Final Report

Project code: 2320-0010

Smart Hybrid Quantum Task Scheduler for resource management, scheduling, operation and maintenance of on-premise quantum computer systems

Submitted by

Lin Xueshun, A0274618W
Li Shenghua, A0274682U
Chen Bihan, A0274496M

Supervised by

Bharadwaj Veeravalli
College of Design and Engineering
Department of Electrical and Computer Engineering

Project Sponsor

Institute of High Performance Computing (IHPC), (A*STAR)

Submitted on

14/06/2024

Table of Contents

1. Introduction

This project consists of two main components: a quantum-classical hybrid resource management system and an advanced maintenance framework for monitoring the operational status of quantum computers through IoT sensor data. The initial phase involved designing the database schema and back-end structure for the resource management system. Concurrently, an in-depth analysis of the IoT data available for the quantum computer maintenance system was performed, comparing and evaluating various anomaly detection and forecasting models. The two subsystems work collaboratively, with the resource management system using signals from the anomaly detection system to intelligently reschedule classical and quantum tasks. And finally came out this comprehensive Smart Hybrid Quantum Task Scheduler.

2. Project formulation

2.1 Executive Summary

This project aims to develop a task scheduling and distributed computing node simulation system, while also designing algorithms to monitor the state of quantum nodes. As shown in the diagram, the system is divided into two major parts: task scheduling and anomaly detection.

In the task scheduling part, users upload tasks (quantum tasks, classical tasks, and hybrid tasks) through the front end. User information is stored in a MySQL database, while tasks are stored in Redis. Tasks are then entered into the Tasks Scheduling and Management module, where they are allocated to different computing nodes (QPU and CPU) for computation. The results are returned to the front end for users to view. The Computing Cluster Management module manages both types of clusters.

For QPU, temperature sensors and pressure sensors are installed in the machine room to monitor data. The time-series data is uploaded to a Time Series Database for storage. The data is then sent to the Anomaly Detection System for detection and judgment. If an anomaly is detected, it triggers the Maintenance Mechanism for QPU self-check and fault recovery. The Anomaly Detection System sends the state of the QPU to the Computing Cluster Management to better manage the clusters.

The detailed structure of this project is shown in fig.1 below:

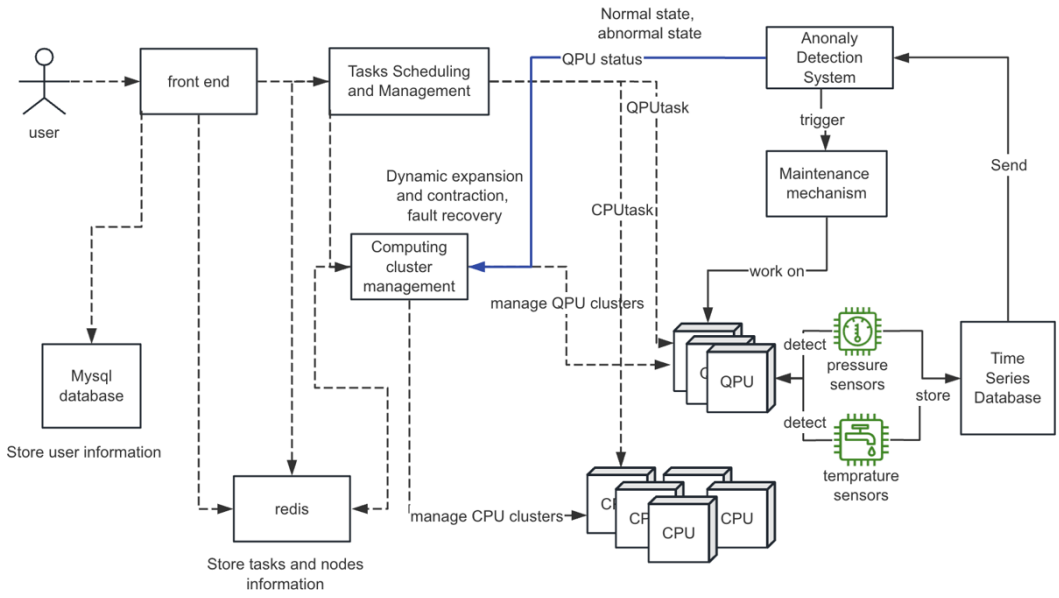

Figure 1: Overall system structure

2.2 Project Adaptations

Originally envisioned to leverage cloud-based quantum computing resources, this project aimed to revolutionize the integration and utilization of quantum and classical computing resources within a hybrid cloud environment. However, the availability of an on-site quantum computing resource provided by A*STAR prompted a significant shift in the project's focus. Consequently, the project's scope has evolved from a hybrid cloud setup to an on-site deployment model. This transition includes the development of a comprehensive

maintenance system designed to enhance the quantum computer's availability and improve its coordination with classical computing resources.

2.3 Project Framework Design

2.3.1 Front-End

A simple front-end interface would be leverage for this project, achieving the following functionalities:

- **Task Submission and Management:** Initial features will include interfaces for task upload and management, tailored to the needs of quantum computing resource users.
- **Monitoring and Configuration:** Developers and administrators will have tools for data analysis, system monitoring, and configuration management.

The expected front-end design is presented as fig.2 and fig.3 below:

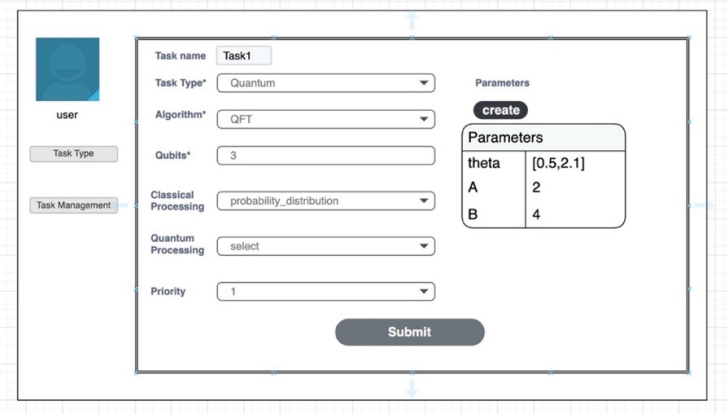

Figure 2: User Input Interface

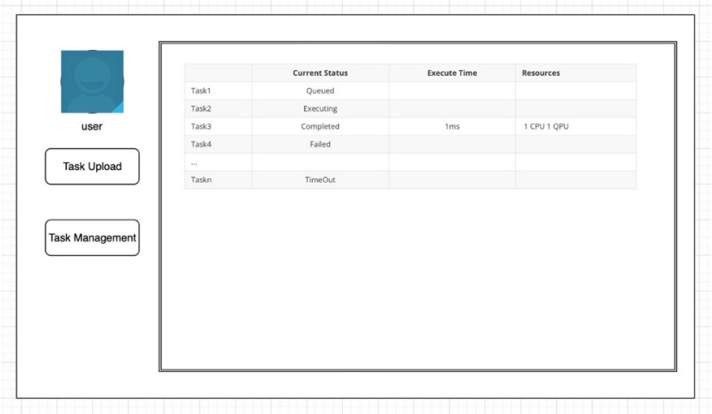

Figure 3: Task Management Interface

2.3.2 Python Side

The maintainance system would be wrote in python for the following responsibilities:

- **Input and Processing:** The system will process time-series sensor data for data cleaning, standardization, normalization, feature engineering, and anomaly detection by using machine learning.
- **Output:** Processed data will be formatted for the Java backend, including anomaly detection results denoting the quantum computer's status and availability.

2.3.3 Java Side

The resource management back-end would be developed in Java, will oversee the subsequent duties:

- **Input and Processing**: The Task Generator module generates different types of tasks (quantum tasks, classical tasks, hybrid tasks) with different execution times and statuses. Tasks are then submitted to the TaskScheduler for scheduling.
- **Resource Scheduling Strategy**: Strategies will include First-Come, Priority Queue, with dynamic adjustment based on quantum computer state.
- **Output**: The output design of the system primarily revolves around robust logging mechanisms to ensure comprehensive tracking and monitoring of task and node activities.

2.3.4 Backend Database Design

This Entity-Relationship (ER) diagram (fig.4) represents the database schema designed to manage quantum computing resources, user interactions, and task assignments.

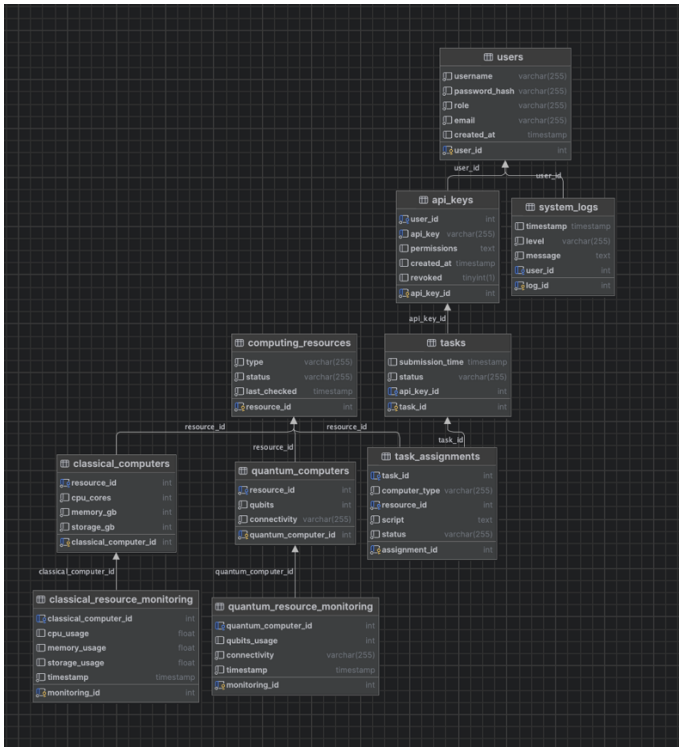

Figure 4: database schema ER diagram

A crucial security feature in this database design is the incorporation of an "API Keys" table, which effectively reflects our token system. This token system facilitates secure and controlled access to the quantum computing resources. The "API Keys" table is designed to store tokens associated with user accounts, where each token is represented by the api_key field. These tokens enable users to interact with the system through an API, providing a programmatic way to submit tasks, query resource status, and manage computing operations. Each API key is uniquely identified by an api_key_id and is linked to a user via the user_id foreign key. The permissions field within this table specifies the scope of actions that the token grants to the user, delineating what operations can be performed when using this API keyrevoked field.

3. Project schedule

Three Gantt Charts designed in different stages to represent the project management timeline

The initial Gantt chart(Fig. 5) was created at the onset of this project. Due to an incomplete understanding of the project requirements and nuances, this chart possibly included some unnecessary details and lacked certain steps.

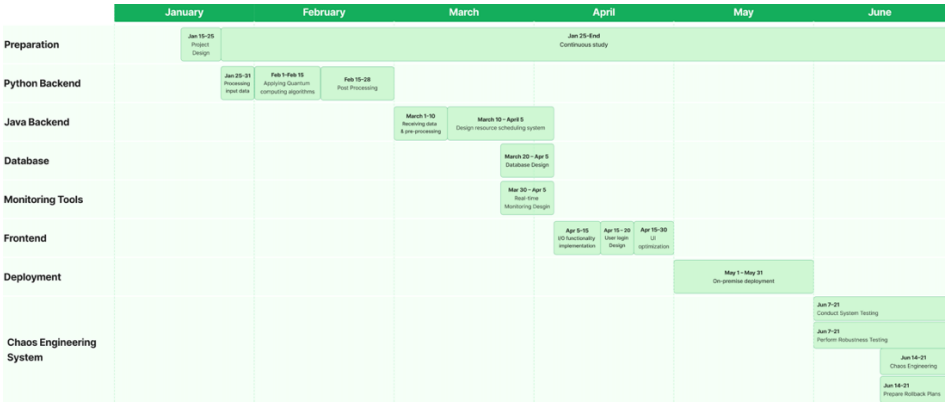

Figure 5: Gantt Chart at the beginning of the project

However, as the project progressed, shortcomings of the plan was realized. Consequently, based on progress and a deeper understanding, a second Gantt chart(Fig.6) was created at midest point of this project. And Fig.7 is the final description of the schedule spans for the whole project, some goals were changed due to the time limitation such as the front-end. Whereas some are added for the better system integration such as a forecasting model design.

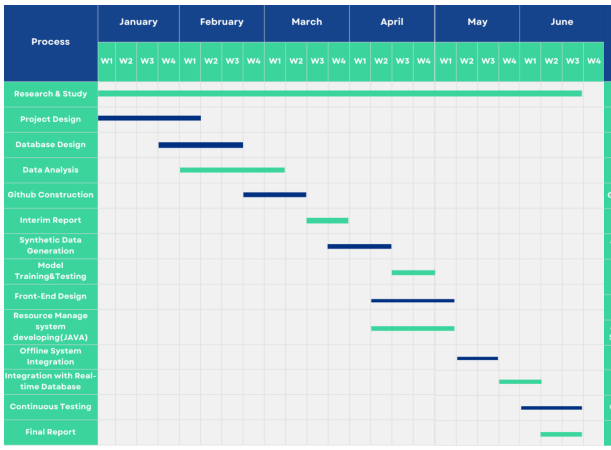

Figure 6: Gantt Chart at middle point

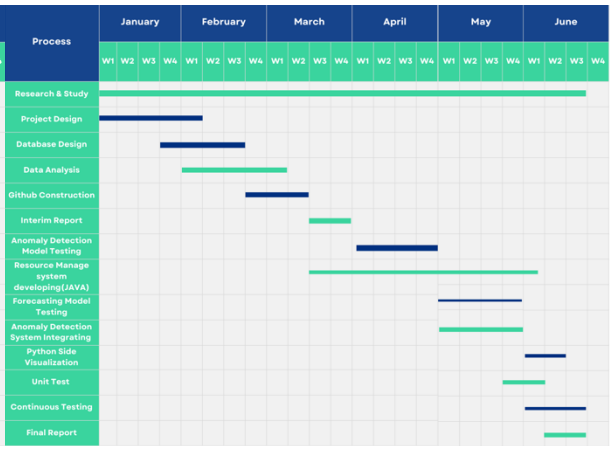

Figure7: Final Gantt Chart Schedule

4. Literature review

The emergence of quantum computing marks a significant leap in computational power, potentially addressing challenges beyond the reach of classical computing. However, as noted by M. Perelshtein et al. (2022), current quantum systems are limited and may require millions of qubits to achieve their full promise. A promising solution is the integration of quantum and classical systems in a hybrid model, a direction with increasing research interest.

Nguyen et al. (2023) introduced iQuantum, a cloud-based simulation toolkit designed to model hybrid quantum-classical computing environments, demonstrating its essential role in the development of quantum software and systems. iQuantum addresses the current limitations of quantum computing resources by providing an efficient platform for resource management strategy design. Its simulation capabilities enable researchers to validate their algorithms and system designs, thereby facilitating the practical application of quantum computing. Highlighting the importance of hybrid quantum-classical computing, iQuantum enhances computational capabilities and promises significant breakthroughs in various fields, including drug discovery and optimization problems.

Ahmed et al. (2023) delve into the critical integration of quantum accelerators with High Performance Computer (HPC) systems by developing of a sophisticated monitoring system employing Internet of Things (IoT) technology, which captures and analyzes environmental factors to enhance quantum computer fidelity. This proactive approach uses feature engineering and machine learning models to adaptively calibrate quantum computers based on environmental changes. Their research is not only enhancing the fidelity of quantum computing in data centers but also contributing to a broader understanding of the environment's influence on quantum processor units (QPUs). Such insights are vital for maintaining optimal performance of quantum computers in the complex, real-world settings

of HPC centers, highlighting the team's innovative steps in the quantum-classical convergence.

The aforementioned studies provide a robust theoretical foundation for the resource and maintenance systems pertinent to this project. However, the current project diverges notably in its application; it is oriented towards deploying a resource management system for an on-premises quantum computer resource, diverging from the cloud-centric environment associated with the iQuantum project. Furthermore, this project stands apart from Ahmed et al.'s work as it does not incorporate QPU fidelity data for optimization due to the limited availability of on-premises quantum resources and the inherent complexities of executing related experiments. As a result, this project will devise and apply custom strategies to tackle these specific challenges.

5. Analysis of industry and company

5.1 Industry Overview

The quantum computing industry is rapidly advancing, with significant investments from tech giants, startups, and research institutions. This sector is moving towards practical applications, focusing on solving real-world problems across various fields. Over the next decade, quantum computing is set to revolutionize industries by addressing complex challenges across automotive, logistics, finance, pharmaceuticals, energy, and chemicals. This technology is expected to optimize energy grids, expedite drug discovery, and improve complex system simulations like weather forecasting. The advancement of hybrid quantum-classical computing technologies is key to tackling current industrial application issues.

