From Data to Decisions: ML-Based Job Failure Analysis for the ALICE experiment

Erlend Lauvås Skutlaberg

Master's thesis in Software Engineering at

Department of Computer science, Electrical engineering and Mathematical sciences,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

June 2024

Western Norway
University of
Applied Sciences

Abstract

This paper researches the possibility of utilizing machine learning to locate factors contributing to a failing job at the ALICE grid, focusing on the grid site at the University of Bergen. A prototype system has been developed for data collection, management, analysis, and machine learning. The analysis data originates from the ALICE monitoring system, MonaLISA, and its grid middleware JAliEn. A custom transformer model is utilized in the research, which addresses memory constraints in the project test environment by processing subsets of the complete input related to a job execution at a time.

Acknowledgements

First and foremost, I would like to express my heartfelt gratitude to my project supervisors, Bjarte Kileng, Matthias Richter, and Håvard Helstrup, for their unwavering expertise and guidance throughout this project. Your insights, feedback, and encouragement have been invaluable, shaping the course of my work and ensuring its quality.

Additionally, I extend my sincere thanks to my family for their continuous support and unwavering motivation. Their belief in me has been a driving force, and I am grateful for their presence on this journey.

With warm regards,

Erlend Lauvås Skutlaberg

Contents

Acronyms

AI Artificial Intelligence.

ALICE A Large Ion Collider Experiment.

CVMFS CernVM File System.

JAliEn Java ALICE Evironment.

LHC Large Hadron Collidor.

LSTM Long Short-Term Memory.

ML Machine Learning.

MSE Mean Squared Error.

NLP Natural Language Processing.

WLCG Worldwide LHC Computing Grid.

Chapter 1

Introduction

1.1 Motivation

Machine Learning (ML) is a branch of Artificial Intelligence (AI) that has grown quickly in recent years in terms of data analysis tasks [34]. It is a technology that can extract meaningful patterns and structures in data. There are multiple use cases for ML, where language translation tasks are one of several examples where it has proven exceptionally well [42]. This motivates the project to utilize this powerful technology for its data analysis.

A Large Ion Collider Experiment (ALICE), is situated at CERN's Large Hadron Collidor (LHC) in Geneva, Switzerland. The experiment revolves around a study of the physics of strongly interacting matters at high energy densities [11]. Data procured in this experiment is preserved for further analysis, where several computational tasks, referred to as jobs, are generated to analyze this data. These are distributed across multiple computer nodes within a grid infrastructure, as the computational power required is substantial.

The grid sites utilized by the ALICE experiment sometimes experience job failures, where a job cannot successfully execute its tasks. These jobs consume computing resources without yielding any valuable output. The project aims to find the causes of job failure to provide more insight into why a job fails. This information can be quite valuable, especially if the causes can be addressed. A reduction in job failures reduces wasteful computing, which

conserves resources in terms of power and costs. The first step towards this goal is to locate the problem's origins.

1.2 Research questions

The primary objective of this project is to investigate the feasibility of utilizing Machine Learning to locate the underlying causes of job failure at its test subject, the University of Bergen's grid site. In pursuit of this goal, the research will address the following key questions:

- **RQ1:** How can AI help us discover the underlying causes and factors contributing to job failures?
- **RQ2:** What strategies and mechanisms can be devised for the efficient optimization of computer resources based on insights derived from this AI model?

These research inquiries will guide the exploration of ML-driven data analysis within the context of failing jobs at the University of Bergen's grid site.

Chapter 2

Background

This section will provide some background information relating to Grid Computing, ALICE, and Machine Learning. The provided information will be utilized in the following chapters.

2.1 Grid Computing

Grid computing utilizes the collaboration between interconnected computers to obtain higher performance and more resources. It has several benefits, one of which is the ability to solve more resource-heavy problems. Grid computing also provides an infrastructure for collaboration between institutions and organizations, as described in the book "Grid Computing" by Barry Wilkinson [44].

Grids have a decentralized structure, in which authorized users can submit tasks. Not one person or organization has control over the complete system, but rather a part of it. The essence of grid computing is having distributed computing resources working towards a common goal. The distribution yields a need for certain standards to ensure successful executions of the computational tasks. Problems might occur if the (local) test environment of a task differs from the execution environment. In short, grid computing provides an interface to heterogeneous resources, which can be across multiple administrative domains [16].

2.2 CERN

The European Organization of Nuclear Research, known as CERN, is an organization that has been researching the elementary particles and forces that build this universe since 1954 [15]. CERN has a particle accelerator called the Large Hadron Collidor (LHC), which is the largest and most powerful particle accelerator in the world [9]. It currently (Jan. 2024) supports nine experiments, each of which is part of a collaboration of scientists and institutions worldwide[8]. The experiments use specific detectors (located in the LHC) designed to capture different physical phenomena.

2.3 ALICE

The ALICE experiment, functioning as a versatile, heavy-ion detector located within the complex of the CERN Large Hadron Collider (LHC), is dedicated to the comprehensive examination of physics observables originating from collisions of particles [11]. Notably, the data generation is formidable, amounting to tens of petabytes during a typical operational year. A portion of this extensive data is utilized for subsequent in-depth analysis [28] [24].

2.3.1 ALICE Computing

ALICE leverages the World LHC Computing Grid (WLCG), a sophisticated computing infrastructure, in conjunction with the Java ALICE Evironment (JAliEn) grid middleware for the storage, processing, and analysis of data [24]. The WLCG serves as a pivotal computational resource reservoir consisting of multiple grid sites located worldwide, offering support to numerous CERN initiatives.

2.3.2 CernVM File System (CVMFS)

CVMFS is the current software distribution for the ALICE grid. It is a read-only file system, whose purpose is to deliver software in a fast, scalable, and reliable way [20]. The system is implemented as a POSIX (The Portable Operating System Interface) file system [7], which can be added to an existing file three [40]. Software stored within the CVMFS can be run directly from it, without any additional installations.

2.3.3 Java ALICE Evironment (JAliEn)

ALICE utilizes the grid middleware JAliEn. It consists of a collection of several services and serves as the user interaction with the ALICE grid. The interaction is a Linux-like console, in which certain commands are available such as job submission. Users can connect to JAliEn using their CERN-issued grid certificate. Interactions with its services are also possible via scripts connecting to the WebSocket endpoint. An example of this is the Python package Alienpy [39], which is a package that connects to the WebSocket endpoint to enable interactions with JAliEn via Python scripts.

One of the main components of JAliEn is JCentral. This component is responsible for the job queue, named TaskQueue. A lot of functionality is related to this queue, such as job submission, registering job outputs, job status transition, job matching, and job tracing. All of which is managed by JCentral. It also controls the data management, deciding file placement for every grid job (such as the trace file of a job). The location of these files is annotated in a File Catalogue, which users can interact with (via JAliEn) to locate and access these files [24].

2.3.4 MonaLISA

ALICE utilizes the MonaLISA system to monitor its grid sites. It is a monitoring system, which is built up of independent self-describing agent-based subsystems, registered to MonaLISA as dynamic services [5]. These services are responsible for monitoring and passing along the information. Its agent-based architecture enables additional services to register themselves, to be utilized in other services or clients that depend on this information. MonaLISA has a layered architecture, consisting of four logical layers illustrated in Fig. 2.1. The first layer consists of "regional or high-level services (HLS)", namely repositories and clients. This layer exists for the consumers of the monitored data, which they can utilize for storage, analysis, etc. The second layer consists of proxies, enabling secure and reliable communication in a scalable way. The proxies can multiplex the data received from MonaLISA services to all the subscribed clients, meaning that the services only have to send the same information once. The third layer consists of the services, responsible for the monitoring tasks. The fourth layer is a lookup service (LUS), a network of services enabling dynamic registration and discovery of the other system components found in the other layers. MonaLISA services

can find each other and be found by clients needing their monitoring data.

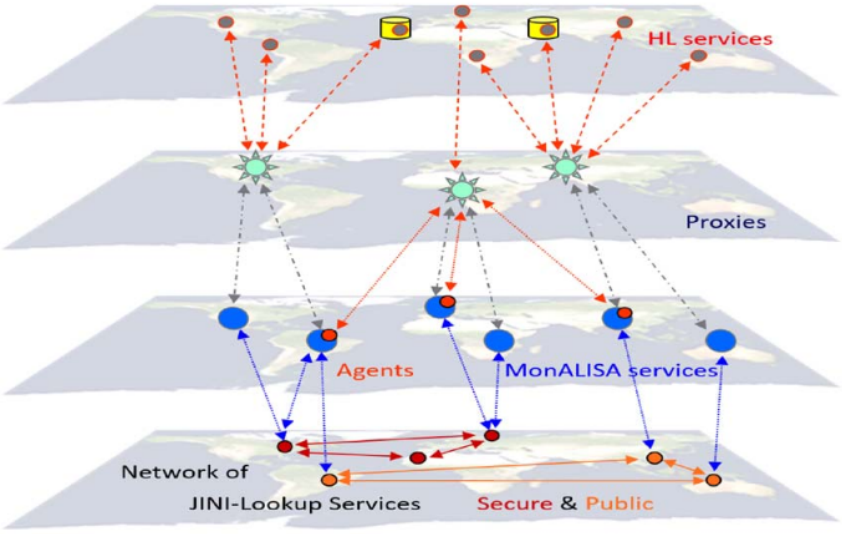


Figure 2.1: The four logical layers of the MonALISA architecture, from top to bottom: 1. high level services, 2. proxies, 3. distributed agents, 4. lookup services. [5]

2.4 Machine Learning

Machine Learning is described as a computational paradigm where algorithms develop the capability to solve problems through the analysis of historical data [22]. This is done by training a machine-learning model. A model contains both an architecture, which describes the structure, and a set of numerical weights that are optimized during training. The architecture of a model defines how the weights connect to each other, and how inputs interact with these weights. There exist different learning approaches in Machine Learning, one of which is supervised learning.

2.4.1 Supervised Learning

The supervised approach utilizes data that has a defined label for each input, a target value that we want to predict as accurately as possible. The training

process consists of passing an input through a model that outputs a predicted value. This value is then compared to the label to calculate loss (or other measurements), which quantifies how far the predicted value is from the label. The goal of the training is to minimize this loss. Several mathematical functions can be utilized to calculate loss, but a common example is the Mean Squared Error (MSE), illustrated below:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.1}$$

MSE calculates the average squared difference between each predicted (y_i) and target ($\hat{y}_i$) value [14]. The weights are updated concerning this loss, where a common optimization technique is gradient descent. First, we calculate the derivative of the loss function (here MSE) with respect to each individual weight. Then, we utilize gradient descent in order to minimize loss. This is done by following the negative gradient of each of the derivative functions [3]. In essence, the derivative will show us whether increasing or decreasing the weight will minimize loss, and we alter the weights accordingly. This is often done in batches, which are subsets of the total training data. Training in batches ensures that model optimizations are based on several inputs. A separate validation dataset, which is not exposed to the model during training, is utilized to measure how well the model performs on unseen data. Models can train for several epochs, where an epoch is one training iteration where the model has "seen" each input exactly once.

2.4.2 Transformer architecture

The transformer architecture was first proposed in the paper "Attention Is All You Need" by Ashish Vaswani et al [42], and has since been proven exceptionally well in Natural Language Processing (NLP) [6] and computer vision tasks [23]. Transformers leverage self-attention mechanisms to focus on important parts of the input and ignore less relevant parts. The suggested self-attention mechanism by Vaswani et al is called Scaled Dot-Product Attention, illustrated in Fig 2.2.

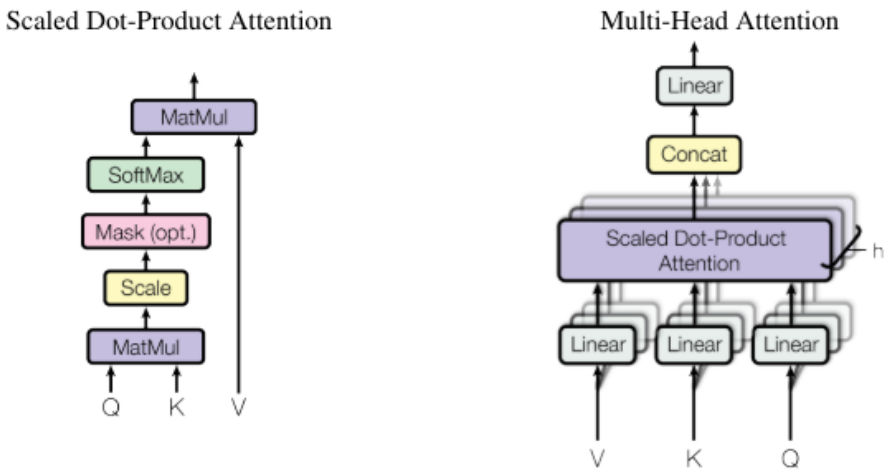


Figure 2.2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel [42]

The Scaled Dot-Product Attention utilizes three vectors, denoted query(Q), key(K), and value(V). The article "Unpacking the Query, Key, and Value of Transformers: An Analogy to Database Operations" by Mohamed Nabil explains the purpose of these vectors as such [26]:

"In transformers, the query is the information that is being looked for, the key is the context or reference, and the value is the content that is being searched."

These vectors are derived from linear layers, where each linear layer corresponds to a specific purpose(query, key, or value). Linear layers perform a matrix-vector operation, where the input vector (x) is scaled by a matrix (W)[25]:

$$y = xW \tag{2.2}$$

The matrices in the linear layers consist of trainable weights, which are optimized in the training phase. The outputs are transformed representations of the input, learned during training. A single element of the input is passed through the linear layers at a time, where an element can be a vector if the input is a matrix (this is often the case in NLP). This results in a collection of vectors that are merged in their respective matrices (Q, K, V), where different rows contain information from separate parts of the input.

The query and key matrices are combined through a series of operations, resulting in a matrix of attention scores. This matrix is referred to as an attention matrix and contains information about how strong the relations are between the elements in the input. This matrix is applied to the value matrix to emphasize parts of the input that are more relevant than others. This is how the transformers generate contextual relevance in the input.

Several self-attention mechanisms are applied to the input in parallel in transformers. This is called Multi-Head Attention, where multi-head refers to multiple self-attention mechanisms and thus multiple query, key, and value matrices. Several linear layers are utilized for each of the matrix types (Q, K, V), such that each "Head", which is a single instance of self-attention, produces a different representation of the input. The outputs from the different heads are concatenated and transformed, resulting in a rich, contextual representation of the input.

