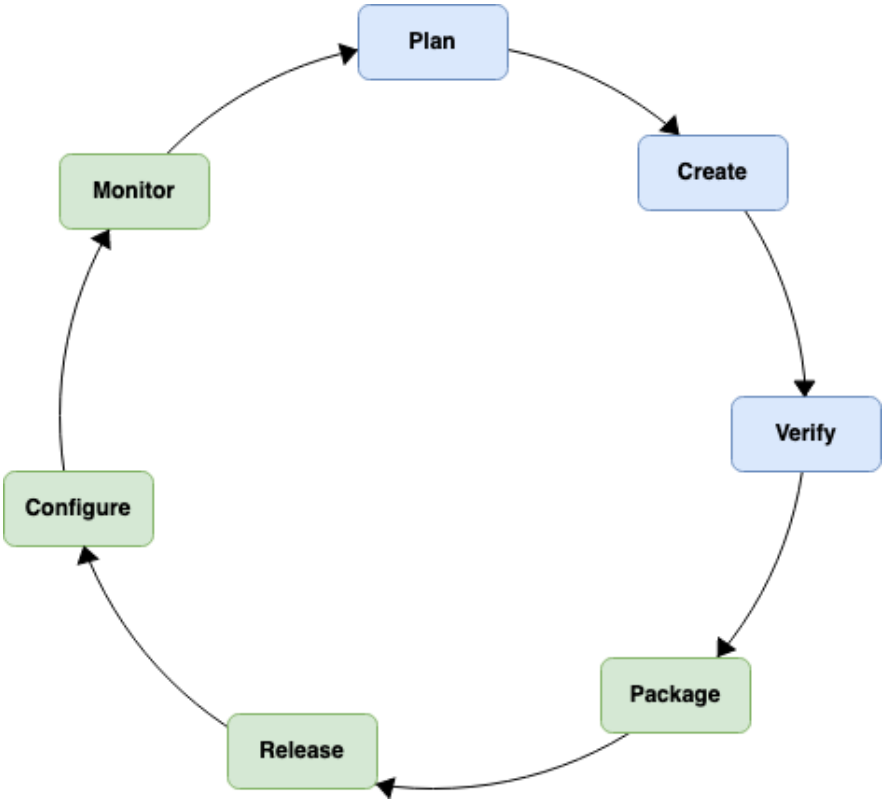


Figure 2. Standard MLOps Paradigm

Thereafter, the *release* phase verifies that the packaged model is suitable for the production environment by analysing factors such as system performance (memory utilisation, CPU/GPU load). These models are then fine-tuned to achieve the target objective in the *configure* phase. The *monitor* phase aims at detecting issues causing performance degradation, triggering an additional iteration of the MLOps loop. The factors considered when analysing the performance degradation includes system performance, and performance of the model (drift, outliers, biases, non-representativeness of input features, etc.).

M. Antonini et al. [10] have designed a framework that can be used to deliver adaptation and evolving capabilities to resource-constrained IoT sensors such as 32-bit microcontrollers. The authors have modified the standard paradigm to accommodate resource-constrained devices.

These phases are carried out taking into account that the models will be deployed on resource-constrained devices. However, there are a few additional phases such as *adapt & optimise*, which aims to optimise the trained model to fit in the target computing platform.

G Symeonidis et al. [7] also discuss the MLOps lifecycle in their survey paper, and describes the MLOps paradigm in three basic procedures: collection, selection, and preparation of data. A simplified form of such a paradigm can be found in Figure 3. After collecting, evaluating, and selecting the data that will be used for training, the process of creating models and training them is automated. This produces more than one model which can be tested and experimented in order to produce a more efficient

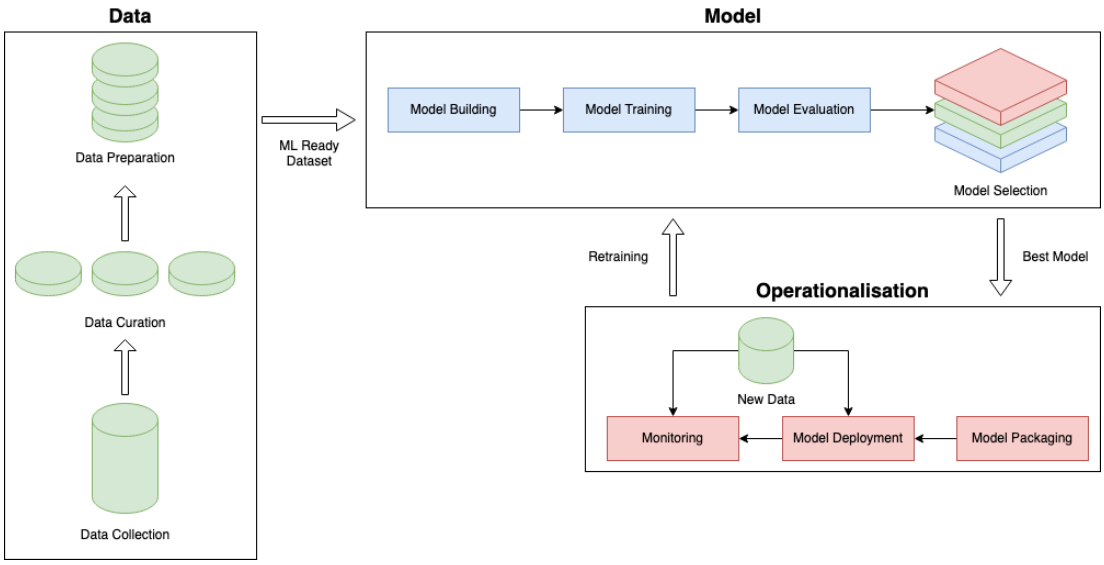


Figure 3. MLOps Lifecycle

and effective model. Finally, the model can be monitored, collecting new data which will be used to re-train the model, this ensuring its continuous improvement.

### *2.2.2. Edge-MLOps Designs*

The paradigms discussed in 2.1 have become standard practice in modern MLOps process development, with modifications that suit specific use cases. These processes are utilised in architectures that enable the deployment and execution of ML models on the edge.

For example, in the Tiny-MLOps framework, M. Antonini et al. [10] have proposed an anomaly detection system as depicted in Figure 4 When the digital samples reach the embedded device from the sensors, they are managed by the Data In Manager component, which is the entrypoint of the Tiny-MLOps loop. Data is moved from the computing entity to the other via the Data Comm Managers, which handle complex networking protocols.

The *Data In Manager* forwards incoming data into the *Local Inference Engine*. If the inference engine detects an anomaly, the data is sent to the *Pool Inference Engine* to be inferred by a pool of models. The model pool can decide if the anomaly identified by the model in the *Local Inference Engine* is a false positive.

The *Knowledge Base Manager* stores the data associated with the normal and anomalous behaviours. It also stores the models trained for the *Pool Inference Engine* and the embedded device.

The *Pool Inference Engine* keeps monitoring the performance of the model in the embedded device. If this model shows an error rate above a predefined threshold, the *Pool Inference Engine* can trigger the replacement of the model to improve the system's performance. This selected model is forwarded to the *Embedded Model Deployment Agent*.

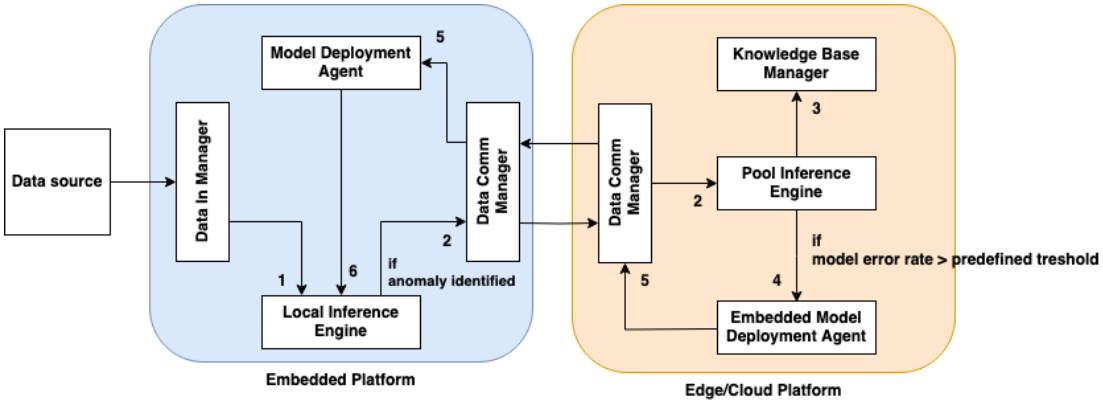


Figure 4. Proposed anomaly detection system supported by the TinyMLOps loop

The *Embedded Model Deployment Agent* receives the model to deploy over the embedded platform, packages it together with the feature extraction pipeline and forwards the package to the *Data Comm Manager*. This is passed into the *Model Deployment Agent* that will replace the existing model in the *Load Inference Engine*.

E. Raj et al. [4] present a framework to automate and operationalise the Continuous Delivery (CD) and Continuous Integration (CI) of ML devices to edge devices. The framework uses a hybrid edge and cloud solution. The *cloud solution* consists of requirements such as the inclusion of a ML pipeline for model training and re-training, CI and CD, source code management, data storage, fleet analysis. The requirements of the edge device includes continuous integration and data streaming with IoT devices, CI and CD from cloud to edge, personalisation, and the ability to perform ML inference.

There are two main layers of the framework:

1. Cloud orchestration layer (runs on the cloud platform)
2. Edge inference layer

Compared to Tiny-MLOps, this solution is geared more towards devices which are less resource-constrained, and are capable of utilising the potential of CI and CD practices. Tiny-MLOps uses its Pool Inference Engine in the cloud platform to train models from new data. The Cloud orchestration layer in this solution serves an equivalent purpose. The main task that is carried out in this layer is model training. This is due to the availability of high performance and compute resources. It utilises a ML pipeline to facilitate model training on the cloud.

Since a full-fledged ML pipeline is used, it is capable of handling specialised ML models for edge devices by spawning pipelines for the specialised models. The same functionality cannot be achieved in Tiny-MLOps, as a pipeline is not used to model the Pool Inference Engine. The steps in the ML pipelines are data ingestion, model training and re-training, model evaluation, model packaging, and model registering. In the packaging phase, the tested model is packaged into a container for standardised deployments. Thereafter, the model is registered and stored in a registry where it is ready for quick deployment into an edge device. The pipelines also come with a ML model repository and container storage.

The Edge Inference layer serves a functionality equivalent to the Embedded Platform in TinyMLOps. It focuses on orchestrating the operations for the IoT and edge devices, enabling real-time ML and inference. The layer communicates with the data source using MQTT. Having the added advantage of following CI/CD practices, it can connect with pipelines to facilitate model deployments in the form of containers. These models are able to optimise and re-train themselves when needed, while constantly learning to serve better. By using containers, the models can be tailored to specific architectures, improving hardware stability and standardisation among nodes in a network.

Targeting frameworks for less resource-constrained edge devices allows this additional level of flexibility. SensiX++ [11] follows a similar design in fetching models from pipelines in the form of container images.

#### MLOps Maturity Model

When comparing MLOps with Devops practices, the main differences include factors such as the introduction of additional testing procedures such as data and model validation in the Continuous Integration (CI) phase. Processed datasets and trained models are delivered to their respective environments using Continuous Delivery (CD). The main factor that differs MLOps from DevOps is the inclusion of a third step known as Continuous Training (CT) [12]. CT is used to monitor data and model performance degradation, and re-train models with newer data to improve model performance.

The authors of [13] have conducted a study to identify activities associated with the adoption of MLOps, and the stages in which companies evolve as they gain maturity and become more advanced. These are outlined in 4 stages.

**Automated data collection** is when a company experiences the transition from a manual to an automated data collection process. For this transition, there is a need for mechanisms to aggregate data from different sources, which can be stored and accessed whenever required [14]. The **Automated model deployment** transition is achieved by implementing provisions for automated model deployment to environments. This is achieved by providing a dedicated infrastructure-centric CI/CD pipeline. The infrastructure required for this transition includes model hosting, evaluation and maintenance [14], means to register, package, deploy models, and integration of software environments for training and deploying models [15]. **Semi-automated model monitoring** provisions for triggers when model performance degradates, and tools for diagnostics, performance monitoring, and addressing model drift. This stage may also include automation scripts to manage and monitor models based on drift. **Fully-automated model monitoring** is the integration of CI/CD and orchestration and CT pipeline to re-train models when performance degrades. For this final transition, there is a need to ensure the certification of models, governance and security controls, auditing of model usage, and reproducibility. Additionally, there should be assurance that data security and privacy requirements are built into data pipelines [12].

### *2.2.3. ML Inference on the Edge*

There is little research done on the area of setting up generic frameworks for running ML inference on the edge. Most existing solutions have catered to a specific scenario.

However, when developing a set of tools to run ML inference on the edge, it is vital that it is designed with the objective of allowing developers to be able to run ML models of their choice.

SensiX++ [11] is a multi-tenant runtime for adaptive model execution with integrated MLOps on edge devices. The functionalities of the platform involves:

1. Preprocessing data from a real-world sensor into a compatible input format.
2. Executing the model within a model-specific framework (e.g. Tensorflow, Pytorch).
3. Serving the inference through a well-defined interface (e.g. REST API).