5.2 Challenges

- **Quantum Error Correction (QEC):** The importance of quantum error correction technology cannot be overstated, as it's crucial for maintaining the stability of quantum states during computations. IBM, a leader in the field, is at the forefront of developing new QEC codes that require fewer qubits, making quantum computing more practical and scalable (Mandelbaum, Steffen, & Cross, 2023).
- **Device Stability:** The physical realization of quantum computing faces significant challenges, including the coherence time of qubits, the accuracy of quantum operations, and the stable operation of quantum devices. Moreover, current quantum hardware, despite rapid advancements and remarkable progress in the field, still faces limitations in solving large-scale industry application problems independently.
- **Scalability:** The scalability of quantum computers remains a critical issue. Current systems face challenges in scaling up due to the complex requirements for maintaining qubit coherence and the intricate control needed for larger qubit arrays.

5.3 Future Trends

- **Hybrid Quantum Computing:** Unlocking commercial benefits by creating business value for clients through the integration of quantum and classical computing. This approach leverages the strengths of both systems to tackle complex problems more efficiently (Terra Quantum AG, 2022).
- **Advanced Optimization Algorithms:** Development of sophisticated algorithms to solve complex optimization problems faster, enabling quicker decision-making processes. This includes enhancements in machine learning models and financial optimizations.
- **Monitoring and Calibration of Quantum Devices:** Implementing comprehensive systems for the real-time monitoring and calibration of quantum devices is crucial. This ensures that quantum systems operate optimally, maintaining the integrity of quantum states throughout computations.

5.4 Company Information

The Institute of High-Performance Computing (IHPC), a premier research institute under the Agency for Science, Technology and Research (A*STAR) in Singapore, specializes in computational modelling, simulation, and artificial intelligence (IHPC) (n.d.). Since its establishment in 1998, IHPC has emerged as a digital frontrunner, employing over 350 research scientists and engineers from diverse cultural backgrounds to address major scientific, industrial, and societal challenges. In the realm of quantum computing, IHPC is dedicated to harnessing quantum computers' potential to achieve practical quantum advantages in fields such as optimisation, machine learning, chemistry, finance, and healthcare. By adopting a hardware-agnostic, full-stack approach and collaborating closely with the Centre for Quantum Technologies (CQT) at the National University of Singapore and the National Supercomputing Centre (NSCC) Singapore, IHPC contributes significantly to Singapore's National Quantum Computing Hub. This effort focuses on developing quantum computing capabilities and exploring applications through industry collaborations, aiming to bridge the gap between quantum-accelerated solutions and real-world problem statements optimally deployed on near-term quantum hardware.

6. Python side – Anomaly Detection System

6.1 Dataset Description

The dataset utilised for this project was sourced from IoT sensors affixed to Quantum computers within the laboratory. Communication was consistently maintained with the laboratory technicians throughout the project's duration. The data from these IoT sensors were aggregated in real-time within an InfluxDB database, noted for its specialisation in handling time-series data. For the purposes of this research, the technicians retrieved and sent three distinct databases, each spanning different time periods (2024/01/11 to 2024/01/19, 2023/09/26 to 2023/10/04, and 2023/08/31 to 2023/09/23), and stored them in a local SQLite database.

All three databases are sharing the same structure and is shown in the following diagram(fig.8) :

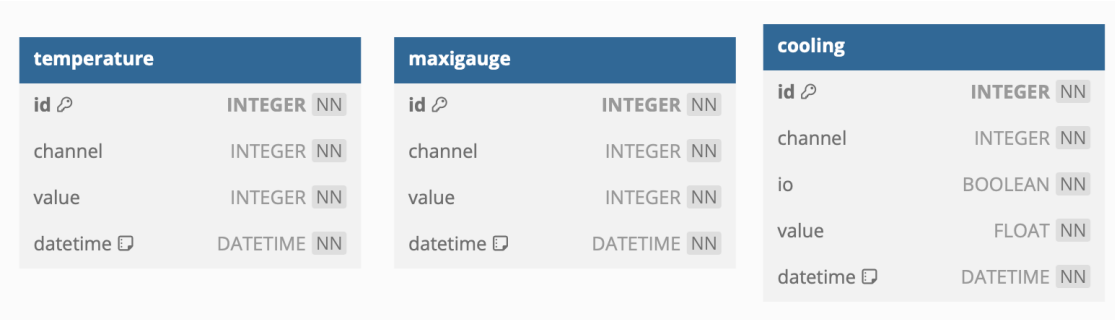

Figure 8: Database Structure

Figure 6 illustrates that each database comprises three types of datasets: temperature (inside the fridge), maxigauge (indicating pressure data), and cooling (refering to the cooling water temperature). Each dataset includes a unique identifier serving as the primary key, a channel to denote the sensor's location, a value representing the direct measurement, and a datetime stamp marking the moment the measurement was recorded. Additionally, there is an extra

"io" column within the cooling datasets, which signifies the input/output channel associated with the cooling pump.

Lists of channel numbers alongside their corresponding channel names are displayed in table below:

temperature		Maxigauge(pressure)		Cooling	
1	PT1	1	VaccumCan	0	Compressor1
2	PT2	2	PumpingLine	1	Compressor2
5	Still	3	CompressorOutlet		
6	MXC	4	CompressorInlet		
		5	MixtureTank		
		6	VentingLine		

Table 1: Channel Information Details

Different channels represent the various locations on the quantum computer where sensors are attached, e.g., "MXC" denoting the Mixing Chamber, a key part of dilution refrigerators used to reach the ultra-low temperatures required for quantum computing as stated by Zu, Dai, and de Waele (2022).

Python scripts were also developed and included in the Appendix for creating the database schema, importing datasets from all databases, and subsequently merging all data into a unified dataset.

The summary statistics(fig.9) of the unified dataset were generated for understanding the dataset, and box plots(attached in appendix) of each column also created for reference.

	MXC	PT1	PT2	Still	VaccumeCan	PumpingLine	CompressorOutlet	CompressorInlet	MixtureTank	VentingLine	Compressor1 Input	Compressor1 Output	Compressor2 Input	Compressor2 Output
count	111308.000000	111308.000000	111308.000000	111308.000000	111308.000000	111308.000000	111308.000000	111308.000000	111308.000000	111308.000000	111308.000000	111308.000000	111308.000000	111308.000000
mean	4344.972470	77223.887314	47069.206525	49612.422228	0.677389	1.447109	474.204652	613.025146	240.385987	27.220085	18.456965	27.321137	18.549016	26.282779
std	16502.949537	85905.819608	91759.954806	94184.002454	25.235829	21.624272	331.219764	185.387634	336.136158	155.025638	1.578901	5.129376	1.648024	4.686868
min	0.000000	35593.000000	3083.690000	725.326000	0.000004	0.000501	-3.360000	18.400000	3.870000	0.008440	15.386110	15.388330	15.398890	15.352220
25%	9.020200	35874.200000	3131.080000	899.378000	0.000006	0.022300	-0.748000	638.000000	5.220000	0.695000	17.895000	28.085560	18.041670	27.085550
50%	11.552800	36064.400000	3141.760000	1652.870000	0.000006	0.022700	642.000000	695.000000	5.940000	1.790000	18.563330	29.232220	18.658330	27.969440
75%	15.208300	52888.400000	17855.300000	40944.400000	0.000011	0.023500	689.000000	712.000000	722.000000	3.180000	19.288330	29.708890	19.329440	28.456670
max	101971.000000	294546.000000	295307.000000	295659.000000	1000.000000	524.000000	1890.000000	1400.000000	738.000000	1000.000000	29.307220	51.294450	30.452220	53.442780

Figure 9: Summary statistics

6.2 Data Analysis

Taking the period from 2024/01/18 to 2024/01/19 as an example for the explaination, some sample visualizations would be given as follows, plotting in log scale (so observing fluctuations at low value ranges clearly) and shifted (by dividing the first element to every element in the series) so all signals aligned with each other.

The cooling water temperature plot is visualized below and expain the event happened.

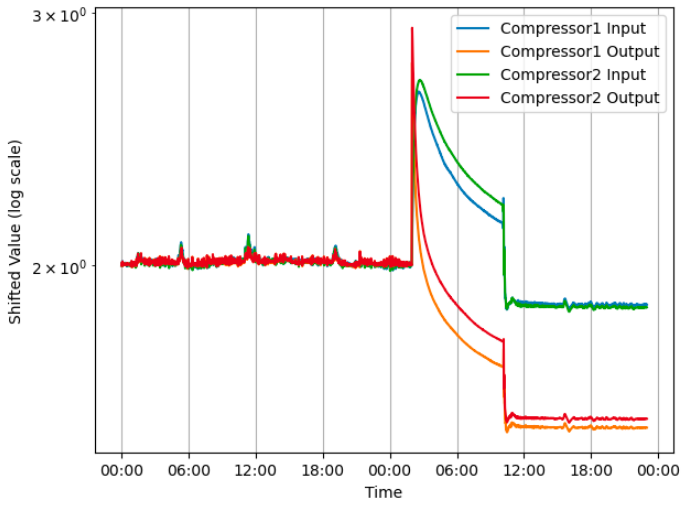

Figure 10：Variation of cooling water temperature

Here we can see a spike at around 2024/01/19 1 o'clock AM, which is caused by a known cooling pump faliure as told by the technician.

Upon examining the temperature data(fig.11), it was noted that the temperature across all channels experienced a rise precisely at the moment of pump failure. The MXC value decreased to 0 towards the end, which is attributed to its upper limit of 100K; any value exceeding this threshold is reduced to 0. In contrast, other channels did not exhibit a pattern of returning to normal levels, indicating that the issue with the cooling pump was not resolved at the end. A small spike at aroud 1PM on 18th is also occurred, due to the operation of a RF switch.

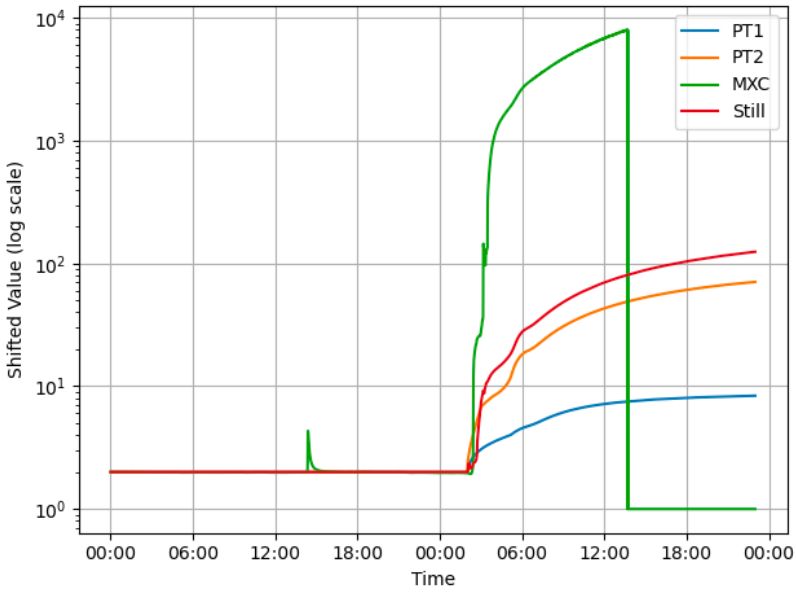

Figure 11: Variation in temperature

The pressure data(fig.12) of different channels seem like don't follow a similar pattern. The VacuumCan's value is continuelly rising, however, all other channels keep constant to a fixed value at the end. It’s still struggling to find out what’s happening to the pressure data.

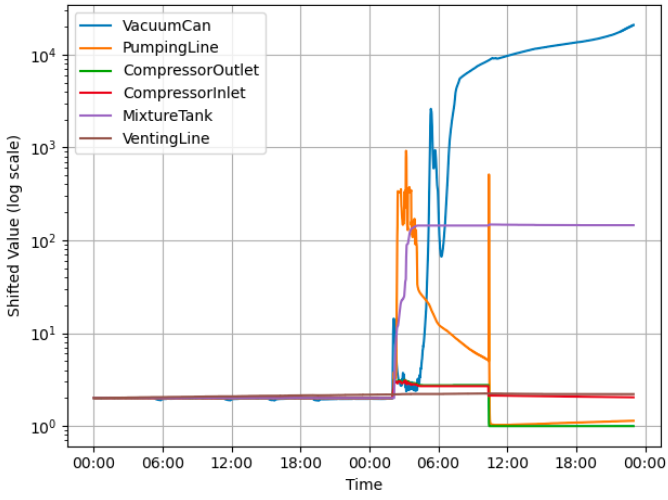

Figure 12: Variation in pressure

The Fig.13 illustrates the overall temperature change during the full operational cycle, taken from the example period from of the Quantum Computer, from startup to shutdown. After consulting with the technician, the definition of normal data has been established: it typically

focuses on the MXC channel, where the flat and stable segment in the middle of the graph represents the normal data.

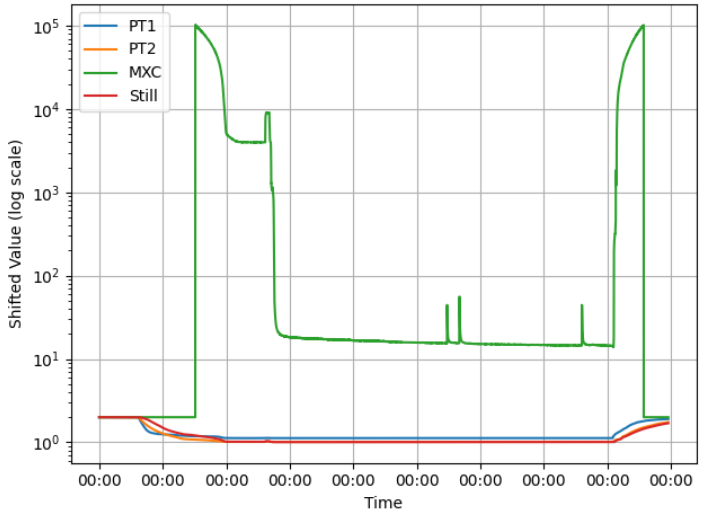

Figure 13: Full cycle overall temperature change

One thing to note is that since the sampling frequency of the different channels, the method of interpolation is used to ensure the separation of each time step is 30 seconds.

A correlation matrix(fig.14) was also derived from the unified dataset. It's surprising that some pressure channels, like VaccumeCan and PumpingLine, have almost no correlation with any other channels, alongside instances of highly correlated channels. These calculated correlation coefficients might prove beneficial for future work.

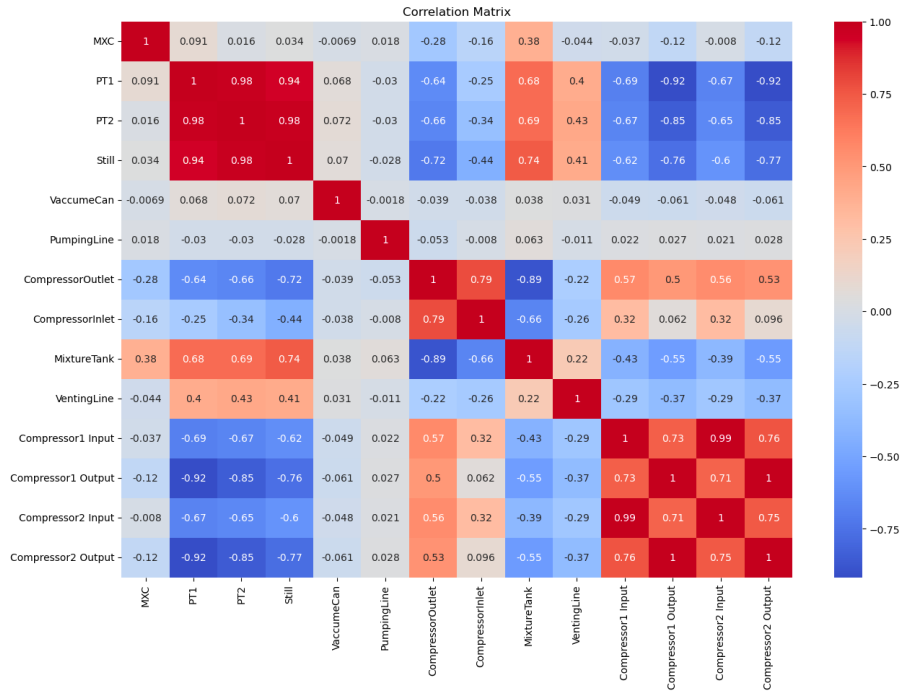

Figure 14: Correlation Matrix

To streamline the project while maintaining its original objectives, the focus is placed on the MXC channel, which is related to the definition of normal data, and two other channels most correlated with MXC: MixtureTank and CompressorOutlet.

6.3 Anomaly Detection

This section details the exploration journey of various anomaly detection models to find the most suitable fit for this kind of multi-variate IoT data anomaly detection. Explicit analysis of the results is also discussed.

6.3.1 Naïve Anomaly Detection Model

The exploration began with a naïve anomaly detection model. The core methodology of this model involves continuously calculating the gradient of the data. If the gradient shows an increasing trend, the algorithm notes this, and when the value exceeds a certain threshold, the

anomaly detection system reports the anomalies. Initially, all channels were considered, and based on the number of channels reporting warnings, the system would print out warning statuses of different levels. To denoise the data and smooth the overall curve for better gradient calculation, a moving average filter was also applied in this case.