The transformer architecture proposed by Vaswani et al (Illustrated in Fig. 2.3) is an encoder-decoder structure. The encoder processes the input, while the decoder produces an output. The suggested decoder is auto-regressive, meaning that one output element (i.e. a word) of the complete predicted output is generated at a time. Complete outputs are produced through iterations, where the current (incomplete) version of the predicted output is given as input to the decoder in each iteration. More detailed information about the transformer architecture can be found in the papers "Attention Is All You Need" by Ashish Vaswani et al [42] and "Transformers in Vision: A Survey" by Salman Khan et al [23].

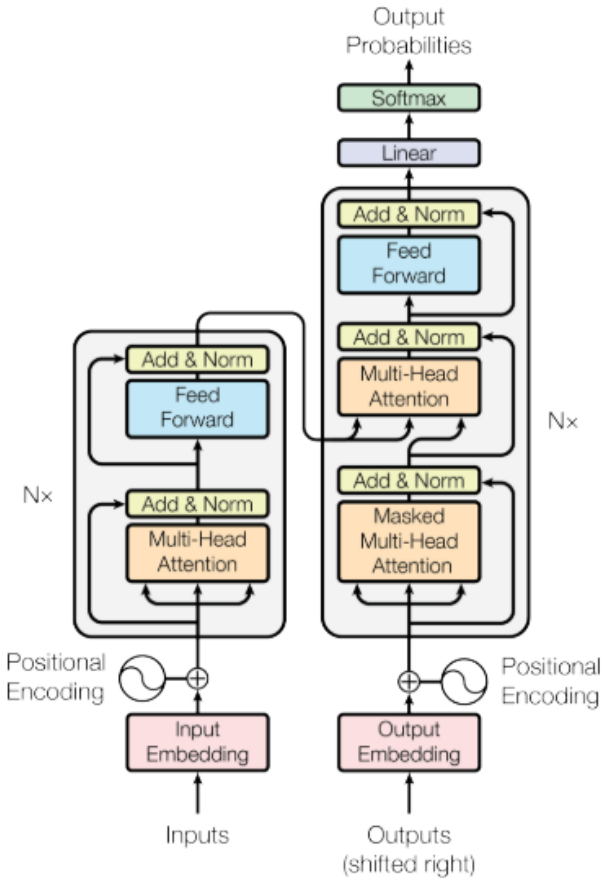


Figure 2.3: The Transformer - model architecture. [42]. Consists of an encoder(left) and a decoder(right).

Chapter 3

Methodology

This chapter discusses the methodologies utilized in this project. It is important to have a research methodology, as this structures the project, as well as increases the efficiency, validity, and reliability. The software development branch is a bit unique within the realm of research, as the implementation of various systems is a big part of such projects. They need to address the narrative of the project and work within its defined environment.

The project utilizes the Design Science methodology as its research methodology. It describes an iterative process of implementation and evaluation, as well as ensures that it is built on a knowledge base in the respective field. The following section discusses the different aspects of Design Science as described in the paper "A Three Cycle View of Design Science Research" by Alan R. Hevner [17].

3.1 Design science

Alan R. Hevner discusses how the interaction between the Environment, Design Science Research, and Knowledge Base works in a Design-science project. The form of the interaction is through cycles, or iterations included in the development of an application. It is a way to ensure that multiple prospects are considered in the process of implementation. The three cycles; Relevance, Design, and Rigor cycle form a methodology for software development, in which multiple aspects are considered.

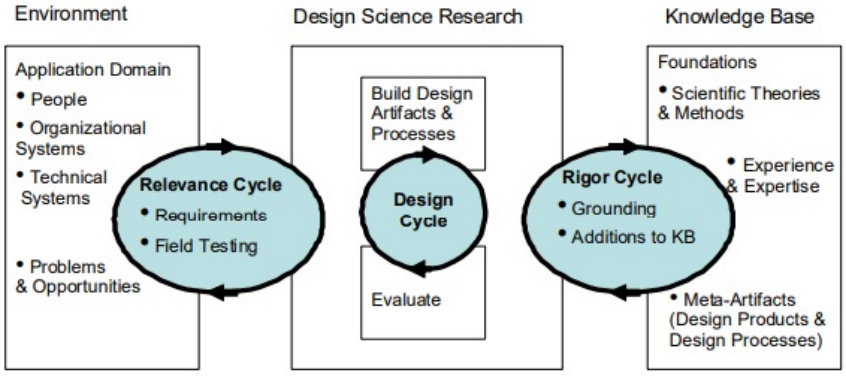


Figure 1. Design Science Research Cycles

Figure 3.1: The design sciences research cycle [17]. Illustrate the iterative nature of the methodology by three cycles, Relevance, Design, and Rigor cycle. It is a methodology that ensures that environment and knowledge have been considered during the development of a system.

According to Alan R. Hevner, the design science cycle can be reduced to three central components, combined by three cycles: Relevance Cycle, Design Cycle, and Rigor Cycle.

3.1.1 The Relevance Cycle

The relevance cycle defines the interaction between the environment, called the application domain, and the Design science research. The environment possesses certain system requirements, which are defined by people or systems on which the application will be dependent. It can produce problems and opportunities, in which some acceptance criteria are set. The cycle defines an iteration of field testing, where the application will be tested directly toward the acceptance criteria, which again will determine whether additional iterations are necessary. The iterations might also show the need for additional requirements, in which further acceptance criteria can be included for the subsequent iterations. This helps ensure that the adoption of the application in the given environment will work within the requirements posed by its domain.

The project environment in terms of the Design Science model consists of project supervisors, a test environment for the code, and ALICE-related

systems. These have posed certain requirements to the system developed in the project. The relevance cycle consists of defining certain key features, implementing them, and exposing them to the test environment, as well as the project supervisors. Weekly meetings with the supervisors have been held during the project. These meetings usually consisted of presenting the work done and receiving feedback in the form of additional requirements or tips for solving current problems.

3.1.2 The Rigor Cycle

The Rigor Cycle ensures that past knowledge has been considered in the development of the application. There exist practices, principles, systems, and design patterns that have been proven successful in the development of an application. This knowledge might not only speed up the development process but can also help establish a more profound foundation. It is however worth mentioning that not all aspects of the application should be derived from previous knowledge, as this might be a blocker for the process of innovation. The scientific findings obtained in the current design-science project might one day appear in the knowledge base for another, as this is the progression of science.

The project has leaned on experience and expertise in its development process. The expertise of supervisors and the JAliEn development team has helped the project a lot, especially concerning the utilization of ALICE-related systems. Litterateur and tutorials have been quite helpful as well in terms of design and development of different systems in the project. Several aspects of this project were initially unknown to the author, in which the rigor cycle was visited to obtain the knowledge required.

3.1.3 The Design Cycle

The Design Cycle is the core of design science projects. Through an iterative process of development, new features are added and evaluated to ensure that they work as expected. The results from the evaluation work as feedback to the implementation, to make sure that requirements (given from the Relevance cycle) are upheld. The essence of the design circle is to have a fine balance between development and evaluation, as well as ensuring that the relevance and rigor cycle is present in both.

The project's design cycle mainly consisted of implementing features, defined in the relevance cycle, and evaluating them. The evaluation for certain features was in the form of unit tests, while other features relied only on user tests to confirm that they functioned as intended. The implementation was revisited in the case of undesired functionality and further tested until a satisfactory result was reached.

3.2 The Machine Learning Lifecycle

The project utilizes Machine Learning for its data analysis, in which the Machine Learning Lifecycle framework (fig. 3.2) would be natural to utilize in this project. The ML life cycle is a development framework that divides the development of a Machine Learning model into specific stages, structuring an ML project [4]. These stages include:

- Business goals: Define an overall goal
- ML problem framing: Define how the model would help us reach this goal
- Data Processing: Define data sources, collect and process data, and feature engineering
- Model Development: Define an appropriate model for our data and goal. Train the model with different parameters to obtain an optimal model
- Deployment: Expose the model to a production environment
- Monitoring: Ensure that the model serves the overall goal. Further development of the model might be needed.

It can be adopted in a broad variety of ML projects as it describes the necessary steps in the creation of a model. We will further discuss how these stages were utilized in the project.

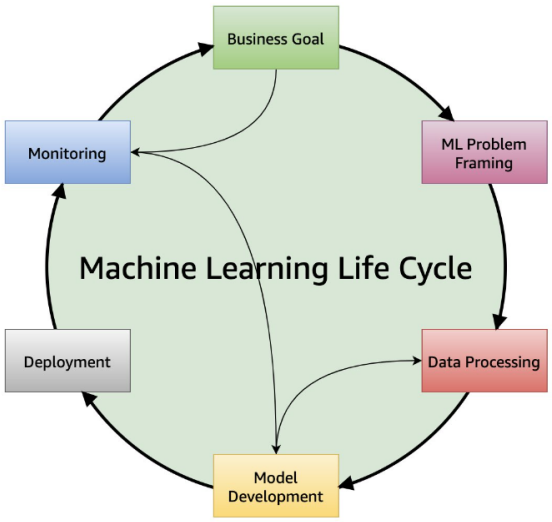


Figure 3.2: The lifecycle of Machine Learning development framework [4]. Divides the development of a model into six different stages, structuring the development process. The first stage is defining a Business Goal.

3.2.1 Business goal

This project aims to find causes of job failure, with an overall goal of reducing computational waste for the ALICE project, focusing on the grid site at the University of Bergen. The waste in question is resources tied up in the execution of failing jobs. This is a broad definition, but it provides the constraint that the data in question needs to be related to a job execution.

3.2.2 ML Problem Framing

How can Machine Learning help achieve the goal of reducing computational waste? There are several choices on model architectures and overall approach. The discussion of whether to use supervised or unsupervised learning depends on our overall goal. Unsupervised learning is a great tool for discovering underlying structures and relationships in our data. This could give us a broader understanding of the data patterns for successful and failing jobs. Supervised learning is another approach, where we would try to predict a target value for our data. The natural target value of a job's end status incentivizes the project to use this approach. We therefore defined the ML

problem as: "Can Machine Learning predict whether a job execution succeeds or fails?". This is a binary classification problem, meaning that there are two possible classes that the model can predict, namely success or failure.

3.2.3 Data Processing

Data processing is an important step in every ML project. It incorporates the process of data collection, feature selection (which attributes to use as input), and feature engineering (creating new attributes with the data available). The data utilized in the training process of the model directly influences the performance, robustness, and generalization capabilities of a model. Utilizing data relevant to the problem is a key factor in ML.

The first step in the data processing stage is data collection, making the data accessible for our model. Possible data sources are limited by our business goal and ML problem, which states that the data must be related to job executions and have a corresponding label. The computing environment of a job, namely the grid site, is a data source that upholds these conditions.

ALICE utilizes MonaLISA to monitor its grid. This system comes with a GUI, which (amongst other things) gives an overview of the several services monitored by MonaLISA (Fig. 3.3). Services related to the UiB grid site are grouped, which gives us an easy overview of the available data.

There are several services monitored at the UiB site, each of which contains different measurements. These need to be tied to a job execution to address our ML problem. Additionally, we need a target value for our data, namely job end status. This information is not available through MonaLISA, meaning that we need an extra data source.

The ALICE grid middleware JAliEn traces each job execution. Each trace is available in the trace file, which essentially is a log containing information such as which grid site executes the job. This file contains valuable information that can be utilized to link data from the MonaLISA services to specific job executions, such as job-id and worker node (computer at the grid site executing the job). Each job also has a related JDL file (Job Description Language file), which can be accessed through JAliEn. Certain properties are defined in this file such as the TLL (Time To Live), job arguments, and the executable file for that specific job [2]. This file might also be of interest, as it contains additional information about the job execution. JAliEn

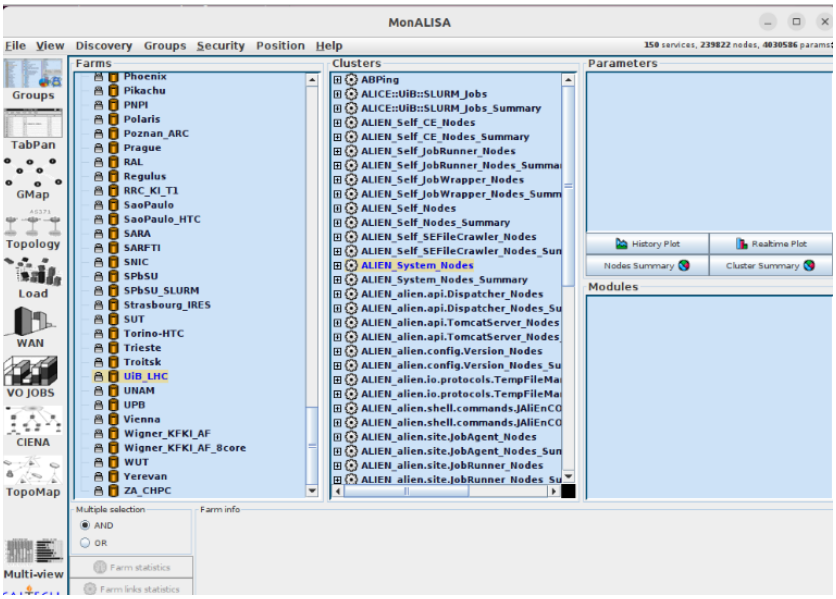


Figure 3.3: Screen snippet of the MonaLISA GUI. It provides an overview of the available services. The services are grouped into farms, and their measurements are grouped in Clusters.

is also responsible for job status transitions, assigning status to jobs. There are several possible statuses for a job, such as "RUNNING", "DONE", and several different error statuses [21]. These statuses are accessible via JAliEn. The project will focus its efforts on jobs that ended with the status "ERROR_E", jobs that terminated with an execution error, and "DONE", jobs that successfully terminated.

A combination of these two data sources, MonaLISA and JAliEn, provides the necessary information to access resource usage related to a job execution. The project stores this data locally, to ensure its availability for longer periods. The project utilizes the grid site of the University of Bergen as its test site but aims to be versatile and usable throughout the ALICE grid. This provides the acceptance criteria that the systems responsible for collecting this data are configurable.