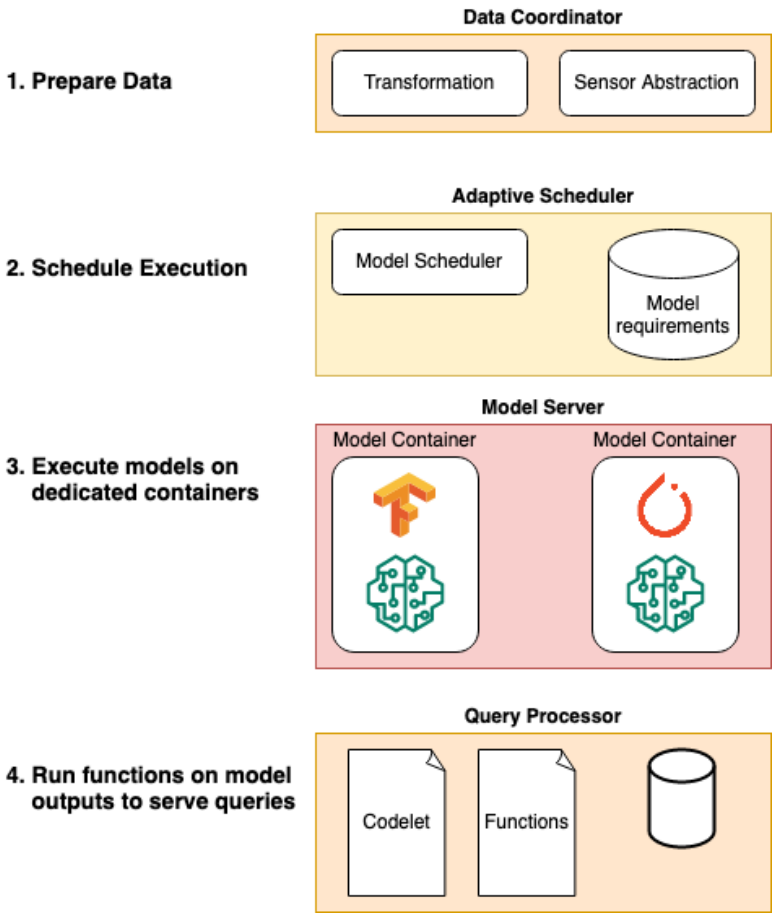


Figure 5. SensiX++ system architecture

Its system architecture consists of four main components: data coordinator, adaptive scheduler, model server, and query processor. The *Data Coordinator* interfaces with the sensors connected to the host device decoupling data production from data consumption. It transforms sensor data into different formats, meeting the requirements of different models.

The *Adaptive Scheduler* decides the execution schedule of multiple models balancing throughput and latency. It allows models to specify their latency requirements, and provides best-effort performance while meeting those latency budgets. The *Model Server* enables and optimises the deployment of multiple models

on edge devices. It creates model-specific containers, which are assigned different processors (CPU, mobile GPU, TPU by the scheduler. The *Query Processor* interfaces with external applications and handles their queries. This allows the execution of functions that process the outcome of model containers. These functions can be provided by developers during the deployment phase.

Developers can define functions using two files; *codelet*, and *functions* files. The *codelet* file provides a number of executable functions that use the outcome from one or more functions. The *functions* file lists models and/or other functions whose outputs are necessary for the execution, and the type of its outcome. The query processor also maintains a storage of model outputs to serve historical queries. This is served in the form of API endpoints.

### *2.2.4. Solution Evaluation Methodologies*

Once a solution has been developed, it must be evaluated in order to measure its effectiveness. The first part of the evaluation process is selecting a hardware platform to run the solution in. Edge MLOps [4] runs their evaluation on a Raspberry Pi 4, NVidia Jetson Nano 2, and Google TPU edge. SensiX++ uses an NVidia Jetson AGX, and a Goofle Coral accelerator. The NVidia and Google edge accelerators are powerful units capable of running relatively heavy workloads for edge devices.

On the other end of the hardware spectrum, Tiny-MLOps [10] uses a 32-bit microcontroller flashed with MicroPython firmware, which is a lean and optimised open-source implementation of Python 4.3 for MCUs. However, it uses an edge gateway that is implemented using a Raspberry Pi 4 board with 8GB RAM, which hosts the knowledge base manager and pool inference engine.

Once the appropriate hardware has been selected, the next step in the evaluation process is coming up with a scenario the solution can be used in. Tiny-MLOps [6] has done its validation on a real-world industrial scenario consisting of a wastewater management plant. However, the initial set of models were trained using the first 60% of a NASA dataset. These were stored in a model repository consisting of 20 models. The models were fed using the features (kurtosis, skewness, mean) computed over a continuous non-overlapping time window of 500ms. A threshold of 0.6 was set to detect anomalies.

Edge MLOps [4] has implemented an experimental setup of a similar nature, where they continuously monitor the air quality and conditions in rooms equipped with different IoT devices. Sensor data is sent to a gateway, which then uses MQTT to send the information to a broker the edge device is connected to. The objective was to train ML models to forecast the air quality of a room in the next 15 minutes. Prediction errors were evaluated with a loss function (RMSE) comparing the predictions and the targets.

Conversely, SensiX++ [11] follows an approach were various vision and audio recognition models are tested and compared against two baselines. As a result, this does not use a scenario to evaluate its solution. The main metric that is evaluated is its effectiveness by measuring model inference throughput (number of inferences per second). It also measures efficiency by measuring the hit ratio, which is defined as the

ratio between the number of data samples sent for inference and the total number of data samples generated. A higher hit rate indicates higher resource efficiency.

## 2.3. Commercial and Open-Source Tools and Services

The integration of MLOps practices with edge devices can be achieved using existing tools and services. The objective of this subchapter is to briefly describe the MLOps capabilities of these platforms with the edge. Finally, the capabilities of this platform will be compared with the discussed existing tools in the Evaluation chapter.

### *2.3.1. AWS IoT Core and Greengrass*

AWS IoT Core [16] is a managed cloud service that enables devices to connect securely and interact with the AWS cloud. It provides a foundation for implementing Edge and MLOps capabilities. AWS Greengrass [17] extends IoT Core to edge devices allowing local processing and data management. It allows devices to run AWS Lambda functions and sync data with the cloud when connected.

Greengrass brings the power of edge computing by allowing its users to deploy Lambda functions directly on the edge devices, enabling real-time processing, reducing latency, and improving responsiveness. Additionally, it provides local storage capabilities allowing devices to collect, process, and store data locally.

Greengrass supports ML inference at the edge by running Lambda functions that utilise ML models. Greengrass allows the deploying and updating ML models on the edge, and ensures that each device is running the latest model that is available to them.

When it comes to monitoring, AWS IoT Core provides device management capabilities, allowing to remotely monitor and manage edge devices. IoT Core lets users perform tasks such as deploying software updates, monitoring device health, and remotely troubleshooting issues.

AWS IoT Core and Greengrass can be seamlessly integrated with other AWS services like IoT Analytics, IoT Events, and IoT SiteWise to build comprehensive MLOps pipelines and automate workflows for managing and monitoring edge devices and ML models.

### *2.3.2. Azure IoT Edge*

Azure IoT Edge [18] is a cloud-managed service provided by Microsoft that extends Azure services and capabilities to edge devices, allowing for local processing and decision making. It provides a runtime environment that allows users to deploy and run containerized modules on edge devices, which can be developed using various programming languages and frameworks.

Azure IoT Edge allows ML models to be deployed as modules on edge devices, enabling real-time inference and decision making without relying on cloud connectivity. It supports popular ML frameworks such as TensorFlow and ONNX, allowing engineers to train models in the cloud and deploy them at the edge with ease.

The MLOps capabilities of Azure IoT edge involve version control, model deployment, and monitoring functionalities.

### *2.3.3. PlatformIO*

PlatformIO [19] is an open-source ecosystem for embedded development that supports a wide range of hardware platforms and development frameworks. Edge capabilities in Platform IO enable developers to build applications for edge devices such as microcontrollers and single-board computers by providing a unified development environment.

MLOPs capabilities in PlatformIO facilitate the integration of ML models into embedded systems by providing tools and libraries that enable the deployment of models, ensuring efficient execution and resource utilisation. The MLOps capabilities support popular ML frameworks such as Tensorflow, PyTorch, and scikit-learn, allowing seamless integration with existing workflows.

Developers can use PlatformIO to train and optimise ML models on more powerful hardware platforms and deploy optimised models on edge devices. The platform provides features for model conversion, quantization, and optimization, enabling developers to adapt models to the constraints of edged devices without sacrificing performance.

### *2.3.4. Eclipse IoT*

Eclipse IoT [20] provides an Edge Computing framework that enables the deployment, and management of applications at the edge of the network. Some of the capabilities of this platform include edge device management, data collection and processing, and local analytics.

The platform provides a set of tools and libraries for building Edge applications, including Eclipse Kura, Eclipse Leshan, and Eclipse Kapua. The MLOps components of Eclipse IoT enable the deployment, monitoring, and management of ML models at the edge.

It integrates popular ML frameworks like Tensorflow, PyTorch, and scikit-learn, allowing developers to train and deploy ML models directly on edge devices. The MLOps capabilities include versioning, model deployment, monitoring, and model lifecycle management.

These capabilities are facilitated by providing infrastructure for collecting and processing data at the edge, enabling real-time inference and decision making without relying on cloud connectivity.

The combination of Edge and MLOps capabilities in the platform empowers developers to create intelligent edge applications that can process and analyse data locally, reducing latency, improving responsiveness, and enabling real-time insights.

## 2.4. Summary

The purpose of this chapter is to conduct a comprehensive and critical summary of existing research on the topic of integration of MLOps practices with edge devices by producing knowledge, identifying research gaps, evaluating previous research, and providing context for new studies. By examining and analysing published literature and existing solutions, we have gained deeper understanding of the current state of knowledge, and identified areas for further investigation.

The study begins by analysing the basic concepts that should be discussed in this topic; Edge Computing, Machine Learning, and MLOps. These topics are analysed by studying survey papers related to the respective topics which discuss the current academic state of knowledge in these areas, while highlighting the knowledge gaps that are left to be explored.

The next topic investigates existing literature conducted in the concepts that were discussed, which going over designs that have been used for work similar to delivering ML models into edge devices. Thereafter, we explore existing methodologies of running ML model inference in edge devices, followed by how such solutions could be evaluated by taking existing research into consideration.

The final topic looks into commercial and open-source tools and services that have functionalities similar to delivering and running ML models on edge devices. The objective of studying existing solutions is to identify MLOps capabilities of these platforms with the edge and comparing them with each other.

# 3. DESIGN AND ARCHITECTURE

## 3.1. Project Scope

The objective of this project is to **research, design, and develop a platform that enables seamless integration of MLOps practices [21] with edge devices**. As a result, the platform that is being developed does not have to support the requirements of the *automated data collection* stage discussed in chapter 2.2.2. However, it should be able to support *automated model deployment*, where it can easily connect to dedicated CI/CD infrastructure, and consist of means to keep track of previous and existing models. It should also support *semi/fully automated model monitoring stages* by providing means to monitor model performance, diagnostic tools, as well as certificates and other security controls that address security and privacy requirements built into data pipelines.

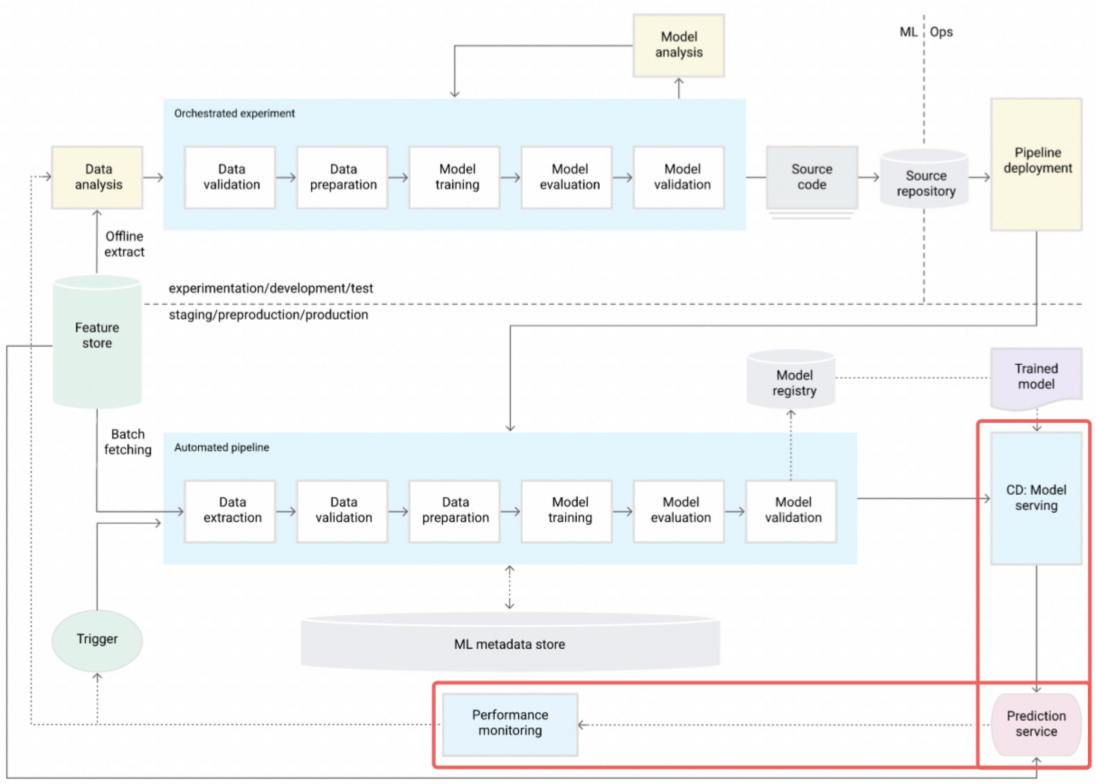


Figure 6. Sample MLOps pipeline guidelines [22]

Figure 6 shows an example MLOps architecture by Google Cloud [22] that accommodates the requirements of the *fully-automated model monitoring* stage, as a part of its MLOps guidelines. MLOps is a vast field of research, and the area highlighted in red in Figure 6 indicates the scope of this project in reference to its objectives. However, instead of having a prediction service, there should be a component that is responsible for running inference within the edge device.

