Technical University of Applied Sciences Würzburg-Schweinfurt

Deployment of a Cloud Based Machine Learning Pipeline to the Edge

Author: Hamza Muhammad Naseer

Matriculation Number: 4019112

Bachelor of Engineering Mechatronics

Company Name : Bosch Rexroth AG

Company Supervisor: Felix Rauterberg

First University Supervisor : Prof Dr. Andreas Schiffler

Second University Supervisor: Prof. Dr. Bastian Engelmann

Contents

List of Abbreviations

AWS Amazon Web Services

ML Machine Learning

UUID Universally Unique Identifier

API Application Programming Interface

UI User Interface

DB Database

ECS Elastic Container Services

DAG Directed Acyclic Graph

EC2 Elastic Container Service

MWAA Managed Workflow for Apache Airflow

OS Operating System

VPC Private Virtual Cloud

RT Real Time

NRT Non-Real Time

PoC Proof of Concept

CHI Component Health Index

MHI Machine Health Index

1. Introduction

1.1 Research Topic and Motivation

The research topic revolves around the product "CytroConnect Solution", which is a data-centric machine health monitoring tool. The advantage of CytroConnect compared to traditional machine health monitoring tools is that since it is completely data based and does not require extensive physical/mathematical models of the machines being monitored nor any domain-specific knowledge for it to function. With cloud-based solutions it is also easy to expand the resources required to run it for any size of application.

However, cloud-based solutions are not preferrable in locations where internet access is restricted or in very remote locations with inconsistent internet connections such as at some offshore oil rigs or within mining applications deep underground or inside mountains. Edge Based computing can be very useful in these cases to keep all of your data on premises, enabling much faster access and processing times.

1.2 Current Situation / Problem

The current software stack exists completely on the cloud using a third-party cloud service provider AWS (Amazon Web Services). Since cloud storage is easily expandable, not much focus was given on optimization of size of the models and containers used for applying the ML pipeline. Furthermore, the orchestration software used (Apache Airflow) is also resource intensive. Most edge devices have limited storage and computation abilities. Furthermore, a consistent internet connection to an edge device cannot be guaranteed at all times so the cloud–based solution cannot be directly ported to the edge device.

1.3 Objective of the Thesis

The goal is to have a proof of concept of a hybrid machine learning pipeline consisting of parts from both the cloud and the edge. This pipeline must be run successfully on an edge device, here in this case being the ctrlX Core. Meaning, getting appropriate Machine Health Index "MHI" scores on our edge device.

The performance of this edge device implementation must be compared to the cloud-based solution. Furthermore, effort must be made to maintain similarity to the current-code base by making minimal functional changes for ease of integration of this use-case into pre-existing software stack to enable ongoing support for the platform.

1.4 Structure of the Thesis

This bachelor thesis is divided six chapters. The first chapter introduces the topic and the problem statement, provides an overview of the current situation, and explains the goal to be achieved. The second chapter goes over the core concepts required for understanding the further chapters. The third chapter explains the current cloud-based architecture and goes into details about the functionality of each stage of the pipeline. The fourth chapter discusses the creation of a hybrid architecture and its actual implementation. The fifth chapter covers the evaluation of the hybrid architecture implementation from the previous chapter. The sixth and final chapter concludes the thesis, highlighting potential future research directions.

2. Theoretical Basics

This chapter offers a framework for comprehending the ideas and innovations that underpin the investigation conducted for this thesis. Theoretical concepts in the context of machine learning and deployment are introduced.

2.1 Cloud Based Architecture and Edge Based Architecture

2.1.1 Cloud Based Architecture

Cloud computing is an encompassing term to describe using massive data centres to store, manage, process, and transform data and other applications remotely through the internet. This approach has been widely adopted by many industries and companies of varying sizes due to the convenient end-to-end capabilities and quick deployment of resources usually provided by cloud service providers [1].

Since all the data storage and processing is done at the data centre, that means that they are able to offer virtually any amount of storage and processing capabilities required due to resource pooling. Furthermore, they are also able to scale resource allocation automatically depending on the load provided, adding another layer of convenience for the end user. This allows users to have support for occasional spike in usage and parallel processing abilities without having to invest permanently in additional data infrastructure thus reducing costs. [2]

Since many of the large cloud-service providers are based in many countries around the globe, this also means that you can provide low latency operations to sites and projects based in any country. [3]

2.1.2 Edge Based Architecture

Since the wide-spread adoption of cloud computing, there has been growing concerns attached to its use as referenced in [4] which edge computing aims to address. Edge computing is the idea of processing and storing your data closer to the data source, either

on a private local cloud or a physical device present on premise.

Cloud service providers require a consistent connection to the internet because the data is transmitted constantly to a central server where it is processed. This connection also needs to have high bandwidth to ensure the large amounts of data are transferred in a timely manner. However, applications based in very remote locations where consistent internet connection is not guaranteed would have severe difficulties functioning appropriately if relying solely on the cloud solution. This is where Edge computing has an advantage since all the processing is done locally. In geographical locations where data centres are far from location of the application, there could also be high latency issues which could also cause additional disruptions in operations.

Furthermore, cloud storage for applications with sensitive data is another point of contention since constant data transmission or the storage on cloud itself is more susceptible due to the complexity in setup of cloud infrastructure, there is likelihood of security misconfiguration providing unintended access to foreign users which can result in loss of intellectual properties and financial losses. Therefore, some companies have also employed policies which prevent any of their data from ever leaving their facilities which makes it very difficult to use commercial cloud computing services for their applications. In contrast, edge computing devices store and process all the data locally, and in some cases don't even have to be connected to the internet at all, which makes it much harder for them to be a victim of data-theft.

High costs for operations using cloud service providers can also be an issue for small to medium scale businesses because cloud service providers have multitudes of charges related to their operations such as data transmission costs to move the data to the data centres, data storage expenses and minimum resource allocation costs even during less resource-intensive periods. Whereas for edge computing, there is minimal data transmission costs because it has to be transferred locally, there is no minimum resource usage cost because it can be switched off during periods of inactivity.

2.1.3 Hybrid Approach

The actual solution which highlights the strengths of both types of computing while effectively addressing the limitations is an architecture using both in tandem as also proposed in [5].

In a hybrid deployment, the combination of edge and cloud can be used to get the most cost-effective resource allocation for tasks. Since edge devices are usually too underpowered to train intensive models on, this can be offloaded to the cloud where larger resource pools can be used to train this much more quickly and efficiently. This also saves from large and expensive data infrastructure having to be bought by individuals. Furthermore, edge devices can be used for all the preprocessing tasks and inference, ensuring that only the most relevant data is sent to the central server, thereby reducing network bandwidth, and reducing some of the cost as well.

2.2 Real Time Data Streaming vs Batch Processing of Data

In the context of machine learning, there are two methods for processing data: batching and streaming. Data is gathered and processed in fixed-size batches or pieces through the process of batching. This approach can analyse bigger amounts of data more effectively at once. Conversely, streaming allows for continuous monitoring and prompt reactions by processing data as it comes in. Batching is typically utilised for activities that can tolerate some delay and involve processing bigger volumes of historical data at once, whereas streaming is frequently used for applications where low latency and real-time insights are critical.

Performance wise, the trade off is mainly in the type of resource used. Since data streaming inherently works with a smaller amount of data, it requires less storage, but a much higher CPU consumption since the frequency of operations is much higher, whereas batching usually has less amount of operations per time period, however has to deal with much larger amount of data, so it requires more active storage.

However, smaller batch sizes can be utilised to have relatively faster response time to anomalies and mitigate one of the downsides of batching.

2.3 Model Registry

A model registry is a repository used to store and version trained machine learning (ML) models. It contains many tools to improve reproducibility and tracking for ML projects and simplifies the standard ML Lifecycle[6]. Previously model parameters and experiment results were tracked in local files and excel sheets, which makes it exceedingly difficult to make use of all the information being collected because of dispersion.

Most model registries use unique identifiers (UUID) to store models and might also include functionality to track multiple versions of the models. Besides storing the model, themselves, they can also store metadata e.g., hyperparameters used for training the models, the training data used and model types etc. Furthermore, model registries also contain some sort of UI (User Interface) for users to be able to interface with all the information and compare performance between different models and runs. Finally, they also might contain some sort of programmatic API (Application Programming Interface), which can be used to query the models and or their respective metadata easily.

Besides tracking models and their information, in real-world uses case, many different models might be used and compared before getting deployed to production. Each of these different experiments for a single use-case can also be grouped together and have their metrics logged. This is commonly referred to as Experiment Tracking in various model registry providers.

Model registries increase trackability and reproducibility of models during any ML related project. Which enables these projects to be deployed to production in a standard fashion and more efficiently.

2.4 Container Technologies

A container is a standard unit of software that packages up code and all its dependencies, so, the application runs quickly and reliably from one computing environment to another [7].

Containers are especially useful in isolating and modularising distinct parts of a Machine Learning pipeline, where there are different dependencies required at each stage e.g. A machine learning pipeline with multiple steps, where the first step is data cleaning with the next step being model training, with some large sized training library being used for the model training stage. Since this training library is not required during data cleaning stage and is a considerably large size, it can be removed from this stage, which will improve the performance of the edge device during this step because redundant overhead is removed.

There are multiple options available for containerisation including completely simulated. Virtual Machines, Docker Containers and Snap etc. The ones relevant for this thesis will be Docker and Snap which will be explained in a bit more detail. Figure 1 shows how a container is structured. The underlying infrastructure and operating system are completely isolated from actual applications being run using the container engine, which performs the virtualization tasks and creates standardised instances which contain the applications. This container engine enables various machines to use the same code without changes.