Data management is important, especially when operating with several data sources. The project uses the term "data management" to describe the process of obtaining and utilizing the data after it is stored locally. Data need to

be retrieved and held in an organized way. A data representation in the code, that encapsulates data retrieval and operations, makes it easier to utilize the data further.

3.2.4 Model Development

The Model Development stage is the stage where we choose an appropriate Machine Learning model for our problem and train it on our data. The model is constrained by data type and our defined learning approach. The project operated with time-series data, a version of sequential data, which it aims to utilize for supervised learning. The Long Short-Term Memory (LSTM) architecture is quite suitable for such tasks, as it can capture long-term dependencies in sequential data, where the information from previous inputs is available when processing the following inputs [18]. The Transformer architecture can also be utilized to process sequential data [43] and has (as mentioned in the background) proven quite well on different tasks. This incentivizes the project to research whether job end-status prediction is one of these tasks.

3.2.5 Monitoring and Deployment

This stage is dedicated to the deployment of a model, in which it is placed in a production environment. The continuous monitoring of a model is essential, as cases might occur that are not present in either the training or validation set. This serves as a verification that it works as expected. Further training might be necessary, which the monitoring process should show.

3.3 Development method

This project has a focus on quick development. The development method utilized consisted of a planning phase, followed by an implementation phase. In the planning phase, system models were created and key features determined. These were the first features implemented, and new functionality was implemented when needed. The described method does not perfectly align with any specific software development methodologies, but it shares some common traits with Agile development, as it has an iterative cycle of adding functionality [32].

Chapter 4

Design and Implementation

This chapter discusses the design and implementation of the various subsystems developed in this project. There are two data collectors and one data analysis system utilized, where one of the data collectors is an already existing tool that is further configured. A simplified overview of the complete system is illustrated in Fig. 4.1.

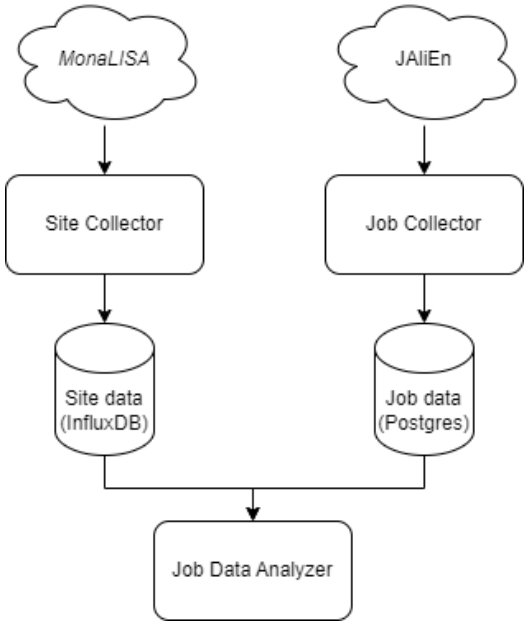


Figure 4.1: A simplified model architecture of the complete system developed in the project. Two data collectors are utilized, one that stores data from MonaLISA and one from JAliEn. Both of the collectors have their respective databases due to the type of data they retrieve. This data is further utilized in the Job Data Analyzer.

The data utilized in the project are stored in two databases; a time-series Influx Data database[19], and an object-relational PostgreSQL database [27]. The reasoning for this is the data type collected in the respective collectors. The Site Collector gathers site attributes at different timestamps, a time series datatype. The simplest way of storing this data is to utilize a time-series database, such as InfluxDB. InfluxDB operates with buckets instead of tables, where each bucket groups the same fields and differentiates separate measurements with a timestamp. The Job Collector stores job attributes related to specific job executions. All jobs contain the same properties and are collected once per job execution. This presents a more "tabular-like" dataset, in which a relation database is a fitting structure, storing the properties in a relational table consisting of columns and rows.

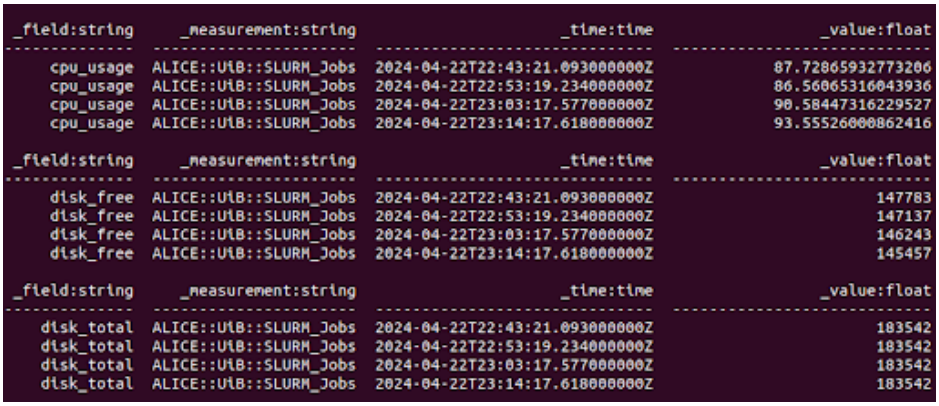
Note: The test environment for this project is set up on a computer node at the University of Bergen (separate from its grid site), accessed through

an SSH tunnel. This node is set up with PostgreSQL, Influxdb, Docker, CVMFS, Python, and Java. The node is also set up with an additional disk of 100TB.

4.1 The Site Collector

The purpose of the Site Collector is to collect grid site attributes from the services monitored by MonaLISA. There already exists a GitHub repository for this exact purpose, called EPN2EOSMonitor [38]. This application registers a new client to MonaLISA to access data from the EPN2EOS data transfer, which is a service monitored by MonaLISA. The exact purpose of this service is not relevant to the project and will therefore not be discussed. The data it receives from this service is further stored in an InfluxDB bucket, whose properties can be configured in an influx.properties file. This application can be tailored to our purposes by redefining which service to gather data from. An additional property file was created in our forked version of this repository: config.properties. It serves as a simple way of re-configuring the client such that the Site Collector can collect data from different grid sites, addressing our acceptance criteria that the developed system in this project is usable across the ALICE grid.

The application can only be configured to store data from one service at a time with the current implementation. The application is therefore running as a docker container. A container is a sandboxed process, meaning that it runs in a separate, isolated environment. The container has all the required dependencies within this environment, where it runs its software and configurations [13]. Data from each service that is of interest is collected through an instance of the Site Collector, storing it in their own respective InfluxDB bucket. A screen snippet of data stored in one of the project buckets is shown in Fig. 4.2.



_field:string	_measurement:string	_time:time	_value:float
cpu_usage	ALICE::UiB::SLURM_Jobs	2024-04-22T22:43:21.093000000Z	87.72865932773206
cpu_usage	ALICE::UiB::SLURM_Jobs	2024-04-22T22:53:19.234000000Z	86.56065316043936
cpu_usage	ALICE::UiB::SLURM_Jobs	2024-04-22T23:03:17.577000000Z	90.58447316229527
cpu_usage	ALICE::UiB::SLURM_Jobs	2024-04-22T23:14:17.618000000Z	93.55526000862416

_field:string	_measurement:string	_time:time	_value:float
disk_free	ALICE::UiB::SLURM_Jobs	2024-04-22T22:43:21.093000000Z	147783
disk_free	ALICE::UiB::SLURM_Jobs	2024-04-22T22:53:19.234000000Z	147137
disk_free	ALICE::UiB::SLURM_Jobs	2024-04-22T23:03:17.577000000Z	146243
disk_free	ALICE::UiB::SLURM_Jobs	2024-04-22T23:14:17.618000000Z	145457

_field:string	_measurement:string	_time:time	_value:float
disk_total	ALICE::UiB::SLURM_Jobs	2024-04-22T22:43:21.093000000Z	183542
disk_total	ALICE::UiB::SLURM_Jobs	2024-04-22T22:53:19.234000000Z	183542
disk_total	ALICE::UiB::SLURM_Jobs	2024-04-22T23:03:17.577000000Z	183542
disk_total	ALICE::UiB::SLURM_Jobs	2024-04-22T23:14:17.618000000Z	183542

Figure 4.2: A (incomplete) screen snippet from the data stored in an InfluxDB bucket. Similar fields are grouped and differentiated by a timestamp. Each field measurement contains (amongst other things) the name of the field and the measurement, a timestamp, and a corresponding value.

4.2 The Job Collector

The purpose of the Job Collector is to store job attributes related to job executions. This is done by retrieving, filtering, and storing data accessed via JAliEN. The collector has a service-oriented architecture, whose services are managed by a central controller as illustrated in Fig. 4.3.

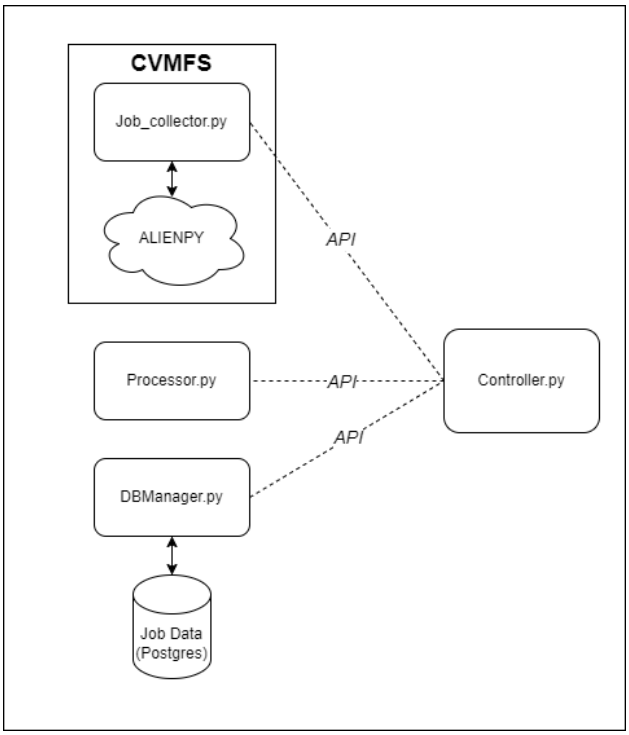


Figure 4.3: Job Collector Architecture, designed as a micro-service architecture. It contains a central controller that utilizes three services (via their respective APIs) to collect, filter, and store job properties. The system utilizes a Postgres database to store its collected properties.

The application contains four components:

- **The Controller** is responsible for the application logic and application flow, integrating all of the services into the application.
- **The Job Collector** service collects job attributes by interacting with the Python version of JAliEn, Alienpy. It encapsulates alienpy calls in functions with descriptive names, such as getJDL and getTrace.
- **The Processor** service is responsible for processing and parsing all data, formatting it for further use.
- **The DBManager** service is responsible for all database operations. It is the only component connected to the PostgreSQL database, where we store the job properties. It contains functions that encapsulate SQL queries, populated by the function input.

The decision of a service-oriented approach was based on the application's purpose. It would need features such as data collection, filtering/cleaning, and storage. These components would need a natural integration into the application. Dividing them into different services separates the domain of concern as well as keeping the application organized. It was initially developed as a highly coupled monolithic architecture, but during the development of the Job Data Analyzer (which will be discussed in the next section), it became clear that it could make use of the services within the Job Collector. A migration to a micro-service architecture was done to address this. An "REST-like" API layer is attached to each of the services. This API maps its calls to functions within the respective services, making them accessible throughout the system (they currently only available through localhost).

The application is dependent on some initialization parameters, defined in the configuration file config.json. It contains information about the database (table name and fields to store), grid sites to monitor, application iteration interval, and the URLs for the respective APIs. This configuration enables it to be set up in different environments and collect job properties from jobs executed on different sites. The application starts with an initialization phase, where the controller reads the configuration file and sets up the database table (if it doesn't exist) with help from the DBManager. It then starts its first iteration, which consists of the following:

1. Collect id and job-status of running jobs: JAliEn keeps an overview of

running/recently run jobs (Collector).

2. Remove jobs that are still running or are already stored in our database (Controller / DBManager). Some end statuses are ignored, such as zombies.
3. Collect trace and JDL for the remaining jobs (Collector).
4. Parse and filter out the properties defined in the config.json file (Processor / Collector)
5. Store the properties in the database (DBManager)
6. Repeat steps 1-5 at iteration interval (Controller)

4.3 Job Data Analyzer

The Job Data Analyzer system is where we utilize our collected data. It is designed as a tightly coupled application that handles data analysis tasks and machine learning. The system is built up of three main components; Data Management, Data analysis, and Machine Learning.

All systems within the Job Data Analyzer have their dedicated purpose, but they all share a common component; a Job Data Handler. This component connects to the DBManager service within the Job Collector to collect job attributes. This is the only component that interacts with the Job Collector, illustrated in Fig. 4.4. It comes with some predefined functions to handle the job attributes, one of which is dividing them into error and success attributes.

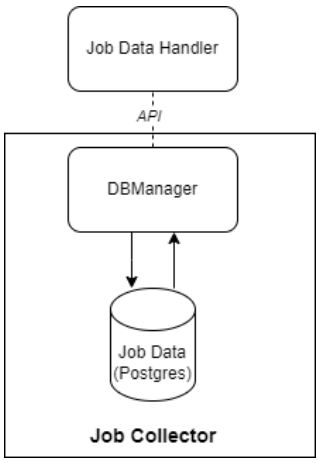


Figure 4.4: Model of the interaction between the Job Data Handler and the DBManager in the Job Collector. The interactions are done via an API, which enables the Job Data Handler to utilize database operations.

4.3.1 Data Management

This system has an object-oriented data management approach. All data related to a job execution is stored within a Python object called a DataBox. The box holds the data and contains all data operations related to a single job execution. For data operations on multiple jobs, a DataTerminal is utilized, which essentially is a collection of DataBoxes. The DataTerminal is initialized with a collection of job parameters, which is used to initialize a collection of DataBoxes. The attributes needed are Job ID, worker node, and start/end execution time, which are accessed through the Job Data Handler. These attributes are used to filter out the grid site data related to a job execution. The DataBox utilizes an InfluxManager for this, which populates predefined queries with job attributes to obtain the correct site data. The data flow is visualized in Fig. 4.5.