### 3.2. Continuous Delivery

In DevOps, Continuous Delivery (CD) is an automated approach in which developers produce software in quick cycles in an automated manner such that reliability is ensured. The objective of CD is to build, test, and release software with a greater speed and frequency. In the context of MLOps, CD is no longer about a single software package or service. Instead, it is a model training pipeline that should automatically deploy an inference service [22].

Traditional MLOps practices deploy ML models to a target environment to serve predictions. This deployment can be microservices with REST API to serve online predictions, an embedded model to an edge or mobile device, or part of a batch prediction system [22]. An important step in CD is to ensure that the deployed model is compatible with the target infrastructure. For example, the packages required for inference must be installed in the serving environment.

When developing a tool that enables the continuous delivery of models into edge devices, it is important to consider the different possibilities in which models can be made available. For example, a model could be trained and packaged using a serialising library such as Pickle [23] for Python, or ONNX [24], which is an open format that represents ML models. Therefore, it is important that the tool should allow the model deployment process to be as generic as possible. The authors of SensiX++ [11] and Edge MLOps [4] use Docker containers to package and deploy models in a standardised manner. Additionally, there are several advantages to having models run in containerised environments.

The reasons for SensiX++ for using containerised deployments include:

- Model abstraction and process isolation by running multiple models in individual containers representing their runtime dependencies.
- Being able to create model-specific containers meeting their package requirements.
- Assignment to different processors (CPU, Mobile GPU, etc.).
- Assignment to different processors Being able to dedicate a separate container to facilitate post-processing functions.

Edge MLOps uses containers for deployments such that:

- A model repository and container storage could be used to enable fast deployment.
- Provides a means of standardised deployment.
- Almost all existing CD pipelines facilitate model deployments in the form of containers.
- Containers can be tailored to the specific architectures of the edge devices.
- Docker containers are a well-established standard in industry.
- Ease of rollback operations in the event of failures.

The main disadvantage of packaging and deploying ML models as container images is that it would not be possible to execute them on non-linux based embedded real time operating systems. It is assumed that most use cases that involve running ML inference on the edge would also consist of edge devices with operating systems capable of running containers. As a result, given the advantages offered by using

Docker containers to package, deliver, and deploy ML models, it was decided to use them for the CD part of the MLOps pipeline.

### 3.3. Inference

One of the biggest challenges of using a platform that facilitates the deployment of models of any type is that the method of inference must be made as generic as possible. Since the model is delivered and deployed into the edge device in the form of a Docker container, the image should have the functionality to take input from a data source (sensor), preprocess the data, run inference, and provide an output.

This is achieved through the introduction of an "**Inference API**". The objective of the Inference API is to act as an interface between the sensors and model. The API accepts input through an MQTT input topic, and provides an output to a different topic. The MQTT broker resides within the edge device, which will be discussed later in the "Proposed Architecture".

The steps taken by the inference API is as follows:

1. Load model into memory
2. Preprocess sensor input
3. Run ML inference on input
4. Return inference results

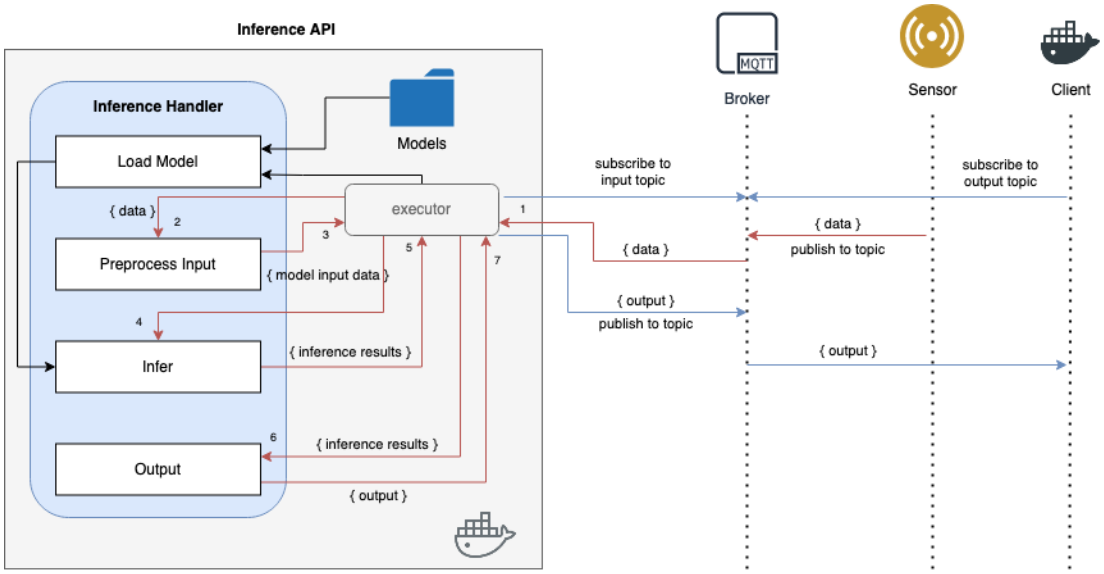


Figure 7. Inference API workflow

Figure 7 shows the flow of how data from a sensor is processed and returned. When the ML model is deployed, the inference API subscribes to a predefined MQTT input topic, and a user-defined client application subscribes to a predefined output topic.

The Inference Handler consists of a set of functions; some of which should be overridden by the developer. The components of the model container image consists of the inference API and supporting files.

- **Executor** - controls the sensor input and inference API data workflow. The actions taken by the executor should not be overridden by the developer.
- **Models directory** - consists of the packaged model file, and any supporting files used in the preprocess or inference process.
- **Load Model** - loads the model and any supporting files from the models directory into memory. The function must be overridden by the developer to utilise any libraries/dependencies of the model. This function is only executed when the model is deployed.
- **Preprocess Input** - validates and transforms the input obtained by the sensor to match the input expected by the ML model. This must be overridden by the developer.
- **Infer** - uses the preprocessed input against the model loaded into memory and obtains the inference results. This function should be overridden by the developer.
- **Output** - transforms the inference results into a format that is human readable, or readable by a client application. This function should be overridden by the developer.

The expected workflow of the API during inference is as follows:

1. The **Executor** reads data from the sensor.
2. The **Preprocess Input** function is run by the **Executor** providing the sensor data as input.
3. Results of the function are obtained by the **Executor**.
4. The **Infer** function is run by the **Executor**, providing the preprocessed input data as input.
5. Inference results are obtained by the **Executor**.
6. Inference results are fed to the **Output** function by the **Executor**.
7. Human/client application readable output is fetched by the **Executor** and published to the predefined output topic.

### 3.4. Performance Monitoring

Continuous Training (CT) is the main step which helps differentiate MLOps from DevOps practices [12]. Performance monitoring is a major component of the CT process, which is used to monitor the performance of data and models to detect degradation. If the the monitor identifies that the performance falls below a given threshold, it will request the CI pipeline to re-train the model with newer data.

CT is used to monitor data and model performance degradation, and re-train models with newer data to improve model performance. CT is a vital component that must be included in order to meet the objective of integrating MLOps practices with edge devices.

In addition to being able to detect performance degradation (drift), keeping track of data and model performance provides many benefits from an analytical standpoint [25]. Some of which include:

- **Compliance** - meeting regional regulations for IoT networks may require precise measurements at the edge to monitor decision making and feedback loops on the physical plane [26].
- **Ease of use** - may require data to be processed close to the IoT node [26].
- **System stability** - stability under heavy load demands scalability and throughput [27].
- **System safety** - systems that interact with their surroundings may benefit from physical proximity to models in order to speed up decision making [28].

### *3.4.1. Performance Monitoring KPIs*

John M.M. et al. [13] discuss the preconditions that should be met for the transition of a company's MLOps practises into a stage where its model monitoring is fully automated. At such a stage, performance monitoring should ensure:

- Model certification
- Governance and security controls
- Model explainability
- Auditing of model usage

In order to meet these requirements, it was decided to design the platform such that the performance monitoring system should keep track of the following:

- Input data from sensor(s)
- Model inference results
- CPU usage during inference
- Memory usage during inference
- Time taken for inference

Although there are many other factors that can be considered for performance monitoring. The above were decided for this thesis project due to their importance, simplicity, ease of implementation, and time constraints of the project.

### *3.4.2. Data Collection, Storage, and Retrieval*

Data about the sensor input, inference results, CPU, memory, and inference time are taken **immediately after inference** and stored in a JSON file with the following format.

```
{
    "id": <INFERENCE REQUEST ID>,
    "input": {<SENSOR INPUT>},
    "output": {<CLIENT APPLICATION READABLE INFERENCE RESULT>},
    "cpu": <CPU USAGE>,
    "ram": <MEMORY USAGE>,
    "time": <INFERENCE TIME IN NANOSECONDS>
}
```

The JSON file containing this data is saved with the current epoch timestamp as its name (e.g. 1683102806.json) under a predefined directory structure (e.g. " /.edgeops/history") in the edge device.

SensiX++ [11] consists of a component called the "Query Store and Data Server", whose purpose is to serve query requests. This consists of a set of microservices as API endpoints.

Similarly, the retrieval of the performance monitoring JSON files are made available to be retrieved from storage through a collection of REST API endpoints. The endpoints are as follows:

- `GET ./performance` fetches all performance data files stored in the device
- `GET ./performance/:start_timestamp/:end_timestamp` fetches all performance files within a given timestamp range
- `DELETE ./performance` removes all performance data from storage
- `DELETE ./performance/:start_timestamp/:end_timestamp` deletes all performance data within a given timestamp range

These REST endpoints are made available using a simple HTTP server running locally in the edge device. DELETE requests are also supported such that the user has the option to delete existing data in order to preserve storage space within the edge device.

### *3.4.3. Implementation*

The position of performance monitoring in the MLOps infrastructure for edge devices is use-case dependent. Some scenarios may require the model performance to be monitored within (or close to) the edge device, and others on the cloud. A tool that allows MLOps integration should accommodate both of these scenarios.

For both scenarios, the engineers designing the MLOps pipeline must define how often performance monitoring should be done. The tool allows this to be achieved by requesting a schedule for the performance monitoring job. The syntax of this schedule is the same as how jobs are scheduled in Linux devices with [29].

If the engineer chooses to do performance monitoring in the cloud, this could be specified in the tool. During this step, a remote HTTP endpoint must be provided for the platform to upload the performance data to. This endpoint must

- Accept multipart/form-data to retrieve a .zip file
- Consist of a parameter that helps to map the data its respective edge device

Afterwards, the performance monitor follows the provided cron schedule and fetches all of the data files related to performance in the edge device. Then, it packages them into a .zip file and uploads it to the provided endpoint. If the service receives a 200 OK response from the file upload, it will remove the existing data files in the device. Thereafter, it is the task of the engineer to unpack the obtained .zip file from the server-side and implement their own performance monitoring logic. If the performance falls below a given threshold, the server-side implementation should then trigger the CI pipeline to re-train the model with the new data.

The main disadvantage of this approach is that it does not allow the use of the API endpoints in the device, which compromises the flexibility of any monitoring logic being implemented. If the project requires performance monitoring to be done at the edge, the user has the option to implement the monitoring logic themselves, and package it into a container image. The deployed container can make use of the API endpoints available in the device and analyse the performance data accordingly.

Similar to the cloud approach, the API endpoints fetch the data files and package them into a .zip file, which can be unpacked and processed by the user-defined application in the container image.

## 3.5. Proposed Architecture

A platform that enables the integration of MLOps practices with Edge devices should facilitate the components that have been discussed in this chapter. Thus, the proposed architecture is split into two sections.

1. Platform architecture describes MLOps process in general and how it integrates with edge devices.
2. Edge architecture discusses the components and infrastructure running inside edge devices that facilitate the functionalities of the platform.