A sample of the system printout is shown as below:

```
Time: 2024-01-18 14:18:53+00:00, Warning Level: Low
Time: 2024-01-19 01:51:57+00:00, Warning Level: High
Time: 2024-01-19 02:15:01+00:00, Warning Level: Medium
Time: 2024-01-19 02:55:08+00:00, Warning Level: Low
Time: 2024-01-19 12:54:58+00:00, Warning Level: Low
Time: 2024-01-19 13:24:03+00:00, Warning Level: Low
Time: 2024-01-19 13:43:06+00:00, Warning Level: Low
```

Figure 15: Warning message in detecting anomalies of temperature

The part of the implementation detail is attached below:

```
# Calculate the differences in temperature
 temperature_deltas = np.diff(temperatures)
 # Calculate the derivatives (rate of temperature change)
 derivatives = temperature_deltas / time_deltas
 # Detect points where the absolute value of the derivative exceeds the threshold
 abnormal = np.abs(derivatives) > threshold
 # Filter out consecutive abnormal points, only keeping the first in a series
 abnormal_indices = []
 for i in range(len(abnormal)):
     if abnormal[i] and (i == 0 or not abnormal[i - 1]):
         abnormal_indices.append(i)
 # Return the timestamps for points of abnormal temperature rise
 abnormal_times = [timestamps[i+1] for i in abnormal_indices]
```

This method is simple and straightforward but may prove challenging for this setting, particularly when dealing with more complex data. Implementing this naïve model requires pre-defining the gradient threshold and the appropriate moving average window size for each channel. Unfortunately, these parameters are often implicit and may require extensive experimentation to determine before setup.

6.3.2 Classification Model with Manual Labels

Machine learning analysis was conducted for another experimental purpose. Due to the absence of quantum computer fidelity data. It was hypothesised that the status of the quantum

computer correlates with the cooling water temperature at the beginning since only when the cooling pump works, can the QC reach the normal working temperature. Consequently, a dataset was manually labelled, assigning the quantum computer's status(not working/small error/normal/crash down) of a certain period based on the fluctuation levels of the cooling water temperature, as depicted in Figure 16.

	Start time	End time	Label
0	2023-08-31 00:03:34+00:00	2023-08-31 16:50:57+00:00	notworking
1	2023-08-31 16:50:57+00:00	2023-09-01 16:26:48+00:00	small error
2	2023-09-01 16:26:48+00:00	2023-09-02 03:03:50+00:00	normal
3	2023-09-02 03:03:50+00:00	2023-09-02 18:01:21+00:00	small error
4	2023-09-02 18:01:21+00:00	2023-09-04 02:02:54+00:00	normal
...	...	...	...
94	2024-01-18 12:52:21+00:00	2024-01-18 18:29:58+00:00	normal
95	2024-01-18 18:29:58+00:00	2024-01-18 20:33:36+00:00	small error
96	2024-01-18 20:33:36+00:00	2024-01-19 01:58:27+00:00	normal
97	2024-01-19 01:58:27+00:00	2024-01-19 11:05:03+00:00	crash down
98	2024-01-19 11:05:03+00:00	2024-01-19 23:41:26+00:00	notworking

Figure 16: Labelling of the status of the quantum computer

The status of the quantum computer at each timestamp was labelled using the previously mentioned dataset (fig.17). This dataset was then aggregated based on continuous status periods, and four features were extracted for each column(fig.18), which is mean, standard deviation, maximum, and minimum respectively.

	MXC	PT1	PT2	Still	VaccumeCan	PumpingLine	CompressorOutlet	CompressorInlet	MixtureTank	VentingLine	Compressor1 Input	Compressor1 Output	Compressor2 Input	Compressor2 Output	Label
2023-08-31 00:03:34+00:00	0.0	293078.0	293761.0	293753.0	0.020	0.08880	-1.44	150.0	710.0	998.00	15.53500	15.58111	15.52722	15.52444	notworking
2023-08-31 00:04:57+00:00	0.0	293078.0	293761.0	293753.0	0.020	0.08870	-1.44	150.0	710.0	998.00	15.53500	15.58111	15.52722	15.52444	notworking
2023-08-31 00:05:00+00:00	0.0	293077.0	293743.0	293714.0	0.020	0.08870	-1.44	150.0	710.0	998.00	15.53500	15.58111	15.52722	15.52444	notworking
2023-08-31 00:07:34+00:00	0.0	293077.0	293743.0	293714.0	0.020	0.08870	-1.44	150.0	710.0	998.00	15.56556	15.60444	15.56056	15.55056	notworking
2023-08-31 00:08:00+00:00	0.0	293073.0	293755.0	293698.0	0.020	0.08870	-1.44	150.0	710.0	998.00	15.56556	15.60444	15.56056	15.55056	notworking
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2024-01-19 23:40:10+00:00	0.0	266635.0	221527.0	207953.0	0.132	0.00337	-3.07	679.0	723.0	2.76	15.72500	15.76389	15.70222	15.67111	notworking
2024-01-19 23:40:26+00:00	0.0	266635.0	221527.0	207953.0	0.132	0.00337	-3.07	679.0	723.0	2.76	15.72278	15.75889	15.72500	15.69667	notworking
2024-01-19 23:40:49+00:00	0.0	266683.0	221597.0	208058.0	0.132	0.00337	-3.07	679.0	723.0	2.76	15.72278	15.75889	15.72500	15.69667	notworking
2024-01-19 23:41:11+00:00	0.0	266683.0	221597.0	208058.0	0.132	0.00337	-3.10	679.0	723.0	2.76	15.72278	15.75889	15.72500	15.69667	notworking
2024-01-19 23:41:26+00:00	0.0	266683.0	221597.0	208058.0	0.132	0.00337	-3.10	679.0	723.0	2.76	15.66111	15.70222	15.65333	15.62222	notworking

Figure 17: Unified dataset updated with status labels

	MXC_mean	MXC_std	MXC_max	MXC_min	PT1_mean	PT1_std	PT1_max	PT1_min	PT2_mean	PT2_std	...	Compressor1 Output_min	Compressor2 Input_mean	Compressor2 Input_std	Compressor2 Input_max	Compressor2 Input_min	Compressor2 Output_mean	Compressor2 Output_std	Compressor2 Output_max	Compressor2 Output_min	Label
0	0.000000	0.000000	0.0000	0.0000	292841.199463	118.279602	293079.0	292677.0	293512.638012	127.378589	...	15.53778	15.593608	0.044508	15.91555	15.50389	15.573225	0.043337	15.90000	15.47056	notworking
1	0.000000	0.000000	0.0000	0.0000	89461.399440	54745.254843	293039.0	53574.4	91833.354974	74316.304378	...	15.61722	19.096523	0.556487	20.77611	15.58667	28.988713	1.782011	30.68778	15.53500	small error
2	67917.493004	22280.606430	101899.0000	0.0000	51620.516975	1842.991690	53568.1	44737.8	16468.647335	3801.457257	...	29.48722	18.823113	0.138651	19.15222	18.47833	28.602906	0.204584	28.96166	28.02889	normal
3	6602.297281	6299.751238	33171.6000	4003.8100	36350.415356	1524.294281	44737.8	35609.7	3910.783365	997.447884	...	28.84555	18.410253	0.231968	19.85611	18.18000	27.738623	0.217624	29.04945	27.52111	small error
4	381.867785	1541.758568	9528.8300	14.4251	35831.753907	248.359680	37358.5	35701.2	3168.696470	325.367656	...	28.82778	18.392287	0.069354	18.60722	18.16722	27.802532	0.078367	28.15500	27.55444	normal
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
88	13.627745	3.477726	42.4182	12.5308	36047.598993	14.070926	36072.5	36015.3	3133.087503	3.789402	...	28.91778	18.233198	0.084377	18.43222	17.91778	28.104952	0.163309	28.75778	27.70389	normal
89	12.731520	0.054513	12.8499	12.6464	36047.684444	10.977629	36069.0	36024.4	3128.898333	3.887066	...	28.90222	18.212634	0.272686	19.01833	17.83834	28.034176	0.298834	29.18833	27.65000	small error
90	12.494573	0.110298	12.7875	12.2775	36005.894332	11.972238	36030.9	35985.9	3129.947166	3.580847	...	28.74778	18.040602	0.075438	18.28556	17.68167	27.887219	0.177135	28.96667	27.51556	normal
91	36804.059899	24481.840134	78846.9000	11.9076	136003.630678	48485.077821	213188.0	35987.3	58994.395358	34991.933630	...	15.38833	24.155561	3.686657	30.45222	15.56833	25.027078	6.011359	53.44278	15.37833	crash down
92	18910.829489	36933.479417	101835.0000	0.0000	247972.181818	14561.386387	266683.0	213188.0	178958.803256	28942.794255	...	15.45000	15.684419	0.078633	16.05444	15.42167	15.636323	0.069286	15.95389	15.36000	notworking

Figure 18: Resulted dataset after feature extraction

The feature dataset continued with standardisation, train test split, and finally trained and tested using the random forest algorithm. The whole progress and python scripts is attached in the appendix, and the results in the summary and confusion matrix is shown below:

	precision	recall	f1-score	support
crash down	0.00	0.00	0.00	1
normal	1.00	0.90	0.95	10
notworking	1.00	1.00	1.00	1
small error	0.78	1.00	0.88	7
accuracy			0.89	19
macro avg	0.69	0.72	0.71	19
weighted avg	0.87	0.89	0.87	19

Figure 19: machine learning result

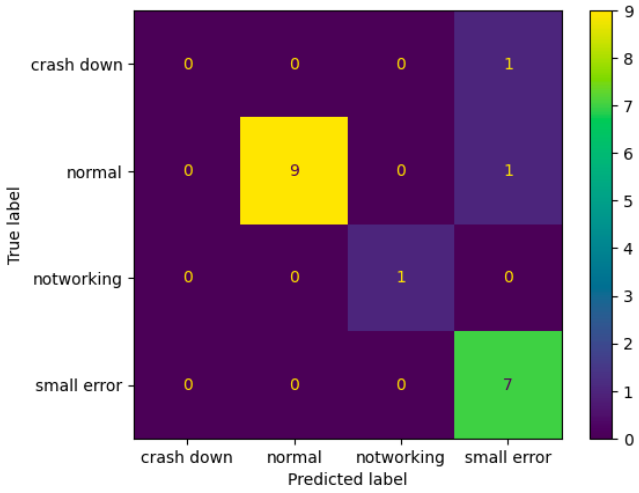

Figure 20: Resulted confusion matrix

The accuracy of this method is reasonably good, around 80%. However, the classification process requires manually labelled data, which is costly, especially when large volumes of new data are introduced. Additionally, this manually labelled data may not accurately represent the true status of the quantum computer. For instance, using the cooling water temperature as a reference for labelling has shown delays compared to changes in the temperature inside the fridge, which is more likely to be correlated with the actual status of the quantum computer.

6.3.2 Isolation Forest

Isolation Forest is an unsupervised machine-learning algorithm tailored for anomaly detection. Unsupervised means there's no need for the labels of the data, only the original data samples are required. Thus, the problem from the last method is solved as this project works with the dataset without labels. The Isolation Forest functions as an ensemble method like Random Forest, it combines the predictions of several decision trees to determine a final anomaly score for each data point. Unlike other methods, it directly learns the data distribution and immediately isolates anomalous points (Maklin, 2022). This approach is especially effective for identifying thresholds and distinguishing between normal and abnormal data. Moreover, it also supports multivariate time-series data and thus excellently fits this task's purpose. The mechanism details is demonstrated in Fig.21 ("Point-Denoise: Unsupervised outlier detection for 3D point clouds enhancement," 2024).

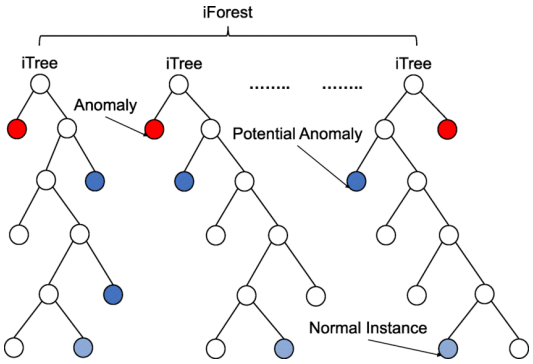

Figure 21: Principle of Isolation Forest Algorithm

This project utilized the Isolation Forest (IForest) model from the 'PyOD' library, which stands for Python Outlier Detection—an anomaly detection toolkit tailored for Python. Within the IForest algorithm, several hyperparameters can be adjusted to optimize the model's performance under various conditions. Notable among these are such as the "contamination" parameter, indicating the maximum percentage of data points classified as anomalies, and "n_estimators," which determines the number of trees in the ensemble,

usually 1000. Other considered hyperparameters also include max_samples and max_features, which denote the percentage of rows and the percentage of features for every tree to train, to avoid being overfitted.

To ensure a fair comparison across all models, including those introduced later, hyperparameter tuning was necessary. However, since there is no direct metric for evaluating an anomaly detection algorithm, this tuning was carried out using a supervised machine learning approach. The process involved the following steps:

1. An IForest model was initialized with a predefined set of hyperparameters and then evaluated to distinguish inliers (normal data points) from the dataset. This was achieved by fitting the model to the data and predicting labels, where inliers were assigned a label of 0.
2. A linear regression model was subsequently applied to the inliers. Given the multivariate nature of this project's dataset, one variable served as the target label, while the others were used as features. The dataset was then split into training and testing sets. The linear regression model was trained on the training set, and predictions for the target variable were made on the testing set. The performance was assessed using the root mean squared error (RMSE), with a lower RMSE indicating better predictive accuracy of the linear regression model, thus suggesting more effective anomaly detection by the corresponding IForest model.
3. This procedure was repeated for different sets of hyperparameters. For each combination, the IForest model was instantiated, inliers identified, and the RMSE calculated. The results were stored in a dictionary for comparative analysis.

All optional hyperparameters and the best result chosen are listed below:

contamination	0.01	0.05	0.1	**0.2**
n_estimators	1000	**1500**	1750	2000
max_features	0.4	0.6	**0.8**	1
max_samples	0.4	0.6	**0.8**	1

Table 2: Hyperparameters chosen for IForest. The best results are highlighted in bold

Building with the best hyperparameter selected, the IForest is then fitted and predicted based on the overall data for anomaly detection. Below is an Anomaly Detection example by IForest:

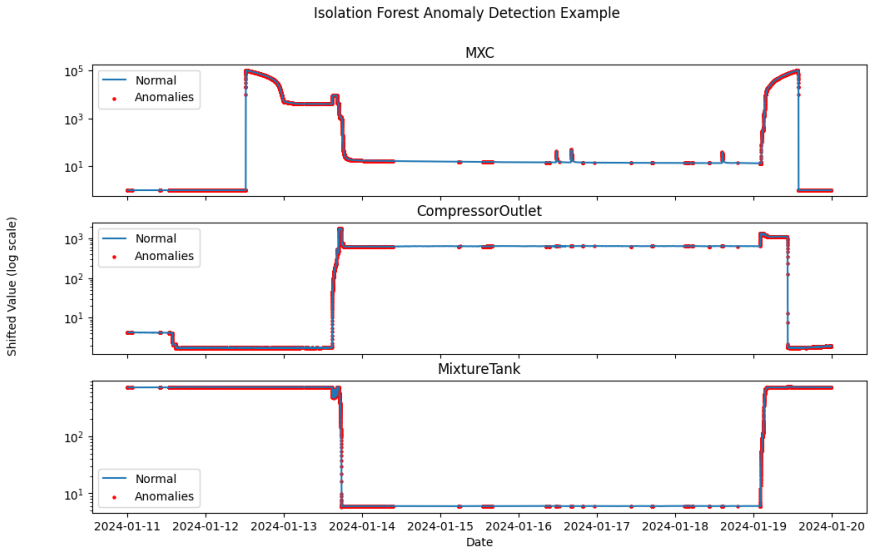

Figure 22: An IForest Anomaly Detection Example

Take the period from January 18, 2024, to January 19, 2024, as an example again. As mentioned before, normal data is defined as only the flat line in the middle of the graph. All other fluctuations or initial values are considered anomalies. The graph indicates that the IForest model performed generally well. It successfully identified most of the small fluctuations in the middle and the sudden increases and drops on either side of the graph. However, some initial values were mistakenly classified as normal, and some flat lines at the beginning and small amounts in the middle were incorrectly recognized as anomalies. This likely occurred because the IForest model does not inherently understand the definition of normal data. It performs anomaly detection based on the overall data distribution, which can lead to undesired results. Some values also appear normal if the definition is unclear, causing the model to make sometimes reasonable but incorrect detections.

6.3.3 Local Outlier Factor (LOF)

LOF stands out as another valuable unsupervised machine learning method, whereas it works quite differently from Isolation Forest. The LOF aims to pinpoint outliers by focusing on local neighbourhoods rather than the entire dataset distribution. It operates on a density-based principle, leveraging nearest neighbour search to detect anomalous data points("Local outlier factor," n.d.). A key benefit of LOF lies in its ability to detect outliers concerning the density of nearby data points within a local cluster. This localised approach enables more nuanced anomaly detection, allowing for identifying outliers that may not be apparent when using global outlier detection techniques.