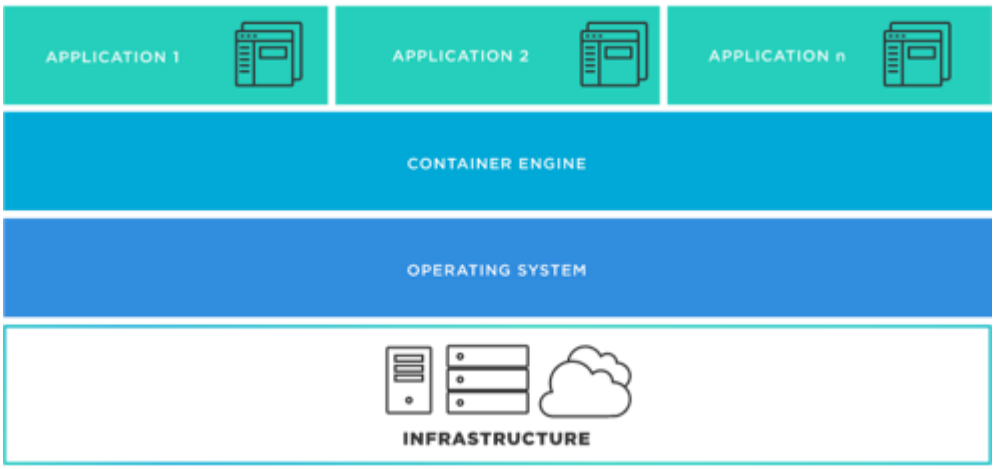

Figure 1: Container Working Principle

2.4.1 Docker

Docker is an open-source platform which provides tools and services for creating and managing containers. Contrary to Virtual Machines, they share the Host OS kernel, which allows it to share the same system resources without abstraction overhead.

The docker daemon is an engine running on top of the host, which takes API requests from the user using the docker-cli or docker client (GUI) to create and manage the containers.

Docker containers are designed to be portable and easy to distribute. An isolated container containing application code, dependencies and environmental variables etc. is called a Docker Image. Each docker image can be created from a "dockerfile" which describes how an image is built.

Since all containers are managed by the docker daemon, that means that any OS which supports the docker daemon is able to run the same containers.

2.4.2 Snaps

Snap is a package management system which provides a way to package applications with their dependencies and runtime environments. They directly extend the functionality of underlying host. Snaps are primarily intended for package distribution on Linux so it's easy to port snaps to different Linux distros which support snap package manager and use them. The main advantage of Snaps compared to Docker containers is that they do not contain an OS within, which makes them much lighter to run.

The application and its environment are also isolated by default within snaps, but they have connectors available called slots and plugs, which can be used to interface your snap with the host systems services and data. Figure 2 shows a diagram explaining how these interfaces could be used to connect between two different snaps. The slot provides the output connection which can be connected to through an interface by a corresponding plug. The type of plug which can interfaced with the slot is defined in the snapcraft.yaml file as well. In an Ubuntu version provided by Canonical called Ubuntu

Core, which is a version of the operating system designed and engineered for IoT and embedded systems, the whole operating system is defined using snap packages, so all the system resources are accessible through slots e.g. home directory access , network access etc.

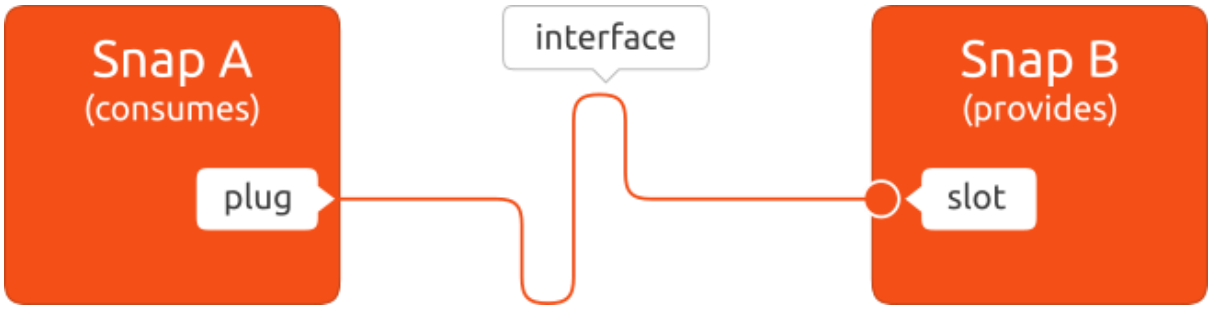

Figure 2: Container Working Principle

A snaps functionality, constituent dependencies and interfaces can be defined using a snapcraft.yaml file [8] . This snapcraft.yaml is used by the snap craft tool to create, build, and package the snaps.

3. Current Cloud-Based Architecture

Since the hybrid architecture plans to implement the pre-existing cloud-based functionality through a different channel, a solid understanding of the constituents of the current cloud architecture is required. An overview of the pipeline is given in Figure 3. Shown is the main functional workflow through each containerised steps along with the data sources and sinks required for them. It also shows additionally the conditional steps using the model registry and training containers and how the whole pipeline is orchestrated.

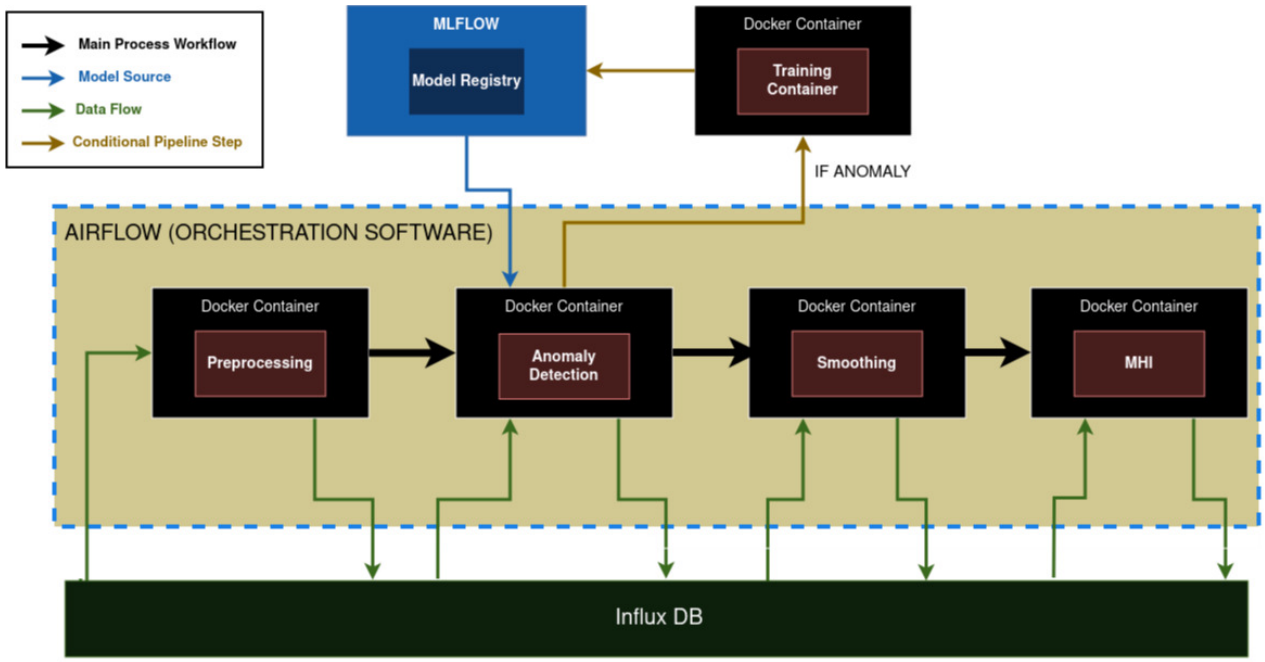

Figure 3 : Cloud Architecture Overview

3.1 Data Sources and Structure

A standard hydraulic application which can use this pipeline can consist of several hydraulic components (pump, motor etc.) where each hydraulic component can contain multiple sensors which provide different kind of time series data.

A component like a pump or a motor could contain multiple sensors to monitor the different parameters during its operation. Each sensor of the component is defined as an individual signal. The signals are constantly fed into the time series database using data acquisition units (DAQS), each individual DAQ is usually responsible for collecting data for multiple different sensors, albeit with a certain sampling rate. So, all the sensor values (signals) relating to one component could be collected using multiple different DAQs if it contains sensors which send data at multiple frequencies. All these DAQs stream the data through an API gateway to the cloud. The actual internet connection varies across different applications, in some places it is being doing using wLAN and in others using purely mobile data.

All the signals relating to a single hydraulic component form to combine a component. And all the components belonging to one single machine or application are grouped together into an MLSystem.

Now using this component information, it could be possible to have different configurations for the machine learning pipelines, i.e., it could be possible to use a different algorithm for the anomaly detection, or have different preprocessing steps and activation conditions, these variants of the whole configurations are all different MLSystems which are still part of the same customer and are grouped together as an Application.

Each sensor time series is stored as a signal in the database.

The database in operation is called InfluxDB, which is a database provider specifically designed to handle time series data. This means that they store huge amounts of time series data much more quickly and efficiently. It also enables them to perform time series related operations on the data within.

3.2 Core Functionality

The core components of the pipelines are the individual stages called Preprocessing, Anomaly Detection, Smoothing and Machine Health Index (MHI). They provide the

current anomaly detection functionality for the CytroConnect product.

All the parameters related to all the pipeline stages and all the data mappings about a singular hydraulic application stored in a config file. This config file is a YAML file constructed at the start of implementation for any hydraulic application and serves as a guideline for all the following operations.

Once each stage is successfully completed, a checkpoint file is created and the time for the last processed data segment. In order to ensure complete coverage and prevent repetition for the same timeframes.