### *3.5.1. Platform Architecture*

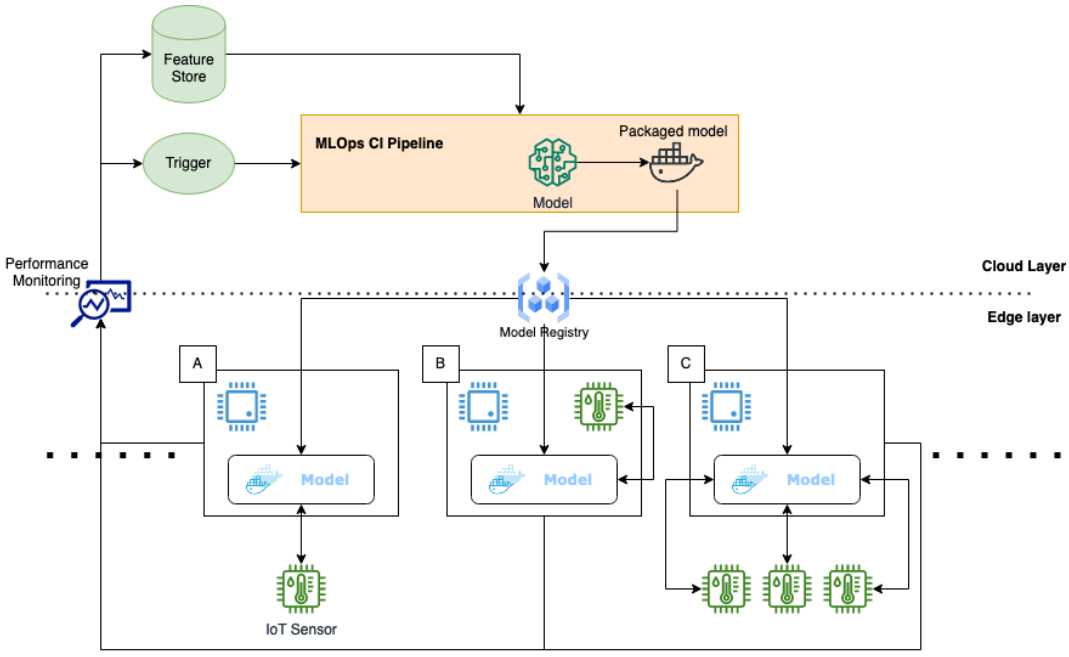


Figure 8. Platform architecture

As shown in Figure 8, the edge-MLOps platform consists of two layers: cloud and edge. The cloud layer infrastructure works towards producing and packaging models

to be delivered to edge devices. These models could be device-specific, or a set of common models for a group of devices.

Although the **cloud layer** is displayed in the architecture, its inner workings are not relevant and out-of-scope for this project. The main component is the **CI pipeline**, which is responsible for training and validating a model for a specific dataset from a **feature store**.

The trained model is then packaged into a Docker image and pushed into a **model registry**. The model registry sits between the two layers, as its implementation is defined by the project architects/engineers. Engineers could choose to host the model image in a registry service (e.g. Docker Hub[22]) or a private cloud service (e.g. AWS ECR, ACR, etc.) in the cloud. The repository could also be located in a self-hosted registry (e.g. Docker registry server[23]) closer to the edge nodes.

The **edge layer** refers to the collection of edge devices in the system. The packaged models produced by the CI pipeline are fetched from the model registry by each edge device.

Edge IoT systems can be of different arrangements. A considerable amount of IoT devices are not suitable for edge computing. These are usually simple **sensors and actuators** that have network access capabilities. These can communicate with edge devices, which have computational resources capable of performing ML inference. These are indicated by the edge device label "A".

It is also possible for the sensors and actuators to be a part of the device itself. A common example of such systems are smartphones, or inexpensive edge computing devices like the NVIDIA Jetson (labelled "B").

Another arrangement is having multiple IoT sensors and actuators within a localised region connect to a single device. This is indicated by the device label "C". There are many other similar arrangements that are possible when designing IoT systems. These were kept in mind when the platform was designed.

Performance and inference data are sent to the performance monitor for analysis. If the performance monitor identifies that the model or device is performing at a level below a predefined threshold, it will send a request to the trigger in the cloud layer to re-train the model with newer data. This new data is sent to the feature store.

Similar to the model registry, the performance monitor sits between the cloud and edge layer in Fig 3.3, due to the fact that its implementation could be done on either layer.

### *3.5.2. Edge Architecture*

Figure 9 displays the architecture and components running within an edge device of the system. The **edge controller** acts as the central components of the entire platform. This acts as the medium for communication between the rest of the components in the system.

**Sensors and actuators** send and receive data from the edge device using MQTT. As a result, an **MQTT broker** is deployed into the edge device to facilitate this communication. The edge controller subscribes to the topics the sensors send data to, and forwards them to the model for inference.

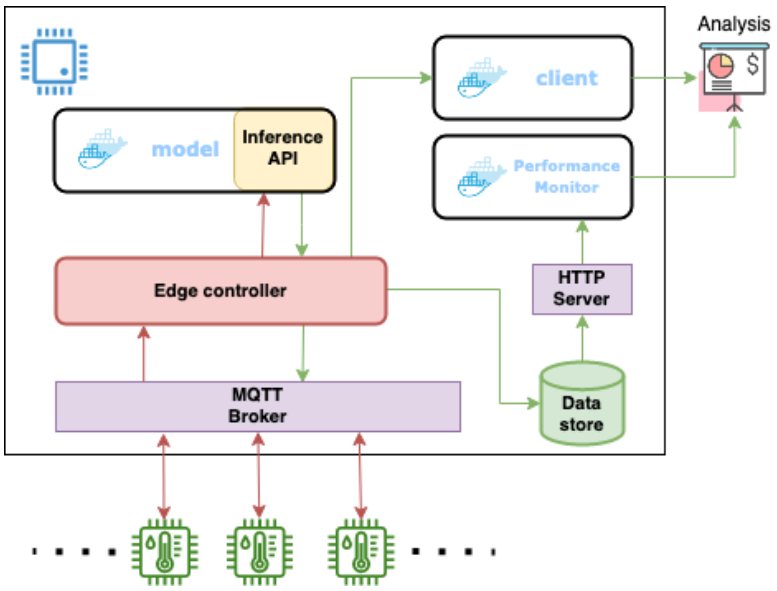


Figure 9. Edge architecture

The **model** is the container image produced by the MLOps pipeline. The image is pulled by the edge controller when the platform is initiated. Then, the edge controller periodically polls the container registry to check if a newer version of the model is available, and pulls it. The model consists of the **inference API**, which takes sensor data from the edge controller, transforms them, feeds into the model for inference, and produces an output.

Once an inference result is obtained, the edge controller performs two actions. The first is that it will store the sensor data, inference results, along with device internal data in the **data store**, which is a set of .json files. This data can be accessed by the **performance monitor** through an **HTTP REST API** server to analyse performance degradation. **Analysis** results can be sent to an external application which may decide if the MLOps pipeline needs to be triggered to re-train the model.

Developers also have the option to define an additional **client** application. This application can obtain inference results from the edge controller. It is a container image that is defined by developers and data scientists. The objective of this is to provide flexibility to perform additional operations according to the requirements of the projects, which are otherwise not possible by the edge platform.

# 4. IMPLEMENTATION

## 4.1. Usage

When designing a platform that enables edge devices to connect to MLOps pipelines, it must consist of tools and services to build infrastructure on edge devices that enable integration with MLOps pipelines. This tool essentially sets up the configurations and components discussed in the edge architecture shown in Figure 9.

An important component cloud layer of the platform architecture (Figure 8) is the CI pipeline responsible for training, and packaging ML models in Docker containers. These packaged models should consist of the Inference API that enables the edge device to run inference on the model. Thus, having a set of tools that assist in the process of packaging models with an inference API that is compatible with the edge device it is being deployed to is essential.

As a result, this platform consists of 2 CLI tools:

1. **Inference toolkit** - packaging models with inference API into a Docker container
2. **Edge manager** - setting up and managing infrastructure components of an edge device

### *4.1.1. Inference Toolkit*

The inference toolkit is a CLI tool whose purpose is to set up the file structure of the inference API shown in Figure 7 to ease the process of packaging a model to be compatible with running inference on edge devices.

The inference toolkit is to be used towards the end of the CI pipeline after the verify phase [5], when the model is ready to be packaged into a container image. It consists of two commands: *init* and *build*.

```
Usage: inference-toolkit <COMMAND>

Commands:
  init  Initialize the model inference handler
  build Package model and inference handler into a
Docker container
  help  Print this message or the help of the given
subcommand(s)

Options:
  -h, --help  Print help
  -V, --version  Print version
```

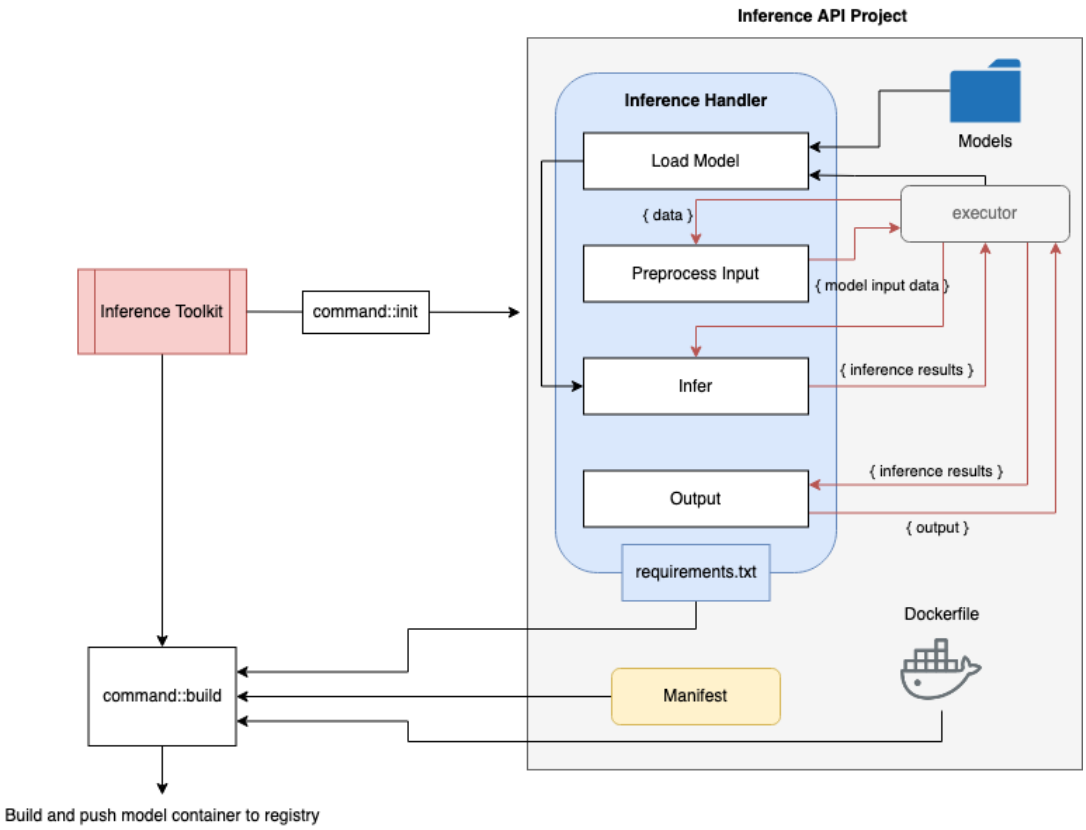


Figure 10. Inference Toolkit

### Init command

The *init* command initialises the project structure of the Inference API Project shown in Figure 10. This produces a set of files and directories in the directory the command is executed in.

This creates the **Inference Handler** files that enable inference, which were discussed in detail in chapter 3.3. The inference handler also comes with a *requirements.txt* file, where the engineer must provide the dependencies that should be installed to run inference.

In addition to the inference handler, the tool also creates a manifest and Dockerfile. The purpose of the manifest file is to follow a declarative abstraction approach similar to the SensiX++ project [11]. It is a JSON file that contains information about the model, and what is required to run inference. It consists of:

- Name of the model
- Version
- Directory and name of the model file
- MQTT topic to expect an input from
- MQTT topic to provide an output of the inference results to

```
{
"name": "scania-aps",
    "version": "0.0.1",
```

```
"mqtt-input-topic": "inference/sample/scania/aps/req",
"mqtt-output-topic": "inference/sample/scania/aps/res",
"model-dir": "./models",
"model-name": "gbdt_model.pkl"
}
```

These values should be modified by the user to match the architecture of the project.

The Dockerfile is a template whose values will be populated during the build command. Although it is not required, users may make modifications to the Dockerfile if the project contains additional complexities.

The usage of the init command of the CLI tool is as follows:

```
Initialize the model inference handler

Usage: inference-toolkit init <PROJECT_NAME>

Arguments:
  <PROJECT_NAME>  Name of the model inference project

Options:
  -h, --help  Print help
  -V, --version  Print version
```

This creates a directory with the provided project name. The value of the "name" parameter in the manifest file is also changed to the project name. The version defaults to "0.0.1".

### Build command

The build command takes input from the requirements and manifest files and modifies the Dockerfile template. Using the Dockerfile, it builds a Docker image which is ready to be deployed into a registry.

The actions taken while running the build command are as follows:

1. Install dependencies for the Docker image from the requirements file.
2. Provide the path to the model file for the executor.
3. Provide the MQTT input and output topics to the executor.
4. Increment the patch version [30] in the manifest file version by 1.
5. Build a docker image "`<name>:<version>`" from the values provided in the manifest.

The command does not push the built image into a registry. This implementation should be done in the CI pipeline by the engineer. The usage of the build command is as follows.

```
Package model and inference handler into a Docker container

Usage: inference-toolkit build <PATH>
```

```
Arguments:
  <PATH>  Path of the inference handler project

Options:
  -h, --help  Print help
  -V, --version  Print version
```

### *4.1.2. Edge Manager*

The edge manager is a CLI tool that creates and manages the MLOps components of an edge device (Figure 9). This tool is more complex than the inference toolkit as it consists of a set of commands and subcommands.

```
Usage: edgeops <COMMAND>

Commands:
    init      Initialize the edge device
    add     Add components to the edge device
         client     Initialize the edge device client
         model     Add additional models to the device
         monitor     Add monitoring to the device
    pull     Pull images manually
         client     Pull client image
         models     Pull model image
         monitor      Pull monitor image
    help     Print this message or the help of the given subcomma

Options:
  -h, --help  Print help information
  -V, --version  Print version information
```

**Init command**

The *init* command sets up all of the components in the edge device that is required for integration with the MLOps pipeline.

```
Initialize the edge device

Usage: edgeops init [OPTIONS]

Options:
   --id <ID>           ID for the edge device
   --host <HOST>       Hostname of the model container registry
   --username <USERNAME> Username of the model container registry
   --passwd <PASSWD>   Password of the model container registry
   --repo <REPO>       Model container repository
  -h, --help              Print help information
```

```
  -V, --version            Print version information
```

When running the init command, the tool prompts for an ID for the device, container registry host, username, and password it can connect to, and the repository for the container consisting of the ML model. Alternatively, the user can provide these values as flags if the process requires automation.