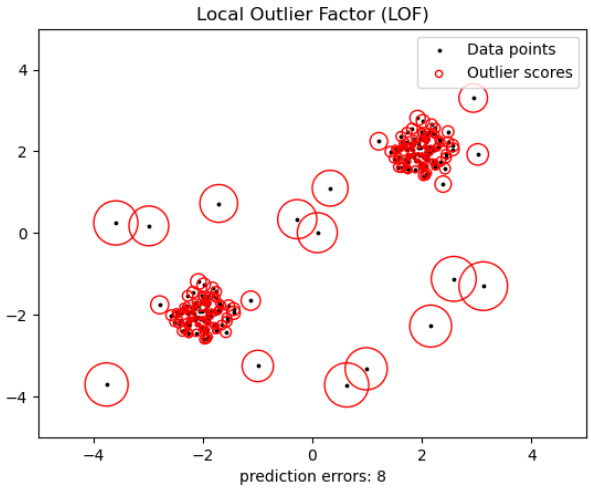

Figure 23: Principle of the LOF algorithm

As the demonstration graph shows above ("Outlier detection with local outlier factor (LOF)," n.d., fig23), the more a data point deviates from the cluster, the larger the red circle is drawn on it, which means a higher outlier score is obtained for that point.

The LOF model is also available in the PyOD library and will be utilised again. Before setting up and comparing the models, it is again necessary to tune the hyperparameters first. The considered hyperparameters for the LOF model include contamination, which is the

same as defined in IForest and indicates the expected percentage of outliers in the data. Unlike IForest, LOF uses local clustering for anomaly detection, and the n_neighbors parameter defines the number of neighbours in a cluster for computing the local density deviation of a given data point. The algorithm parameter determines the specific algorithm used to compute the nearest neighbours, while the metric parameter defines the distance metric used to calculate the distances between data points.

The method for tuning the hyperparameters of the LOF model follows the same procedures used for the IForest model. The entire process was repeated for LOF as well.

All optional hyperparameters and the best result chosen are listed below:

contamination	0.01	0.05	**0.1**	0.2
n_neighbors	20	200	**2000**	5000
algorithm	auto	ball_tree	kd_tree	**brute**
metric	euclidean	manhattan	**minkowski**	

Table 3: Hyperparameters chosen for LOF. The best results are highlighted in bold

Building with the best hyperparameter again, fit the LOF model and predicted the overall data for Anomaly Detection. Below is an Anomaly Detection example of LOF:

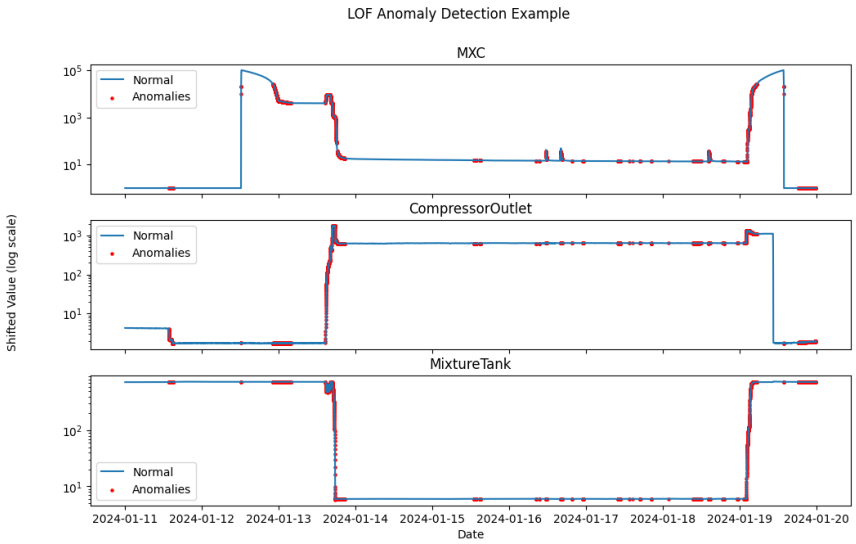

Figure 24: A LOF Anomaly Detection Example

For comparison purposes, the detection example for the period from January 18, 2024, to January 19, 2024, is considered again. Analyzing the graph, the LOF model successfully detects most of the fluctuations. However, it incorrectly classifies many of the flat initial values as normal, and some high-value flat lines are also incorrectly identified as normal. These results do not align with the defined characteristics of the dataset, indicating that further exploration is necessary.

6.3.4 Autoencoder

The Autoencoder is a specific type of neural network that employs unsupervised learning to replicate the patterns inherent in its input data. It encompasses two main components: an encoding function, which converts the input data into a lower-dimensional representation, and a decoding function, tasked with reconstructing the input data from the encoded form(Patel, 2018). The autoencoder serves various purposes, including noise reduction, dataset augmentation, feature extraction, and, notably, Anomaly Detection, by calculating the reconstruction loss from the decoded information. Given its reliance on Unsupervised Learning principles, the autoencoder seamlessly integrates into the comprehensive investigation of this project.

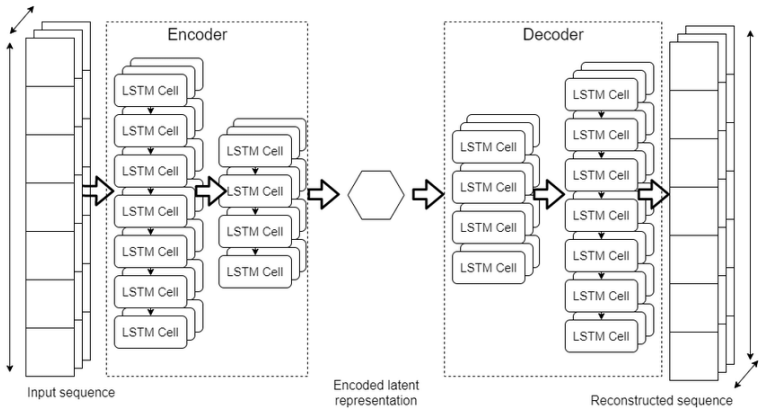

Figure 25: Neural Network Architecture of the Autoencoder

Similar to the diagram("Unsupervised marine vessel trajectory prediction using LSTM network and wild bootstrapping techniques," 2024, Fig. 25) shown above, the LSTM layer is leveraged for the Autoencoder design for this project because it is proficient in capturing temporal dependencies and patterns, making it particularly well-suited for processing time series data.

It's essential to recognize that although the autoencoder operates as an unsupervised learning algorithm, it necessitates the extraction of normal data to effectively learn the typical patterns across the entire dataset. Fortunately, for this project's dataset, discerning normal data is relatively straightforward, as it consistently appears continuously within the middle sections of each dataset piece, as described in the data analysis section. This requirement is significantly more feasible than manually labelling the data.

Before constructing the Autoencoder, it is crucial to preprocess the data with two important steps: creating fixed-size window slices for the time-series data and applying a normal data scaler to each window slice. After this preprocessing, the dataset is divided into training, validation, and test sets. The training set is used to directly train the model. The validation set, which the model has not seen during training, is used to evaluate the model's performance at each epoch to prevent overfitting; the best parameters are fine-tuned based on the validation loss. The test set, which remains untouched during training, is used to evaluate the overall performance of all models. The key implementation details are shown in the code snippet below:

```
def create_sequences(values, time_steps=TIME_STEPS):
    output = []
    # cut the data into pieces with length of time_steps
    for i in range(len(values) - time_steps + 1):
        output.append(values[i : (i + time_steps)])
    return np.stack(output)

def preprocess_single_dataframe(df):
    # ad_scaler is fitted on the overall normal data
    scaled_data = ad_scaler.transform(df)
```

```
    x_train = create_sequences(scaled_data)
    return x_train
# split train validation test data by setting different indices
x_train_list = [preprocess_single_dataframe(normal_data_list[i]) for i in train_indices]
x_val_list = [preprocess_single_dataframe(normal_data_list[i]) for i in val_indices]
x_test_list = [preprocess_single_dataframe(normal_data_list[i]) for i in test_indices]
```

The LSTM autoencoder model is constructed using the TensorFlow architecture. Before setting up the autoencoder, a warm-up cosine decay learning rate scheduler is pre-defined. This scheduler is crucial in deep learning because an inappropriate learning rate can significantly affect model performance: a rate that is too small may cause the model to get stuck in a suboptimal solution, while a rate that is too large can make the model unstable and difficult to converge, possibly further leading to the gradient vanishing or gradient disappearing problem. The learning rate scheduler automatically adjusts the learning rate during training. Specifically, the cosine decay learning rate scheduler used in this project adjusts the learning rate following a cosine curve as epochs progress. Additionally, a warm-up stage is included to allow the model to start training at a steady rate at the beginning of each cycle. Since this warm-up cosine decay scheduler is versatile for most cases, it would be mainly used for every deep-learning structure defined in this project. Fig. 26 is an example of the dynamic adjust learning rate variation curve created. Relative implementation is also attatched in the Appendix.

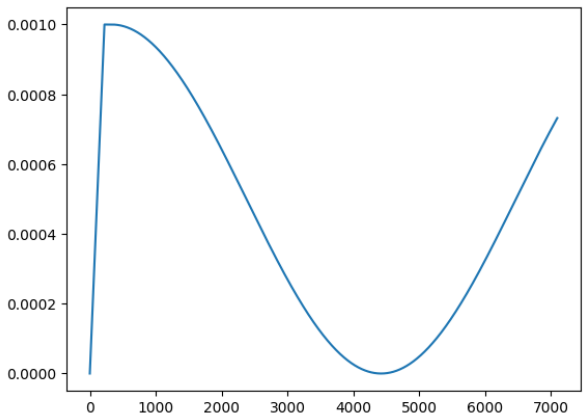

Figure 26: Warm-up Cosine Decay Learning Rate Scheduler Variation Example(Learning Rate VS. number of steps)

The Keras Tuner Hyper model was leveraged to tune the hyper-parameters of the LSTM autoencoder. To enable the best search, the model is defined flexibly using various hyperparameters, including:

1. *use_conv*: A Boolean value determines an optional convolution layer for potentially extracting important features from the input data, reducing dimensionality to simplify the data and decrease computational complexity, smoothing out noise to handle irrelevant variations, and capturing multi-scale features for a more comprehensive understanding of the data patterns before the LSTM processes the sequential aspects.
2. *gaussian*: A float determines the standard deviation of a Gaussian layer to potentially introduce controlled randomness or noise to the input data, which can help regularise the model and prevent overfitting. Enhancing the LSTM's ability to capture meaningful sequences and improving overall model performance.
3. *use_bidirectional*: A choosable bidirectional Boolean option for LSTM layers can possibly enhance the model's performance by processing the input data in both forward and backward directions. This allows the model to capture dependencies from both past and future contexts and may provide a more comprehensive understanding of the sequential data.
4. *num_lstm_layers_en & num_lstm_layers_de*: selectable integer number of LSTM layers, ranging from 1 to 3, respectively, for the encoding and decoding section, ensures the model can be tailored to specific problem requirements, balancing performance and computational efficiency.
5. *use_batchnorm*: Boolean choice of whether to use a batch norm layer after each encoder or decode layer. The normalisation may help accelerate the training process and reduce the sensitivity to the initialisation of network weights.
6. *lstm_units:* A selectable integer number of LSTM neurons. Similar to the functionality of adjusting the number of LSTM layers. A low number of neurons makes simple and faster training, and a high number of neurons increases the model's capacity to learn complex patterns. 32, 64, 128, and 256 are provided as the options here.
7. *dropout_rate1 &2*: Two dropout layers with adjustable dropout rates and may help improve the model's robustness and prevent overfitting. Dropout works by randomly setting a fraction of the input units to zero during training, which prevents the model from relying too heavily on any particular neurons.
8. *regularizer1&2*: Float L2 regularisation hyperparameter, which adds a penalty proportional to the square of the magnitude of the weights. Encourages smaller, more evenly distributed weights and helps improve generalisation by also reducing overfitting.
9. *clipnorm:* A float gradient clipping parameter used to prevent the problem of exploding gradients during training. It works by scaling down the gradients if their

norm exceeds a specified threshold. This helps stabilise the training process by ensuring that the gradient updates do not become excessively large.

10. *batch_size, start_lr, target_lr, epochs_per_period:* Other hyperparameters for defining the warm-up cosine decay learning rate scheduler like batch size, start learning rate, target learning rate, and epochs per period, where epochs per period denote the number of epochs over which the learning rate decays from the start learning rate to the target learning rate following a cosine schedule.

The experiment was defined for 100 trials, 500 maximum number of epochs for each trial, and 15 epochs as the number of patience with the tuning algorithm of Bayesian optimisation. The best hyperparameter was identified after the experiment and concluded in the following table:

use_conv	False	*lstm_units*	256
gaussian	0.12	*dropout_rate1*	0.1
use_bidirectional	False	*dropout_rate2*	0.1
num_lstm_layers_en	1	*regularizer1*	0.004247
num_lstm_layers_de	1	*regularizer2*	0.001699
use_batchnorm	True	*clipnorm*	0.3
batch_size	128	*start_lr*	4.5257e-05
epochs_per_period	90	*target_lr*	0.00043

Table 4: LSTM autoencoder best hyperparameter

Analysing these hyperparameters, both the encoder and decoder are configured with a single LSTM layer, indicating that a simple and fast model is optimal for this dataset. The convolutional layer is not used, possibly because its inclusion might degrade performance by not aligning well with the temporal nature of the data. The bidirectional setting is false, suggesting that increasing model complexity with bidirectional LSTMs could lead to overfitting. The model utilises 256 LSTM units, striking a balance between the capacity to learn complex patterns and computational efficiency. The combination of regularisation techniques, such as dropout rates, L2 regularisation, and gradient clipping,

indicates that there may be a tendency for the model to overfit during training, and these measures are in place to mitigate that risk.

The deployment result of this best LSTM autoencoder is shown in the graphs below:

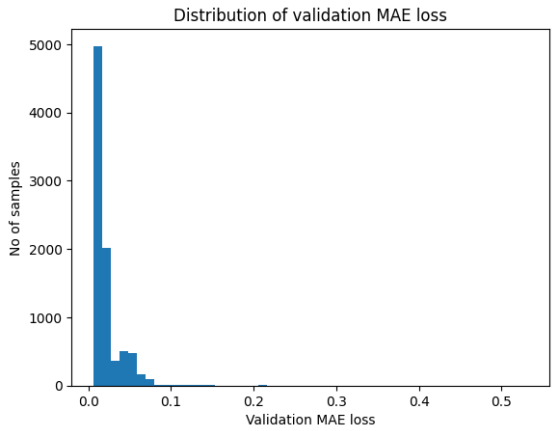

Figure 27: Distribution of the Validation Loss

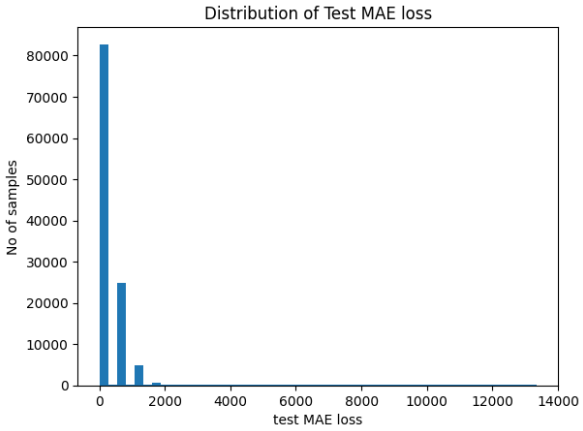

Figure 28: Distribution of the Test Loss

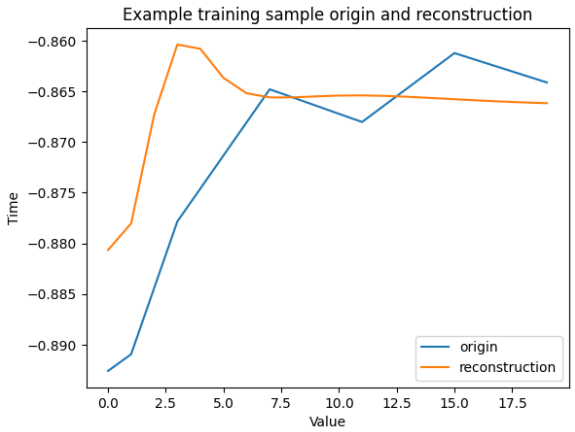

Figure 29: Normal Data Origin and Reconstruction

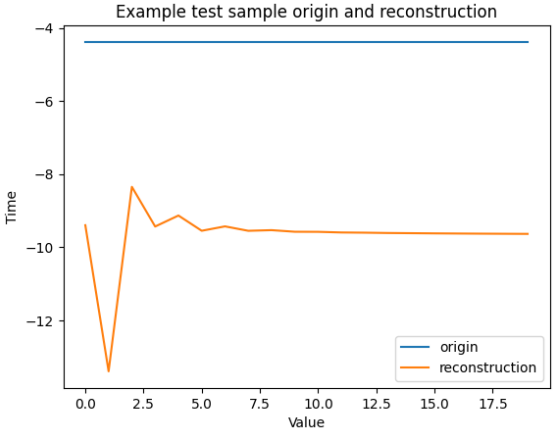

Figure 30: Abnormal Data Origin and Reconstruction

Comparing Figures 12 and 13, it is evident that the test data, which includes both normal and abnormal samples, exhibits a significantly higher reconstruction loss than the training validation data, which consists solely of normal samples. This indicates that the autoencoder has effectively learned the patterns and statistics of the normal data. When fed with normal data during testing, it reconstructs it with remarkably low loss, typically not exceeding 0.6. However, when abnormal data is introduced, the reconstruction loss can soar up to 14,000. Figures 14 and 15 provide further details: the training reconstruction data closely resembles

the original normal data, while the test data with an initial abnormal value shows a substantial difference from the reconstructed data, resulting in a high reconstruction loss. A threshold is set at 0.85 percentage of the maximum training loss, classifying any data with a reconstruction loss exceeding this threshold as abnormal. A coefficient of 0.85 is chosen to enhance the autoencoder's sensitivity and ensure that no anomalies are missed.