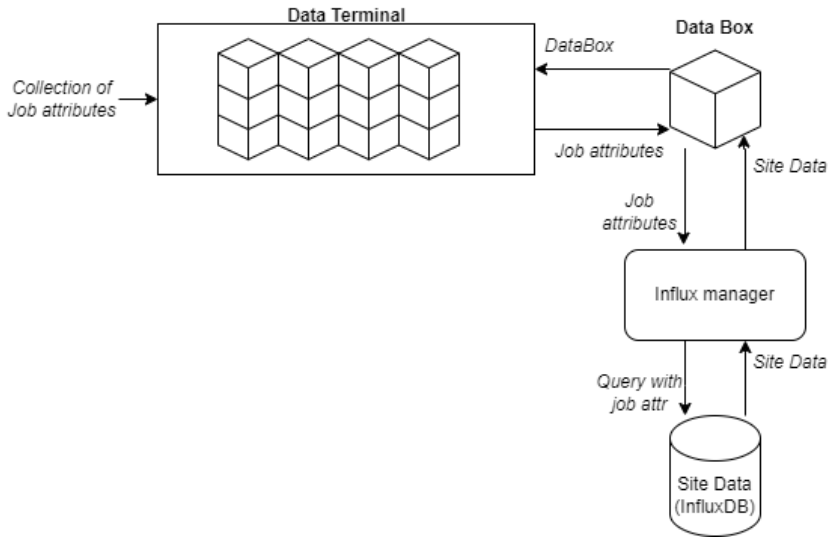


Figure 4.5: Model illustrating the flow of data in the Data Management component. A collection of job properties is passed as an argument to the DataTerminal, which utilizes it to create new instances of the DataBox. Each DataBox is initialized with a set of job parameters, which all relate to the same job execution. These are further passed to the InfluxManager, which populates predefined queries with the job properties to filter out the grid site data related to this job. This data is held within the DataBoxes, which are managed by the DataTerminal.

There are two configuration files associated with Data Management; data_conf.json, which enables us to define which influx buckets to load data from, and data_cleaning.json, used to define which fields to keep from each bucket as well as which fields to log (for ML usage). These files are loaded in the DataTerminal and its information is passed to each DataBox. These files give us customization options, enabling us to retrieve data from several buckets and filter out fields we want to study further.

The site data, retrieved from InfluxDB, is initially loaded into a dictionary called "data" in the DataBox. The key-value pair in the dictionary is (bucket name, bucket data), where bucket data is a Pandas DataFrame, a data structure for tabular data [1]. The several DataFrames (one for each bucket), held as values in the data dictionary, are further merged into a single DataFrame, where each field gets its column and values with the same timestamp are

grouped in the same row, as illustrated in Fig. 4.6. The project refers to this DataFrame as a Data Matrix. The matrix contains all the grid site data related to the job execution. It has a more "human-readable" format, and further data operations are quicker, both implementation and computational-wise since we don't have to group similar fields before applying the operations.

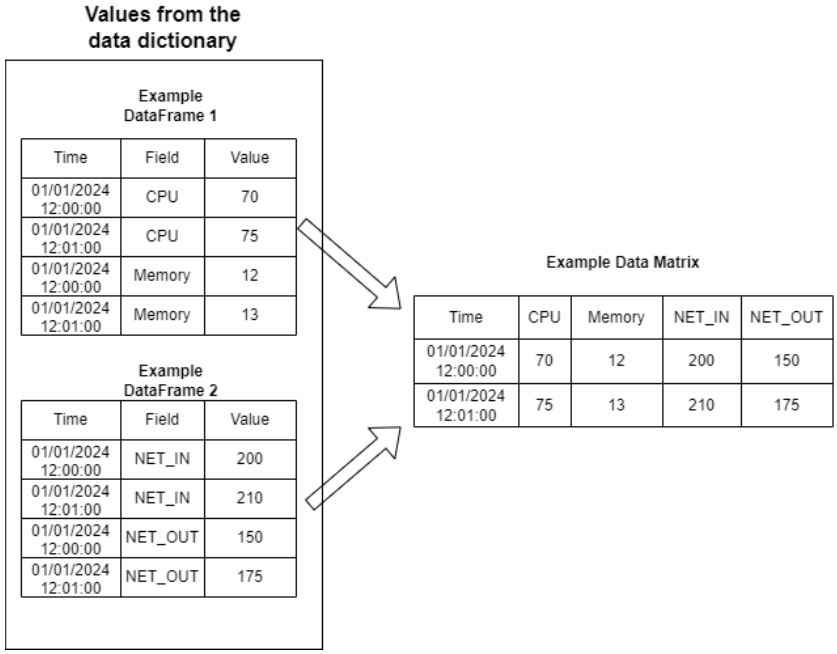


Figure 4.6: A model illustrating the creation of the data matrix. The DataFrames held in the data dictionary are merged into a single DataFrame, called a Data Matrix, where similar fields are grouped in the same column and values with identical timestamps in the same row. All tables in this model solely serve as an illustration and only contain example data.

4.3.2 Data Analysis

The Data Analysis component utilizes the DataTerminal and a visualizer to compare and visualize the data. It is implemented as an interactive Python script, where certain predefined functions can be run. Some configuration is possible through the script, such as defining how many jobs to compare. The functionality is mainly geared towards data analysis tasks, but it also

contains a function for ensuring that given job attributes have corresponding site data. The Data analysis component contains the following operations:

- **Filter out job attributes that contain grid site data**: This creates a new database table within the Job Collector. The table consists of job attributes columns aligned with a "bucket" column, which indicates that these attributes have associated data in the corresponding bucket. This is the table we load attributes from in all other operations, hence this function needs to be run first.
- **Correlate features to label**: This operation calculates and plots the Pearson correlation score for field - job end status pairs. All available buckets and fields are used in this operation.
- **Plot comparison of defined fields**: This operation compares the values between failed and successful jobs. It plots the field values ((x, y) $\Rightarrow$ (execution time, field value)), and two boxplots (one for done and one for error jobs) of all fields defined in the data_cleaning.json file. The failed and successful jobs are differentiated with a different color in the value plot. The plots generated from this function are merged into two PDFs, one for the value plots and one for the boxplots for easier studying of the data. This operation is only done on the fields defined in the cleaning_conf.json file.
- **Create field to field plot**: Plots a pair of fields, one on the x-axis and one on the y-axis. The failed and successful jobs are differentiated with a different color in the plot. The project has not utilized this operation in its analysis, but the idea is to research if there exists a correlation between fields, and if that differentiates between done and error jobs.

All data operations related to job executions are contained within the DataTerminal and DataBox. The visualizer displays the given data via plotting or merging plots to PDFs. The data analysis component works as an intermediary between the Job Data Handler, DataTerminal, and Visualizer (Illustrated in Fig. 4.7). It provides the application flow.

This tool can be utilized to study our data more carefully. The idea is to get a broader idea of how the fields differentiate between failed and successful job executions. This is especially useful in the feature extraction process, which is the process of selecting which fields to utilize as data for the machine learning model.

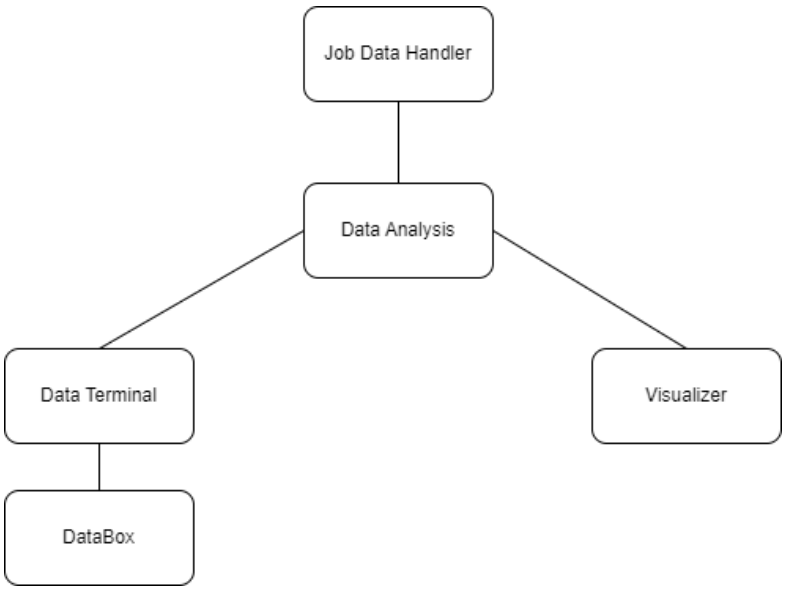


Figure 4.7: Model of the Data Analysis component. The data analysis is an interactive script that utilizes the DataTerminal, to retrieve grid site data and perform data operations, as well as a visualizer to display the data in the form of plots or PDFs.

4.3.3 Machine Learning

Machine Learning is the last part of the Job Analysis system. It consists of three main components; a training script, a custom dataset, and a model. It is implemented with the PyTorch libraries, which is a tensor library for deep learning [29]. The training of this model is done through the train_model script, which utilizes a custom dataset (via a data loader) to train a machine-learning model. The job attributes are accessed through the Job Data Handler, like the other components in the Job Data Analyzer system.

Our dataset, the JobDataset, is implemented as a class that inherits functionality from the PyTorch Dataset module. This module functions like an abstract class, which can be customized to load and convert our data to a format that the model understands. The models within the PyTorch library expect a tensor input, which essentially is a numeric list/matrix that can be loaded and utilized on a GPU. These custom Datasets can be utilized in a PyTorch Dataloader, which creates an iterator for the Dataset. The

JobDataset is initialized with a collection of job attributes. It consist of two main functions

- len(): Returns the number of elements in the collection.
- get_item(i): Takes an index as argument. Retrieves the job attributes located at this position to create a Data Matrix (via the DataTerminal). This matrix is converted to a tensor, which the function returns.

The DataLoader creates an iterator with the help of these functions by passing indexes in the range of the dataset to the get_item function. Our dataset is initialized with a collection of job attributes. Each call to the get_item function prepares the data for a single job execution. The job attributes are fed into a DataTerminal, which filters out fields and buckets as defined in the data_conf and data_cleaning files, and creates the data matrix. This matrix is then extracted from the DataTerminal and converted to a Pytroch tensor matrix (visualized in Fig. 4.8). Such custom datasets can be passed as an argument to a PyTorch dataloader, which creates an iterator for the dataset.

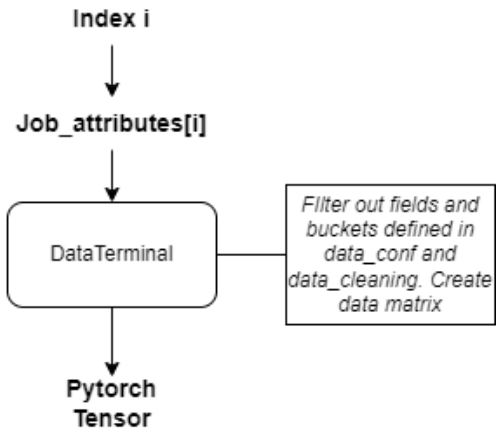


Figure 4.8: Flow of get_item function in the custom dataset. It takes an index as an argument and retrieves the job attributes located at that position in the collection. These attributes are utilized to initialize a DataTerminal, which creates the Data Matrix. This matrix contains all the (grid site) data related to a single job execution. The matrix is further converted to a tensor, which the function returns.

The model is a class that inherits functionality from the PyTorch nn.Module.

This also functions as an abstract class, in which we can customize a machine learning model. PyTorch comes with predefined machine-learning layers (i.e. a transformer), which we can combine to create a custom model. The project's model, called StatefulTransformer, contains three components:

- A transformer layer (transformer model architecture) to process the input.
- A linear layer to reduce the dimensionality of the output from the transformer.
- A Sigmoid activation function, which squeezes the output from the linear layer to a value between 0 and 1. The output from the Sigmoid function is the prediction for the processed input.

The interaction between the input and these components is defined in the "forward" function of the model, which is a function that all PyTorch models are expected to have. The model is initialized with parameters, such as input size, number of heads (in the transformer), etc.

The training script utilizes the job handler to retrieve job attributes, the JobDataset to create an iterator for the data (via a DataLoader), the StatefulTransformer, and a visualizer to display i.e. training progression. Two datasets are utilized, one for training the model and one for validation. The training consists of extracting data from one job at a time and passing it through the model to make a prediction. The prediction is then compared to the true target value, the actual job end status, to calculate the loss. The loss is then utilized to update the weights to minimize loss. We iterate over the training dataset multiple times and validate it between each iteration to see how well it performs on unseen data. A simple illustration of the training process is found in Fig. 4.9.

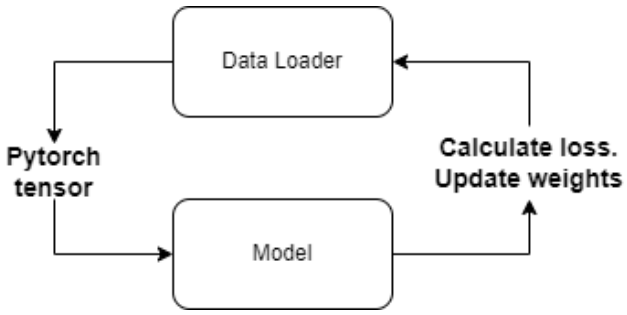


Figure 4.9: The training loop in the train_model script. The DataLoader iterates over its given job attributes to load and convert grid site data to a PyTorch tensor. This tensor is passed through the model to make a prediction, which is compared to the true target value for that job execution to calculate loss. The loss is further utilized to update the weights in the model (to minimize this loss).

4.4 Code structure

The code in this project is located in three different GitHub projects: Job Collector, Site Collector, and Job Data Analyzer. The Job Data Analyzer project is dependent on data collected from the two collectors, as well as the DBManager service within the Job Collector. The project has had a focus on object-oriented implementation, as this naturally divides functionality responsibilities. The project's source code can be found in the appendix.

Chapter 5

Analysis and Assessment

The project consists of three components which together create a system for data analysis and machine learning. This has been utilized to study the collected data and train a machine-learning model. In this chapter, we will evaluate this system design and data credibility. The chapter will also discuss how the collected data was utilized to create a machine-learning model.

5.1 System evaluation

5.1.1 Job Collector

The Job Collector is implemented as an object-oriented application with a micro-service architecture. There are several benefits to micro-service architectures. They are often quite flexible, often resulting in a maintainable, available, and scalable application [30]. The availability of this architecture comes to good use in this project. It provides the opportunity to reuse much of the functionality of the application, such as database access. The application's entry point is the controller, acting as an intermediary between the application's three services: collector, processor, and DBManager. This object-oriented design divides responsibilities and separates concerns, making the application quite organized. The services are accessed through their own respective API layer, which maps REST-like requests to the object's functions. This provides system-wide interactions (the APIs are currently limited to the local host) with the services, enabling the reuse of their func-

tionality.

