University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Carbon-Aware Big Data Processing in Cloud Infrastructures

Richard Arthurs
2023

Abstract

The amount of CO_2 released per unit of electricity produced on a power grid differs depending on the time of day, and significantly fluctuates depending on the geographical location due to variations in the methods of power generation. When a flexible big data workload is being processed without taking into account these fluctuations in carbon intensity, the result is more carbon emissions are generated than necessary.

This project aims to utilise this opportunity for optimisation by delaying the execution of jobs based on carbon intensity forecasts, between the window of the job being submitted and when it needs to be completed. This is done by the use of many interconnected components which provide information to the scheduler, which makes real-time updates based on this information to the execution times of a queue of jobs. These jobs are then submitted at their optimised time to a cluster on a cloud infrastructure for processing.

The result, carbon footprints of big data jobs are reduced due to the jobs being executed at times when the national grid is utilising more clean power. A vast variety of job profiles were experimented with, these were submitted at varying times throughout the day. From this, it was calculated that a job submitted to the scheduler with a 24-hour deadline can expect to see a reduction of 11.5% in its carbon footprint based on the median value.

Education Use Consent

Contents

1 Introduction

1.1 Motivation

Reducing carbon emissions is a crucial step in addressing climate change, this is caused by greenhouse gases like CO_2 that trap heat in the atmosphere. This leads to devastating effects such as rising global temperatures, more frequent heatwaves, forest fires, rising sea levels, tropical storms, and droughts. The key to combatting climate change is sustainability in the long term, as relying less on non-renewable energy will enable us to move more rapidly towards a carbon-neutral future. In addition to its environmental benefits, reducing our dependence on fossil fuels can also help to decrease our vulnerability to price spikes and supply shortages caused by international sanctions, war, or costly import fees. By decreasing our carbon emissions, climate change concerns can be addressed, sustainable practices promoted, and economic growth stimulated.

National grids across the world each have a carbon intensity value at one point, this is the amount of CO_2 released per unit of electricity produced value at any point. This metric constantly fluctuates, this can be dependent on time and location among many other factors (Radovanović et al., 2023; Trihinas et al., 2022). This value can be predicted by forecasts, which give an indication of how the value will change in the near future. This presents an opportunity to offset the time which electricity is used to a time when the energy being utilised by the grid is more green (from renewable resources). This optimisation opportunity can be benefitted from in the processing of big data jobs. Some big data jobs are flexible, this means they can be completed at an arbitrary time between the submission time and some deadline, i.e. the developer or system does not require the results immediately or as soon as possible. Here an opportunity is presented to optimise the execution time in the given window of submission time and deadline, the aim is to execute the job with the most optimal timing considering runtime and carbon intensity forecasts, in order to reduce the overall carbon footprint of the job. This can be achieved by using previously executed runtimes of the same or similar jobs, utilising regularly updated predictions of the carbon intensity, in order to find the optimal time to execute the job.

Furthermore, it can be estimated that there can be a reduction in carbon emissions of around 10%-20% is the scenario, from a load shifting architecture in EV charging for instance (Cheng et al., 2022). Across the world, data centres solely are responsible for 2% of global carbon emissions, and between 2010 and 2018 global data centres compute instances increased by 550% (Masanet et al., 2020). Any percentage which can be saved becomes more substantial every day, and there are only a small amount of companies taking advantage of this opportunity, and those who are, only are employing the strategy for their own internal data processing jobs. More importantly, there are no cloud services which offer this functionality for job submissions on their frameworks, without having to implement an external carbon aware scheduler.

1.2 Problem

In the data science world, big data jobs are submitted throughout the day all across the globe, each one can vary drastically in size, complexity, and runtime. This presents many challenges, for one, estimating the runtime of these jobs can be complex with numerous factors involved. For instance, hardware capabilities when working with cloud infrastructures have a big impact on the runtime, but these could easily be bottle necked by other factors such as data format, network bandwidth and the algorithm being used to process the data. An example of a job which would be hard to predict would be a natural language processing (NLP) job, something like sentiment analysis or categorising topics. Due to the large amounts of unstructured text involved, which can have a vast range of complexity and content. This sort of job would require substantial bench marking and profiling in order to give a gauge of possible runtime if repeated similarly.