Similarly, a final anomaly detection example of test data by autoencoder is plotted below to compare with other models. The period is still chosen from January 18, 2024, to January 19, 2024.

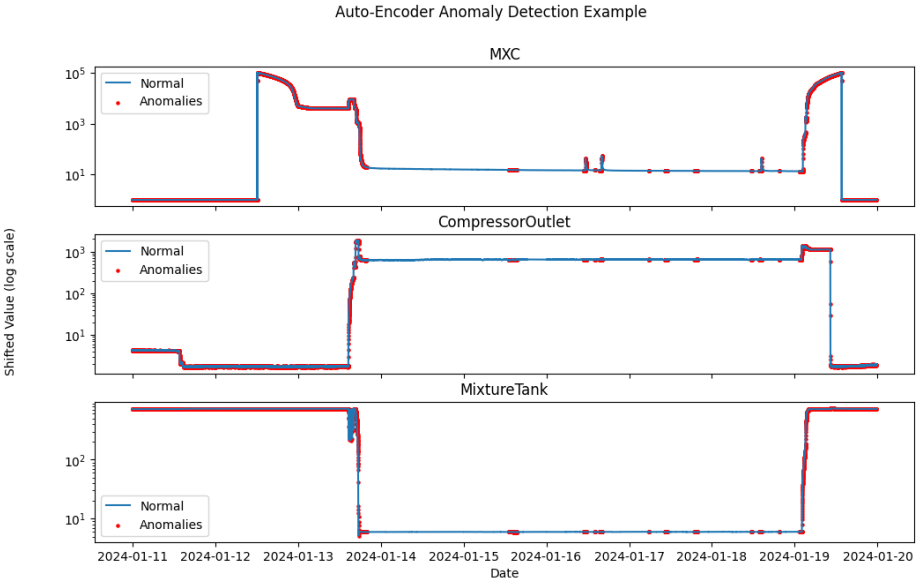

Figure 31: An Autoencoder Anomaly Detection Sample

This time, the autoencoder successfully detected almost all desired anomalies with high accuracy, making only a few minor mistakes. It effectively identified both large and small fluctuations, as well as initial abnormalities, correctly labelling them as anomalies. This success is attributed to the autoencoder learning patterns from the training set. Unlike other models that make decisions based on assumed data distributions, the autoencoder detects anomalies by comparing the test data to the learned characteristics of normal data.

6.3.5 Anomaly Detection Bench Mark

In classical machine learning, models are typically trained with features and labels, allowing for straightforward performance assessment using metrics such as recall, precision, and the F1-score, which combines the two. However, specific labels are often absent in unsupervised problems like anomaly detection, and the definition of abnormality can be ambiguous. While visualising results and manually reviewing them can help evaluate model performance, this approach can be problematic for audiences lacking the necessary background knowledge. Additionally, tasks involving complex or voluminous data may require significant effort for manual visualisation review. Therefore, a direct metric for evaluating anomaly detection models is essential. Fortunately, previous studies have successfully developed and validated meaningful evaluation metrics for this purpose. And that is the Excess-Mass(EM) and Mass-Volume(MV) introduced by Nicolas(2016).

While some aspects of the paper extend beyond this project's scope and aren't deeply explored here, the general concepts of Excess-Mass (EM) and Mass-Volume (MV) can be summarised as follows. Excess-Mass (EM) measures the difference between the total mass (or density) of data points above a certain threshold and the volume (or space) they occupy. It essentially captures how much of the data is considered anomalous beyond a certain point, emphasising regions with high density but low volume. Mass-Volume (MV), on the other hand, focuses on the smallest volume that contains a given proportion of the data's mass. It looks at how compactly the normal data can be enclosed, helping to identify regions where anomalies are likely to lie outside. By using these criteria, it becomes possible to evaluate the effectiveness of anomaly detection algorithms even in high-dimensional datasets where labelling is impractical.

Leveraging these two concepts and applying the author's method in this project, two metrics are obtained by generating two line plots, respectively, in the figures below. The naïve model and classification model did not participate in this evaluation as they are only partially completed.

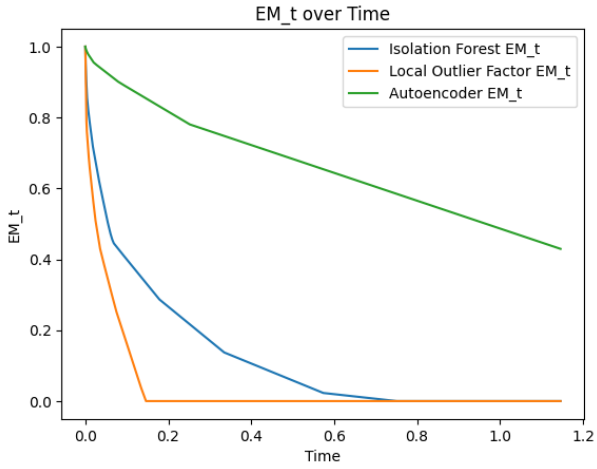

Figure 32: Excess-Mass Evaluation of Models

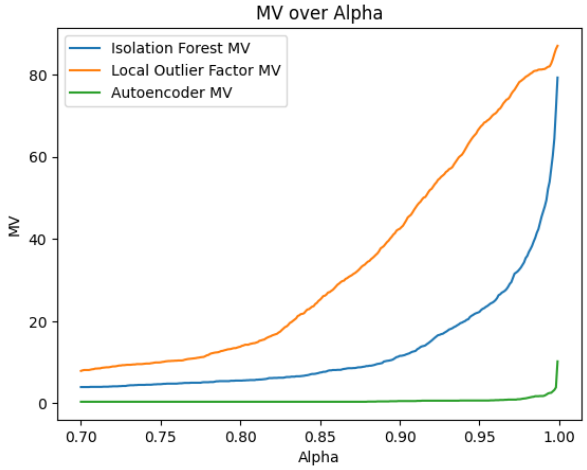

Figure 33: Mass-Volume Evaluation of models

The area under EM(t) and MV(t) forms the intended metric EM and MV. Specific values are presented in the table below:

	EM	*MV*
IForest	0.1378	3.8249
LOF	0.0435	10.0861
Autoencoder	0.7605	0.1827

Table 5: Performance of each Anomaly Detection Model

According to the paper's definitions, **higher EM values and lower MV values** indicate better performance in anomaly detection tasks. It is evident that the Autoencoder outperforms the other two algorithms based on these metrics, showcasing its exceptional and stable capability in anomaly detection.

6.4 Time-Series Forecasting

This section briefly explores the study of multi-variate time-series forecasting to accelerate the anomaly detection process. The goal is for the forecasting model to predict short-term future data, which will then be input into an anomaly detection model. This approach aims to detect anomalies before they actually occur. All details of the analysis and findings will be discussed in this section.

6.4.1 LSTM Forecasting Model

Similar to the LSTM autoencoder discussed in the anomaly detection section, the structure for forecasting also uses labels, but with a key difference: the training label is not the data itself. Instead, a horizon (a window of future forecasting steps) of 20 is sliced from the data and used as the label for the model, and training data is changed to 60 steps as the window size, to suit better for this fast-changing time series dataset, where one step represents 30 seconds. With only minor adjustments, the LSTM autoencoder can be adapted for forecasting, such as modifying the decoder shape to match the horizon size. Other procedures, like preprocessing, remain largely the same as in the anomaly detection part. Below is a code snippet that defines the functions used for the preprocessing steps in multi-variate time series forecasting.

```
def windowed_dataset(df, window_size, horizon):
    X = []
    y = []
    # iterate through the dataset and create the windowed dataset
    for i in range(len(df) - window_size - horizon +1):
        # data across the window size is the feature
        feature = [a for a in df[i :i+window_size]]
        X.append(feature)
        # next value after the window size is the label
        label = df[i+window_size:i+window_size+horizon]
        y.append(label)
    return np.array(X), np.array(y)

def train_val_test_split(X, y, train_size, val_size):
    # split the dataset into training, validation and test set
    X_train, y_train = X[:train_size], y[:train_size]
    X_val, y_val = X[train_size:train_size+val_size], y[train_size:train_size+val_size]
    X_test, y_test = X[train_size+val_size:], y[train_size+val_size:]
    return X_train, y_train, X_val, y_val, X_test, y_test
```

Same model but with different tasks, therefore the hyperparameter is tuned again to best suit the need of this forecasting task, similarly in 100 trials and 500 maximum epochs. The details of the chosen hyperparameters are listed in the table below:

use_conv	False	*lstm_units*	128
gaussian	0.14	*dropout_rate1*	0
use_bidirectional	True	*dropout_rate2*	0.1
num_lstm_layers_en	1	*regularizer1*	0.00074
num_lstm_layers_de	1	*regularizer2*	0.00289
use_batchnorm	False	*clipnorm*	0.7
batch_size	512	*start_lr*	1e-07
epochs_per_period	70	*target_lr*	0.00024

Table 6 : LSTM forecaster Best Hyperparameters

The selection of a single LSTM layer for both the encoder and decoder indicates that the dataset requires a simple and fast model. The activation of the bidirectional option suggests that the time series forecasting task is more complex than anomaly detection, necessitating the model to learn patterns from both directions. Additionally, the increase in clipnorm from 0.1 to 0.7 implies that the forecasting task requires more regularisation to prevent overfitting. Other parameters remain largely unchanged, likely due to the same training dataset being used.

6.4.2 N-Beats

N-BEATS stands for Neural Basis Expansion Analysis, different from LSTM forecaster, it is a highly successful **non-recurrent** neural network architecture for time-series forecasting. It operates using a series of basic building blocks, each containing a stack of only fully connected layers followed by a fork architecture with separate forecast and backcast layers. These layers generate forward and backward expansion coefficients to create the final forecast and backcast outputs. The architecture employs "doubly residual stacking," creating

a hierarchical structure with residual connections across layers for better interpretability and efficiency. Each block processes the input step-by-step, focusing on specific data components, and the partial forecasts from individual blocks are combined to form the overall prediction, similar to boosting ensembles(Filho, 2023).

This multi-stack with multi-block structure is demonstrated in the figure below(Dancker, 2024, Fig. 34)

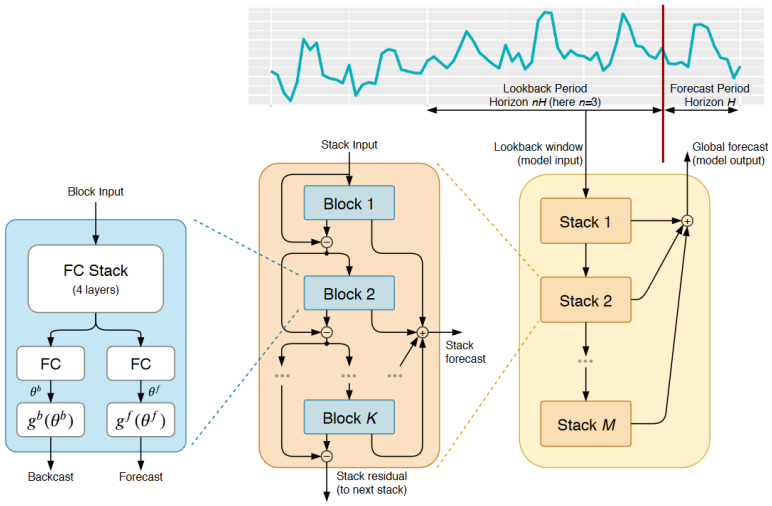

Figure 34: N-Beats Model's Architecture

The hypermodel is utilized again for the hyperparameter search of the N-Beats model.

Adjustable hyperparameters include:

1. *num_neurons*: Integer number of neurons refers to the individual units within each fully connected layer of the network. Similar to the lstm_units in the definition of LSTM forecaster. Options include 16, 32, 64, 128, 256.
2. *num_layers*: Integer number from 1 to 5 of fully connected layers within each basic building block of the network. Each layer consists of a set of neurons that process the input data sequentially. Having multiple layers in each block allows the model to learn and represent more complex patterns and relationships within the data.
3. *num_stacks*: Integer number of also 1 to 5 stacks of basic building blocks used in the network. Each stack consists of multiple building blocks, and these stacks are sequentially connected to form the overall architecture. This stacking mechanism enables the model to capture different levels of patterns and components in the time series data, improving the overall accuracy and robustness of the forecasts. Also ranging from 1 to 5.

4. *activation_func*: choice of mathematical function applied to the output of each neuron in a fully connected layer. It introduces non-linearity into the model, enabling it to learn and represent complex patterns in the data. Options include ReLU, Leaky ReLU, and ELU.
5. *dropout_rate*: The float dropout rate for the dropout layer after each fully connected layer. Used for regularization purposes. Which wasn't included in the original paper but is worth experimenting with for the small dataset used in this project.
6. *clip_norm, weight_decay, batch_size, start_lr, target_lr, epochs_per_period*: similarly defined here for the warm-up cos decay learning rate scheduler.

The experiment was conducted in 100 trials, with 5000 maximum epochs for each trial, and 200 for the number of early-stopping patients. The fine-tuned parameter result is listed in the table below:

num_neurons	32	*num_layers*	2
num_stacks	2	*dropout_rate*	0.05
Activation_func	leaky_relu	*clipnorm*	0.1
batch_size	128	*start_lr*	1.483e-05
target_lr	3.522e-03	*epochs_per_period*	170

Table 7： N-Beats Model Best hyperparameters

6.4.3 XGBoost

XGBoost, an efficient implementation of gradient boosting, is commonly used for classification and regression but can also be adapted for time series forecasting with the correct preprocess of the input data, directly available in the xgboost library. It builds an ensemble of decision trees sequentially, correcting errors at each step, and uses regularization to prevent overfitting. This scalable and fast algorithm is ideal for handling large datasets, making it potentially a powerful tool for predicting future values in time series forecasting.

Random search method is used to find out the best parameters for XGB Regressor, including:

1. *n_estimators:* Number of trees to be built in the model. A higher number generally increases model complexity and accuracy but also increases computation time and risk of overfitting.
2. *learning_rate:* Similar definition for the one in the neural network, float number of step size at each iteration while moving towards a minimum of the loss function.

Lower values make the model more robust but require more trees, while higher values can make the model faster but risk overshooting.

3. *max_depth:* This sets the maximum depth of each tree in an integer. Deeper trees can model more complex relationships but are more prone to overfitting. Shallower trees are simpler and more generalizable.
4. *min_child_weight:* This is the minimum sum of instance weight needed in a child. Higher values prevent overfitting by requiring larger sums of weights, leading to more conservative models.
5. *gamma:* This is the minimum loss reduction required to make a further partition on a leaf node of the tree. Higher values lead to more conservative models as they require greater gains for splits.
6. *subsample:* This parameter denotes the fraction of samples to be used for building each tree. Lower values prevent overfitting but too low values can lead to underfitting.
7. *colsample_bytree:* This parameter specifies the fraction of features (columns) to be randomly sampled for each tree. Using a subset of features can help prevent overfitting and improve generalization.

All optional hyperparameters and the best result after the experiment are listed below:

n_estimators	100	500	1000	**1500**	2000
learning rate	0.01	0.03	0.5	0.1	**0.2**
max_depth	**3**	4	5	6	7
min_child_weight	1	2	**3**	4	5
gamma	**0**	0.1	0.2	0.3	0.4
subsample	0.6	0.7	**0.8**	0.9	1
colsample_bytree	0.6	0.7	**0.8**	0.9	1

Table 8: Hyperparameters chosen for the XGBoost Model. The best results are highlighted in bold

6.4.4 ARIMA

The Autoregressive Integrated Moving Average (ARIMA) model, in contrast to complex neural network models, is a straightforward yet effective tool for forecasting future values based solely on the short-term statistical characteristics of time-series data. In ARIMA, the 'AR' component represents autoregression on its own past data, the 'MA' component represents a linear combination of past forecast errors, and 'I' denotes the differencing order

required to make the data stationary(Ariton, 2021). This model is readily available in the statsmodels library.

Unlike neural networks, ARIMA does not learn from window-sliced data but instead fits the entire dataset to predict the next future steps. Therefore, this project leverages the approach that involves fitting a separate ARIMA model for each piece of short-term data in real-time for each single channel. To achieve optimal performance, the auto_arima function from the pmdarima library is used, which automatically identifies the best parameters for AR, I, and MA orders for each data segment of 60 timesteps.