Some implementations in this system are less than optimal, such as the parsing of the trace file (by the processor service). Data from the trace file is mainly parsed by pattern recognition. This method is highly vulnerable to changes in the layout, where the change of keywords in the trace could lead to faulty parsing. There exists a Java trace class, which can be utilized for parsing [41]. This tool is maintained by the JAliEn development team, which ensures its reliability and adaption to changes in the trace layout. This should ideally be integrated into the processor, as it minimizes the possibility of faulty parsing of a trace.

The services have a clear field of concern, but some logic from the processor has floated over to the controller. The system contains the needed functionality and fulfills the project's acceptance criteria for data collection by being configurable to collect job properties from jobs executed on different sites at the ALICE grid. The services and the controller have been unit-tested which ensures that they function as expected. Additional integration tests are done via user testing, to verify that the complete system works as intended.

5.1.2 Site Collector

The Site Collector is an already existing tool, which is further extended to accept some additional configurations. It is not a lot to evaluate, given that it is not developed in this project. The additional configuration does however enable data collection from several different MonaLISA services, which addresses the project's acceptance criteria for data collection.

5.1.3 Job Data Analyzer

The Job Data Analyzer contains logic for job data management and analysis. The system was developed in the later stages of the project, where time limitations made for some "less than optimal" decisions to finish the prototype and make some room for data analysis. There are optimization opportunities in this system, some of which will be discussed in this section.

The data is managed mainly through two objects; the DataBox and DataTerminal. All logic in terms of data transformations and statistical calculations are done through these objects. Representing the data as an object is an

organized approach, as data operations are confined within the object that holds it. It simplifies the further use of the collected data as it can be retrieved by initializing the object. The process of initializing the DataTerminal could however be optimized, which currently is done by passing it a collection of job attributes. These attributes are accessed via the Job Data Handler, which interacts with the DBManager service in the Job Collector. Therefore, utilizing the DataTerminal in other programs requires that the job properties are gathered before initializing it. A better approach could be moving the functionality from the Job Data Handler to the DataTerminal. This would not only result in a cleaner architecture but also make the initialization of the DataTerminal more flexible, as we could initialize it with i.e. a time interval or an integer representing the number of jobs to load.

The collector and processor services are not currently being utilized in the Job Data Analyzer system, but their availability can be further utilized for monitoring a machine-learning model. Presently running jobs are ignored by the controller(in the Job Collector), as they do not have an end status yet. The service makes these jobs accessible such that we can predict their end status before their termination, and validate as they terminate. This can provide continuous monitoring of a model, ensuring that it performs well on present jobs and not just the historical data it is trained on.

The Data Analysis component utilizes the DataTerminal and a visualizer to perform statistical calculations and visualize the data. This is a well-suited prototype, which visualizes the data in the form of plots and PDFs. There are currently only a few possible operations available but additional functionality can be implemented on demand. The functionality for verifying that job attributes contain corresponding site data should however be moved, as it is not a data analysis task. Here we have at least two possible approaches; verification before storing (Job Collector), and verification before loading (DataBox/ DataTerminal). The simplest solution is probably to verify in the DataTerminal before loading, where we can choose not to include jobs without grid site data. In addition, with the proposed update of moving the Job Data Handler logic to the DataTerminal, the job attributes can be removed from the database since the terminal can use the DBManager service in the Job Collector.

The data analysis components have only been user-tested to verify that it function as intended. Additional unit tests should be implemented to verify

the correctness of the code.

5.2 Data Credibility

The data credibility is only assured by the queries defined in the influx manager, as well as an additional check in the Databox, which filters out data between its defined start and end execution time. The queries do however ensure that the site data retrieved are relevant to the conditions given(job id, worker node, start and end time), as they are used as filtering conditions in the queries. This gives our utilized data more credibility, although additional verification such as checking that each Databox object contains all the expected fields for each collected MonaLISA service would give us more assurances.

Currently, the project imputes missing values with the mean for each field. This is done for each individual job when creating the data matrix. It is also done for the datasets utilized to calculate the correlation, in the case that a field is missing for every measurement related to that job. This is a simple approach to address missing values, which enables us to preserve our sample size. Replacing missing values with the mean does not change the central tendency of our dataset since the mean is a measurement of this [33]. It does however introduce some bias to our data, where we assume our missing values are similar to the ones observed. A print within the calculation of the correlations shows that there exist jobs that are missing a complete field, but that they are few (in the utilized dataset) compared to the total amount of measurements (Fig. 5.1).

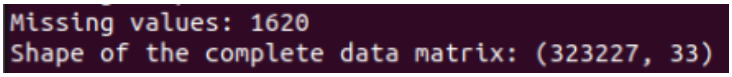


Figure 5.1: A print showing that 1620 cells (field measured at a specific time) in the dataset consisting of 500 jobs are missing values. This dataset contains fields gathered from the Self_nodes bucket. The complete shape of the matrix is printed below, showing that the complete matrix contains about 10 million cells (10 million measurements in total).

The number of missing values can be explained by one job missing three fields or three jobs missing a field, given that the average amount of rows

per job is about 650 (num rows/num jobs), which is not a lot compared to the 500 jobs utilized in the calculation. This might be an indication that the jobs missing a complete field are in the minority, but we cannot state that confidently, and thus additional checks for this should be implemented.

5.3 Machine Learning analysis

The Job Data Analyzer system provides some tools for feature selection and machine learning. These, alongside data collected from the job and Site Collectors, were utilized in the creation of a machine-learning model with the purpose of predicting whether a job would succeed or fail. A transformer architecture was selected to address this problem.

5.3.1 Data Processing

The first step in the model creation is data processing. In this step, the Job Collector was set up to collect several job-related attributes, where four of these properties are necessary to filter out the corresponding site data. These properties are:

- Job ID: A unique ID for each job execution
- Workernode: The computer node executing the job
- Start execution time: The data and time at which a job has started its execution.
- End execution time: The date and time at which a job has terminated its execution.

The necessity of these attributes became clear when studying the collected grid site data, as the services we monitor (from the UiB grid site) separate its measurements by job ID and worker node. Some MonaLISA services were recommended to utilize in the research by the JAliEn development team (Costin Grigoras, personal communication, Nov 10, 2023). The data collected contains information relating to a jobs resource usage, via the ALICE::UiB::SLURM_Jobs service, and resource usage at the host (the grid site at UiB), via the ALIEN Self Nodes and ALIEN System Nodes services. These are stored in three different InfluxDB buckets: Slurm, Self_nodes, and

System_nodes. The project will further refer to the bucket names when discussing data from these services.

Data collected from these services are utilized for analysis and machine learning. There are over 100 fields in total present in these services, where a field is a type of measurement (i.e. CPU usage). Some of these might not be as relevant to the problem as others. By studying the data closer we might be able to locate a smaller subset of fields that are more likely to impact the success or failure of a job. This is the purpose of the data analysis script in the job analysis system. The project utilizes the grid site data related to job executions in its analysis. More data are available, such as certain properties from a job's JDL file, but this is not utilized in the analysis and creation of the model.

Job end statuses are encoded to numerical representations, 0 (error jobs) and 1 (done jobs), to perform mathematical/statistical operations on them. They are further referred to as labels for job executions. The project's first analysis step was utilizing the Pearson correlation coefficient in an attempt to determine how closely each fields relate to the label. The Pearson correlation coefficient is a measurement of linear association between two variables [37]. Its values range between -1, indicating a strong negative association, and 1, indicating a strong association, while a value of 0 indicates no association at all. For this analysis, a subset of 500 jobs were randomly selected. At this point (May 2024) the project had data from about 1000 error job executions. About 25% of the available error jobs were therefore present in this analysis. A correlation histogram of the fields, retrieved from the Slurm bucket, and its labels can be found in Fig. 5.2.

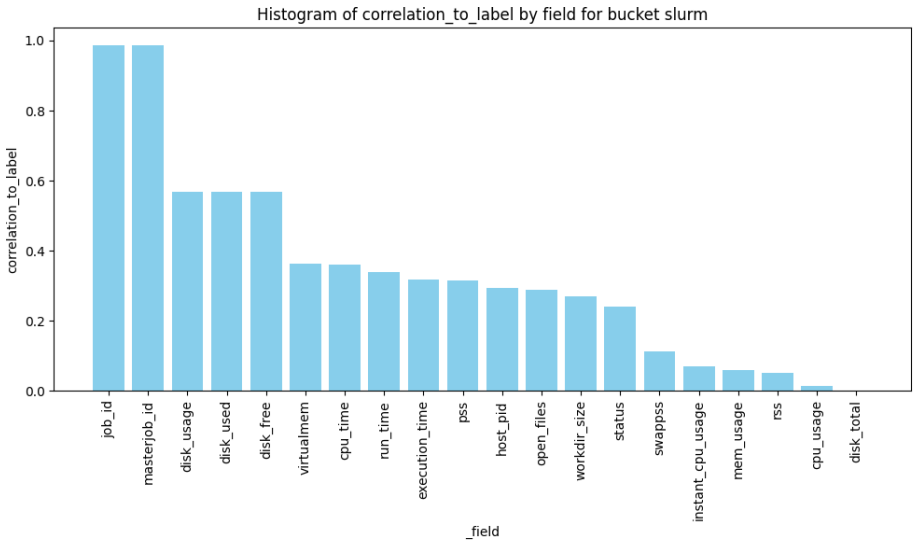


Figure 5.2: Histogram plot of the Pearson correlation between field values and labels for the Slurm bucket. 500 jobs were utilized in this calculation, with an even distribution between done and error jobs.

There are a total of 4 histogram plots created, one for each service monitored except the system nodes service, which has two associated histograms due to its shared number of fields. Most of the fields are in the range of 0 to 0.6 in correlation score. This indicates a somewhat positive linear association between some of the fields, but it isn't as clear as first hoped. This however doesn't mean that there doesn't exist a strong association between the fields and labels, just that it might not be linear.

Certain comparison plots were created to study the differentiation in field values between done and error jobs. A subset of 500 (new) randomly selected jobs were utilized for this, evenly distributed between done and error jobs. A value plot was created to display all values measured for each field. Done and error jobs are differentiated with a different color in the plot, blue for done and yellow for error jobs (Example found in Fig 5.3). In addition to the value plot, a boxplot was created with the same job data (Example found in Fig. 5.4). The boxplot is a popular way to visualize data and is especially useful for displaying multiple datasets [31]. The traditional boxplot displays the distribution of several datasets in simple figures. A combination of value

plot and boxplot gives us a greater understanding of how the several fields differ between done and error jobs. The plots were merged into two PDFs for organizational purposes, one for the value plots and one for the boxplots. This simplifies the process of studying the plots since we can visualize both PDFs at the same time while being able to scroll up and down to quickly change between plots.

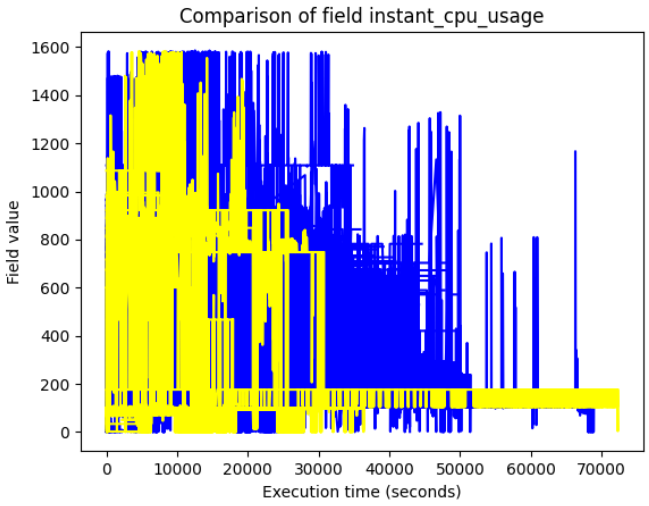


Figure 5.3: A linear plot of the measured values of the field "instant_cpu_usage" gathered from the Slurm bucket. The x-axis represents the execution time (in seconds), and the y-axis represents the field values. Done jobs are presented in blue, while error jobs are presented in yellow. 500 randomly selected jobs were utilized in the creation of this plot.

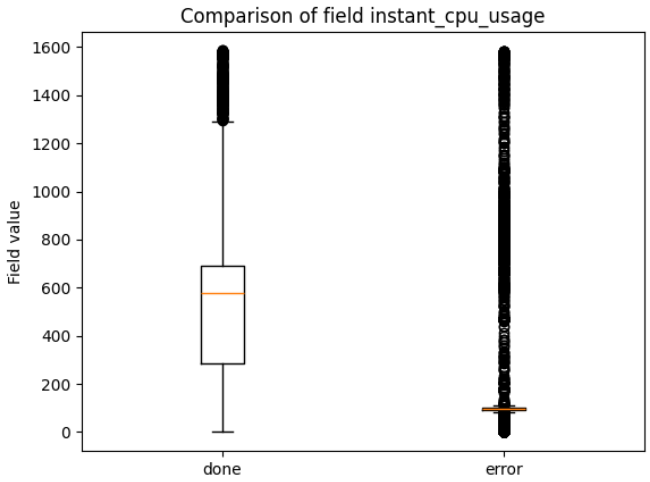


Figure 5.4: A boxplot of the measured values of the field "instant_cpu_usage" gathered from the Slurm bucket. Done jobs are presented on the left, while error jobs are presented on the right. The y-axis represents the field value. Consists of data from the same 500 jobs utilized to create the linear plots.

The value plots were mainly utilized to study the progression of values during the execution of a job. The example of "instant_cpu_usage", illustrated in Fig. 5.3, indicates that error jobs running for longer time periods contain lower field values in the later execution stages compared to done jobs. It is however not clear if this is representable for most jobs or if these jobs are outliers. This is where the boxplot is handy, as it displays the data distribution. A boxplot consists of four main components (explained in context of the boxplot illustrated in Fig. 5.4):

- Interquartile range (IQR): Represented as a box in the plot. Ranges from the 25th(Q1) and 75th (Q3) quantiles. 50% of the data is therefore located in this interval.
- The median: Displayed as an orange line within the box and is located on the 50th (Q2) quantile.
- The whiskers: Two lines that extend up and down from the edges of the box. The lower whisker extends to the smallest data point that is within the range of 1.5 * the range of the box (specifically; Q2 -

1.5*IQR). Respectively, the higher whisker extends to the highest data point that is within the range of 1.5*IQR (Q3 + 1.5*IQR).