During the execution of the init command, the tool will set up an MQTT broker, directory structure for the data store, HTTP server to fetch files from the data store, and the edge controller service, which runs in the background. Thereafter, it will fetch the model from the registry using the given credentials.

The initialization also creates a configuration, which is a simple JSON file consisting of values to run the edge-MLOps services.

```
{
    "fleet_id":"1",
    "models":[
     {
     "host":"registry.hub.docker.com",
     "username":"savidude",
     "password":"&t5zDJ9muR0xjh$xsYtg",
     "repository":"savidude/scania-aps",
     "topics":{
     "input":"inference/sample/scania/aps/req",
     "output":"inference/sample/scania/aps/res"
     }
     }
    ],
    "client":{
     "host":"registry.hub.docker.com",
     "username":"savidude",
     "password":"&t5zDJ9muR0xjh$xsYtg",
     "repository":"savidude/scania-aps-client"
    },
    "monitor": {
     "type": "local",
     "host":"registry.hub.docker.com",
     "username":"savidude",
     "password":"&t5zDJ9muR0xjh$xsYtg",
     "repository":"savidude/scania-aps-monitor",
     "schedule":"*/5 * * * *"
    },
    "model_pull_schedule":"*/5 * * * *"
}
```

### Add command

The *add* command is used to add components to the edge device. These components are defined by their individual subcommands: client, monitor, and model. Similar to

the init command, these commands prompt for the container registry and repository details (which could also be given as flag values for automation), with the exception of the monitor, which additionally requests for the type (local or remote), and a schedule for which it should run.

```
Add monitoring to the device

Usage: edgeops add monitor [OPTIONS]

Options:
   --type <TYPE>       Monitor type: local or remote
   --host <HOST>       Hostname of the client container registry
   --username <USERNAME> Username of the client container registr
   --passwd <PASSWD>   Password of the client container registry
   --repo <REPO>       Client container repository
   --schedule <SCHEDULE> Monitor execution Cron schedule
  -h, --help              Print help information
  -V, --version           Print version information
```

**Pull command**

Similar to the add command, pull also consists of 3 subcommands for the models, client, and monitor containers. These *do not* require any prompts as all component information is fetched from the configuration file. If there are multiple models in a single device, it will attempt to pull all models from their respective registries.

## 4.2. Languages and Tools

As discussed in the design and architecture chapter, in order to develop a platform that runs on edge devices which enable the integration with MLOps pipelines, the best option is to use a CLI tool.

When selecting a tool to develop CLI applications, there are a few things that must be considered:

- The platform being developed must be capable of running on edge devices with limited resources.
- Must be capable of working with containers (Docker).
- Consist of libraries that can work with various communication protocols (MQTT, HTTP).
- Consist of tools and libraries capable of monitoring system components (CPU, memory)

Some possible options for programming languages that could be used are:

- C: a low-level programming language that is highly efficient and has a small memory footprint. It is commonly used for embedded systems and other low-level applications.

- Go: a relatively new programming language designed with efficiency in mind. It has a small memory footprint and provides built-in support for concurrency.
- Rust: a systems programming language designed for performance and reliability. It has a small memory footprint and provides built-in memory safety features.
- Consist of tools and libraries capable of monitoring system components Lua: a lightweight scripting language that is often used for embedded systems and other low-power applications. It has a small memory footprint and is highly customizable.

All of these languages are capable of using common communication protocols and can perform simple system monitoring operations.

Lua is geared towards low resource-intensive such as 32-bit microcontrollers. As a result, it does not contain native support to work with Docker images, although there are some third party tools, they are not maintained regularly and have little community support.

C would be an ideal choice for developing applications for low-powered devices. However, the libraries that are available are not user-friendly and easy to use as the alternatives.

Go and Rust are ideal languages of choice for this work. They are commonly used to develop CLI applications. Docker has developed an official SDK for Go [31], which enables working with the Docker API. Rust consists of reliable community-driven tools that enable working with the Docker API.

However, when developing applications for embedded devices, it is an ideal tool for systems programming [32]. Some reasons as to why Rust is ideal to be used with embedded systems include the option to have dynamic memory allocation, interoperability with C applications, portability with a variety of systems including small microcontrollers, and it is strongly community driven. Thus, it is clear that Rust would be an ideal choice to develop a platform to integrate MLOps with edge devices.

### *4.2.1. Creating a Command Line Interface*

The CLI component of the tool was developed using the Clap crate [33] (Rust package). Apart from being able to create applications with common argument behaviour, it generates help prompts automatically, does suggested fixes for users, shell completions, etc. The tool also comes with a feature called "derive" [34], which allows the creation of subcommands.

### *4.2.2. Creating Edge Device Components*

A large part of the edge architecture for MLOps consists of containers. Naturally, this requires that the edge device is capable of running containers. Information about the implementation behind pulling and running containers is described in chapter 4.2.2.

The edge device also consists of communication servers. The project uses the Rust Hyper [35] crate to create a simple HTTP server. The server manages REST API endpoints for the retrieval of performance monitoring data, discussed in chapter 3.4.2.

Most of the communication within the device goes through an MQTT broker. Existing Rust libraries do not have adequate support to run MQTT brokers within edge devices. As a result, the platform uses the Eclipse Mosquitto [36] Docker image to run a broker.

## 4.3. Continuous Delivery

When considering the implementation of the CD design, two factors need to be considered.

1. Implementation of the Inference API (inference toolkit)
2. Fetching and running packaged models on the edge

### *4.3.1. Inference API*

The inference API enables edge devices to be able to run inference on the model by providing an input and obtaining an output in an expected format. The inference toolkit discussed in chapter 4.1.1 consists of two functionalities: initialising the inference API project, and packaging the model into a Docker container.

When initialising the project, it gets the version of the project from its Cargo package version. Thereafter, this version is compared with the existing releases in the GitHub repository, to identify the project template compatible with the current version of the tool. Thereafter, it downloads the project template in the form of a .zip file, and unpackages it in the directory provided by the user.

This template contains all of the functions and configurations of the model the user must modify. The user must implement the functions of the Inference Handler discussed in chapter 3.3 to fit the input requirements of the model, and output requirements of any client applications.

In addition to the Inference Handler, the user should also modify the manifest file in the project, and specify an MQTT topic to expect an input to, and a topic to produce an output to. The toolkit then configures the executor to subscribe to the input topic, and publish to the output topic.

Once the Inference handler has been implemented, the build command (chapter 4.1.1) can be used to package the model, and interface it with the Inference API such that inference such that inference can be performed on the incoming sensor data. It uses information from the manifest file to create a docker container with the given name, and tag it with the version provided in the manifest. If the manifest version has not changed, it will increment its patch version [30] by 1 and build the image. Thereafter, the user must push the image into the required registry.

### *4.3.2. Fetching and Running Models on the Edge*

Containers are pulled and maintained using the Rust Shiplift crate [37]. Shiplift is a community-driven Rust package that uses the Docker API, that is capable of

performing the same actions as a Docker client. This package was used to pull images from a registry using the given credentials, as well as run containers with specified arguments.

## 4.4. Inference

As indicated in the Inference architecture in Figure 10, the inference toolkit provides a process template for inference. The user follows the steps guided by the template and implements the code such that it will use the packaged ML model and run inference on incoming data.

As described in the Inference design in chapter 3.3, the user must implement the functionality for 4 inference steps: model loading, preprocessing, inference, and output. The template for this functionality is provided by the inference toolkit upon initialization. This is made available in the Python programming language. For any future work of this thesis, the cool could be modified to include other programming languages for ML inference.

Following are the Python functions to be implemented by the user, where the arguments indicate the resources required for each inference step:

1. `def load_model(models_dir, model_name)`: Loads a model. This function should fetch the packaged model file and save into memory as an object of the trained model.
2. `def preprocess_input(input_data, models_dir)`: Handle structured input of any format and restructure into a format that fits the model.
3. `def infer(model, input_data)`: Run inference on restructured input data.
4. `def output(inference_result)`: Convert inference results into a format that can be sent for analysis (e.g. JSON)

Once the packaged model is deployed into an edge device, the inference process is handled by the executor as shown in Figure 11. The executor first subscribes to the MQTT input topic which it expects to receive sensor data from. Then it executes the load model function and keeps the model object in memory, such that it would not have to fetch the model each time a sensor input is received.

Thereafter, it executes the preprocess input, infer, and output functions respectively, and forwards the output results back to the edge controller. This was the initial implementation of the executor. However, a few flaws of this approach were identified during evaluation.

1. The controller does not keep any record of the input and output topics of a model, which would cause issues in the future when auditing.
2. If multiple sensor readings are inferred successively, there is no mechanism to map an inference output to a sensor input.
3. Erroneous mappings of sensor input to inference outputs lead to issues during performance monitoring.

To address these issues, a handshake mechanism was introduced between the inference API and edge controller, as indicated in Figure 12.

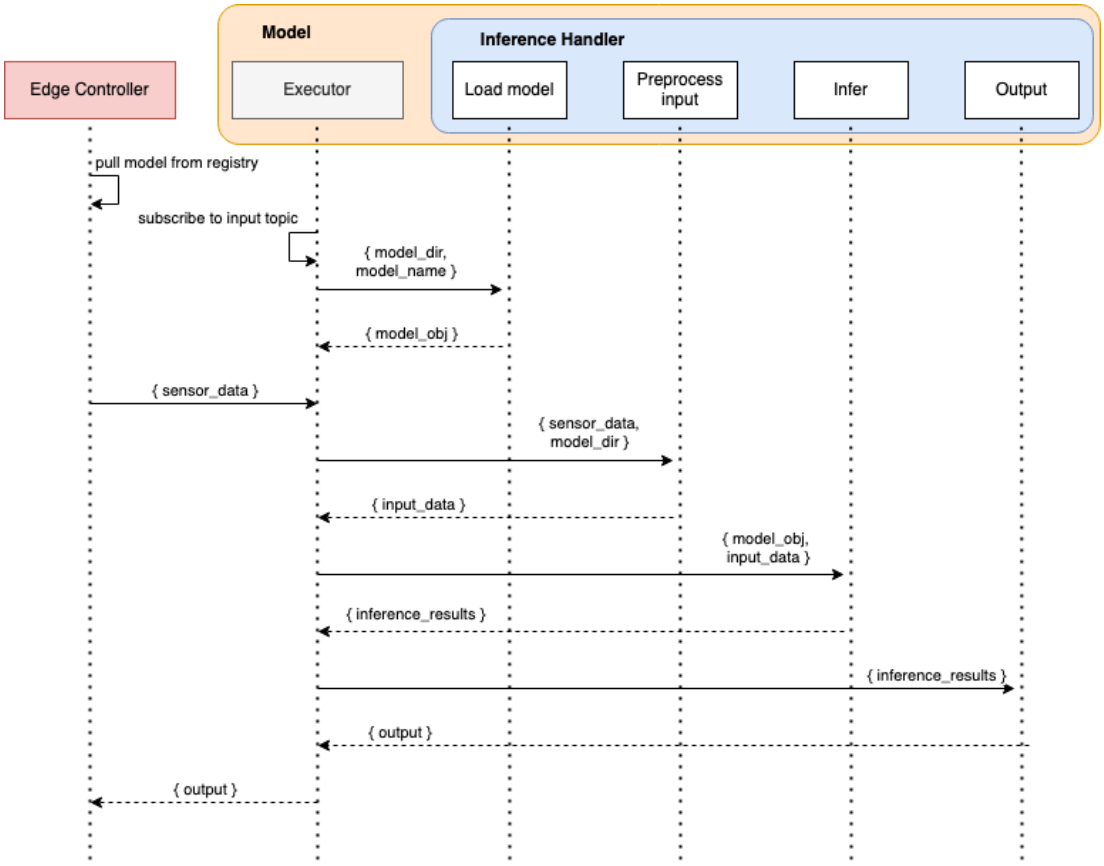


Figure 11. Executor process

This handshake is initiated when the ML model is pulled from the registry. Both the inference API and edge controller have predefined hardcoded topics to which they expect to receive handshake information to.

- The controller subscribes to a **handshake topic**.
- The inference API subscribes to a topic it expects to receive an acknowledgement (acknowledgement topic) from the controller.

When a model image is pulled from a repository and started by the executor:

1. The **executor** sends the input and output topic names to the **handshake topic**.
2. The **controller** receives this information and generates an **internal set of request and response topics** that is only known between it and the inference API. These topics are sent to the **acknowledgement topic** to be received by the inference API.
3. Then the **executor** subscribes to the **internal request** topic.
4. The **controller** subscribes to the **internal response** topic.

This handshake process allows the controller to keep track of the sensor data being received, where the data is being sent to, and the output produced for each input. This will be useful for monitoring and auditing purposes.