6.4.5 Forecasting Model Result and Bench Mark

This section presents the experimental results and comparisons. Each model is evaluated on a previously unseen test set, and three different samples are created, representing the forecasting for stable data, fluctuating data, and upward-trending data.

LSTM model forecasting examples:

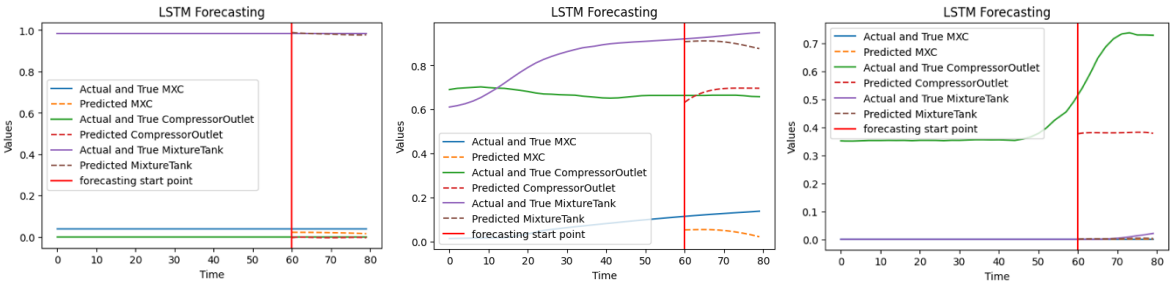

Figure 35: LSTM forecasting examples, correspond to stable, fluctuated, and upward-trending respectively

N-Beats model forecasting examples:

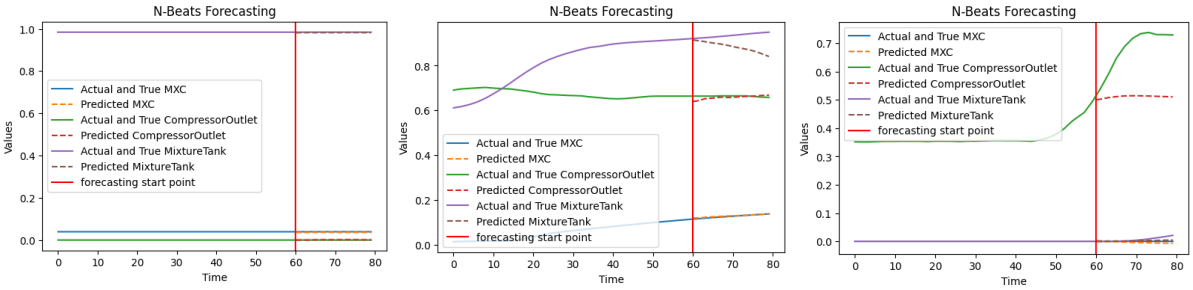

Figure 36: N-Beats forecasting examples, correspond to stable, fluctuated, and upward-trending respectively

XGB model forecasting examples:

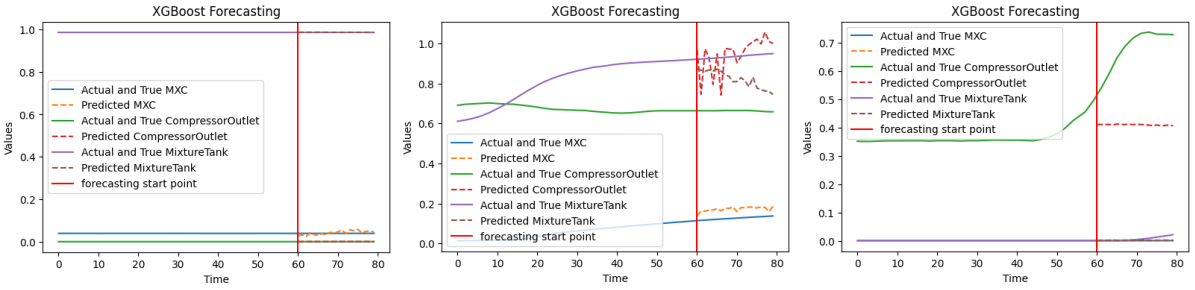

Figure 37: XGB forecasting examples, correspond to stable, fluctuated, and upward-trending respectively

ARIMA model forecasting examples:

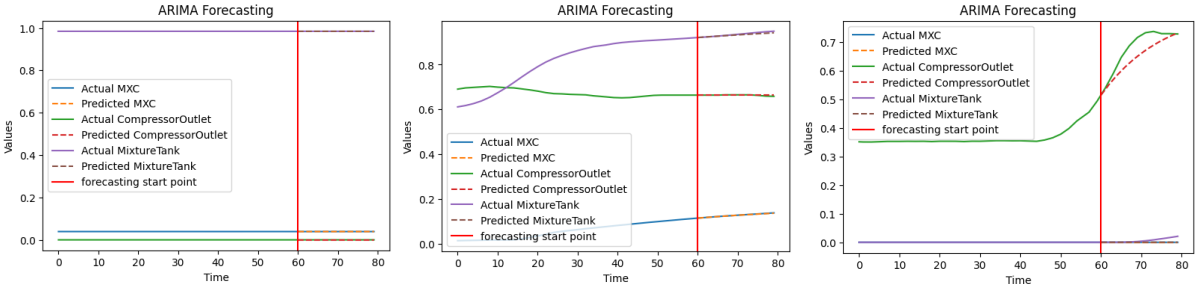

Figure 38: ARIMA forecasting examples, correspond to stable, fluctuated, and upward-trending respectively

The evaluation metrics are given in the table below for the direct comparison of each model's performance. Visualisation is created for MAE and MSE metrics for a better interpretation.

	MAE	**MSE**	**RMSE**	**MAPE**
LSTM model	0.016734	0.002812	0.022222	254983.40625
N-Beats model	0.003982	0.000366	0.005104	124617.56250
XGB model	0.011226	0.001496	0.015691	420009.81250
ARIMA model	0.000645	0.000096	0.001109	1131.41520

Table 9: Forecasting model evaluation result

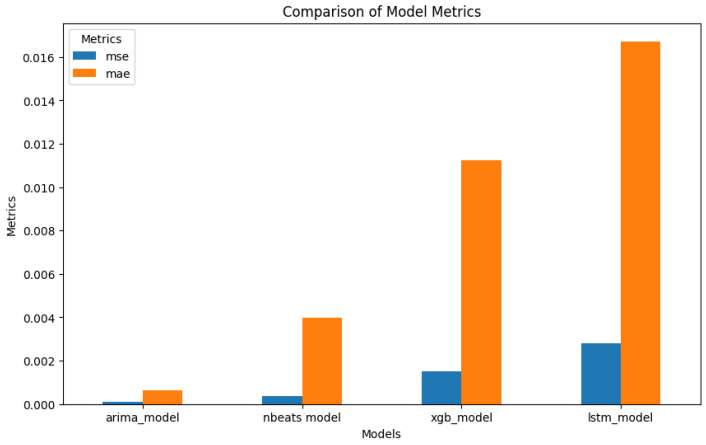

Figure 39: MAE and MSE comparison of models

Analyzing the forecasting example and evaluation results reveals a surprising outcome: all metrics consistently indicate that the simplest ARIMA model outperforms all other forecasting models. Conversely, the most complex LSTM model ranks the worst among all models. The N-Beats model, composed solely of fully connected layers, ranks second, suggesting that simpler models may perform better with limited data.

When focusing on the forecasting examples, all three models perform well on stable forecasts except for the LSTM model, which consistently fails to correctly align the starting point of the forecast and tends to predict lower values across different samples. This likely results from the dataset predominantly containing flat and low values, causing the models to lean towards forecasting lower values. Similar issues are observed in other machine learning models, although the N-Beats model performs slightly better in this regard. Despite having a low validation MSE in forecasting, the XGBoost model produces noisy forecasts on the test data, which might also due to the limited dataset so it cannot congerge to the best result.

6.5 System Outcome

The Anomaly Detection system has been successfully integrated, connecting the best anomaly detection model, the LSTM autoencoder, with the best forecasting model, ARIMA. This integration followed the deployment process outlined in Figure 40. The result is an enhanced anomaly detection system, as illustrated in Figure 41. With this one-step forecasting, the system can now detect anomalies in real-time as early as possible.

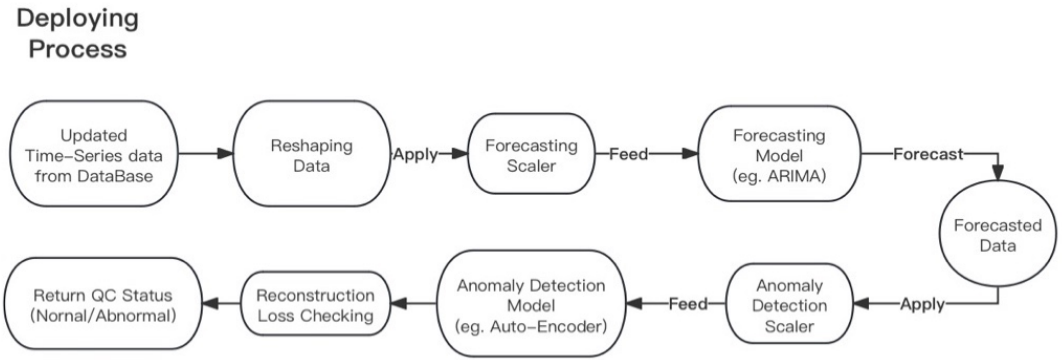

Figure 40: Anomaly Detection System Deploy Process

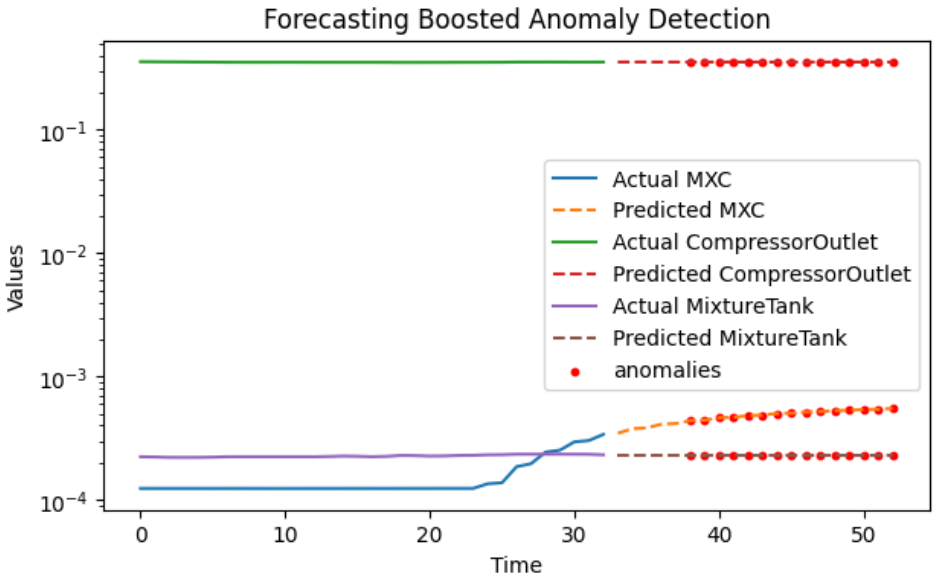

Figure 41: Forecasting Boosted Anomaly Detection

7. Java side – Resouce Management System

7.1 Java-side Design Introduction

The Java side focuses on the Tasks Scheduling and Management module and the Computing Cluster Management module.

The system aims to design a hybrid computing task scheduling system that handles classical computing tasks, quantum computing tasks, and hybrid computing tasks. The system includes a task generator, task scheduler, compute node management, storage system, logging system, and test module, featuring task status management and fault recovery functions.

Using Java and the Spring Boot framework to establish this system. It can generate and process classical computing tasks, quantum computing tasks, and hybrid computing tasks. It can generate and manage compute nodes, support concurrent access, and possess dynamic scaling and fault recovery capabilities for compute nodes. The system's core is the kernel part shown in the diagram, which can be replaced with different inputs and outputs. Currently, a series of test files are used as input, and logs are used as output to test the kernel.

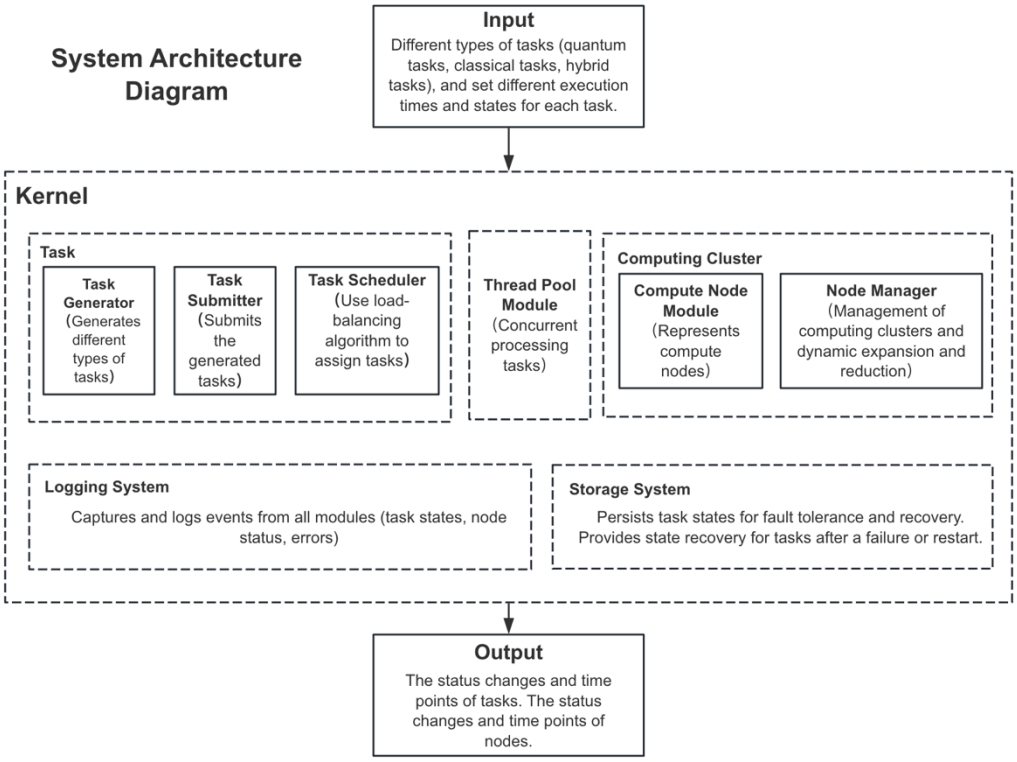

Figure 42: Task management system architecture

For tasks, a lifecycle of a task is designed with five statuses, as shown in the diagram.

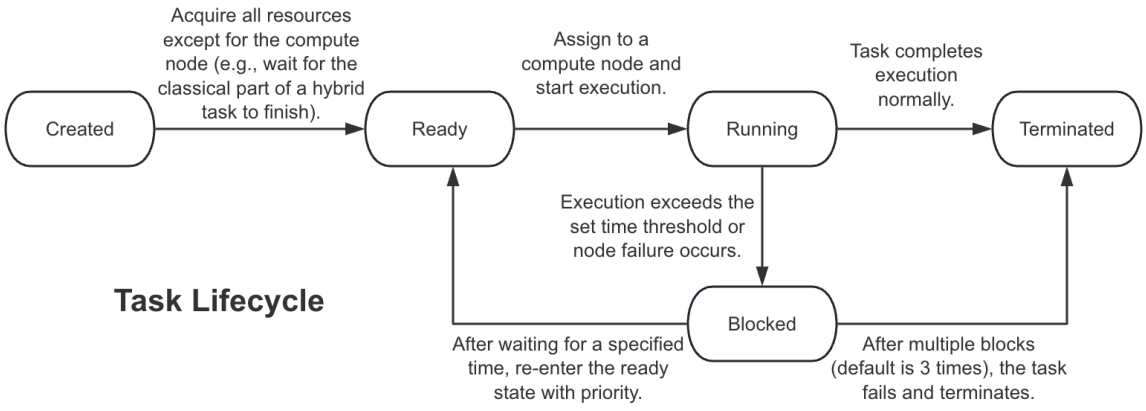

Figure 43: Lifecycle of a task

In the system, tasks transition through five distinct statuses, each representing a stage in their lifecycle. The Created status is the initial state where tasks are identified and initialized but have not yet acquired the necessary resources for execution. Once a task obtains all required

resources except the compute node, it transitions to the Ready state. In this state, tasks are prepared and waiting to be assigned to a compute node for execution. When a task starts using the compute node to perform its operations, it enters the Running state. If a task encounters an issue, such as exceeding its execution time threshold or a node anomaly, it moves to the Blocked state. Here, the task is temporarily halted and cannot continue until the issue is resolved. Finally, the Terminated status indicates that the task has either completed its execution successfully.