- Outliers: Displayed as circles outside the whiskers. Represents all data points that are located outside the whiskers.

Note: The quartiles in a dataset are the values of the elements that split it into 4 equal pieces in the sorted version of the dataset. For a sorted dataset of 1000 elements, Q1 is the element's value at index 250, Q2 is the value at index 500, and Q3 at index 750.

The boxplot illustrated in Fig. 5.4 shows a clear difference in the error and done jobs data distributions for the field "instant_cpu_usage". The interquartile range is much bigger for done jobs, indicating that they generally operate on a wider interval of values. It confirms that the lower field value for error jobs is not an outlier as the IQR is somewhere in the range of 100-200, while the IQR for done jobs is in the range of 300-700. This field might therefore be of interest for further study, given its visual difference between field values of error and done jobs.

A combination of these plots has been utilized to locate a subset of fields for further study. The subset contains the fields: cpu_usage, instant_cpu_usage, workdir_size, open_files, virtualmem, and cpu_time. All of these fields have visual differences between error and done jobs in their respective plots (they can be found in the appendix). These fields, in addition to a column containing "seconds since execution start", are the input features for the project's machine-learning model. The complete input is the Data Matrix (created in the DataTerminal) with these fields, converted to a PyTorch tensor. This is a tensor matrix where each row contains field values measured at a specific time.

Some of the fields are pre-processed before being utilized to train the model. Columns that contain big numerical values or operate in widely different ranges are mathematically logged (log base 10). This normalizes the field values by bringing the dataset closer to a similar scale. This technique can improve performance and training stability for machine learning models [12]. This is done by defining the "field_to_log" parameter in the "cleaning_conf.json" file.

5.3.2 Model Development

The project's model utilizes a transformer model to predict whether a grid job succeeds or fails. The purpose of this is to research whether the transformers' attention mechanisms can find a context in the raw grid site data, and utilize this to perform a prediction. Its architecture consists of four encoders and a single decoder, containing four heads (four instances of self-attention) and is illustrated in Fig. 5.5.

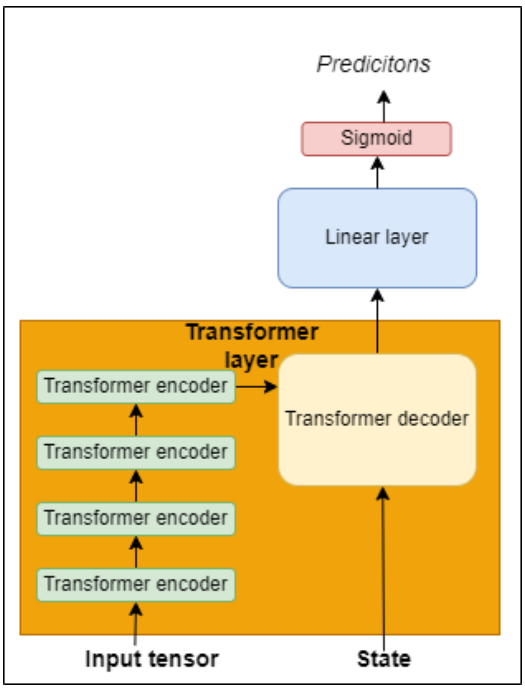


Figure 5.5: The projects transformer architecture. It utilizes six transformer encoders to process the input and one decoder to process encoder outputs in context with a state. This state refers to a collection of previously seen (raw) encoder inputs. The prediction is generated by a linearly transforming of the decoder output to a single numerical value, which is squeezed to a number between 0 and 1 by a Sigmoid function.

The model iterates over the input matrix, processing a single row before adding it to a "state". The state is a collection of previously processed rows, namely previous measurements. An example model of the data extraction is illustrated in Fig. 5.6. This state, as well as the input from the encoders, is

the decoder input. The decoder processes the last encoder output in combination with this state. The idea is to process each row in context with recent measurements. The decoder output goes into a classifier, which is an ML component that transforms its inputs into a class (which in our case is either success (1) or failure (0)). This classifier consists of a linear layer, which reduces the dimensionality of the decoder output to a single numerical value. This value is the input for a Sigmoid activation function, which is a (non-linear) mathematical function that squeezes its inputs to a number between 0 and 1. The output from the Sigmoid function is the model prediction or rather, a part of it. Each job produces several predictions since each row is passed through the decoder individually. The complete prediction is an average of these values.

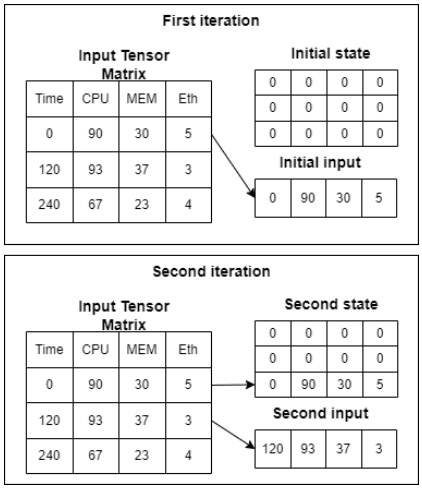


Figure 5.6: Individual input and state extraction from the complete input matrix. The column headers in the input tensor matrix are just for an overview of possible fields. Column headers are not present in Pytorch tensors, which are strictly numerical.

This architecture currently serves as a prototype. It is developed to address the memory system constraint at the test node. The whole input matrix, representing a job execution, would ideally be processed at once, as a complete context of the execution could be learned. This was however not possible with the available memory. The current model implementation processes a subset of the input matrix at a time.

Our dataset consists of about 2000 jobs. This was divided into two datasets, a training dataset containing 90% of the data, and a validation dataset containing 10% of the data. Both datasets were evenly distributed between error and done jobs. The training was done with 10 epochs, where an epoch is a complete pass-through of the training dataset, where the model has seen each element exactly once. The batch size was set to four, meaning that the model's weights are updated after every 4th job, ensuring that multiple job predictions are utilized to update the weights. The model trained for about three days, but did not have a great progression, illustrated in Fig. 5.7.

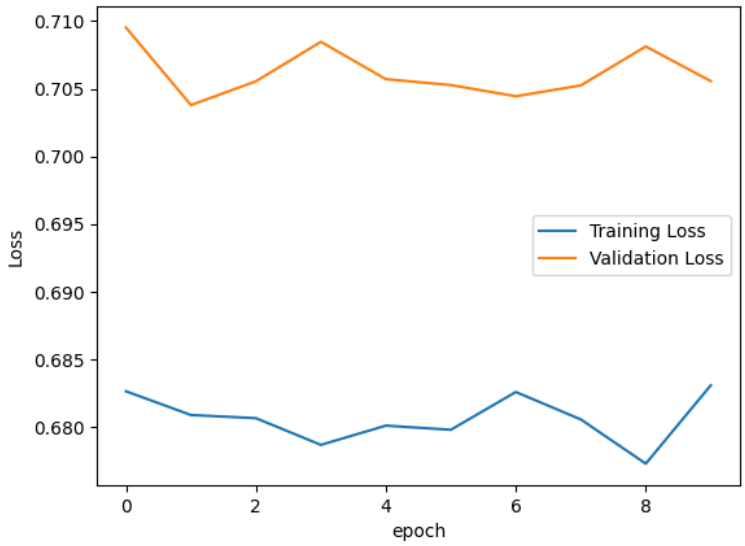


Figure 5.7: A plot of the training and validation loss. The x-axis represents epochs, and the y-axis represents loss. The loss is calculated after each epoch. The plot shows that the model does not train very well, as the loss, for both the training and validation dataset, is close to the initial loss (at the first epoch) for all epochs.

The plot shows that the model does not have any substantial improvements during training, as the loss measured after each epoch is close to its initial loss. The training has resulted in a poor-performing model, which is confirmed by a confusion matrix, illustrating predictions on the validation set (Fig. 5.8).

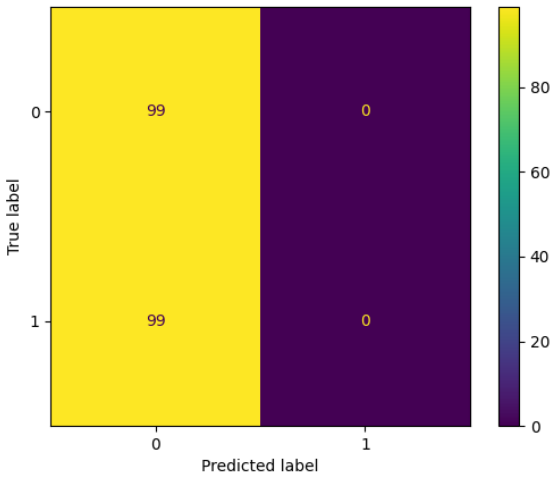


Figure 5.8: Confusion matrix of the predicted values. The y-axis illustrates the true labels and the x-axis predicted labels. It shows that the model predicts a failing job for every job execution, which results in a 50% accuracy since the validation dataset has an equal amount of done and error jobs.

The model has a 50% accuracy, which it achieves by predicting a failed job for each job execution in the validation set. Several factors can explain the poor performance, one of which is the model itself. Processing a single row at a time produces a substantial amount of predictions for each job. Averaging these values skews the final prediction towards the majority of predictions, as well as extreme values (values close to 0 or 1). The encoder input size should ideally be increased as well to provide a wider context and reduce the amounts of predictions. Another explanation for the poor results could be the input features, as they might not contain the relevant information.

The model training process is generally an iterative process. When achieving unsatisfactory results, the usual next step is to go back the to model development and data processing stages and continue the research. Possible steps include data analysis, possibly identifying additional features and further development of the model (which can be a complete change of model architecture). The project could unfortunately not perform another iteration, and it must conclude the development with unsatisfactory results.

Chapter 6

Discussion

The project has developed a system for job data collection and machine learning, which serves as a foundation for job execution analysis for the ALICE project. The system can be configured to store data from multiple (ALICE) grid sites, as the Job Collector passes the "site" property to Alienpy to retrieve the related data and the Site Collector subscribes a client to the services defined in its configuration. Its use cases are mainly tasks related to job data collection and analysis. As of now, there are only a few data analysis tasks available. It is however a foundation that is meant to be further developed. Its flexible nature provides the possibility to study a wide range of grid site measurements related to job executions. Additional machine learning problems are also possible to research with this system.

The data analysis and machine learning did not yield conclusive results, meaning further research is needed. The next sections will discuss the research done, the system design, and the utilized methodologies.

6.1 The research problem

The research problem revolves around identifying reasons for job failure with the help of machine learning. This is formulated as two research questions, which we will now discuss.

- **RQ1:** How can AI help us discover the underlying causes and factors contributing to job failures?

There are several approaches to researching this question, such as different learning approaches in ML. The project's approach concerning RQ2 was to identify possible candidate factors (grid site measurements) by data analysis and seeking confirmation by (supervised) machine learning. An accurate model can be viewed as an additional verification that the input contains information about the problem. Accurate models derive an output, close to the label, by applying transformations to the input. This indicates a relationship between the input and the label. This approach does not strictly utilize a model to discover the underlying causes but rather as a tool for verification, which can help us determine if its inputs are factors contributing to job failure. The project cannot determine the efficiency of this approach concerning job failure, since the created model is inaccurate, meaning that we did not get any confirmation that the selected input features are causes of job failure.

Unsupervised learning is another approach to address RQ2. The goal of unsupervised learning is to investigate the underlying structures in a dataset. A common unsupervised learning task is clustering, which aims to divide data into meaningful groups based on the statistical sides of the data [10]. Clustering can be utilized to study the data more carefully, as groups dominated by failing jobs might occur, indicating that the model has found patterns in the data that relate to its end status.

- **RQ2:** What strategies and mechanisms can be devised for the efficient optimization of computer resources based on insights derived from this AI model?

Some strategies and mechanisms could possibly be utilized to optimize computer resources in the event that the model would be accurate. Early stopping of a job execution can be a simple strategy. Stopping jobs that are likely to fail is a possible approach, but this might also affect jobs that would succeed and do not address the cause of the problem. Another approach could be to give a job more resources if it seems to fail, i.e. if CPU usage is found to be a leading cause, then providing additional CPU cores for a predicted error job might help the job succeed. This approach does however depend on the availability of additional resources at the site.

The proposed strategies utilize the model as a decision-making tool, which can provide actionable intel. This way of using a model is also suggested in a similar project, provided in the paper "A Study of Job Failure Prediction at

Job Submit-State and Job Start-State in High-Performance Computing System: Using Decision Tree Algorithms" by Anupong Banjongkan [35], where decision trees (a machine learning model type) are utilized to predict whether a job execution will succeed or fail. The paper obtains an accuracy of about 85% by training several decision trees (models) on data from workload logs collected from the National Electronics and Computer Technology Center (NECTEC) in Thailand and Los Alamos National Laboratory (LANL) in the USA. The paper suggests that the model can help users make "efficient justifications", helping the users make informed decisions while their job is running.

The project does not provide any conclusions to its research questions. The work done is however not futile, as it has brought us closer to answering them. Certain systems related to data collection, management, and analysis were developed to address the questions. These systems can be utilized as a foundation for further projects, bringing them closer to the core of the research; data analysis and machine learning. Their design will be discussed in the next section.

6.2 System design

The project has developed a prototype system for data collection, management, and analysis. The system currently works for its intended use, although additional testing and optimizations are needed. The implementations in the later stages of the project have less than optimal design and lack testing, specifically the job data analysis system. The job data analysis system has only been user-tested. There exist no unit tests, which is a bad practice as these tests serve as an additional confirmation that the code works as intended.

In Chapter 5, we discussed some possible optimizations for the system, such as moving logic within the job data handler to the data terminal and utilizing the Java trace class to parse the trace file. The analysis suggests that the system in its current state cannot be considered a final product, but a prototype for a suggested architecture that could be improved. The project therefore proposes a revised architecture for data collection and management(Fig. 6.1). This considers the named optimizations, providing a complete data collection and management system.