The job of the scheduler is to determine when the ideal time to execute a job is, based on carbon intensity predictions, within the window the job has to be completed in. The insertion of these delays for executing the job, versus executing the job when it is submitted, is the main action which reduces emissions. Carbon intensity predictions change throughout the day, meaning the scheduler must also keep updating target execution times of jobs in order to reduce the carbon footprint of the job as much as possible.

As forecasts are updated throughout the day and the closer the current time gets to the forecast time, the more accurate the forecast predictions should be. As a result, the scheduler is required to dynamically adjust the execution times of the jobs in real-time. For instance, the first optimal time for the job to be run may not necessarily be the optimal time after updated predictions are published later in the day. The scheduler also has the job of submitting these jobs to the cluster when they reach their execution time, and if multiple jobs have the same execution time, it should submit the jobs concurrently at that time. This means there is a requirement for functionality which allows the scheduler to submit the multiple jobs to the cluster at once, while detaching from these jobs once they are submitted, removing them from the queue then continuing to schedule any remaining jobs.

1.3 Aim

The aim of the project is to develop and analyse the effectiveness of a carbon aware job scheduler which offsets execution times of data processing jobs based on carbon intensity forecasts with the goal of minimising carbon emissions. To accomplish this, the following will be required:

- Describe the problems associated with carbon intensity forecasts and analyse historical data to infer information about the reliability of such predictions.
- Develop a command-line job submission tool which receives the job filenames, arguments, deadline and estimated runtime as an input from the user. This information must then be validated and uploaded to a database file.
- Develop a scheduler program with reads jobs from the database file then optimises their execution times according to the deadlines supplied, based on carbon intensity forecasts. Then maintain this queue of jobs while making dynamic changes based on forecast updates, while also submitting the jobs to a cluster when they reach their optimised execution time.
- Complete experiments using the system on a cloud provided cluster, measuring the performance of the system and using varied deadline window (this term refers to the window between the submission time and deadline time for a specific job) sizes, and varied runtimes, to infer more information of the impact which these have on total carbon emission reductions.
- Measure and evaluate the performance of the system based on how well it performs its purpose of reducing carbon emissions, and discuss how this could be adjusted and improved. While also identifying any trends in the results if they are present.

2 Background and Related Work

2.1 Carbon Intensity Forecasts

The forecasts in this section produce average carbon intensity signals, details on the difference between this signal and the marginal carbon intensity signal are in the next subsection 2.1.1. The forecasts used in this project are nationwide carbon intensity forecasts which only include carbon dioxide emissions involved with electricity generation, these include every metered power station, imported power, transport of electricity, and distribution losses while also accounting for embedded wind and solar generation (MacMillan, 2021). The only limitation of this monitoring is that it does not include embedded generation.

When observing how the carbon intensity variates across the UK, the differences are massive. Carbon intensity values less than 5 are frequently observed in Scotland, while values exceeding 250 are common in the south of England. It does seem like a massive wasted opportunity that the vast majority of data centres in the UK reside in England and not Scotland, where the clean energy is (ESO, 2023).

It is important to ensure that not only highly reliable forecasts are being used, but also understand how they are calculated and how accurate they are likely to be. Carbon intensity as a parameter is a lot more difficult to predict than other related ones, such as electricity demand. Because they are many factors which contribute to this unpredictability, such as electricity demand, wind speed, solar levels, hydro activity and the actual CO2 emissions produced depending on the composition of fuel. Each fuel is assigned a carbon intensity, grams of carbon dioxide per kilowatt-hour, any renewable energy sources are just allocated zero for this value and imports from other countries are assigned an average value based on the countries overall composition.

The service used for this was the Nation Grid ESO (Electricity System Operator) Carbon Intensity API, which provides 48-hour carbon intensity predictions which are updated hourly (Datopian, 2023). To increase the reliability of their predictions, they use a state-of-the-art supervised Machine Learning regression model, which increases the accuracy of the forecasts over time.