Tasks undergo specific transitions between these states based on their execution conditions. A task moves from the Created state to the Ready state after acquiring all non-compute node resources. For instance, in a hybrid task, this occurs after the classical task component is completed. From the Ready state, tasks proceed to the Running state upon being assigned to a compute node, initiating active execution. If a task surpasses its pre-defined execution time limit or faces a node anomaly while in the Running state, it transitions to the Blocked state. In the Blocked state, tasks pause for a designated period before re-entering the Ready state, now prioritized at the top of the scheduling queue. This prioritization ensures that blocked tasks are handled promptly upon re-entry. If a task encounters persistent issues leading to three instances of blockage, it transitions to the Terminated state, signifying failure due to repeated blockages. Conversely, tasks that complete their execution without issues transition directly from the Running state to the Terminated state, indicating successful completion.

7.1 Input&Output Design

7.2.1 Input Design

The Task Generator module generates different types of tasks (quantum tasks, classical tasks, hybrid tasks) and sets different execution times and statuses for each task. The execution times of tasks include those that complete normally and those that may block. After generating the tasks, the system logs the creation time and details of the tasks for tracking and analysis.

The Task Submitter module submits the generated tasks to the TaskScheduler for scheduling. The types of submitted tasks include tasks that complete normally, potentially blocking tasks, sequentially submitted tasks, and concurrently submitted tasks. The system logs the submission time and details of the tasks to track the execution process and the system's responsiveness.

7.2.2 Output Design

The output design of the system primarily revolves around robust logging mechanisms to ensure comprehensive tracking and monitoring of task and node activities. The system employs SLF4J and Logback frameworks to record detailed logs throughout various stages of task execution and node status transitions. These logs serve as a crucial tool for debugging, performance analysis, and audit purposes.

The logging system captures the creation, submission, and completion of tasks along with precise timestamps for each status. This allows for a chronological trace of task lifecycle events, facilitating the identification of performance bottlenecks or failures. Furthermore, the system records changes in node states, documenting the exact time points at which nodes

switch between normal and abnormal states. This is particularly important for tracking the behaviour of quantum nodes, which are prone to intermittent anomalies.

7.3 TaskScheduler Strategy

The TaskScheduler is designed to efficiently manage the scheduling and execution of tasks by configuring appropriate thread pools and implementing a task prioritization mechanism. The configuration of thread pools is tailored to the specific requirements of classical and quantum tasks to ensure optimal performance and reliability.

For classical tasks, the TaskScheduler configures multi-threaded pools to facilitate concurrent execution. This approach leverages the capabilities of modern multi-core processors, enabling the simultaneous processing of multiple classical tasks. Concurrent execution is essential for maximizing throughput and ensuring that the system can handle high volumes of traditional computational tasks efficiently.

In contrast, quantum tasks require atomic execution due to the sensitive nature of quantum computations. To meet this requirement, the TaskScheduler configures single-threaded pools for quantum tasks, ensuring that each task is executed in isolation. This configuration prevents interference and maintains the integrity of quantum operations, which is crucial for achieving accurate and reliable results in quantum computing.

Task prioritization is another critical aspect of the TaskScheduler's strategy. Tasks that re-enter the ready state after being blocked are assigned the highest priority. This prioritization ensures that tasks interrupted or delayed due to node issues are promptly rescheduled and executed before other tasks. By giving precedence to previously blocked tasks, the system reduces the likelihood of prolonged delays and improves the overall responsiveness and reliability of the task scheduling process.

In summary, the TaskScheduler employs a strategic approach that integrates thread pool configuration and task prioritization to manage the execution of tasks efficiently. By configuring thread pools to match the specific needs of classical and quantum tasks and prioritizing tasks based on their execution history, the TaskScheduler ensures a balanced, reliable, and high-performance task scheduling system.

7.4 Computing Cluster Manager module

The Node Manager in this system plays a crucial role in the creation, allocation, and dynamic scaling of compute nodes, specifically managing classical and quantum computing nodes. This component ensures that tasks are assigned to appropriate nodes and that resources are efficiently utilized to maintain optimal performance and reliability.

The system is designed to handle two types of compute nodes: classical nodes and quantum nodes. Quantum nodes can experience both normal and abnormal states, with a set probability of 10% for anomalies. These anomalies, lasting for about 10 minutes, can impact the system's ability to process quantum tasks effectively. Although the quantum node status is currently simulated within the system, the design includes provisions for future integration with external Python scripts that will periodically check and update the status of these nodes. This periodic check will be implemented as a scheduled task that queries the status and updates the system accordingly.

The Node Manager employs a list-based structure to manage the tasks assigned to classical and quantum nodes. This design facilitates the tracking and dynamic allocation of tasks, ensuring that each task is processed by the appropriate type of node. For instance, classical tasks are handled by classical nodes, while quantum tasks are directed to quantum nodes. The

dynamic nature of the Node Manager allows for adjustments in the number of active nodes based on the system's current load.

Overall, the Node Manager is a critical component that ensures the effective management of compute nodes within the system. Its ability to dynamically scale resources, coupled with the efficient allocation of tasks to classical and quantum nodes, provides a robust framework for maintaining high performance and reliability. Future enhancements will include integrating real-time status updates from Python scripts, further improving the system's ability to handle quantum node anomalies and optimize task scheduling.

7.5 Fault Recovery

In the development of a robust task scheduling and distributed computing system, fault recovery is an essential component to ensure the resilience and reliability of the system. The designed system utilizes Redis technology for efficient fault recovery, enabling the restoration of task states after system failures or restarts, thereby ensuring the continuation of unfinished tasks.

To achieve effective task recovery, the system meticulously records various task attributes in Redis. Each task is assigned a unique identifier (task ID), which serves as the primary key for tracking and managing the task. Additionally, the type of each task—whether it is a classical task, a quantum task, or a hybrid task—is documented to ensure appropriate handling based on the task's specific requirements. The current status of the task, along with a timestamp for each status change, is also recorded. This comprehensive logging facilitates precise tracking of task progression and the system's ability to pinpoint the exact state of each task at any given time.

The task recovery strategy is designed to uphold both task fairness and the atomicity of quantum tasks. The recovery strategy is defined based on the current state of the task and the nature of the failure encountered. By ensuring that the task classes and their subclasses implement the Serializable interface, the system can effectively serialize and deserialize task states within Redis. This capability is crucial for maintaining the integrity and continuity of tasks across system failures.

Upon system failure or restart, the task scheduler can invoke recovery mechanisms to retrieve the persisted task states from Redis. This process involves reconstructing the task queues and reinstating the task statuses to their most recent states. For quantum compute nodes, which are prone to anomalies, the system can dynamically reassign tasks to other nodes based on real-time status checks. This ensures that tasks disrupted by node failures are promptly rescheduled and prioritized according to their urgency and the severity of the failure.

The implementation of Redis for task state persistence and recovery not only enhances the system's fault tolerance but also provides a scalable solution for managing distributed tasks. By leveraging Redis' high-performance in-memory data store capabilities, the system can achieve rapid recovery and maintain high availability, even under conditions of frequent node failures or system restarts. This robust fault recovery mechanism is integral to the system's overall design, ensuring continuous and reliable task processing in a distributed computing environment.

7.6 Test Conclusion

7.6.1 Testing Methodology

In this study, the TaskScheduler was subjected to a comprehensive suite of tests to validate its performance, reliability, and fault tolerance. The testing strategy comprised four primary tests:

1.Classical Task Scheduling:

Objective: To verify the correct scheduling and execution of classical tasks.

Method: Classical tasks with varying execution times (ranging from 1 to 10 seconds) were generated and scheduled. The system logged the initialization, creation, scheduling, and execution times, ensuring the task completed within the expected timeframe.

2.Quantum Task Scheduling:

Objective: To ensure quantum tasks are scheduled and executed atomically, adhering to their unique execution requirements.

Method: Quantum tasks, which included a mandatory 1-second safety check followed by execution times between 0.5 and 10 microseconds, were generated and scheduled. The system logged each phase, including safety checks and execution, to confirm atomic execution without interruptions.

3.Hybrid Task Scheduling:

Objective: To validate the correct sequential execution of hybrid tasks, which consist of both classical and quantum components.

Method: Hybrid tasks were generated, requiring the classical component to execute first (1 to 10 seconds) followed by the quantum component (0.5 to 10 microseconds after a 1-second

safety check). Logs were reviewed to ensure sequential execution and proper handling of both task types.

4.Node Failure Handling:

Objective: To test the system's fault tolerance and its ability to recover from node failures.

Method: Simulated node failures were introduced, requiring the TaskScheduler to fetch running tasks from Redis, set their status to BLOCKED, and re-schedule them. The system logged each recovery step, including task fetching, status updating, and re-scheduling.

7.6.2 Results Interpretation

The results are as below:

```
2024-06-13 22:57:12.920 Test 1: Schedule Classical Task
2024-06-13 22:57:12.920 INFO: Initializing TaskScheduler...
2024-06-13 22:57:12.920 INFO: Creating Classical Task...
2024-06-13 22:57:12.920 INFO: Scheduling Classical Task...
2024-06-13 22:57:12.920 INFO: Classical Task executing for 1820 ms...
2024-06-13 22:57:14.746 INFO: Classical Task scheduled successfully.
```

Figure 44: Schedule Classical Task

Test 1 shows the successful scheduling and execution of classical tasks within the expected timeframes demonstrated the system's capability to handle traditional computing tasks efficiently.Logs indicated precise task management, with all tasks completing without delays or errors, affirming the robustness of the classical task scheduling mechanism.

```
2024-06-13 22:57:11.901 Test 2: Schedule Quantum Task
2024-06-13 22:57:11.901 INFO: Initializing TaskScheduler...
2024-06-13 22:57:11.901 INFO: Creating Quantum Task...
2024-06-13 22:57:11.901 INFO: Scheduling Quantum Task...
2024-06-13 22:57:11.901 INFO: Quantum Task performing 1s safety check...
2024-06-13 22:57:12.912 INFO: Quantum Task executing for 4413 µs...
2024-06-13 22:57:12.917 INFO: Quantum Task scheduled successfully.
2024-06-13 22:57:12.917 ------------------------------------------------
```

Figure 45: *Schedule Quantum Task*

Test 2 shows the atomic execution of quantum tasks, including the mandated safety checks, confirmed the system's ability to manage quantum-specific requirements.The execution logs verified that all quantum tasks adhered to their specified execution times and safety protocols, highlighting the scheduler's precision in handling quantum tasks.

```
2024-06-13 22:57:14.750 Test 3: Schedule Hybrid Task
2024-06-13 22:57:14.750 INFO: Initializing TaskScheduler...
2024-06-13 22:57:14.750 INFO: Creating Hybrid Task...
2024-06-13 22:57:14.750 INFO: Scheduling Hybrid Task...
2024-06-13 22:57:14.750 INFO: Hybrid Task - Classical Part executing for 6234 ms...
2024-06-13 22:57:20.990 INFO: Hybrid Task - Quantum Part performing 1s safety check...
2024-06-13 22:57:21.991 INFO: Hybrid Task - Quantum Part executing for 8750 µs...
2024-06-13 22:57:22.001 INFO: Hybrid Task scheduled successfully.
2024-06-13 22:57:22.001 ------------------------------------------------
```

Figure 46: *Schedule Hybrid Task*

Test 3 shows the sequential execution of hybrid tasks, with accurate logging of both classical and quantum components, validated the scheduler's ability to manage complex task types.

The successful completion of hybrid tasks without interference between the classical and quantum parts showcased the scheduler's effectiveness in orchestrating multi-faceted tasks.

```
2024-06-13 22:57:10.880 Test 4: Handle Node Failure
2024-06-13 22:57:10.880 INFO: Initializing TaskScheduler...
2024-06-13 22:57:10.880 INFO: Simulating Node Failure...
2024-06-13 22:57:11.886 INFO: Fetching Running Tasks from Redis...
2024-06-13 22:57:11.886 INFO: Setting Task Status to BLOCKED...
2024-06-13 22:57:11.886 INFO: Re-scheduling Blocked Tasks...
2024-06-13 22:57:11.886 INFO: Node Failure handled successfully.
2024-06-13 22:57:11.886 ------------------------------------------------
```

Figure 47: *Handle Node Failure*

Test 4 shows the system's response to simulated node failures, including task state recovery and re-scheduling, demonstrated its fault tolerance. Detailed logs of the recovery process confirmed that the scheduler could effectively manage and recover from node disruptions, ensuring task continuity and minimal downtime.

```
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 11.143 s - in com.judy.ihpctasks.TaskScheduler.TaskSchedulerTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  13.419 s
[INFO] Finished at: 2024-06-13T22:57:22+08:00
```

Figure 48: Conclusion

Overall, these tests substantiate the TaskScheduler's efficiency in managing and executing diverse task types, maintaining atomicity for quantum tasks, and demonstrating robust fault recovery capabilities.

8. Future work needed

Due to the time limitation and restriction of the resources, there are few future works that could be considered to futher improve the whole system.

- **Refined Model Training:** Future iterations could benefit from training the model on a larger dataset, which would improve its ability to generalize. Additionally, experimenting with different window slice sizes could enhance performance in both anomaly detection and forecasting models. In this project, a window size of 20 was used for anomaly detection and 60 for forecasting, but testing a variety of sizes may yield even better results. Furthermore, exploring additional more state-of-the-art model structures could also be a future topic of this project.
- **Integration of Python and Java systems:** Currently, Python-based anomaly detection systems and Java-based resource scheduling systems operate basically independently. Future work may focus on enhancing the interaction between the two systems. By achieving more seamless integration, systems can work together more effectively, manage resources, and detect anomalies in a timely manner.
- **Development of a user-friendly front-end interface:** To improve usability, a fully intuitive front-end interface can be designed in the future. The interface should provide users with easy access to anomaly detection and resource scheduling functions. Key features may include dashboards for real-time monitoring, interactive charts, and task management tools.
- **Integrates real-time anomaly detection with Grafana:** In order to improve the real-time performance of the anomaly detection system, it is necessary to integrate with A*STAR's Grafana real-time database to detect the status of quantum devices in real time in order to give timely or even early alarms. In addition, the accuracy of the

anomaly prediction model is further improved in the case of accumulating more actual data.

- **Combine quantum computing nodes:** With the increasing demand for quantum computing, it is very necessary to integrate quantum computing nodes into resource scheduling systems. This involves not only managing traditional computing resources, but also dynamically assigning tasks to quantum nodes. The system should be designed to handle the unique properties and constraints of quantum computing, ensuring efficient and reliable execution of tasks.
- **Enhanced fault recovery mechanism:** While the current system has a basic failure recovery mechanism, future work can focus on improving the robustness of this mechanism. This includes developing algorithmic models for predicting potential failures and implementing automatic failover processes, thereby improving the fault tolerance of the system.
- **Comprehensive testing and verification:** The focus of future work should be on extensive testing and validation to ensure the reliability and effectiveness of the integrated system and its ability to meet real business requirements. This may involve unit testing, integration testing, and performance testing in various scenarios, collecting feedback from A*STAR researchers, and iteratively optimizing the system.

References

1. Perelshtein, M., Sagingalieva, A., Pinto, K., Shete, V., Pakhomchik, A., Melnikov, A., Neukart, F., Gesek, G., Melnikov, A., & Vinokur, V. (2022). Practical application-specific advantage through hybrid quantum computing. arXiv preprint arXiv:2201.00001.

2. Nguyen, H. T., Usman, M., & Buyya, R. (2023). iQuantum: A Case for Modeling and Simulation of Quantum Computing Environments. (pp. 21–30).

3. Ahmed, H., Deng, X., Heller, H., Guillen, C., Zulfiqar, A., Ruefnacht, M., Jamadagni, A., Tovey, M., Schulz, M., & Schulz, L. (2023). Quantum Computer Metrics and HPC Center Environmental Sensor Data Analysis Towards Fidelity Prediction. 2, 154–160.

4. Mandelbaum, R., Steffen, M., & Cross, A. (2023). Error correcting codes for near-term quantum computers.

5. Terra Quantum AG. (2022). How hybrid quantum computing creates business benefits today. Terra Quantum White Paper.

6. Institute of High Performance Computing (IHPC). (n.d.). Message from director. Retrieved from https://www.a-star.edu.sg/ihpc/about-us/message-from-director

7. Zu, H., Dai, W., & de Waele, A. T. A. M. (2022). Development of dilution refrigerators—A review. Cryogenics, 121, 103390.

8. Maklin, C. (2022). Isolation Forest. Medium. Retrieved from https://medium.com/@corymaklin/isolation-forest-799fceacdda4

9. Point-Denoise: Unsupervised outlier detection for 3D point clouds enhancement. (2024). Scientific Figure on ResearchGate. Retrieved from https://www.researchgate.net/figure/Isolation-Forest-learned-iForest-construction-for-toy-dataset_fig1_352017898

10. Local outlier factor. (n.d.). Local Outlier Factor - an overview. Retrieved from https://www.sciencedirect.com/topics/computer-science/local-outlier-factor

11. Outlier detection with local outlier factor (LOF). (n.d.). scikit-learn. Retrieved from https://scikit-learn.org/stable/auto_examples/neighbors/plot_lof_outlier_detection.html

12. Patel, M. (2018, December 24). Applied deep learning - Part 3: Autoencoders. Towards Data Science. Retrieved from https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

13. Unsupervised marine vessel trajectory prediction using LSTM network and wild bootstrapping techniques. (2024). Scientific Figure on ResearchGate. Retrieved from https://www.researchgate.net/figure/Architecture-of-LSTM-autoencoder_fig1_352898971

14. Goix, N. (2016). How to Evaluate the Quality of Unsupervised Anomaly Detection Algorithms? arXiv preprint arXiv:1606.03903.