The controller is able to keep track of incoming data and their corresponding inference results by assigning a request ID to incoming data before being sent to the

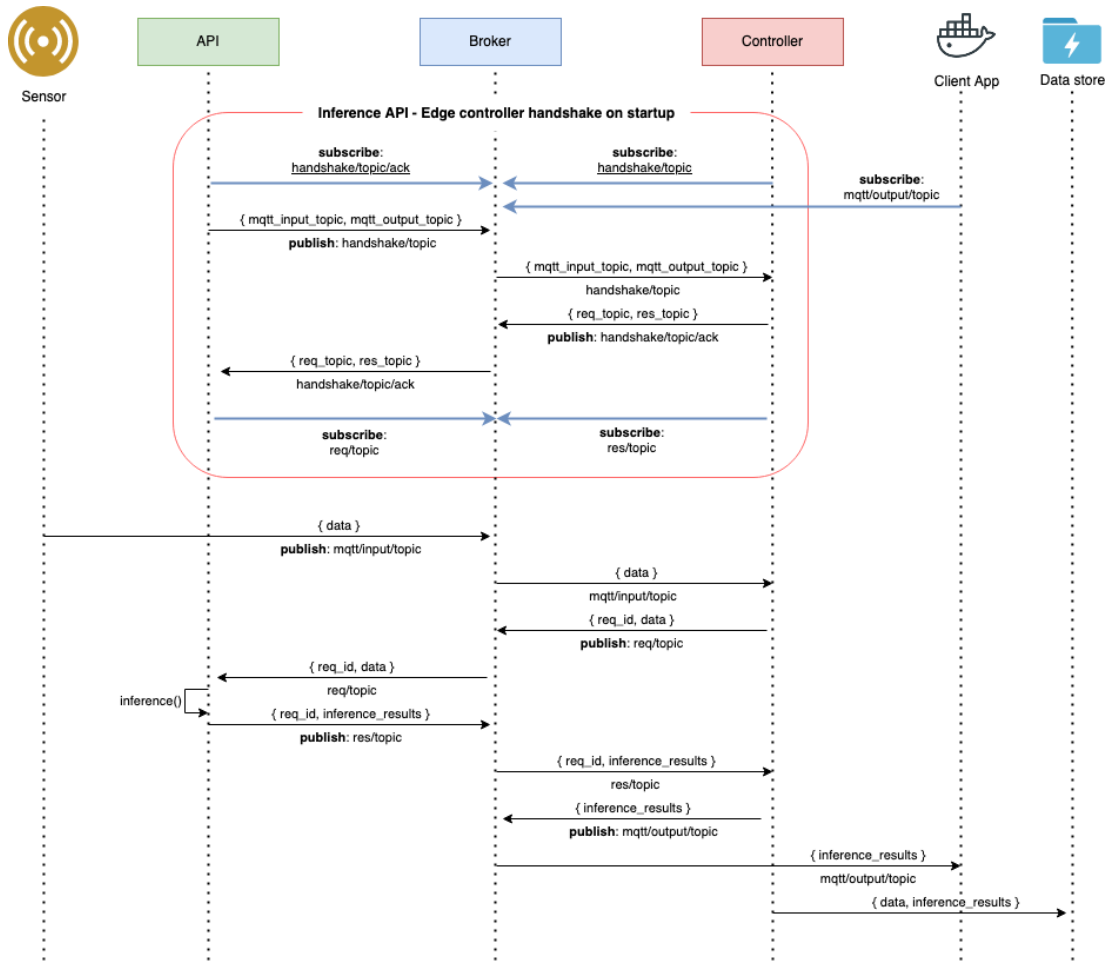


Figure 12. Inference API Handshake

inference API. Once an output is obtained from inference, the API appends the request ID to the response when providing inference results back to the controller. This enables the controller to keep track of sensor data and inference results. The controller then adds this information to the data store, which will be used by the performance monitor in the future.

### 4.5. Performance Monitoring

Implementation of performance monitoring applications in MLOps is use-case specific [38][39]. As a result, the platform provides the necessary tools and services needed to run custom performance monitoring implementations. The main tools offered by the platform are scheduling and a REST API to fetch system and inference data.

### *4.5.1. Scheduling*

Since performance monitoring is run periodically, when initiating the monitor, the user must specify a schedule to which monitoring must be performed. As discussed in the design chapter this schedule is provided in the form of a cron schedule.

If the monitoring is done locally in a Docker container, a new entry is added to the system crontab file to run the image. For example, if the container is to be executed every hour at 15 minutes, the crontab entry would be as follows: `15 * * * * docker run scania-monitor`

### *4.5.2. Rest API*

Apart from being able to be used for CLI application development, Rust is also a widely-used server-side programming language. It is used to create REST servers consisting of microservices.

As discussed chapter 3.4.2, the server has 4 microservices for retrieving and deleting performance stats. These were implemented using the Rust Hyper [35] crate, which is used to create simple HTTP servers. The created microservices does very simple functionalities like retrieving the files in the data store by timestamp, packaging them into zip files, and serving the zip file as a response to the requests.

# 5. EVALUATION

The platform evaluation is conducted in two stages. The first stage was coming up with a use-case scenario and developing a simulation environment to match the use case. Then, the performance of the platform was measured by altering various variables of the simulation environment.

## 5.1. Validation Scenario

The objective of this chapter is to come up with a scenario in which this platform could be used, and to develop a solution for the scenario. To do so, we must first identify a dataset that fits a valid use case, develop a model using the dataset, set up an MLOps pipeline that trains and packages the model, and then implement the edge-level components that utilise the model for inference and analyses its performance.

### *5.1.1. Selecting a Dataset*

The first step in developing a scenario for the platform to be used is to identify a suitable dataset that consists of data for a valid use-case for running ML applications on the edge.

Carvalho et al. [40] present a systematic literature review of ML methods applied to predictive maintenance applications. Predictive maintenance in itself is a very valid use-case for ML applications at the edge, as anomalies must be identified as quickly as possible. When selecting a dataset for a validation scenario for running ML inference on the edge, the following factors were considered:

- Anomalies must be identified immediately.
- The operating environment must consist of hardware resources capable of running ML inference.
- Running inference on the cloud is not a viable option.

In [41], the authors present a dataset aimed to detect component failures in an air pressure system (APS) of trucks. The dataset consists of 76,000 samples, each containing 171 attributes.

This dataset presents an ideal case to develop a validation scenario with as:

- Air pressure system failures must be identified immediately as failure to do so may result in serious damage.
- Computer systems in modern trucks come equipped with hardware capable of running ML inference.
- Trucks operating on the road may not always have internet access. Thus, ML inference cannot be performed in the cloud.

Therefore, it was decided to develop a scenario using this dataset. The dataset consists of data collected from Scania trucks in everyday usage. The positive class represents component failures for a specific component of the APS. The negative class represents component failures not related to the APS.

The names of the 171 attributes have been anonymised. However, this would not affect the sample scenario being developed as the attribute names are not relevant for the application. The attributes consist of single numeral counters and histograms consisting of bins with different conditions. The missing values are denoted by "na".

### *5.1.2. Prediction Model*

Once a suitable dataset has been identified, the next step is to develop a prediction model using entries from this dataset. There are two sets of files in this dataset:

1. Training set with 60,000 examples: 59,000 belonging to the negative class, 1,000 belonging to the positive class.
2. Test set with 16,000 examples: 15,625 belong to negative class and 375 to positive class.

A prediction model was developed for this dataset using the steps followed in [42]. The implementation first identifies a set of features that has the least impact on the outcome. These features are removed during the preprocessing stage.

Thereafter, the implementation takes the missing values ("na") into consideration. The authors have identified that the best approach to address missing values is to perform imputation. As a result, median and MICE imputation is performed on data with missing values.

The author trains the data with various ML algorithms in order to find out the model with the best performance. The model performance was calculated using the accuracy and F1 score. All of the trained models were developed using the scikit-learn Python package. The models that will be trained and evaluated in the pipeline are Logistic Regression, Linear SVM, Decision Tree, Random Forest, XGBoost, Adaboost, and Custom Ensemble. The objective of the pipeline is to select the best performing model to be deployed on the edge devices.

### *5.1.3. Setup*

#### MLOps pipeline

We utilise AWS Sagemaker Pipelines [43] for the MLOps pipeline. Sagemaker Pipelines provides their own set of tools and services to create, automate, and manage end-to-end ML workflows at scale. The concept of Sagemaker pipelines is to define a set of steps in a workflow that suits the project.

The workflow steps are defined by the MLOps engineer. The steps are defined and developed in Python, and can make use of different AWS services if required. The pipeline uses AWS S3 (cloud file storage) to retrieve data and train models. The pipeline steps that were defined for this project were:

- **Preprocess**: fetches data from storage, feature removal and imputation of missing values
- **Model training**: trains a set of models using the preprocessed data and trains a set of models using scikit-learn.

- **Evaluation**: evaluates the performances of all the trained models and picks the best one for deployment.
- **Registration**: copy the model into the Inference API project models directory, and build a Docker image using the inference-toolkit CLI. This image is then pushed to the Docker repository (DockerHub).

#### Hardware and simulation environment

After setting up the MLOps pipeline for model training and registration, the next step is to set up model deployment and testing at the edge, along with the other infrastructure set up by the edge manager. The validation scenario was tested on a:

- RaspberryPi 4 Model B
  - 8GB RAM
  - 64-bit ARM v8 SoC 1.8GHz
- RaspberryPi OS (based on Debian)
- Docker for containerization

The edge-MLOps environment is initialised using the Edge Manager tool, along with the container images for the model and performance monitor.

Although the dataset consists of 60,000 training examples, only 40,000 were used for the training of the initial set of models, and the rest were used to run inference on the edge device. The training examples that were not used initially would eventually be used when the performance monitor triggers new pipeline builds in AWS Sagemaker. In a real world scenario, inference would be performed on an actual truck in real-time. Apart from having a lack of resources to conduct such experiments, APS failures in real life are extremely rare, and it would not be possible to conduct an experiment for a long time given the time constraints of this thesis project.

Therefore, as an alternative, a simulation environment was developed that uses the training data to produce artificial sensor data at given time intervals such that inference can be performed.

The simulation is a simple Python script that reads the training data, converts each entry into a JSON format, and sends the JSON data into the input topic. The simulation script lets the user define the sensor data production interval in the form of a command line argument. This supports the next step when load testing is performed on the system.

### *5.1.4. Implementation*

#### Model and Inference API

Once a model is trained by the ML pipeline, it packages it into a binary using Pickle [23] and is put into the models directory of the Inference API project. In addition to the model, the imputation functions (median and MICE) are also packaged and put into the models directory, as the imputation functions will be used during preprocessing.

The `load_model` function looks for the packaged model file in the given directory with the provided file name for the packaged model. This model is loaded into memory using the `pickle.load()` function, and the loaded model object is returned.

The `preprocess_input` function gets the structured input data that is to be obtained from the set of sensors in the simulator. Then it removes the features that are not required for inference. In the next step, it takes the list of features that imputation is meant to be performed on, and performs median and MICE imputation on those features using the previously packaged imputer functions. The model requires data to be structured in a Pandas Dataframe format. Therefore, the function then converts the preprocessed data into a Dataframe and returns the Dataframe object.

The infer function simply takes the model obtained from the `load_model` function, and the preprocessed input obtained from the `preprocess_input` function, and runs inference. Inference is run on the input data by calling the `model.predict()` function, which returns inference results in the form of a Pandas Dataframe.

Finally, the `output` function takes the Dataframe output returned by the model inference and converts it into a structured output. In this case, it returns a JSON string with a key called "result", which either contains the value "safe" or "failure".

In addition to the inference handler implementation, the requirements.txt file is modified to include use-case specific Python dependencies such as pickle4, pandas, and scikit-learn.

The manifest file is changed such that the input topic is assigned the value "inference/sample/scania/aps/req", and the output topic is assigned "inference/sample/scania/aps/res". The project is given the name "scania-aps", which is what the model image will be called after being built.

### Performance Monitoring

The performance monitoring implementation is done locally, which means that a dedicated container is created for monitoring purposes. For the purposes of this simulation experiment, the scheduler is set to run monitoring once every 15 minutes.

The performance monitor implementation makes a request to the `GET ./performance/:start_timestamp/:end_timestamp` microservice to fetch inference data in the last 15 minutes. This data is obtained in the form of a .zip file consisting of a set of JSON files. The monitoring application iterates over the JSON files, and analyses the prediction with the actual result.

For performance morniting, prediction errors were evaluated with a loss function comparing the prediction with the actual result. MSE (mean squared error) was used as the loss function to evaluate the prediction error. If this value was identified to be below 0.6, the pipeline was triggered to run a new model build.

If a model build is to be triggered, the application fists downloads all of the existing inference data in the edge device using the `GET ./performance` microservice, and uploads them to a prespecified AWS S3 bucket. Uploads to this bucket triggers the Sagemaker pipeline build, where the existing models are further trained with the new data in the bucket. Once the data is successfully uploaded to S3, the monitor deletes all existing performance data using the `DELETE ./performance` service.

## 5.2. Testing

The purpose of developing a validation scenario for the platform is to be able to evaluate and test its performance. In addition to the performance of the platform, the performance of the edge device must also be tested to see the extent to which it can support the platform.

Finally, we compare the platform that was developed with existing tools and services that were discussed in chapter 2.2. This helps understand the strengths and weaknesses of the platform and what areas it could be useful in.

### *5.2.1. Load Testing*

Load testing is done to see how an application or system performs under a load beyond its anticipated capacity or peak usage levels. By doing so, we can identify potential vulnerabilities, performance bottlenecks, or potential crashes. This platform has two components where load testing can be performed in: the *performance monitor* and *inference APIs*.

The load test is conducted by measuring the response time of an API against the rate at which requests are sent. The request rate is controlled by the simulation script written in 5.1.3.

#### Inference API

Load testing was performed on the Inference API, response time was calculated between the amount of time it took to send sensor data from the simulation script, to when a response was observed. The response time was tested against the rate at which requests were sent.