3.2.1 Preprocessing Stage

Preprocessing is the initial step in any machine learning pipeline, providing a foundation upon which any effective models can be built. The raw data received from the sample is sometimes sampled in milliseconds range, down to 1ms in some cases. This is a huge amount of data which is sometimes filled with missing values and wrong values due to sensor errors, and the huge amount of data itself is also not very beneficial to the model training as well. Furthermore, it is also expensive to transfer unnecessarily large amount of data from one step to next.

Once the data batch of the selected timeframes is retrieved from the database. The preprocessing steps can begin.

components are all stated in the config file as shown in Figure 4.

```
preprocess_data:
  options:
    steps:
    - action: realign_index
      options: {}
    - action: early_dropna
      options: &id001
        how: all
    - action: filter_implausible_values
      options:
        implausible_value_threshold: 1000000000000.0
    - action: sample_data
      options: &id002
        sampling_fraction: 1.0
        sampling_head: 0
    - action: handle_nans
      options: {}
    - action: activation_condition
      options:
        activation_rule: df.loc[ (df['sample_feature1'].rolling('5min').mean() >
          30) | (df['sample_feature2'].rolling('5min').mean() > 3)]
        bootstrap_cutoff_time: 5min
    - action: missing_signal_policy
      options:
        policy: strict
```

Figure 4: Preprocessing Steps defined in Config File

These preprocessing steps are for the moment standardized across all applications. Every new hydraulic application which is deployed uses the exact same steps with the same parameters.

The *realign_index* step is used to resample the data to the highest frequency of non-null data. The reasoning is that different sensors might be sending data at different frequencies. In order to properly utilise the high-sampling data, all the other samples must be up-sampled to the same sampling rate.

The *early_dropna* function is used to drop a row if it's the completely nan. This reduces the amount of non-useful data which would be transmitted to the next steps.

The *filter_implausible_values* replaces all the values which are above a certain implausibility threshold with NaN. It is possible sometimes that sensors provide some wrong value which would be detected as a false anomaly, or would cause problem in model training, so it is removed pre-emptively.

The *sample_data* function is used to keep a sub-sample of the input data, it maybe the x initial entries defined by sampling head or a certain random percentage defined by sampling fraction. This is for the cases where even after resampling and removing implausible values, the amount of data is just too much to be useful. However, in the current preset, this function still returns the complete amount of data, no samples are removed.

The *handle_nan* function handles all the remaining NaNs in the data in 4 steps. It first replaces all the NaN with the previous non-NaN value from that column. Secondly it replaces all the remaining NaNs with the next non-NaN value. Finally, if there are still some NaN values left within the data chunk, those are then replaced by 0.

The *activation_condition* function is used to further filter a dataframe based on a custom attribute value defined in the config file. This is the only part of the preprocessing step that is usually changed per component.

The last step is the *missing_signal_policy*, which checks if a component from the config file is missing in the dataframe. Depending on the policy, if it is in "strict" mode this either causes the pipeline to stop or "skip" which appends a column full of NaNs to the pre-processed data in place of the missing signal.

Once all these steps are completed, the processed data is sent back to the database, and the next step in the pipeline can begin.

3.2.2 Anomaly Detection Stage

The anomaly detection stage consists of the actual usage and application of machine learning. Where models are trained and used for inference. The main purpose of this stage is to flag anomalous sensor data received from the preprocessing stage, to minimise damage caused by faulty components.

The stage starts again with requesting data from the database. The requested data is the output of the previous stage(preprocessing). The anomaly detection models for all the components are used to calculate the anomaly score of the source data. The parameters for the ML Model are also defined in the config file .

There are multiple models, which are trained over different time periods, which are all used to inference the input data. Each model first calculates the model score for each sensor signal, then takes a mean over a time slot for all the signals to get an average model score per timeslot. This model score is then scaled between a scale of 1 to 100 which is called Component Health Index (CHI) for the component. All the CHIs (Component Health Index) from multiple models are then also aggregated together to create an average

model score for the component. If this aggregated score is under the anomaly-threshold parameter defined in the config-file, it is declared an anomaly. If there are no models found in the registry for the component, which could possibly be a new component or application, that means there is no knowledge about it, so it is declared an anomaly as well.

Once a timeframe has been declared anomalous, a new model is trained for that chunk. The reason for this is that the anomaly detected could actually just be a process change. This would mean that all old models would always declare the new data anomalous. Hence, a new model is trained.

The model training consists of creating a Docker training container with predefined dependencies. This training container can then be used in the cloud platform to train a model. Once the model is trained successfully, it is added to the model registry, to be able to be used for the next batch.

The model scores from this newly trained model alongside with all the inference results from all the previous models are then sent back to the database to be then further used in the next stage of the pipeline.

3.2.3 Smoothing Stage

At the end of the anomaly detection stage, there are multiple model scores for the same time-chunk for all the components. All of these model scores from different models must be combined into a comprehensible score for each component. The smoothing stage starts by resampling the data to a lower frequency defined in the config file. This is followed by taking the maximum value across each row, basically taking the best model score for the defined time stamp. This is followed by a rolling mean across a window defined in the config file. The final result is a smooth single value per time stamp for the "health" of the component (CHI). The stage ends by uploading these CHI values for the component to the database.

3.2.4 Machine Health Index Stage

Once a CHI value is created for each of the subsequent components of the MLSystem for a specific time stamp, the final Machine Health Index (MHI) can be calculated. The step starts by importing the previously calculated CHI for each of the components. The actual step to define the MHI is very simple. The minimum value of the CHI across all components at a certain timestamp is taken as the MHI value for that timestamp.

3.3 Orchestration

Once all the docker images pertaining to each stage are created, some computing instances must be deployed for the containers to run on. This is managed through AWS ECS (Elastic Container Service). ECS offers a service to deploy serverless clusters called Fargate which enables you to deploy tasks to it without having to take care of the resource management of the underlying instances. Since CytroConnect is used for multiple applications which have multiple components with multiple stages, which all run independently at different times. This makes it very convenient to use because to maintain a multitude of compute instances where resource allocation must be done yourself requires a lot of extra effort. This also integrates seamlessly with our pipeline stages because anomaly detection stage requires data batches to be present in the cloud storage when training using SageMaker and setting up access within the AWS ecosystem is more optimised.

Now that computing instances are setup, it is also need for these tasks to be deployed at regular set intervals and track if they have executed successfully. For this purpose an open-source and widely used management platform called Apache Airflow is used. Airflow creates a scheduled execution task pipeline called a DAG [9], which allows for other parameters such as number of retries or frequency of runs. Each stage of this DAG is called a task, and this task basically runs an instance of the container defined in ECS above. The advantages of these DAGS is that it allows you to manage task dependencies, i.e., if the Preprocessing stage for a component has an error, it automatically stops the execution of the anomaly detection stage. Figure 5 shows an example dashboard for a

DAG, tracking the status of multiple tasks over a period, showing the status of previous runs as well. Since this pipeline is automated and each task's status is not checked manually all the time, the failed task runs can be addressed at a later time with ease.

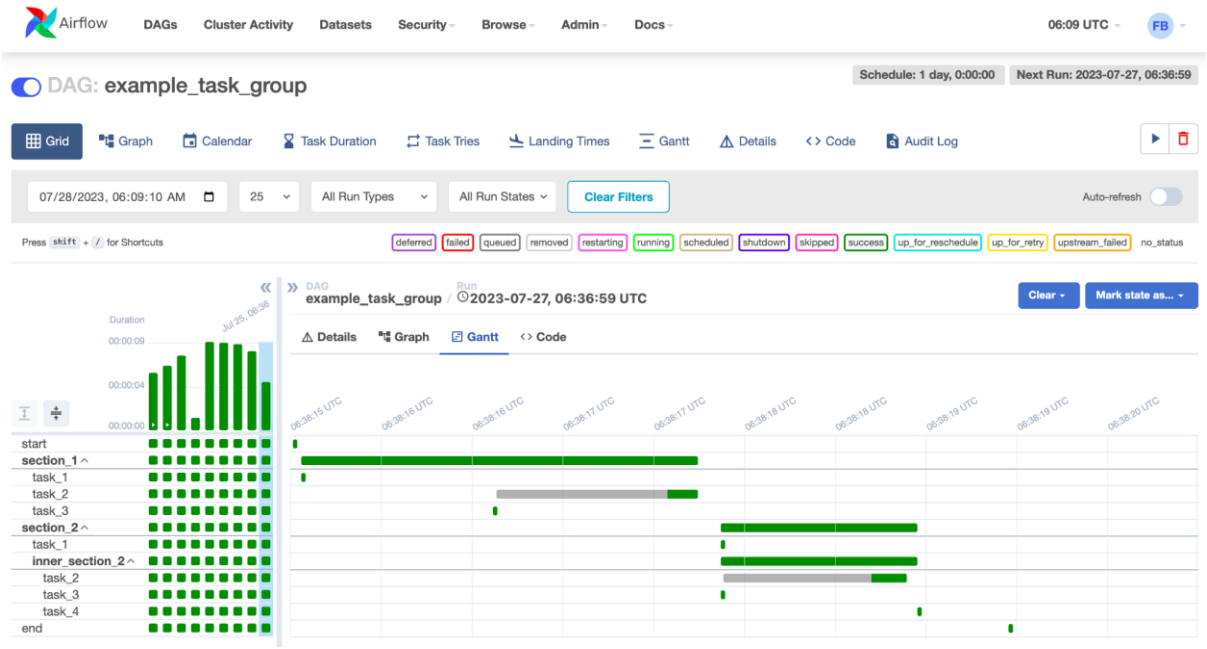

Figure 5: Airflow DAG Task

The instance of Apache Airflow is also run using an AWS service called MWAA, which provides the same advantage as Fargate clusters that it is automatically scalable, and extra care must not be given about maintaining the underlying infrastructure.