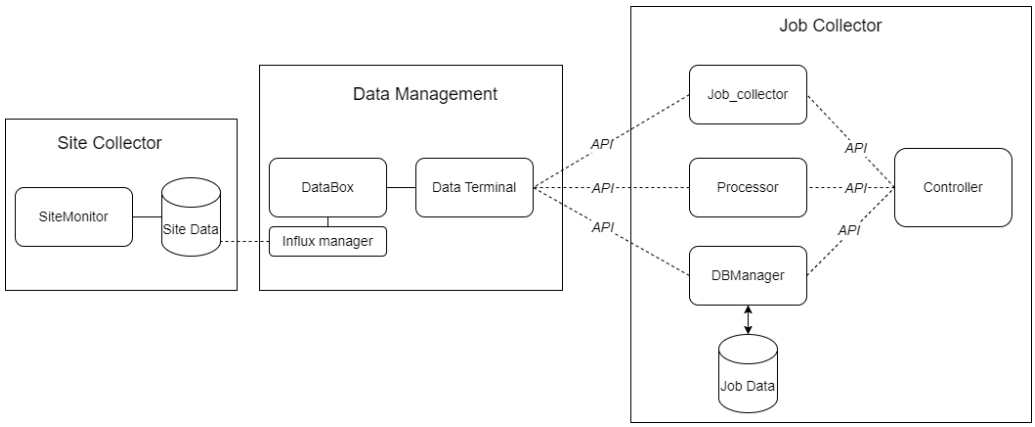


Figure 6.1: A proposed architecture for a data collection and management system for job and grid attributes related to job executions. It addresses the data-related stages of the ML life cycle; Data Processing and Monitoring.

The suggested architecture contains the necessary tools to store, filter, and utilize grid site data related to job executions. The service-oriented design of the job collector enables us to reuse its DBManager to obtain historical job attributes, as well as access the collector and processor to obtain and filter out attributes from jobs that are currently running. This gives the system another dimension, namely real-time monitoring, which can be utilized as additional test data for a machine-learning model. The object-oriented approach of the data representation, via the Databox and Data terminal objects, makes it easy to retrieve and utilize the data as it can be accessed by initializing a terminal with parameters such as the number of jobs to load or a time interval (or both). This architecture is the biggest takeaway from this project, as the data analysis and machine learning produced no conclusive results.

6.3 Methodologies

Several methodologies have been utilized in the project to structure the research and development process, and have partly been utilized with success. The design science methodology was a good fit for the project. It ensured that multiple aspects were considered in the development process, such as system and project requirements. This was however used in addition to a

development methodology that has caused the project some problems. A lot of it is due to poor planning, where the initially developed systems were planned in isolation from upcoming systems. This has caused the project a lot of excess work to integrate the systems, which could have been avoided with a more profound planning phase. The project would've also benefited from a more structured development strategy, something similar to the scrum sprint. The sprints divide the development process into cycles of short periods, in which currently valuable tasks are in focus [36]. This would force the project to prioritize the most valuable tasks at hand, resulting in less time consumption on less relevant tasks. Tests were implemented after a system was deemed "functioning" by user tests, which becomes a problem when this happens in the end stage of a project, and time limitations force the project to use its resources elsewhere.

The machine learning lifecycle was a great way to structure the development of a model as it describes the necessary steps. This helped us plan for some upcoming functionality when integrating the systems, such as continuous monitoring. We did not reach a deployment stage(in either the test or production environment) but made sure that monitoring would be possible with some additional development.

Methodologies play a big role, especially in bigger projects. The project methodologies have been helpful, but the selected development methodology had some drawbacks. Selecting a fitting development methodology enables quicker development (at least in the bigger picture) and ensures that more aspects of the code (such as testing) are accounted for. This is a lesson learned for upcoming projects.

Chapter 7

Conclusion and Further Work

The project successfully designed an architecture for data collection and management that can be utilized across the ALICE grid (Fig. 6.1). The design addresses the data-related stages in the Machine Learning Life Cycle; Data Processing and Monitoring. A prototype of a previous version of this architecture has been developed and utilized for data analysis and Machine Learning. This analysis did not obtain satisfactory results, which provides an inconclusive answer to the project research questions. Certain strategies are however discussed. These remain as recommendations for further research, which can be continued with the foundations laid in this project.

Further work on this project includes data processing and model development steps. A data analysis technique worth researching is Mutual Information(MI). This is a widely utilized technique to measure feature relevance, as it can find linear and non-linear dependencies [45]. As for model development, increasing the input size of the suggested model and adjusting its parameters (such as the number of encoders and the number of heads) could yield a better result. Other architectures can also be considered, as transformers might not be the optimal model choice.

The project utilizes grid-related measurements in its analysis, but cannot verify that these can explain the whole problem. Additional data properties could be included in the research, such as additional job attributes. An example of this is the TLL (Time To Live) of a job. This is the maximum execution time for a job. If exceeded, the job terminates before it finishes.

Additional data are worth exploring if grid site limitations fail to explain the problem.

Appendix A

Source code

The source code for the job data collection system: `https://git.app.uib.no/Erlend.Skutlaberg/job-data-collector`.

The source code for the site collector: `https://gitlab.cern.ch/eskutlab/epn2eosmonitor`.

The source code for the job data analysis system: `https://git.app.uib.no/Erlend.Skutlaberg/data_inspection`.

A.1 Additional analysis plots

Here is a collection of plots of the fields utilized for machine learning. The decision to include these fields was mainly based on the visual differences in data distribution between done and error jobs. This is displayed in the boxplots, where especially the difference in IQR between done and error jobs is quite visual.

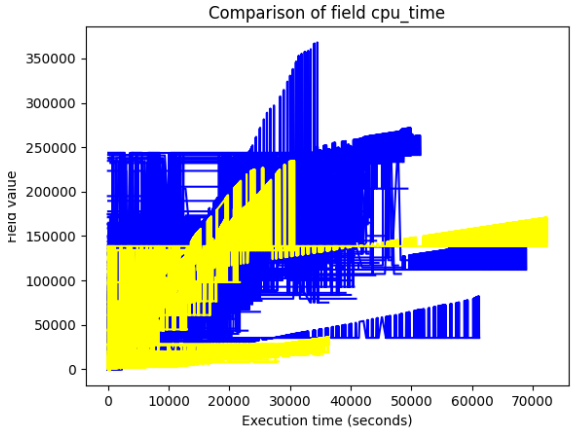


Figure A.1: A linear plot of the measured values of the field "cpu_time" retrieved from the Slurm bucket.

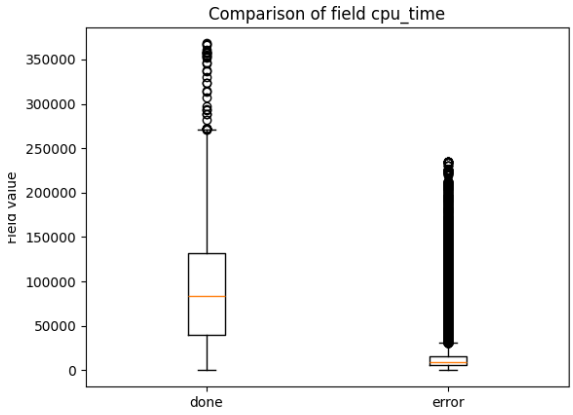


Figure A.2: A boxplot of the measured values of the field "cpu_time" retrieved from the Slurm bucket..

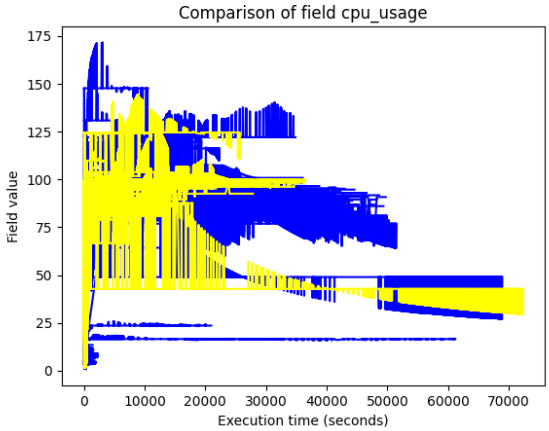


Figure A.3: A linear plot of the measured values of the field "cpu_usage" retrieved from the Slurm bucket.

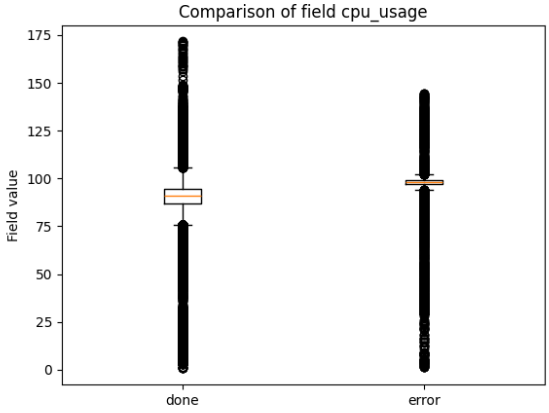


Figure A.4: A boxplot of the measured values of the field "cpu_usage" retrieved from the Slurm bucket.

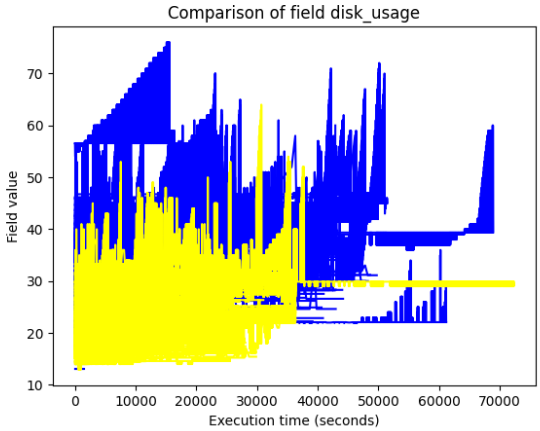


Figure A.5: A linear plot of the measured values of the field "disk_usage" retrieved from the Slurm bucket.

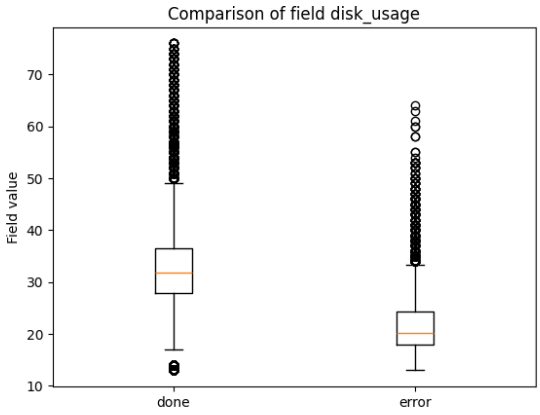


Figure A.6: A boxplot of the measured values of the field "disk_usage" retrieved from the Slurm bucket.

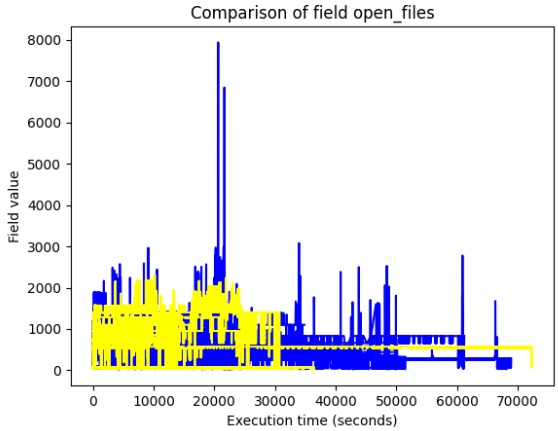


Figure A.7: A linear plot of the measured values of the field "open_files" retrieved from the Slurm bucket.

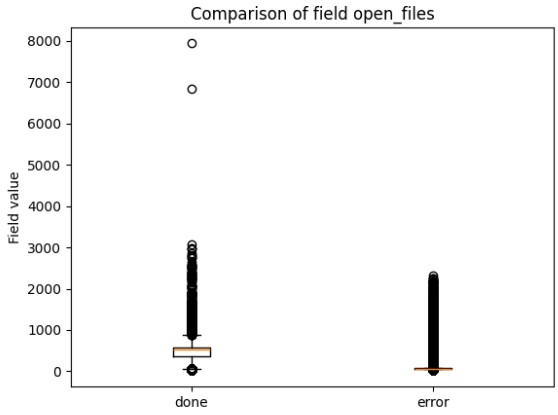


Figure A.8: A boxplot of the measured values of the field "open_files" retrieved from the Slurm bucket.

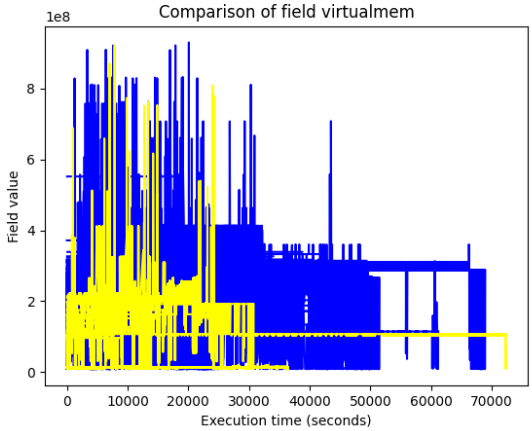


Figure A.9: A linear plot of the measured values of the field "virtualmem" retrieved from the Slurm bucket.

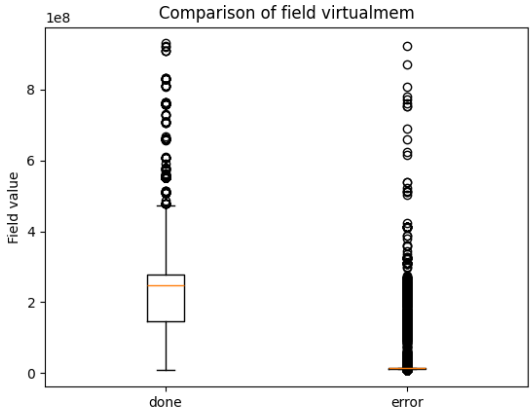


Figure A.10: A boxplot of the measured values of the field "virtualmem" retrieved from the Slurm bucket.