The same validation scenario was developed for AWS Greengrass, which is a commercial tool capable of serving similar purposes to our solution. The performance of this solution was compared against Greengrass in order to justify that the performance is comparable to industry standards. The Greengrass implementation and test was conducted on a separate Raspberry Pi with exact specifications as the one running our solution.

The simulation script controls the request rate, and gets the time taken to observe an output from 100 random samples of responses, and calculates an average value. These values were recorded separately in an Excel worksheet, and plotted as shown in Figure 13.

A similar trend could be observed in our solution as well as AWS Greengrass, although Greengrass performed slightly better in comparison. This can be attributed to the fact that the Greengrass application does not run in a containerised environment. Although using Greengrass provides slightly better performance, it comes at the cost of having to set up OS and application specific libraries on each device individually. Additionally, running inference in containerised environments has the benefit of having the flexibility to work with device-specific models more easily.

The response time for a request increases along with its load. At around 150 requests per second, we begin to see a dramatic increase in this response time. The response

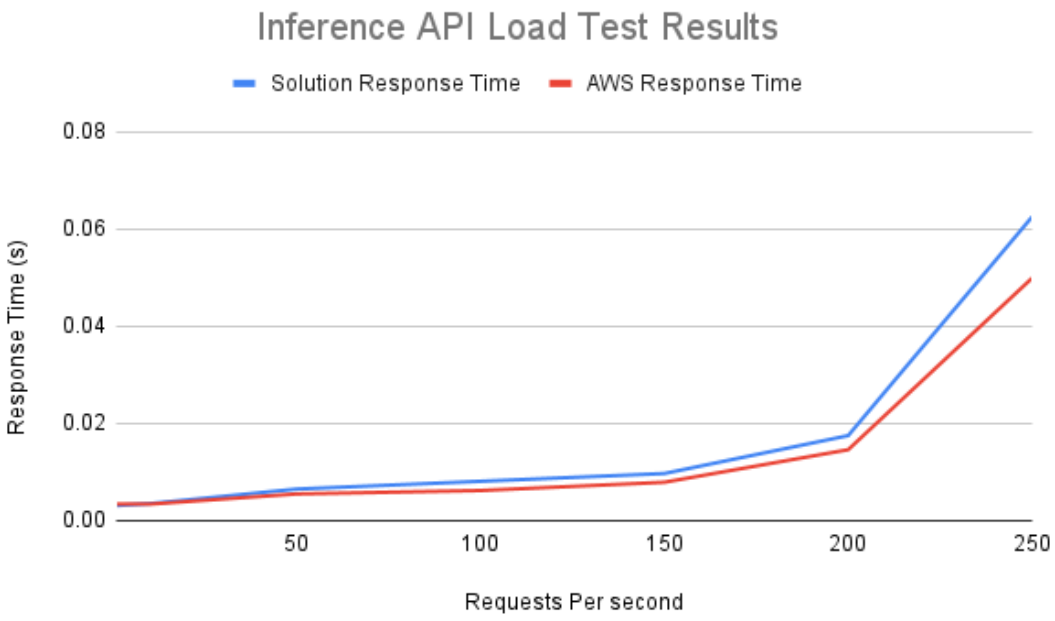


Figure 13. Inference API Load test Results

time at 150 is 0.0097 seconds. In which case, the API is receiving sensor data at a rate of 0.0067 seconds per request, and it is unable to process them at the same rate. As a result, the response times get delayed further until the inference API application eventually runs out of memory.

Both our solution and Greengrass reach a critical point at around 150 requests per second. Although Greengrass appears to have increasingly better performance after 150, it does not add much value as the device is running at a rate where it is already unable to handle the request load, and thus would eventually run out of memory in both cases.

Therefore, we can conclude that a single edge device running on the platform can handle request loads up to 150 requests per second, which is sufficient in most cases that require ML inference at the edge.

### Performance Monitoring API

Load testing the performance monitoring API introduces a complex task as there are many variables that could be adjusted. There are 2 retrieval APIs and 2 deletion APIs. It would not be necessary to load test the deletion APIs as real-life use cases would not make such requests at a large volume.

The retrieval API queries, packages, and returns data from the data store in the form of a zip file. The response time would depend on the size of this file. This load test was conducted when the size of the data store was 121 kB, which records 800 inference instances.

Similar to the Inference API, the performance monitor also reaches a critical point at which it is unable to serve requests at the same rate as it is receiving. The value of this critical point is about 10 requests per second.

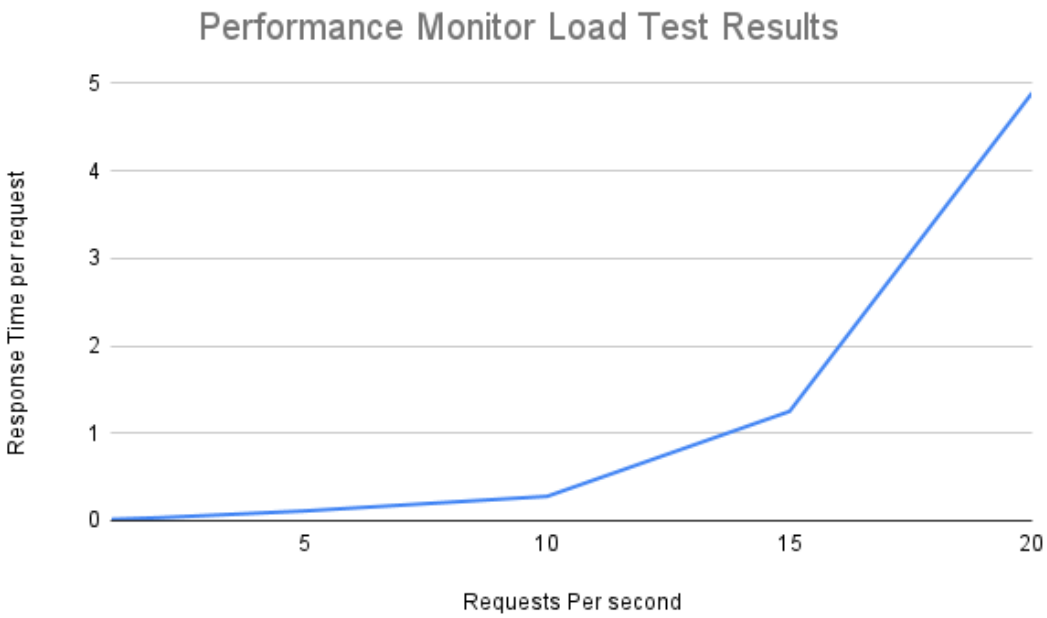


Figure 14. Performance Monitor Load Test Results

In real-world practice, performance monitoring is not done at quick intervals. As a result, being able to serve up to 10 requests per second would not cause any issues to edge systems using this platform.

### *5.2.2. CPU Usage*

The CPU usage was examined by measuring the amount of processing used in the device (by percentage) against time. This experiment was conducted during inference. The usage was measured for different rates at which sensor data was produced for inference.

Figure 15 shows the amount of processing that was used in the edge device during inference. The observations match with the Inference API stress test, where the system begins to reach a critical point at around 150 requests per second.

At 150 req/sec, the usage drastically increases over time, as the system becomes unable to keep up with the load. At 200 req/sec, we were not able to obtain the usage results after 180 seconds, as the application crashes a few seconds after.

The CPU usage was also measured on the AWS Greengrass device to see how well it performs in comparison. Since the Greengrass implementation does not have system performance monitoring by default, the CPU usage was obtained using htop [44], while the timing was done manually.

Figure 16 shows a comparison of the CPU usage between the solution and AWS greengrass when the system receives requests above the critical rate of 150 requests per second. It can be seen that Greengrass performs better in terms of CPU utilisation. However, it eventually reaches full utilisation some time after our solution does.

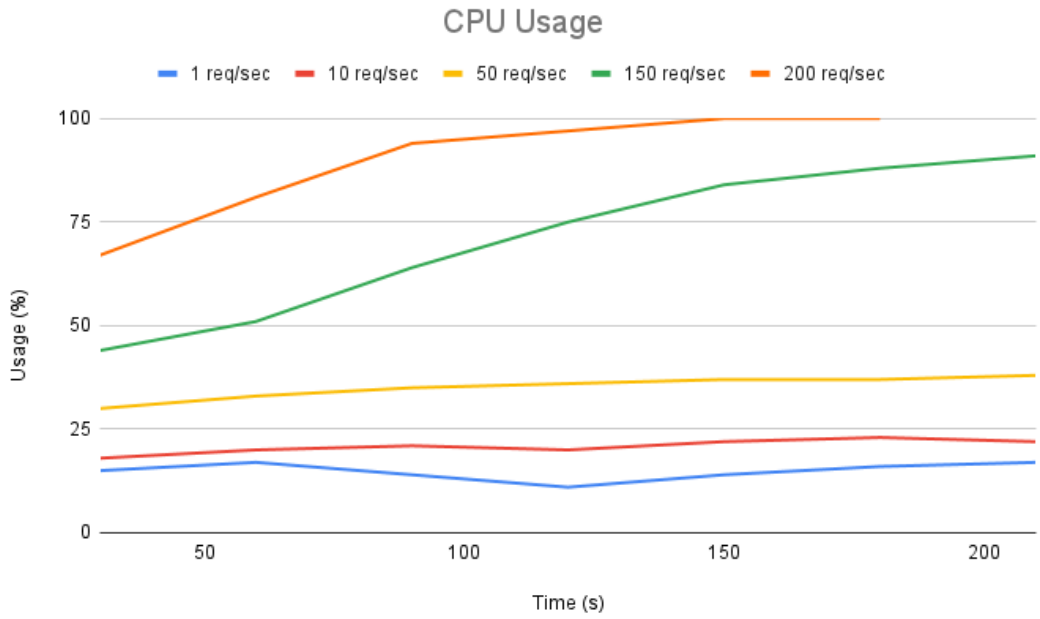


Figure 15. CPU usage

### *5.2.3. Comparison with Existing Tools and Services*

This subchapter evaluates the platform against current tools and services that are also capable of integrating MLOps with edge devices.

The most popular set of tools for the integration of MLOps with Edge in the current market is AWS IoT Edge and Greengrass. One of the major benefits of this toolset in comparison to our platform is the access to Amazon's suite of over 200 services. This also comes as a major downside as the tools are entirely cloud based and dependent on AWS services, which means that adopters of these tools would become vendor locked. Greengrass does not have functionalities out-of-the-box for model performance monitoring, however, performance monitoring applications can be developed and deployed to the edge using Greengrass. In contrast, AWS offers some of the best tools in industry for device monitoring, including device health monitoring.

Azure IoT Edge is another popular cloud platform that can be used. Similar to AWS, this too is cloud-based. However, it does not rely on Azure services, although they can be used with the tools provided. ML applications can be run as containerised modules on edge devices. This has additional MLOps features such as model versioning, and monitoring. However, monitoring cannot be performed on the edge. The process of running inference from sensor data must be defined by the user. Similar to AWS, this too has a good set of device management capabilities.

Platform IO is an Open Source tool that is mostly used to build applications for microcontrollers and single-board computers. It supports ML libraries, although ML application development is not its primary use-case. The platform does not come with monitoring functionalities out-of-the-box, but they can be implemented by the developer. Model integration with MLOps pipelines must be performed manually.

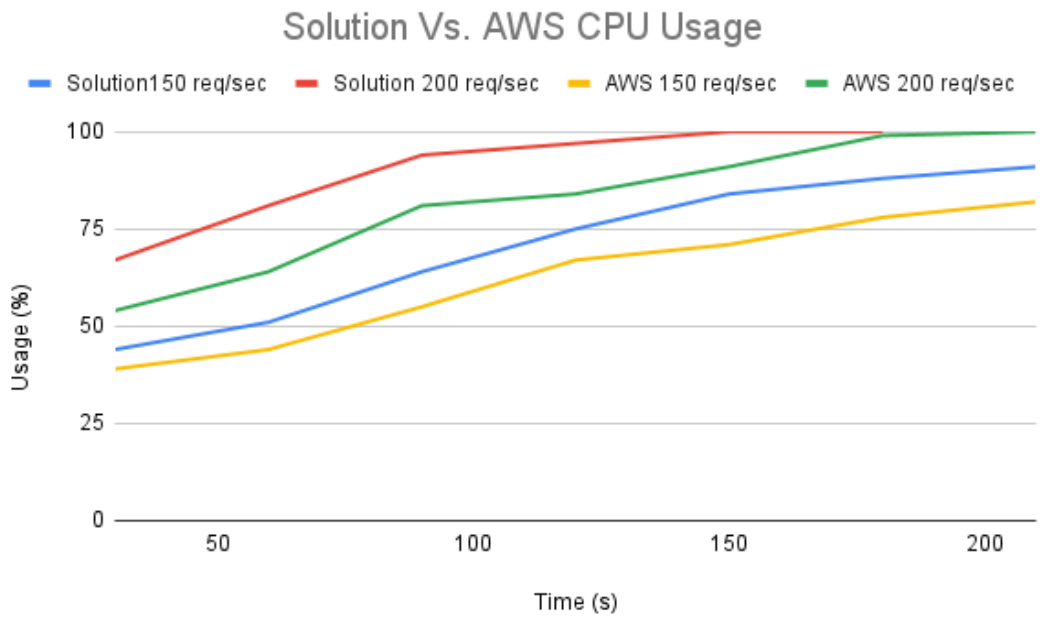


Figure 16. Solution Vs. AWS CPU Usage

Eclipse IoT Edge is another open-source platform that can be used to develop applications on edge devices. It enables the deployment, monitoring, and management of ML models at the edge. Its MLOps capabilities include versioning and model deployment. Model and system monitoring must be user-managed. It supports running on many Operating systems including Linux, Windows, etc.