4. Hybrid Architecture

4.1 Proposed Initial Idea

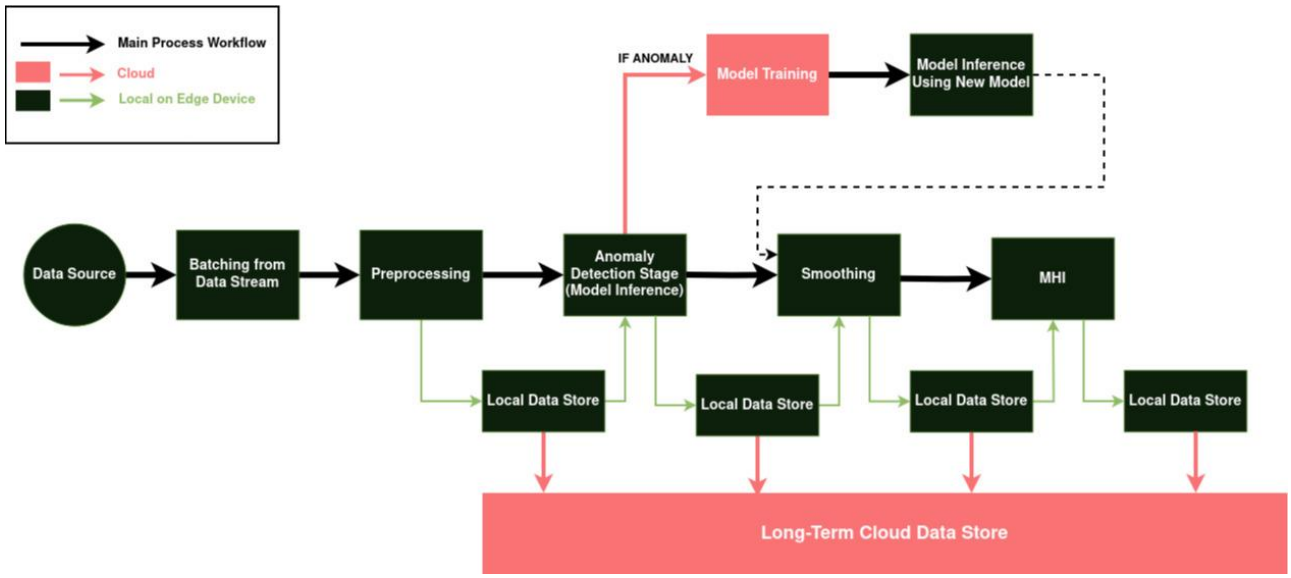

Figure 6 : Proposed Edge Architecture

The initial idea was to create a PoC described in Figure 6 which was architecturally as close as possible to the cloud stack but implemented on an edge device. This would enable easy continued development for the edge stack and would reduce the need to maintain two divergent code bases in parallel.

The input data coming in from the application or sensors is assumed to be in a stream. However, since CytroConnect is a batch-based pipeline, this incoming stream must be batched at a specific interval. The exact interval is a hyper-parameter which can be optimised later.

Once the data is in batches, this can be processed locally on our edge device as it was done on the cloud stack. Once the pre-processing is finished, the data must be stored locally on the edge device to be used by the next stage.
In the next stage, the pre-processed data has to be put through a model. The model is initially trained on the cloud, to reduce resource consumption and decrease the training duration. A trained model is then made available on the edge device to be inferenced from. If no anomaly is detected, as before, the data must be stored locally and made accessible for the next stage in the pipeline. However, if an anomaly is detected and a new model has to be trained, that process is again deployed to the cloud.

The same process is then repeated for both of the post-processing stages as well, performing the function defined in the steps and the saving of the data locally. <

To maintain a consistent structure for all the applications, it was also discussed that, at set intervals, the batches produced by all the stages could also be uploaded to the cloud for long term storage. This maintains consistency with the pre-existing cloud-based applications and reduce the storage used by non-essential data on the edge devices, which usually have very limited storage.

4.2 ctrlX Core

ctrlX Core is a controller device with ability to communicate with multiple fieldbus protocols and also the ability to interface multiple different I/O modules.

The specific version of the ctrlX Core being used for this thesis is the ctrlX Core X3 with 4 GB of eMMC storage and 2 GB of RAM.

The software running inside the ctrlX Core is called ctrlX OS which is a Linux based operating system. This enables easy development for apps for the ctrlX Core and enables the possibility of using third party applications developed for native Linux. Snaps are the primary way to install additional functionality on the ctrlX Core.

The main functionality from the ctrlX software stack being utilised in this thesis is the Data Layer, which is the data broker of the ctrlX Core and provides secure and managed access to the RT and NRT (Real Time and Non-Real Time)data available on the control.

The Data Layer is structured like a tree. Each data channel is called a node. Each data (node) has an address in the ctrlX Data Layer. An address is a hierarchical path to the node. The individual components of the path are separated by a “/”. This structures all data hierarchically.

Figure 7 shows the preprocessing node in the datalayer.

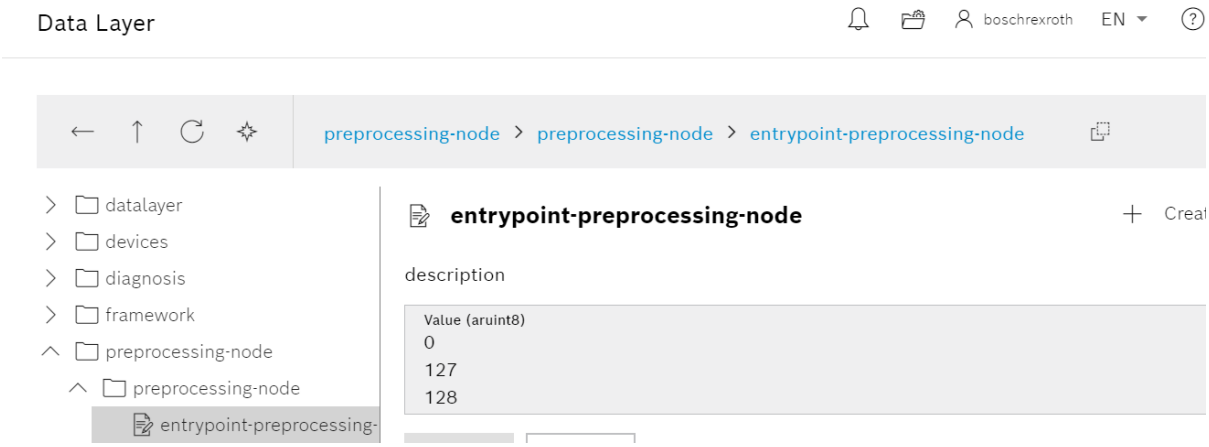

Figure 7 : Current Edge Architecture Implementation

4.3 Current Architecture

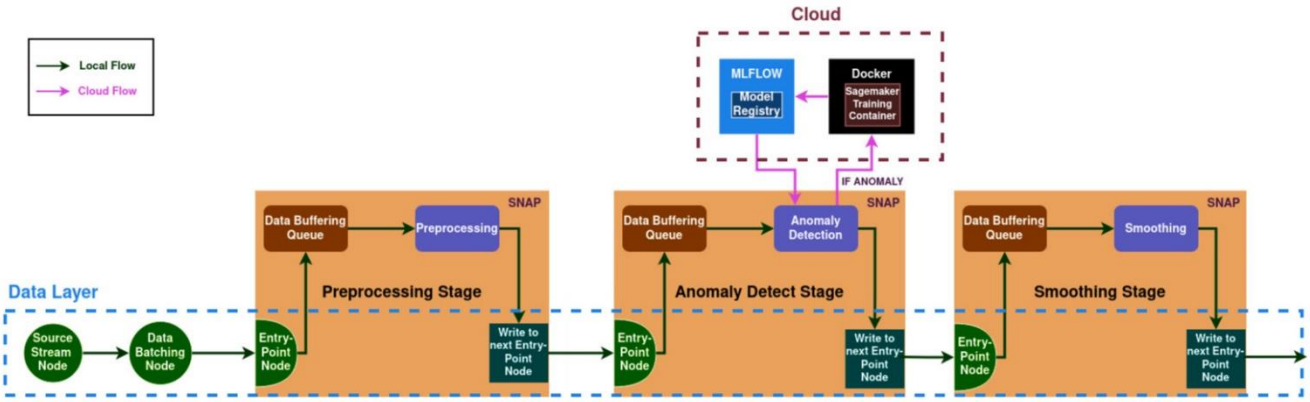

Figure 8 : Current Edge Architecture Implementation

The figure 8 shows the application of the proposal discussed above in Chapter 4.1. The four stages still remain functionally the same, the MHI stage is just not shown in the figure for more explainability. Three things have been changed primarily in comparison to the cloud pipeline. The data transfer mechanism, the orchestration of the pipeline and the type of containers used.

The data transfer is now managed through the Data Layer. A node is created for the input data stream, which can possibly be a stream from various IoT devices. The data stream is then sent to a batching node, where the data is batched on an adjustable time interval parameter. The batched data is then sent to the entry-point-node of the preprocessing stage. The data, which is transferred remains the same, consisting of dataframes. The dataframes must be encoded as unsigned 8 bit-integers to be sent over the Data Layer between the different stages. Some changes must be made within the stages themselves

to be able to function with this updated data transfer mechanism. The changes remain consistent over all the stages. The encoded data frames sent over the Data Layer must be decoded back into dataframes. These data batches are then added to a queue and are accessible through global variables to all the files in that stage container. The reasoning for adding the data batches to a queue is that the processes defined in each stage can take a longer time than the time it takes for the next batch to arrive, in that case the new incoming data might be lost in the absence of a buffer.