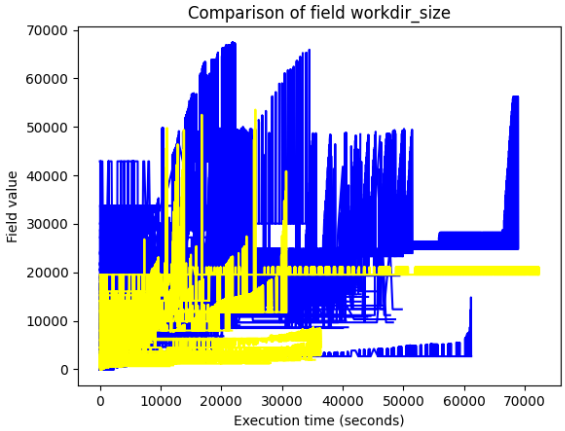


Figure A.11: A linear plot of the measured values of the field "workdir_size" retrieved from the Slurm bucket.

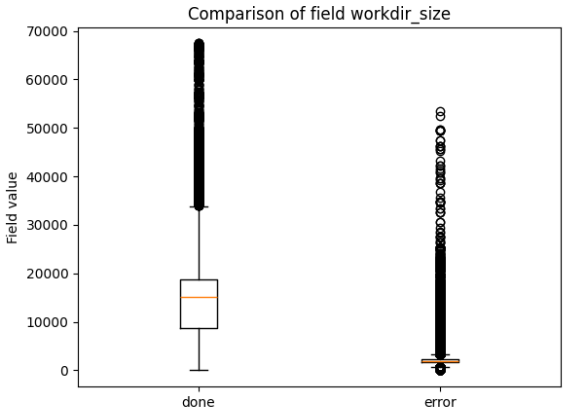


Figure A.12: A boxplot of the measured values of the field "workdir_size" retrieved from the Slurm bucket.

Bibliography

[1] . *pandas.DataFrame*. Pandas. n.d. URL: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html (visited on May 31, 2024).

[2] . *User How-To - Jobs*. CERN. n.d. URL: https://alien.web.cern.ch/content/documentation/howto/user/jobs#JDL_syntax (visited on May 31, 2024).

[3] Khan Academy. *Gradient descent*. Khan Academy. URL: https://www.khanacademy.org/math/multivariable-calculus/applications-of-multivariable-derivatives/optimizing-multivariable-functions/a/what-is-gradient-descent (visited on May 29, 2024).

[4] Amazon Web Services. *Well-Architected Machine Learning Lifecycle*. Amazon Web Services. n.d. URL: https://docs.aws.amazon.com/wellarchitected/latest/machine-learning-lens/well-architected-machine-learning-lifecycle.html (visited on Mar. 13, 2024).

[5] J Balcas et al. "MonALISA, an agent-based monitoring and control system for the LHC experiments." In: *Journal of Physics: Conference Series* 898 (Oct. 2017), p. 092055. ISSN: 1742-6588, 1742-6596. DOI: 10.1088/1742-6596/898/9/092055. URL: https://iopscience.iop.org/article/10.1088/1742-6596/898/9/092055 (visited on May 8, 2024).

[6] Tom B. Brown et al. *Language Models are Few-Shot Learners*. Version Number: 4. 2020. DOI: 10.48550/ARXIV.2005.14165. URL: https://arxiv.org/abs/2005.14165 (visited on May 20, 2024).

[7] CERN. *CernVM-FS*. CERN. URL: https://cernvm.cern.ch/fs/ (visited on May 29, 2024).

[8] CERN. *Diverse experiments at CERN*. URL: https://home.cern/science/experiments (visited on Jan. 25, 2024).

[9] CERN. *The Large Hadron Collidor*. URL: https://home.cern/science/accelerators/large-hadron-collider (visited on Jan. 25, 2024).

[10] Mahnoor Chaudhry et al. "A Systematic Literature Review on Identifying Patterns Using Unsupervised Clustering Algorithms: A Data Mining Perspective." In: *Symmetry* 15.9 (Aug. 31, 2023), p. 1679. ISSN: 2073-8994. DOI: 10.3390/sym15091679. URL: https://www.mdpi.com/2073-8994/15/9/1679 (visited on June 2, 2024).

[11] The Alice Collaboration et al. "The ALICE experiment at the CERN LHC." In: *Journal of Instrumentation* 3.8 (Aug. 14, 2008), S08002–S08002. ISSN: 1748-0221. DOI: 10.1088/1748-0221/3/08/S08002. URL: https://iopscience.iop.org/article/10.1088/1748-0221/3/08/S08002 (visited on Oct. 13, 2023).

[12] Google for Developer team. *Normalization*. Google. 2022-07-18. URL: https://developers.google.com/machine-learning/data-prep/transform/normalization (visited on May 22, 2024).

[13] Docker Inc. *Overview of the get started guide*. Docker inc. n.d. URL: https://docs.docker.com/get-started/ (visited on May 31, 2024).

[14] Rania Echrigui and Mhamed Hamiche. "Optimizing LSTM Models for EUR/USD Prediction in the context of reducing energy consumption: An Analysis of Mean Squared Error, Mean Absolute Error and R-Squared." In: *E3S Web of Conferences* 412 (2023). Ed. by S. Bourekkadi et al., p. 01069. ISSN: 2267-1242. DOI: 10.1051/e3sconf/202341201069. URL: https://www.e3s-conferences.org/10.1051/e3sconf/202341201069 (visited on May 29, 2024).

[15] Christian Fabjan et al. *Technology Meets Research: 60 Years of CERN Technology: Selected Highlights*. Vol. 27. Advanced Series on Directions in High Energy Physics. WORLD SCIENTIFIC, June 2017. ISBN: 978-981-4749-13-8 978-981-4749-14-5. DOI: 10.1142/9921. URL: https://www.worldscientific.com/worldscibooks/10.1142/9921 (visited on Jan. 25, 2024).

[16] Ian Foster. "What is the Grid? A Three Point Checklist." In: *GRID today* 1 (Jan. 2002), pp. 32–36.

[17] Alan R. Hevner. "A Three Cycle View of Design Science Research." In: *Annalen der Physik* 19 (2 2007). URL: https://aisel.aisnet.org/sjis/vol19/iss2/4/.

[18] Mbarek Iaousse et al. "Comparative Simulation Study of Classical and Machine Learning Techniques for Forecasting Time Series Data." In: *International Journal of Online and Biomedical Engineering (iJOE)*

19.8 (June 27, 2023), pp. 56–65. ISSN: 2626-8493. DOI: 10.3991/ijoe.v19i08.39853. URL: https://online-journals.org/index.php/i-joe/article/view/39853 (visited on May 31, 2024).

[19] Influx Data Inc. *Overview*. Influx Data inc. n.d. URL: https://www.influxdata.com/products/influxdb-overview/ (visited on May 31, 2024).

[20] René Meusel Jakob Blomer Predrag Buncic. *The CernVM File System*. CERN. 2024-01. URL: https://jblomer.web.cern.ch/cvmfstech-2.1-0.pdf (visited on May 29, 2024).

[21] JAliEn development team. *JAliEn - JobStatus.JAVA*. CERN. n.d. URL: https://gitlab.cern.ch/jalien/jalien/-/blob/master/src/main/java/alien/taskQueue/JobStatus.java (visited on May 31, 2024).

[22] Taeho Jo. *Machine learning foundations: supervised, unsupervised, and advanced learning*. Cham, Switzerland: Springer, 2021. 391 pp. ISBN: 978-3-030-65900-4 978-3-030-65899-1 978-3-030-65902-8.

[23] Salman Khan et al. "Transformers in Vision: A Survey." In: *ACM Computing Surveys* 54.10 (Jan. 31, 2022), pp. 1–41. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3505244. URL: https://dl.acm.org/doi/10.1145/3505244 (visited on May 20, 2024).

[24] M. Martinez Pedreira, C. Grigoras, and V. Yurchenko. "JAliEn: the new ALICE high-performance and high-scalability Grid framework." In: *EPJ Web of Conferences* 214 (2019). Ed. by A. Forti et al., p. 03037. ISSN: 2100-014X. DOI: 10.1051/epjconf/201921403037. URL: https://www.epj-conferences.org/10.1051/epjconf/201921403037 (visited on Oct. 15, 2023).

[25] Marcin Moczulski et al. *ACDC: A Structured Efficient Linear Layer*. Version Number: 5. 2015. DOI: 10.48550/ARXIV.1511.05946. URL: https://arxiv.org/abs/1511.05946 (visited on May 30, 2024).

[26] Mohamed Nabil. *Unpacking the Query, Key, and Value of Transformers: An Analogy to Database Operations*. IU for Applied Science. 2023-04-19. URL: https://www.linkedin.com/pulse/unpacking-query-key-value-transformers-analogy-database-mohamed-nabil/ (visited on June 2, 2024).

[27] PostgreSQL. *About*. PostgreSQL. n.d. URL: https://www.postgresql.org/about/ (visited on May 31, 2024).

[28] Sarunya Pumma et al. "A runtime estimation framework for ALICE." In: *Future Generation Computer Systems* 72 (July 2017), pp. 65–77.

ISSN: 0167739X. DOI: 10.1016/j.future.2017.02.040. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167739X17302972 (visited on Oct. 16, 2023).

[29] PyTorch. *Pytorch:documentation*. Meta. URL: https://pytorch.org/docs/stable/index.html (visited on May 6, 2024).

[30] Abdul Razzaq and Shahbaz A. K. Ghayyur. "A systematic mapping study: The new age of software architecture from monolithic to microservice architecture—awareness and challenges." In: *Computer Applications in Engineering Education* 31.2 (Mar. 2023), pp. 421–451. ISSN: 1061-3773, 1099-0542. DOI: 10.1002/cae.22586. URL: https://onlinelibrary.wiley.com/doi/10.1002/cae.22586 (visited on May 19, 2024).

[31] Jordan Rodu and Karen Kafadar. "The q–q Boxplot." In: *Journal of Computational and Graphical Statistics* 31.1 (Jan. 2, 2022), pp. 26–39. ISSN: 1061-8600, 1537-2715. DOI: 10.1080/10618600.2021.1938586. URL: https://www.tandfonline.com/doi/full/10.1080/10618600.2021.1938586 (visited on May 22, 2024).

[32] Dominik Rost et al. "Distilling Best Practices for Agile Development from Architecture Methodology." In: *Software Architecture*. Ed. by Danny Weyns, Raffaela Mirandola, and Ivica Crnkovic. Vol. 9278. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 259–267. ISBN: 978-3-319-23726-8 978-3-319-23727-5. DOI: 10.1007/978-3-319-23727-5_21. URL: http://link.springer.com/10.1007/978-3-319-23727-5_21 (visited on May 31, 2024).

[33] Manikandan S. "Measures of central tendency: The mean." In: *Journal of Pharmacology and Pharmacotherapeutics* 2.2 (June 2011), pp. 140–142. ISSN: 0976-500X, 0976-5018. DOI: 10.4103/0976-500X.81920. URL: http://journals.sagepub.com/doi/10.4103/0976-500X.81920 (visited on June 2, 2024).

[34] Iqbal H. Sarker. "Machine Learning: Algorithms, Real-World Applications and Research Directions." In: *SN Computer Science* 2.3 (May 2021), p. 160. ISSN: 2662-995X, 2661-8907. DOI: 10.1007/s42979-021-00592-x. URL: https://link.springer.com/10.1007/s42979-021-00592-x (visited on June 3, 2024).

[35] School of Computer Engineering, Suranaree University of Technology (SUT), Thailand et al. "A Study of Job Failure Prediction at Job Submit-State and Job Start-State in High-Performance Computing

System: Using Decision Tree Algorithms." In: *Journal of Advances in Information Technology* 12.2 (2021), pp. 84–92. ISSN: 17982340. DOI: 10.12720/jait.12.2.84-92. URL: http://www.jait.us/index.php?m=content&c=index&a=show&catid=206&id=1144 (visited on June 2, 2024).

[36] Ken Schwaber and Jeff Sutherland. *What is Scrum?* Scrum.com. URL: https://www.scrum.org/resources/what-scrum-module (visited on May 28, 2024).

[37] P. Sedgwick. "Pearson's correlation coefficient." In: *BMJ* 345 (jul04 1 July 4, 2012), e4483–e4483. ISSN: 1756-1833. DOI: 10.1136/bmj.e4483. URL: https://www.bmj.com/lookup/doi/10.1136/bmj.e4483 (visited on May 21, 2024).

[38] Sergiu Weisz. *EPN2EOSMONITOR*. CERN. URL: https://gitlab.cern.ch/sweisz/epn2eosmonitor (visited on Apr. 22, 2024).

[39] Adrian Sevcenco. *alienpy*. CERN. 2024-04-10. URL: https://pypi.org/project/alienpy/ (visited on May 29, 2024).

[40] Maxim Storetvedt. "A NEW GRID WORKFLOW FOR DATA ANALYSIS WITHIN THE ALICE PROJECT USING CONTAINERS AND MODERN CLOUD TECHNOLOGIES." PhD Thesis. Western University of Applied Sciences, 2023.

[41] Alien Development Team. *Trace java class*. CERN. URL: http://aliendb9.cern.ch/websvn/filedetails.php?repname=alimonitor-lib&path=%2Falimonitor_lib%2Falien%2Fjobs%2FTrace.java (visited on May 14, 2024).

[42] Ashish Vaswani et al. *Attention Is All You Need*. Version Number: 7. 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762 (visited on May 20, 2024).

[43] Qingsong Wen et al. "Transformers in Time Series: A Survey." In: (2022). Publisher: arXiv Version Number: 5. DOI: 10.48550/ARXIV.2202.07125. URL: https://arxiv.org/abs/2202.07125 (visited on May 31, 2024).

[44] Barry Wilkinson. *Grid Computing: Techniques and Applications*. 0th ed. Chapman and Hall/CRC, Sept. 28, 2009. ISBN: 978-0-429-14558-2. DOI: 10.1201/9781420069549. URL: https://www.taylorfrancis.com/books/9781420069549 (visited on Oct. 15, 2023).

[45] Jin Zhou et al. "Sensor-Array Optimization Based on Time-Series Data Analytics for Sanitation-Related Malodor Detection." In: *IEEE Transactions on Biomedical Circuits and Systems* 14.4 (Aug. 2020), pp. 705–

714. ISSN: 1932-4545, 1940-9990. DOI: 10.1109/TBCAS.2020.3002180. URL: https://ieeexplore.ieee.org/document/9115878/ (visited on May 28, 2024).