Table 1 compares the the current platform "Edgeops" with existing tools and services. This helps get an understanding of the type of market the platform would be suitable for. From the analysis, we can conclude that the tool is best suited for organisations that do not want to run or are partially running their applications on the cloud, having edge devices with Linux Operating Systems, require containerized models and applications that can be executed anywhere, and require model and system monitoring capabilities out-of-the-box.

Table 1. Comparison with existing tools

| | Licensing | OS | ML Model | MLOps Features |
|---|---|---|---|---|
| AWS IoT and Greengrass | Proprietary/ cloud-based | Linux | Greengrass module | Device monitoring |
| Azure IoT Edge | Proprietary/ cloud-based | Linux | Container | Model versioning, monitoring |
| PlatformIO | Open-source | Single-board | Executable | User-defined |
| Eclipse IoT | Open-source | Linux, Windows | Executable | Versioning, deployment |
| Our solution | Open-source | Linux | Container | Versioning, monitoring, device management |

# 6. DISCUSSION

There were many reasons that led me to take up this topic for my thesis. Edge computing, ML, MLOps alone are very common topics in which a great deal of research is being conducted. With the rapid advancements in technology, ML on the edge is becoming adopted increasingly [45]. However, methodologies of running ML at the edge is still in its infancy.

Having a platform capable of the integration of MLOps pipelines with edge devices, and a set of tools to accompany this would be of great value in the future as use cases for running ML inference at the edge becomes increasingly common. As a result, I believe that such a tool would have a great market value in the future, which in turn became my main reason for selecting this topic.

During the initial phase of the project, I did not have a clear idea about the concepts that must be studied in order to give value to this topic. Therefore, a literature review was conducted to get an understanding of edge computing, MLOps, as well as to figure out the best approach that could be followed to implement the proposed platform. During the literature review, I came to the understanding that the main components that should be taken into consideration in such a platform are CI, CD, and CT. However, since this platform is specifically catered to the edge, it was realised that the CI implementation is not as important as the others.

Selecting a language to implement the platform was a difficult choice. The two main options were GoLang and Rust. I had experience working with GoLang, as I had previously developed CLI tools for large-scale software tools. However, it was clear that Rust was the ideal language for this project during the analysis. As a result, I had to spend a considerable amount of time learning Rust.

The most challenging part of this project was the design and implementation of the inference handler/ inference API. The plan at the start of the project was to introduce this as a simple Python file. However, as the project progressed, it became evident that running inference is a much more complex task. As a result, the design for the inference handler was made over many weeks with the guidance of many industry experts. In the end, it was agreed that inference must be conducted in 4 steps: model loading, data preprocessing, inference, output formatting. In order to make this process simpler for end users of the platform, it was decided to introduce an additional tool, the "inference toolkit" for this platform. CT was implemented in the form of the performance monitor. I was able to get an idea of what was required to be implemented for performance monitoring during the literature review.

The final part of the project was coming up with a scenario to test the platform. To do so, we required a dataset from a use-case that is relevant to the project. Data from Solita cases could not be used for this as the data contained confidential information. As a result, we had to find a public dataset and make a simulator for testing. Thereafter, the process of testing the scenario was simple and straightforward.

Considering that this is a thesis project which was done in a limited timeframe, there are some areas that could be further improved if this platform was to be used in real world scenarios. One of the main areas of improvement is security. When models are being delivered to edge devices, there must be a certification process for the pipelines to ensure that the model is delivered to its intended device, and for devices to ensure that the model being received is coming from a trusted source. Additionally, secure

communication methods must be implemented in the MQTT broker and HTTP server running inside the edge devices.

Another limitation of the platform in comparison to its competitors is that it can only work with structured data. That is, the platform cannot run inference tasks related to computer vision and audio on the edge. It can only process data that could be sent over MQTT. This could be further improved to support protocols RTSP and websockets in order to process structured data. A significant amount of modern data processing and ML inference is done in batches. Another downside of the platform is that the inference does not allow batch processing out of the box. Although the functions in the inference handler can be adjusted by developers to accommodate batch processing, this might require some additional complex implementations. ML inference can be done in many programming languages. Although Python is the most common language, there are widely used languages such as R, Java, C++, Matlab, etc. In the future, inference handlers could be developed for different languages.

During the validation scenario, it was observed that the same model deployed using AWS Greengrass performed better than our solution. Although Greengrass has the added benefit that it does not run in a containerized environment, there are areas of improvement in the scenario as well as the implementation of the inference API container. The Inference API could be modified to have the imputation functions permanently loaded into memory during startup, and not have to do so each time when sensor data is received.

The research question addressed by this project was "*How can a platform that enables the integration of MLOps practices with Edge devices be developed using state of the art tools and methods?*". Using information gathered from the literature study, we defined an objective to *research, design, and develop a platform that enables seamless integration of MLOps practices with edge devices, while conducting an in-depth study of scientific literature and discussion from their findings*. We managed to achieve this objective with exceeding expectations as we were able to conduct a through study of the field with existing literature, identify knowledge gaps, and develop a solution that addresses the identified knowledge gaps. The developed solution was evaluated to have a performance that matches that of existing commercial and open-source tools and services, while allowing end users to set up Edge-MLOps infrastructure with flexibility without relying on third party software.

# 7. CONCLUSION

The aim of this project is to research, design, and develop a platform that enables seamless integration of MLOps practices with edge devices. This was achieved through the introduction of an integration platform that runs on edge devices, connecting IoT with MLOps pipelines.

The thesis begins by conducting an analysis of related work done on the space of MLOps with edge devices. This identifies various designs and architectures suited for different purposes. Based on the information obtained from the related works study, a design and architecture was created.

Afterwards, the project was implemented following the design. The implemented project was evaluated by creating a scenario in which the platform could be used. This scenario was created using a sample dataset for truck APS systems, along with a simulation script that facilitated a controlled flow of data. The performance of the platform was then tested using the simulation script.

Finally we ended the thesis through a discussion in which I discussed my own thoughts on the topic, my thoughts on the proceedings of the project, and how it could be improved in the future.

# 8. REFERENCES


[1] Murshed M.S., Murphy C., Hou D., Khan N., Ananthanarayanan G. & Hussain F. (2021) Machine learning at the network edge: A survey. ACM Computing Surveys (CSUR) 54, pp. 1–37.

[2] Chen Y.K., Wu A.Y., Bayoumi M.A. & Koushanfar F. (2013) Editorial low-power, intelligent, and secure solutions for realization of internet of things. IEEE Journal on Emerging and Selected Topics in Circuits and Systems 3, pp. 1–4.

[3] Floyer D. (2015), The vital role of edge computing in the internet of things. URL: `https://wikibon.com/the-vital-role-of-edge-computing-in-the-internet-of-things/`.

[4] Raj E., Buffoni D., Westerlund M. & Ahola K. (2021) Edge mlops: An automation framework for aiot applications. In: 2021 IEEE International Conference on Cloud Engineering (IC2E), IEEE, pp. 191–200.

[5] Alla S., Adari S.K., Alla S. & Adari S.K. (2021) What is mlops? Beginning MLOps with MLFlow: Deploy Models in AWS SageMaker, Google Cloud, and Microsoft Azure , pp. 79–124.

[6] Shi W. & Dustdar S. (2016) The promise of edge computing. Computer 49, pp. 78–81.

[7] Symeonidis G., Nerantzis E., Kazakis A. & Papakostas G.A. (2022) Mlops-definitions, tools and challenges. In: 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), IEEE, pp. 0453–0460.

[8] Granlund T., Kopponen A., Stirbu V., Myllyaho L. & Mikkonen T. (2021) Mlops challenges in multi-organization setup: Experiences from two real-world cases. In: 2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN), IEEE, pp. 82–88.

[9] Treveil M., Omont N., Stenac C., Lefevre K., Phan D., Zentici J., Lavoillotte A., Miyazaki M. & Heidmann L. (2020) Introducing MLOps. O'Reilly Media.

[10] Antonini M., Pincheira M., Vecchio M. & Antonelli F. (2022) Tiny-mlops: a framework for orchestrating ml applications at the far edge of iot systems. In: 2022 IEEE International Conference on Evolving and Adaptive Intelligent Systems (EAIS), IEEE, pp. 1–8.

[11] Min C., Mathur A., Acer U.G., Montanari A. & Kawsar F. (2021) Sensix++: Bringing mlops and multi-tenant model serving to sensory edge devices. arXiv preprint arXiv:2109.03947 .

[12] Akkiraju R., Sinha V., Xu A., Mahmud J., Gundecha P., Liu Z., Liu X. & Schumacher J. (2020) Characterizing machine learning processes: A maturity framework. In: Business Process Management: 18th International Conference, BPM 2020, Seville, Spain, September 13–18, 2020, Proceedings 18, Springer, pp. 17–31.

[13] John M.M., Olsson H.H. & Bosch J. (2021) Towards mlops: A framework and maturity model. In: 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, pp. 1–8.

[14] Al-Fuqaha A., Guizani M., Mohammadi M., Aledhari M. & Ayyash M. (2015) Internet of things: A survey on enabling technologies, protocols, and applications. IEEE communications surveys & tutorials 17, pp. 2347–2376.

[15] Sculley D., Holt G., Golovin D., Davydov E., Phillips T., Ebner D., Chaudhary V., Young M., Crespo J.F. & Dennison D. (2015) Hidden technical debt in machine learning systems. Advances in neural information processing systems 28.

[16] (2023), Aws iot core. URL: https://aws.amazon.com/iot-core/.

[17] (2023), Intelligence at the iot edge — aws iot greengrass — amazon web services. URL: https://aws.amazon.com/greengrass/.

[18] (2023), Iot edge | cloud intelligence | microsoft azure. URL: https://azure.microsoft.com/en-us/products/iot-edge.

[19] PlatformIO, Platformio is a professional collaborative platform for embedded development. URL: https://platformio.org/.

[20] Foundation E., Leading open source community for iot innovation. URL: https://iot.eclipse.org/.

[21] Sato K. URL: https://cloud.withgoogle.com/next18/sf/sessions/session/192579.

[22] Mlops: Continuous delivery and automation pipelines in machine learning. URL: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning.

[23] pickle — python object serialization. URL: https://docs.python.org/3/library/pickle.html.

[24] Open neural network exchange. URL: https://onnx.ai/.

[25] Raj E., Westerlund M. & Espinosa-Leal L. (2021) Reliable fleet analytics for edge iot solutions. arXiv preprint arXiv:2101.04414 .

[26] Prince M. (2021), The edge computing opportunity: It's not what you think. URL: https://blog.cloudflare.com/cloudflare-workers-serverless-week/.

[27] Shi W., Cao J., Zhang Q., Li Y. & Xu L. (2016) Edge computing: Vision and challenges. IEEE internet of things journal 3, pp. 637–646.

[28] Beyer B., Murphy N.R., Rensin D.K., Kawahara K. & Thorne S. (2018) The site reliability workbook: practical ways to implement SRE. " O'Reilly Media, Inc.".

[29] Oliveira A. (2023), How to schedule jobs using the linux “cron” utility. URL: https://www.redhat.com/sysadmin/linux-cron-command.

[30] Preston-Werner T., Semantic versioning 2.0.0. URL: https://semver.org/.

[31] Go client for the docker engine api. URL: https://pkg.go.dev/github.com/docker/docker/client.

[32] Embedded devices - rust programming language. URL: https://www.rust-lang.org/what/embedded.

[33] clap - rust. URL: https://docs.rs/clap/latest/clap/.

[34] clap::derive::tutorial - rust. URL: https://docs.rs/clap/latest/clap/_derive/_tutorial/index.html#subcommands.

[35] hyper::server - rust. URL: https://docs.rs/hyper/latest/hyper/server/index.html.

[36] eclipse-mosquitto - official image | docker hub. URL: https://hub.docker.com/_/eclipse-mosquitto.

[37] shiplift - rust. URL: https://docs.rs/shiplift/latest/shiplift.

[38] Cerqueira V., Torgo L. & Mozetič I. (2020) Evaluating time series forecasting models: An empirical study on performance estimation methods. Machine Learning 109, pp. 1997–2028.

[39] Syafrudin M., Alfian G., Fitriyani N.L. & Rhee J. (2018) Performance analysis of iot-based sensor, big data processing, and machine learning model for real-time monitoring system in automotive manufacturing. Sensors 18, p. 2946.

[40] Carvalho T.P., Soares F.A., Vita R., Francisco R.d.P., Basto J.P. & Alcalá S.G. (2019) A systematic literature review of machine learning methods applied to predictive maintenance. Computers & Industrial Engineering 137, p. 106024.

[41] Uci machine learning repository: Ida2016challenge data set. URL: https://archive.ics.uci.edu/ml/datasets/IDA2016Challenge.

[42] Solomon S. (2021), Scania trucks air pressure system failure prediction. URL: https://medium.com/analytics-vidhya/scania-trucks-air-pressure-system-failure-prediction-ad6c43539d38.

[43] Hudgeon D. & Nichol R. (2020), Machine learning for business: Using amazon sagemaker and jupyter. URL: https://aws.amazon.com/sagemaker/pipelines/.

[44] htop. URL: https://htop.dev/.

[45] Chang Z., Liu S., Xiong X., Cai Z. & Tu G. (2021) A survey of recent advances in edge-computing-powered artificial intelligence of things. IEEE Internet of Things Journal 8, pp. 13849–13875.