The existence of this queue is also multi-purpose because it can be used to replace the dedicated orchestration software used in the cloud solution. The processes in each stage are triggered as soon as there is some data in the queue. This method reduces the resource overhead required by the orchestration software.

For the anomaly detection stage, the inference of the models is done on the edge device itself. The model registry from MLFLow in the cloud is still used as a centralised store for the models. This is done for now to maintain consistency with the current cloud implementation. This can be easily changed to a local model registry which can be hosted directly on the ctrlX Core if it is required. The training of the models is also cloud based, using SageMaker training container instances . The models are then stored in the MLFlow model registry, same as the cloud implementation. The trained models are usually very small (few kilobytes), which can be easily downloaded from the cloud due to very low bandwidth requirements. The only time data is transferred from the edge device to the cloud in this stage is if an anomaly is detected, because the data chunk is used to train a new anomaly detection model.

4.4 Changes Made from Cloud Architecture

The ctrlX Core uses a processor which is based on the arm64 architecture, so in order to build snaps for the ctrlX, a machine is needed which is also based in the arm64 architecture. For this requirement, a remote computing instance from AWS (EC2 Instance) based on arm64 was used.

In order for the snap to be able to interact with the datalayer and to be able to run the CytroConnect code, relevant dependencies needed to be installed. Since both these

packages consist of many libraries, with their own dependencies, careful consideration needs to be made to use right versions of the constituent libraries which ensures compatibility.

Besides the proper dependencies, to enable the snaps to be able to communicate with the datalayer, have access to the underlying file system, and also have internet access, the snap needed to have the following plugs: Home, Network and a plug to a custom slot called Datalayer.

Since CytroConnect was initially designed with cloud compute in mind, some changes also had to be made to base library called cytrolib which constitutes CytroConnect.

The first change to be done was the removal of the Domain API calls. All the data related to each application, the databases and the credentials for all the things were called through the domain model API, since it was assumed that in normal working conditions, any application which was registered to work with CytroConnect should have all its constituent data present in the Domain Model, however for our use case, all of this information should be available locally.

Since the data should be available locally, the codebase had to modified to read the application config file locally from the ctrlX Core which will now need to present locally on the device instead of being in an Amazon S3 Bucket.

5. Testing Method and Results

This section outlines how system is tested results are shared. The methods used to evaluate performance, functionality and reliability will be covered, providing a clear picture of how well our implemented solution are working,

5.1 Relevant Metrics

To compare the performance of this local deployment to the cloud-based solution, some metrics need to be defined.

The main metrics which need to be tracked are as follows:

- **Network Usage,** so it can be seen how much bandwidth is saved in comparison to the cloud solution.
- **Memory and CPU Consumption** on the ctrlX Core, so it can be confirmed if the ctrlX Core is adequate for this use case and will not throttle.
- **Pipeline Lifecycle Duration**: Time it takes for one batch of data to get processed through the whole pipeline, to see if the benefits of not having to download and upload models and the data from cloud gives a noticeable performance boost.
- **Batch Size** can also be used as a parameter to optimise the edge pipeline, so that can also be experimented with. Smaller batch size also might make it possible to compute much higher sampling data like 5ms to 10ms, instead of being limited to 100ms or slower.

The tracking of all these metrics is achieved by creating bash scripts, which are able to access these parameters. Along with the total bandwidth, the bandwidth usage per second is also tracked , to track which part of the pipelines use the most amount of data. The CPU and RAM consumption is also tracked here. The system_monitoring file is used to track the parameters across the whole ctrlX Core while the process monitoring tracks the resource consumption per process.

5.2 Test Setup

The test is set up with the intention of tracking a single batch of data through the whole pipeline, going through all four stages.

All the different variants of test data are sent to the entrypoint-node of the preprocessing stage one by one.

For the experiments, the snaps are always relaunched during each different variant test, so the startup times of the nodes are also included, which is the worst-case scenario which can be faced during operation.

There are a total of 5 monitoring scripts running during these test runs. One for each stage process and one for the overall system, so there is some additional overhead and performance might be slightly better during actual use.

The variants of sampling test data are used to simulate various levels of possible functional load during application. The range of the data is from 1ms to a 100ms, which is used to simulate a wide range of possible loads.

The second parameter is the batch duration, which along with also being used for different load sizes, is also used for testing possibilities for faster response times, i.e., if a chunk is processed every 5 minutes instead of an hour, an anomaly could be detected much quicker.

5.3 Results

The results from all the various tests can be divided into the following four sections, which are discussed below.

5.3.1 Processing Times

The Table 1 below shows the time taken for each of the data variants:

Batch Duration / Sampling Rate	1 ms	10ms	100ms
5 Minutes	17 Minutes	5 Minutes	3.5 Minutes
15 Minutes	NaN	8 Minutes	4 Minutes
1 Hour	NaN	27 Minutes	6 Minutes

Table 1: Time taken for various batches to complete pipeline.

For the high frequency data, it is only possible to process 5-minute batches because the processing overhead becomes too much and the stage crashes. When using 15 minutes batches, the initial batch is successfully transferred to the preprocessing data snap and crashes there. However, when using the 1-hour batch, even the initial data transfer to the input node crashes. This is good to know to have an idea about the limitations of the ctrlX Data Layer on this system. The size of 1 hour of 1ms sampling data is 210 MB for reference. For the 10ms and 1ms sampling rate data, all of the batch sizes completed successfully, but it can be seen that it takes a much longer time for the longer sized batches to complete. It is to note that in all of the stages, the longest time is taken in the anomaly detection stage, because that is the stage where the most resource intensive operations are carried out.

Here it can also be seen that the advantage of having different batch sizes to control the amount of the data being processed at a single moment. It makes it possible for the functionality of the 1ms sampling data, whereas it wouldn't be possible at all if fixed batches of 1 hour were being used. The batch size parameter has diminishing returns on the processing time as the sampling rate decreases, because the change in the amount of data is then less drastic. However, in practice since the time it takes for the 5-minute batch of 1 ms data to be processed is longer than actual amount of time it is processing for, that makes it infeasible for actual use in a production environment, because it would mean that it would forever be creating an increasing backlog of data to be processed.

Any batch of data which takes less time to be processed than the size of the batch itself is a valid candidate for an actual use case. However it also important to note that, even though the 100ms batches take much less time than the 10ms batches, there is a huge

amount of data loss, which could most likely mean that a lot of anomalies could he smoothed over during down sampling, i.e. data from a pressure sensor, where the mean might remain the same, but the variance is the actual feature used to identify anomalies, which would get completely smoothed over, hence the 10 ms batch candidates are considered better for this use case.

Batch Duration / Sampling Rate	1 ms	10ms	100ms
5 Minutes	294.1	100	14.3
15 Minutes	NaN	187.5	37.5
1 Hour	NaN	222.2	100

Table 2: Data Point Processed Per Second in Each Pipeline

Table 2 can be used to confirm which configuration is actually the most efficient in processing the data. It takes the total number of data points given as input and divides it by the total amount of time. It can be seen that the 10ms 1 hour batch actually is more time efficient than the 10ms 15-minute batch, because a lot of the algorithms are designed to scale much better with larger datasets and be more resource efficient.

A lot of information can be collected through looking at timing charts for these pipelines can be used already filter out some possible configurations. From just looking at the time efficiency standpoint, the 1-hour batches could already be declared the ideal batch size to use, but looking at other metrics such as memory and ram usage will provide us with other variables to consider.

5.3.2 Memory Usage

Now looking at the charts for memory consumption starting with Figure 9, the batch of 1 hour of data at 10ms because that is the largest batch of data, hence also the most resource intensive. This will be ideal to discuss the resource consumption over the lifecycle of a single pipeline run in more detail because each part of the process is more pronounced.

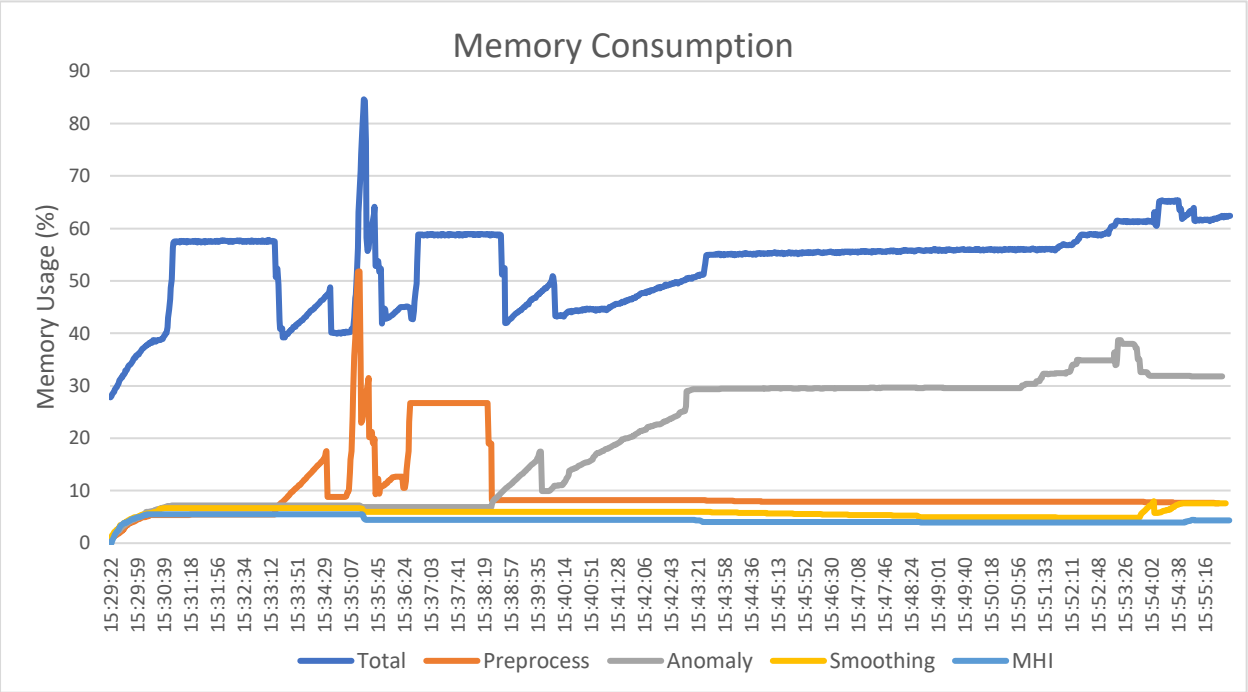

Figure 9: Memory Consumption (1 hour batch of 10ms data)

Even at idle conditions, when no data is being sent to any of the stages, it can be seen that the memory consumption is not 0, because there is still some resource overhead required for the nodes and their respective operations.