15. Filho, M. (2023). Multiple time series forecasting with N-beats in Python. Forecastegy. Retrieved from https://forecastegy.com/posts/multiple-time-series-forecasting-nbeats-python/#what-is-n-beats

16. Dancker, J. (2024). N-beats-the first interpretable deep learning model that worked for time series forecasting. Medium. Retrieved from https://towardsdatascience.com/n-beats-the-first-interpretable-deep-learning-model-that-worked-for-time-series-forecasting-06920daadac2

17. Ariton, L. (2021). A thorough introduction to Arima models. Medium. Retrieved from https://medium.com/analytics-vidhya/a-thorough-introduction-to-arima-models-987a24e9ff71

Appendix

Implementation of warm-up cosine decay learning rate scheduler:

```
def lr_warmup_cosine_decay(global_step, warmup_steps, hold = 0, total_steps=0, start_lr=0.0,
target_lr=1e-3):

        # warm_steps: number of steps linearly increase to the maximum learning rate
        # total_steps: number of steps drop down back to the start learning rate
        # global_step: current number of steps
        learning_rate = 0.5 * target_lr * (1 + np.cos(np.pi * (global_step - warmup_steps -
hold) / float(total_steps - warmup_steps - hold)))

        # target LR: Desired maximum lr
        warmup_lr = target_lr * (global_step / warmup_steps)

        # hold: number of steps keep for the target LR
        if hold > 0:
            learning_rate = np.where(global_step > warmup_steps + hold,
                                     learning_rate, target_lr)

        learning_rate = np.where(global_step < warmup_steps, warmup_lr, learning_rate)

         return learning_rate


# referenced from https://stackabuse.com/learning-rate-warmup-with-cosine-decay-in-keras-and-
tensorflow/
class WarmupCosineDecay(keras.callbacks.Callback):
    def __init__(self, total_steps=0, warmup_steps=0, start_lr=0.0, target_lr=1e-3, hold=0):

        super(WarmupCosineDecay, self).__init__()
        self.start_lr = start_lr
        self.hold = hold
        self.total_steps = total_steps
        self.global_step = 0
        self.target_lr = target_lr
        self.warmup_steps = warmup_steps
        self.lrs = []

    # append the value of learning rate in the list for review
    def on_batch_end(self, batch, logs=None):
        self.global_step = self.global_step + 1
        lr = self.model.optimizer.lr.numpy()
        self.lrs.append(lr)

    def on_batch_begin(self, batch, logs=None):
        lr = lr_warmup_cosine_decay(global_step=self.global_step,
                                    total_steps=self.total_steps,
                                    warmup_steps=self.warmup_steps,
                                    start_lr=self.start_lr,
                                    target_lr=self.target_lr,
                                    hold=self.hold)
        K.set_value(self.model.optimizer.lr, lr)

    # print learning rate at end of the epoch
    def on_epoch_end(self, epoch, logs=None):
        lr = self.model.optimizer.lr.numpy()
        print(f" Learning rate: {lr}")
```

Implementation of the LSTM Autoencoder Hypermodel:

```
class MyHyperModel(HyperModel):
    def __init__(self, timesteps, num_features):
        self.timesteps = timesteps
        self.num_features = num_features
        self.initial_weights = None

    def build(self, hp):
        batch_norm = hp.Boolean('batch_norm')
        conv  = hp.Boolean('use_conv')
        bidirectional = hp.Boolean('use_bidirectional')
        unit = hp.Choice('units', values=[32, 64, 128, 256])
        gaussian = hp.Float('gaussian', min_value=0, max_value=0.2, step=0.01)
        regularizer1 = tf.keras.regularizers.l2(hp.Float('regulizer_1', min_value=0.0001,
max_value=0.01, sampling='LOG'))
        regularizer2 = tf.keras.regularizers.l2(hp.Float('regulizer_2', min_value=0.0001,
max_value=0.01, sampling='LOG'))

        model = keras.Sequential()
        model.add(layers.Input(shape=(self.timesteps, self.num_features)))

        model.add(layers.GaussianNoise(gaussian))

        if conv:
          model.add(layers.Conv1D(filters=32, kernel_size=3, padding='casual',
activation='relu'))
          model.add(layers.MaxPooling1D(pool_size=2))
          model.add(layers.Conv1D(filters=64, kernel_size=3, padding='casual',
activation='relu'))
          model.add(layers.MaxPooling1D(pool_size=2))

        if bidirectional:
          model.add(layers.Bidirectional(layers.LSTM(
              units= unit,
              return_sequences= True,
              kernel_regularizer= regularizer1
          )))
        else:
          model.add(layers.LSTM(
              units= unit,
              return_sequences= True,
              kernel_regularizer= regularizer1
          ))

        if batch_norm:
          model.add(layers.BatchNormalization())

        model.add(layers.LSTM(
            units= unit//2,
            kernel_regularizer= regularizer2
        ))

        model.add(layers.Dropout(rate=hp.Float('dropout_1', min_value=0.1, max_value=0.5,
step=0.1)))

        model.add(layers.RepeatVector(self.timesteps))

        model.add(layers.LSTM(
            units= unit//2,
            return_sequences= True,
            kernel_regularizer= regularizer2
        ))

        if batch_norm:
          model.add(layers.BatchNormalization())

        if bidirectional:
          model.add(layers.Bidirectional(layers.LSTM(
              units= unit,
```

```
                return_sequences= True,
                kernel_regularizer= regularizer1
            )))
        else:
            model.add(layers.LSTM(
                units= unit,
                return_sequences= True,
                kernel_regularizer= regularizer1
            ))

        optimizer = tf.keras.optimizers.Adam(
            clipnorm=hp.Float('clipnorm', min_value=0.1, max_value=1.0, step=0.1),
            weight_decay=hp.Float('weight_decay', min_value=1e-6, max_value=1e-3,
sampling='LOG')
        )

        model.add(layers.Dropout(rate=hp.Float('dropout_2', min_value=0.1, max_value=0.5,
step=0.1)))

        model.add(layers.TimeDistributed(layers.Dense(self.num_features,
                                                      #dtype='float32'
                                                      )))

        model.compile(optimizer=optimizer, loss='mse')

        hp.Choice("batch_size", [128, 256, 512, 1024, 2048])
        hp.Float('start_lr', min_value=1e-6, max_value=1e-4, sampling='LOG')
        hp.Float('target_lr', min_value=1e-4, max_value=1e-2, sampling='LOG')
        hp.Int("epochs_per_period", min_value=10, max_value = 200, step=20)

        return model

    def fit(self, hp, model, *args, **kwargs):
        batch_size = hp.get("batch_size")
        start_lr = hp.get("start_lr")
        target_lr = hp.get("target_lr")
        epochs_per_period = hp.get("epochs_per_period")

        # Adjust learning rates based on batch size
        adjusted_start_lr = start_lr * (batch_size / 128)
        adjusted_target_lr = target_lr * (batch_size / 128)


        total_steps = len(x_train_combined)/batch_size * epochs_per_period
        warmup_steps = int(0.05*total_steps)

        lr_callback = WarmupCosineDecay(
                                    total_steps=total_steps,
                                    warmup_steps=warmup_steps,
                                    hold=int(warmup_steps/2),
                                    start_lr=adjusted_start_lr,
                                    target_lr=adjusted_target_lr)

        # append callback
        kwargs["callbacks"].append(lr_callback)

        return model.fit(
            *args,
            batch_size= batch_size,
            **kwargs,
        )
```

Implementation of the N-Beats Hypermodel:

```
# Referenced from https://colab.research.google.com/github/mrdbourke/tensorflow-deep-
learning/blob/main/10_time_series_forecasting_in_tensorflow.ipynb#scrollTo=llwhXCn9-hQt

# Create NBeatsBlock custom layer
class NBeatsBlock(tf.keras.layers.Layer):
  def __init__(self, # the constructor takes all the hyperparameters for the layer
               window_size: int,
               features: int,
               theta_size: int,
               horizon: int,
               n_neurons: int,
               n_layers: int,
               dropout_rate: float,
               activation: str,
               **kwargs):
    super().__init__(**kwargs)
    self.window_size = window_size
    self.features = features
    self.theta_size = theta_size
    self.horizon = horizon
    self.n_neurons = n_neurons
    self.n_layers = n_layers
    self.dropout_rate = dropout_rate
    self.activation = activation
    self.hidden = []

    self.flatten = tf.keras.layers.Flatten()
    for _ in range(n_layers):
      self.hidden.append(tf.keras.layers.Dense(n_neurons, activation=self.activation))
      self.hidden.append(tf.keras.layers.Dropout(dropout_rate)) # add dropout after each
hidden layer
    # Output of block is a theta layer with linear activation
    self.theta_layer = tf.keras.layers.Dense(theta_size, activation="linear", name="theta")
    # Reshape the output to multivariate
    self.reshape = tf.keras.layers.Reshape((self.window_size+self.horizon, self.features))

  def call(self, inputs): # the call method is what runs when the layer is called
    x = self.flatten(inputs)
    for layer in self.hidden: # pass inputs through each hidden layer
      x = layer(x)
    theta = self.theta_layer(x)
    theta = self.reshape(theta)
    backcast, forecast = theta[:, :self.window_size, :], theta[:, -self.horizon:, :]
    return backcast, forecast

class NBEATS_Forecasting_HyperModel(HyperModel):
    def __init__(self, num_samples, window_size, features, theta_size, horizon):
        self.num_samples = num_samples
        self.window_size = window_size
        self.features = num_features
        self.theta_size = theta_size
        self.horizon = horizon

    def build(self, hp):
        neurons = hp.Choice('neurons', [16, 32, 64, 128, 256])
        layers = hp.Int('layers', min_value=1, max_value=5)
        stacks = hp.Int('stacks', min_value=1, max_value=5)

        stack_input = tf.keras.layers.Input(shape=(self.window_size, self.features),
name="stack_input")
        dropout_rate = hp.Float('dropout_rate', min_value=0, max_value=0.5, step=0.05)
        activation = hp.Choice('activation', ['relu', 'leaky_relu', 'elu'])

        nbeats_block_layer = NBeatsBlock(window_size=self.window_size,
                                         features=self.features,
                                         theta_size=self.theta_size,
                                         horizon=self.horizon,
                                         n_neurons=neurons,
```

```
                                        n_layers=layers,
                                        dropout_rate=dropout_rate,
                                        activation=activation)

        backcast, forecast = nbeats_block_layer(stack_input)
        backcast_reshaped = tf.keras.layers.Reshape((self.window_size,
self.features))(backcast)
        residuals = tf.keras.layers.subtract([stack_input, backcast_reshaped],
name=f"subtract_00")

        for i in range(stacks - 1):
            backcast, block_forecast = NBeatsBlock(
                window_size=self.window_size,
                features=self.features,
                theta_size=self.theta_size,
                horizon=self.horizon,
                n_neurons=neurons,
                n_layers=layers,
                dropout_rate=dropout_rate,
                activation=activation
            )(residuals)

            backcast_reshaped = tf.keras.layers.Reshape((self.window_size,
self.features))(backcast)
            residuals = tf.keras.layers.subtract([residuals, backcast_reshaped],
name=f"subtract_{i + 1}")
            forecast = tf.keras.layers.add([forecast, block_forecast], name=f"add_{i + 1}")

        model = tf.keras.Model(inputs=stack_input, outputs=forecast)

        optimizer = tf.keras.optimizers.Adam(
            clipnorm=hp.Float('clipnorm', min_value=0.1, max_value=1.0, step=0.1),
            weight_decay=hp.Float('weight_decay', min_value=1e-6, max_value=1e-3,
sampling='LOG')
        )

        model.compile(
            optimizer=optimizer,
            loss='mse'
        )
        # learning rate schedular hyperparameters
        hp.Choice("batch_size", [128, 256, 512, 1024, 2048])
        start_lr = hp.Float('start_lr', min_value=1e-7, max_value=1e-3, sampling='LOG')
        target_lr = hp.Float('target_lr', min_value=start_lr, max_value=1e-1, sampling='LOG')
        hp.Int("epochs_per_period", min_value=10, max_value = 200, step=20)

        return model

    def fit(self, hp, model, *args, **kwargs):
        batch_size = hp.get("batch_size")
        start_lr = hp.get("start_lr")
        target_lr = hp.get("target_lr")
        epochs_per_period = hp.get("epochs_per_period")

        total_steps = self.num_samples/batch_size * epochs_per_period
        warmup_steps = int(0.05*total_steps)

        # reuse the cos decay learning scheduler again
        lr_callback = WarmupCosineDecay(
                                    total_steps=total_steps,
                                    warmup_steps=warmup_steps,
                                    hold=int(warmup_steps/2),
                                    start_lr=start_lr,
                                    target_lr=target_lr)
        # append callback
        kwargs["callbacks"].append(lr_callback)

        return model.fit(
            *args,
            batch_size= batch_size,
            **kwargs,
        )
```

Java Side Project Structure:

```
IHPCTasks
│
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com
│   │   │       └── judy
│   │   │           └── ihpctasks
│   │   │               ├── config
│   │   │               │   └── SchedulerConfig.java
│   │   │               ├── core
│   │   │               │   ├── Task.java
│   │   │               │   ├── ClassicalTask.java
│   │   │               │   ├── QuantumTask.java
│   │   │               │   ├── HybridTask.java
│   │   │               │   ├── TaskStatus.java
│   │   │               ├── node
│   │   │               │   ├── ComputeNode.java
│   │   │               │   └── NodeManager.java
│   │   │               ├── scheduler
│   │   │               │   └── TaskScheduler.java
│   │   │               ├── storage
│   │   │               │   └── RedisStorage.java
│   │   │               ├── IhpcTasksApplication.java
│   └── resources
│       ├── application.properties
│       └── logback.xml
│
```

Figure 49: Java Side Project Structure

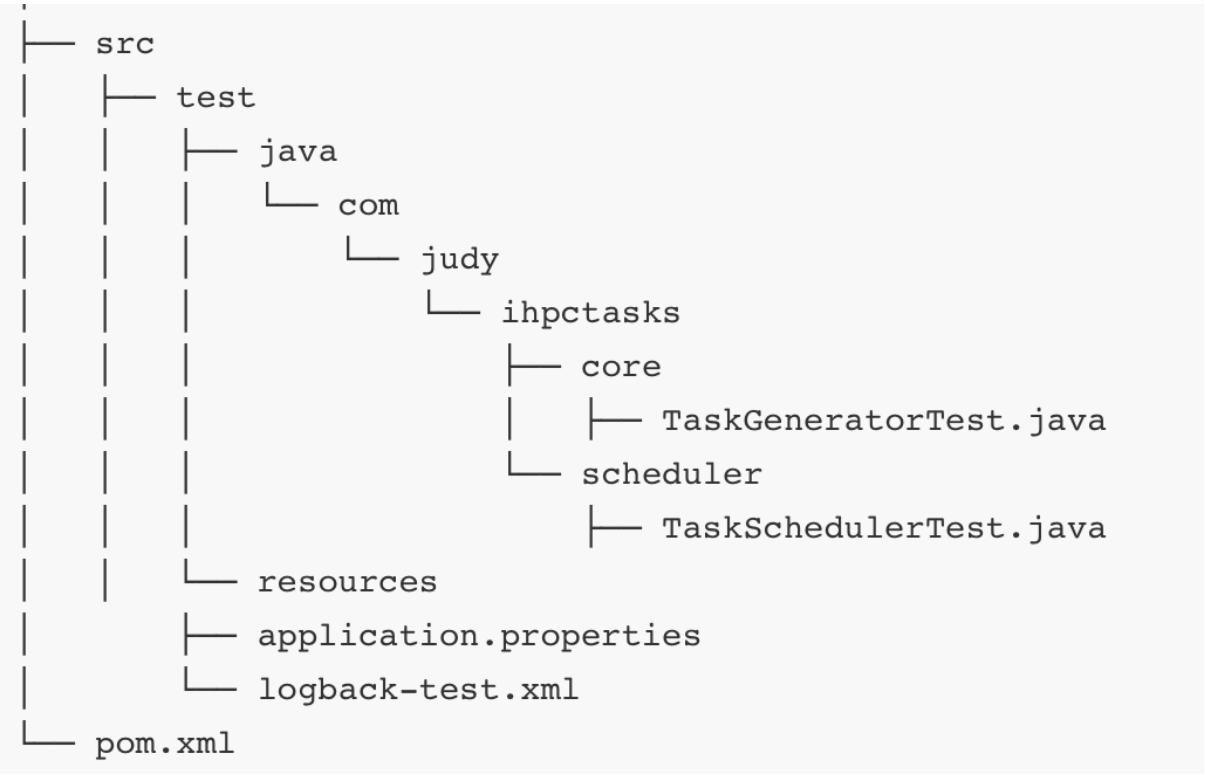

Figure 50: Java Side Project Structure