There is a sustained high memory use when looking at the total memory usage before the preprocess stage even starts, this is because of the initial data transfer through the datalayer to the entrypoint-node.

It can be seen in the charts for batches with a lower amount of data as in Figure 10 which shows the memory usage for 10ms data with a 15-minute batch and for Figure 11 which is 1 hour batch for 100ms data, that initial peak is much lower and also lasts a much shorter duration in comparison.

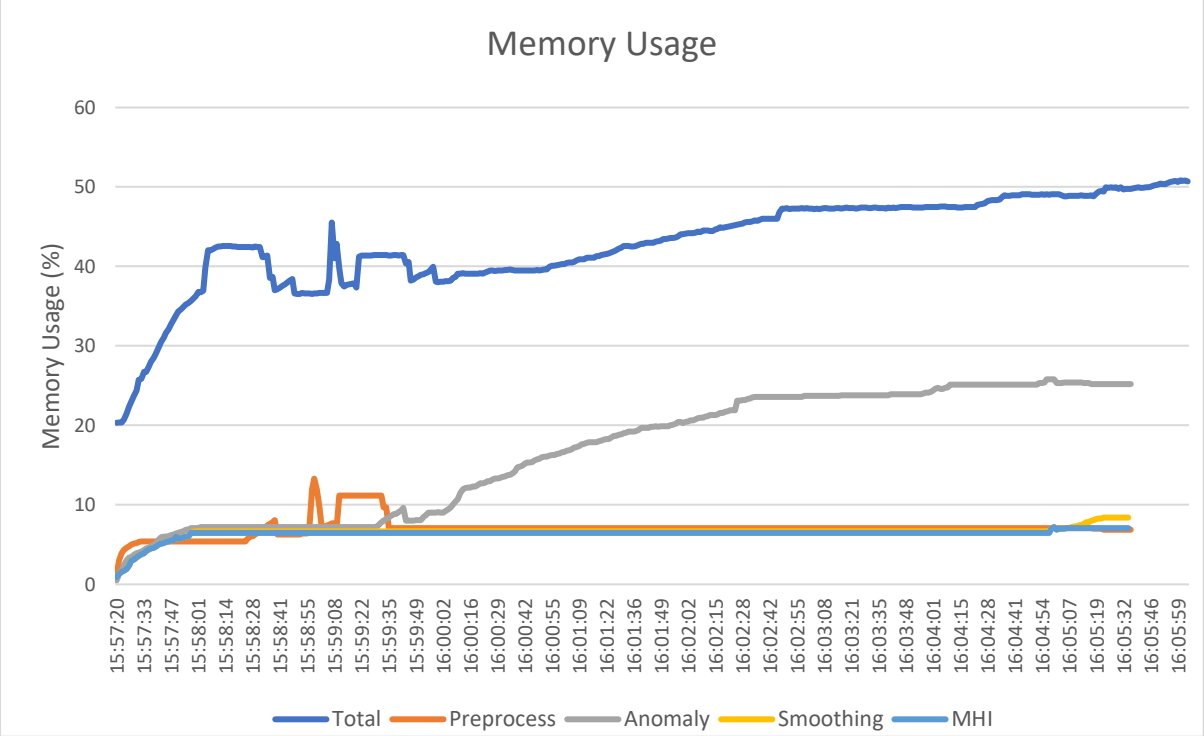

Figure 10: Memory Consumption (15 minutes batch of 10ms data)

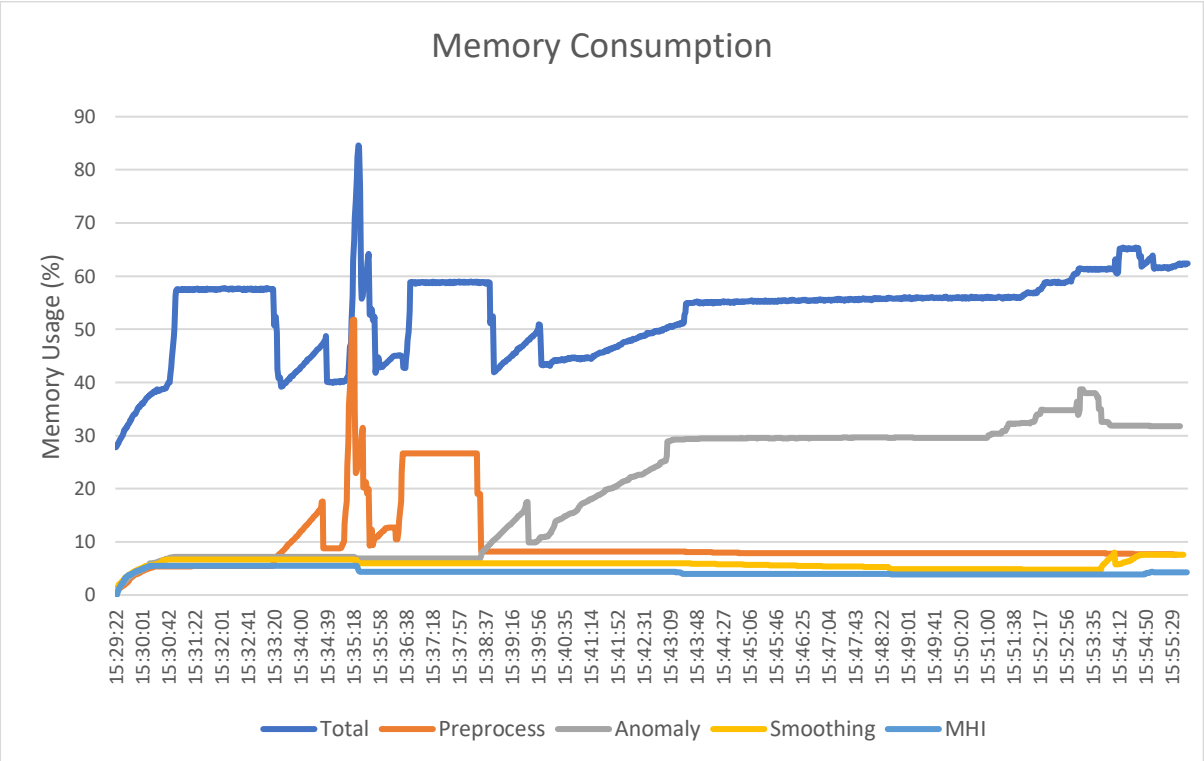

Figure 11: Memory Consumption (1 hour batch of 100ms data)

The small spike seen at the start of every stage is most likely due to the operation of

writing the data received by the entry-point-node to the local queue being used for the operations. When the actual preprocessing stage starts, it can be seen that the memory usage goes down again significantly. This is because most of the operations done during the preprocessing stage are operations to pandas dataframe, which is a very well optimised library, so the memory consumption is low.

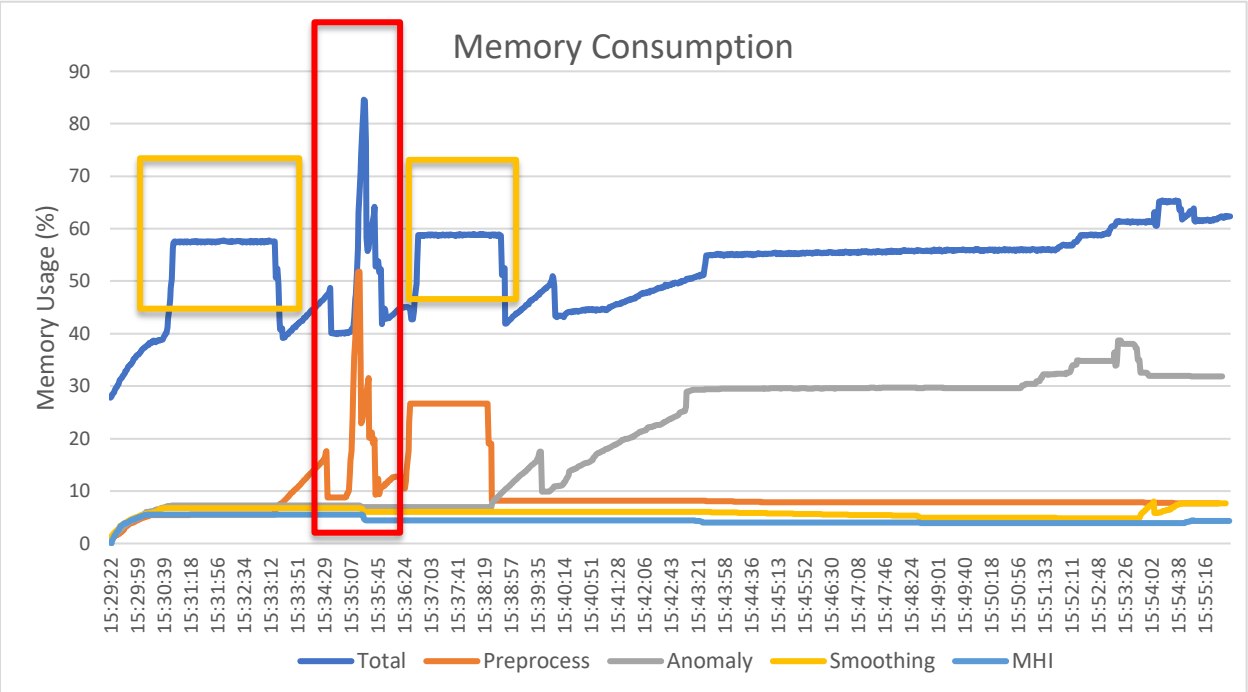

Figure 12: Memory Consumption Spike

The spike highlighted in red in Figure 12 could possibly be due to the resampling function, which can be computationally expensive because it generates a lot of new values. The large increase seen is the data transfer through the Datalayer to the entrypoint-node of the anomaly detection stage. This data transfer is actually slightly bigger than the original one because all signals with low frequency have been up-sampled to be at the same frequency to the highest frequency signal. This can be seen in the boxes highlighted in orange, the values in the first box which represent the initial data transfer are slightly higher and lasts a longer duration that the second orange box which is the preprocessing transfer. After this the preprocess stage usage goes back to idle.

The memory usage is the largest in the anomaly detection stage because it uses libraries which are very memory intensive such as TensorFlow. Model inference is also comparatively more expensive than pandas operations.

But there seems to be a much smaller jump at the end of the anomaly detection stage,

when the data is transferred through the datalayer to the entrypoint-node of the smoothing layer, this is because the amount of data being transferred after this stage is drastically less than the previous stages. Whereas the anomaly detection stage requires multiple features as input for computation , the output is only the model inference, which in this case is a seven time decrease in data because there are seven features which are used as input for the models, but the output is only inferences from one model, because there is only trained model for the component in this case, reducing the values per timestamp from seven to just one.

Since the Smoothing and MHI stage exclusively consists of pandas operations, and involves only down-sampling of data, means that both stages are very computationally inexpensive, as can be seen in Figure 9, 10 and 11 that the memory consumption barely moves away from idle value. There also seems to be no noticeable effect of the monitoring scripts on the memory usage because the sum of the individual stage snaps seems to align almost perfectly with the total memory usage.

Last remark to make about the memory usage is its consistency with expectations, anomaly detection stage uses the highest because it uses intensive libraries, and all the other three stages which only use pandas libraries, have their memory usage ranked according to the size of the data in that specific stage, i.e., Preprocessing then Smoothing then MHI in order of decreasing value.

5.3.3 CPU Usage

The CPU usage graph as shown in Figure 13 should also present similar general characteristics as the memory graphs, however, there seems to be some inconsistencies. The starting peak in the total and usage and all the stages can be attributed to the initial setup of all the entry-point nodes. Following this a similar idle usage for all the stages would be expected till the stage process begins followed by some increase. However, this behaviour is only exhibited by the preprocessing stage and none of the other stages. The starting high values of the total CPU usage can be the initial writing of the data to the preprocessing entrypoint-node, which can be seen being at low idle value then and

starting to increase as soon as the total usage drops at 15:33:21. And after an increased value for a bit, it returns to its idle value again. However, for all the other stages, they seem to have a very high idle usage as well, which does not seem to change at all, even when they are actually processing data. This behaviour has been investigated in order to identify the root problem and rectify it, but no progress has been made in this matter.

The total CPU usage in the figure seems to be more than the sum of the individual stages, this is mostly likely due to the five monitoring scripts running alongside the stage snaps.

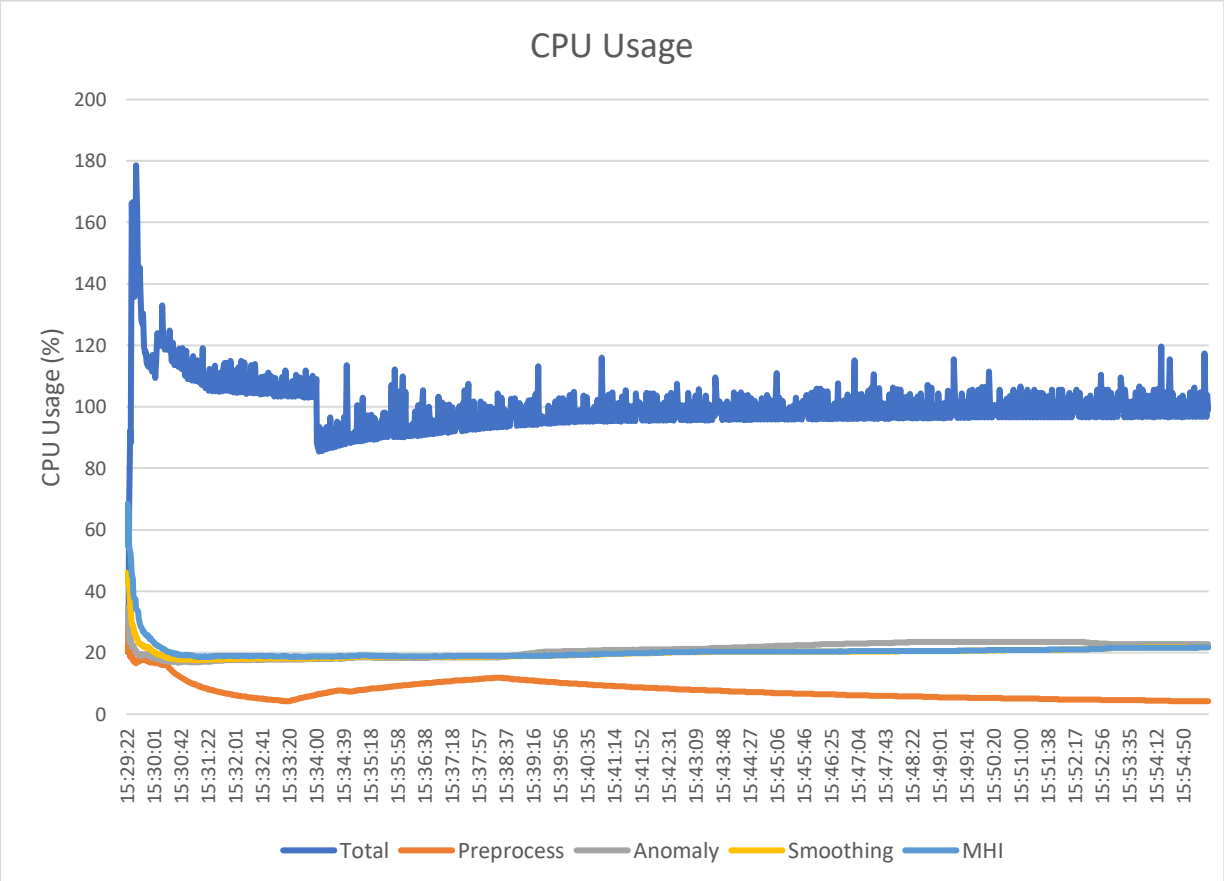

Figure 13: CPU Usage (1 hour batch of 10ms data)

5.3.4 Network Usage

The only connection to the internet in this pipeline is in the anomaly detection stage, all the other steps transfer the data locally through the datalayer to each other, and since they only perform pandas operation, there is no need to use the internet in any case.

There are two places where the internet is used in the anomaly detection stage, once

when the model is downloaded from MLFlow, and in the case there is an anomaly, the data chunk is uploaded to an S3 bucket so it can be trained by a Sagemaker model. And this is represented in the graphs in Figure 14. There is negligible throughput besides one peak. The small fluctuations that can be seen could be due to SSH connections made with multiple terminals to carry out this pipeline test, along with getting the logs from the SageMaker training container as it progresses through the training. This graph only look as such in the case a new model needs to be trained.

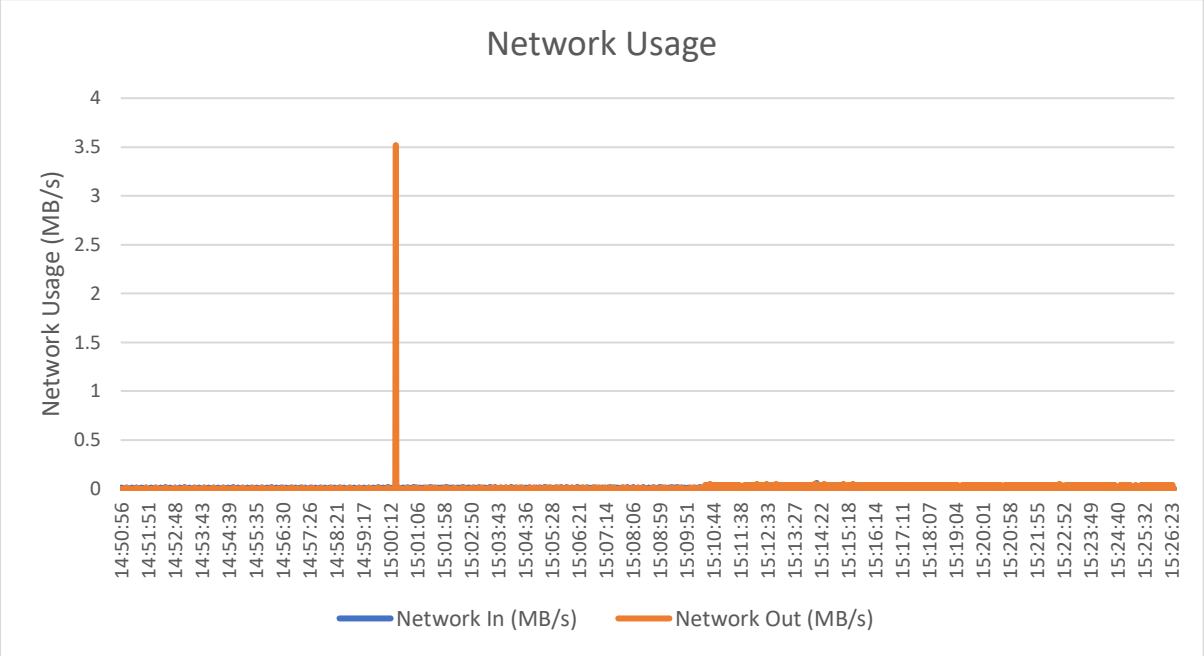

Figure 14: Network Consumption (1 hour batch of 10ms data)

If no new model is being trained, the graph looks like as shown in Figure 15, where it's all random fluctuations, plus the peak in the middle is the downloading of the model from the Model Registry.

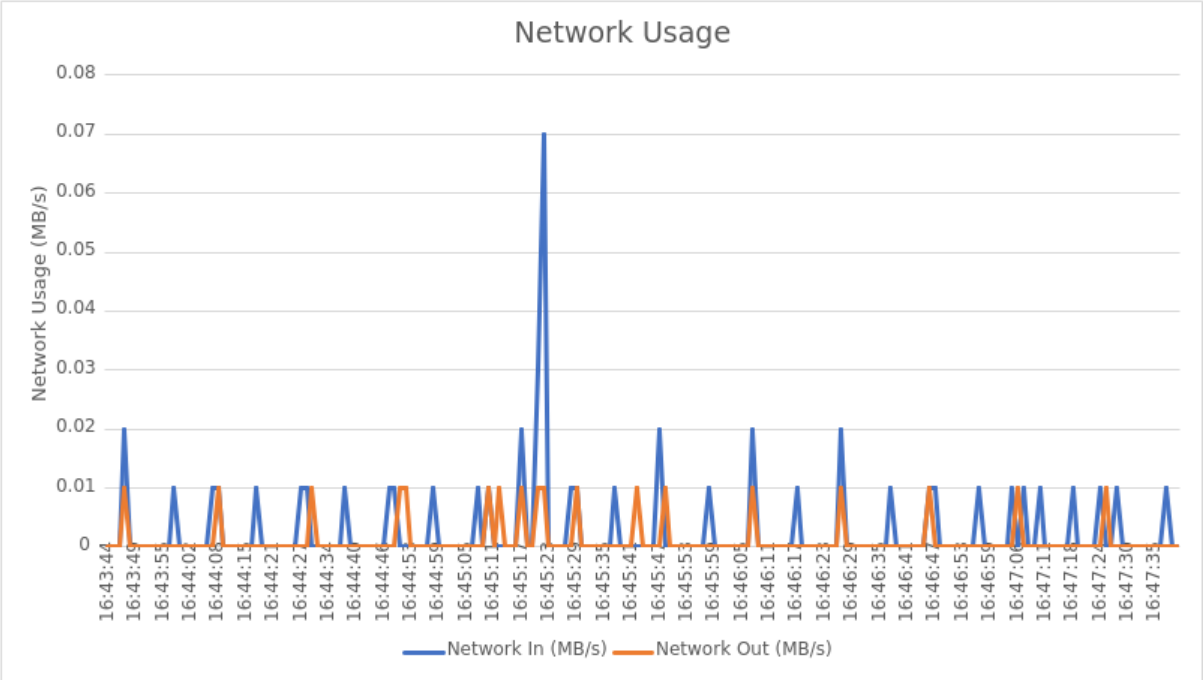

Figure 15: Network Consumption in Pipeline without Model Training

5.4 Evaluation

The main advantage compared to the cloud pipeline is highlighted in the network usage. The only network usage for the pipeline in normal working condition uses about 120kb, to download the model. Every other stage requires nothing. The network usage of the cloud pipeline will be much more, just for the input data upload of 20MB (Size of 1 hour of 10ms data) it takes about three to four mb after compression, and there must uploads and downloads each of the stages among many other sources of network usage. It is not possible to show actual network usage graphs for docker containers because the data inputs to the tasks defined in the ECS clusters, where the containers are deployed is over a private VPC, which means that a lot of the network traffic incoming to the container is actually over the private network and not directly from the internet itself, so it is not displayed properly. The extremely low usage of our pipeline makes it viable to use the CytroConnect pipeline with very limited bandwidth.

The total time it takes for anomaly to be detected from the point it is created depends on the two processes. The first process is the batching, and the second is the actual anomaly detection pipeline. So, for example, if an anomaly takes places 15:05 with the setting of 1

hour batches, first the data would be batched from 15:00 till 16:00, and then it would sent to the pipeline to be processed and only after the anomaly detection stage is completed for this batch would an anomaly be detected. So the time taken for the first anomaly to be detected would be one hour plus however long is required for the pipeline.

Compare the total memory usage in Figure 8, which uses about 55% to 60% of memory vs Figure 9, which has 40% to 45%, and also takes 8 mins for the pipeline to finish in comparison to 25 minutes for the 1-hour batch. When having this isolated case it would actually mean that it is more beneficial to use the big batch because it would take 25 minutes instead of 32 minutes to finish processing one hour of data, but in a real-world use case, it can safely assumed that there will always more than one component for an application, this would mean that, it is only allowed to process a maximum of 2-3 components safely through the pipeline at the same time, and since ctrlX Core is blocked from processing any other batch through the pipeline or 25 or so minutes, the fastest time to get CHI score for an average Application with 5 or 6 components will be 2 hours after the data point is initially generated. Where one hour is the batching time and one hour is the processing time for a batch.

In contrast, assuming the use of a 15-minute batch, since the processing of a batch is completed in 8 minutes, that makes it possible for anomaly detection after just a maximum of 23 minutes. So, this would mean that in a multi-component setting, this would be the most advantageous configuration.
Although, if there is an application with just one component, then it would still be the best to the 1-hour batch because it does compute more efficiently as referenced in Table 2.

6. Conclusion and Next Steps

6.1 Conclusion

In conclusion, the aim of the thesis was to execute the cloud-based pipeline and get some "MHI" scores on our edge device, which was accomplished. The core code and the stage separation principles remain the same, but the data transfer has been delegated completely to the Data Layer. The stage functions have been made passive, i.e., they only react to data provided to them, there is no explicit data sourcing present in the stages themselves. The orchestration for the whole pipeline has been massively optimised, it has been converted to an event-based architecture using the datalayer, so there is no need of an explicit orchestration software causing extra overhead. Instead of having to create a new container instance to process a different component, it can simply be done by adding an extra entrypoint-node from the datalayer. Same goes for new batches of data, the containers are designed in a way to trigger the stage process as soon as it detects incoming data onto its entrypoint-node. This reduces the massive overhead required by orchestration software significantly. The whole pipeline is capable of running on extremely low bandwidth, with only a single step requiring any actual network bandwidth, everything else is done locally. This proof of concept serves as a good indicator for further development of the CytroConnect stack on edge platform.

6.2 Outlook

Since this was only a proof of concept, there is room for improvement in many aspects. Starting with the optimisation of the containers, so each of them have minimum load while idle, as shown by the preprocessing stage, instead of being at high load at all moments.

Another thing to improve the performance of the snap is to remove its dependency on TensorFlow, which is very resource intensive and converting the models to using a lighter library like TFLite. This will reduce the resource usage by the anomaly detection and thus the whole pipeline significantly. Another way of local training could be to use different

ctrlX Cores with hardware accelerators , which are much better suited to training models on edge.

Adjustments can also be made to the current pipeline, like using a local model registry, and assuming that computationally inexpensive models are available, the training could also be done locally, to convert the pipeline from a hybrid solution to a completely local solution.

The final implementation can be the automatic creation of all the number of required entrypoint-nodes depending on the components in all of the stages, just from the yaml config file, which is already accessible.

The data is currently not being stored anywhere beside in the datalayer for data transfer between the stages. For a proper production implementation, an actual datastore must be implemented, this could just be stored using parquet files on the local file system or having a local instance of Influx DB on the ctrlX, snaps for which are already available in the ctrlX online community.

These represent many possible avenues for improvement and an exciting prospect for this project's future development.

References

[1] Google Cloud. “Advantages Of Cloud Computing | Google Cloud.” https://cloud.google.com/learn/advantages-of-cloud-computing (accessed Aug. 12, 2023).

[2] AWS, "Overview of Amazon Web Services - AWS Whitepaper,", early access.

[3] Microsoft. “What Is Cloud Computing? | Microsoft Azure.” https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing#Benefits (accessed Aug. 12, 2023).

[4] S. Chen *et al.,* "Internet of Things Based Smart Grids Supported by Intelligent Edge Computing,", *IEEE Access*, vol. 7, pp. 74089–74102, 2019, doi: 10.1109/ACCESS.2019.2920488.

[5] X. Wang, A. Khan, J. Wang, A. Gangopadhyay, C. Busart, and J. Freeman, "An edge–cloud integrated framework for flexible and dynamic stream analytics,", *Future Generation Computer Systems*, vol. 137, pp. 323–335, 2022, doi: 10.1016/j.future.2022.07.023.

[6] A. Morgunov, "The Life Cycle of a Machine Learning Project: What Are the Stages?,", *neptune.ai*, 21 Jul., 2022. https://neptune.ai/blog/life-cycle-of-a-machine-learning-project (accessed: Nov. 26, 2023).

[7] Docker. “What is a Container? | Docker.” https://www.docker.com/resources/what-container/ (accessed Aug. 15, 2023).

[8] Snapcraft. “The snapcraft.yaml schema | Snapcraft documentation.” https://snapcraft.io/docs/snapcraft-schema (accessed Aug. 15, 2023).

[9] Apache Airflow. “DAGs — Airflow Documentation.” https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html (accessed Nov. 14, 2023).