The equation used to produce the intensity value is as follows – the carbon intensity C_t at time t is found by weighting the carbon intensity c_g for the fuel type g by the power generation P_g, t for that fuel type. This is then divided by the national demand D_t, the values produced represent the number of grams of carbon dioxide per kilowatt-hour of electricity produced.

$$C_t = \frac{\Sigma_{g=1}^{G} P_{g,t} \times c_g}{D_t}$$

This gives an overall gauge of how green the national grid's power is at one point in time, this data will be the basis for all the job execution time offsetting actions.

2.1.1 Marginal vs Average

There are two ways to interpret carbon intensity, namely a marginal carbon intensity and an average carbon intensity, in this subsection the differences between the two measurements will be discussed and why they exist.

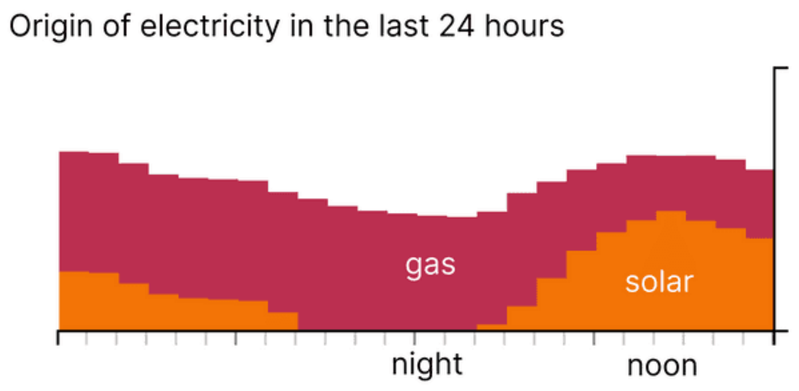


__Figure 2.1:__ This figure displays how the composition of fuel in generators can vary over the course of 24 hours (Corradi, 2022).

Average carbon intensity is the measure of the carbon intensity of the composition of electricity consumed, an oversimplified example of this could be a grid which totally relies on just gas and solar as a means to produce power. This means that all the power plants involved with the production of the electricity contribute to the average signal produced from this metric. In Figure 2.1 there is a diagram of how the dependencies could change throughout the day.

Marginal carbon intensity is a measure of what source will provide the additional electricity required for a task. In other words, marginal carbon intensity predicts the specific fuel or renewable energy source that the national grid will utilize to generate additional electricity needed to fulfil future demands.

This is done by observing a prediction of how the demand for electricity will fluctuate in the future, then looking at what power plants have the capacity to increase their production at this point, for example a gas power plant at one point may be able to increase its production of electricity by one hundred percent on the other hand a wind farm cannot change its production in this manner (one cannot ask the wind to blow faster).

Despite this the predictions for the demand of electricity in the future could include the task you are offsetting, this means the real profile for the power plant is unobservable because it is in the future, and whatever prediction is there can be debated. These difficulties mean it is not only difficult to interpret this signal, but whether it is correct.

In contrast, the average carbon intensity is a much more simple and easy to understand metric which often leads to a superior long-term outcome than that of the marginal signal (Corradi, 2022). The average signal is also easy to transmit and communicate between countries, which is relevant when calculating carbon intensity as a lot of power is imported across the EU. Therefore, due to the lack of ground truth involved with the marginal signal and the effectiveness of the average signal, it is generally considered a better metric for this type of problem.

2.2 Apache Spark and Big Data Jobs

Apache Spark is an open-source, distributed computing framework used for processing large volumes of data (big data). Its purpose is to perform processing tasks on massive data sets at incredibly high speed while optimising hardware efficiency. The framework is around one hundred times faster than Hadoop MapReduce. Apache Spark achieves this by utilising in-memory caching and enhancing processing tasks by using its Resilient Distributed Datasets

(RDDs) abstraction, which enables parallel processing across multiple worker nodes. Spark is also an exceedingly versatile platform which can be run on a variety of environments, including cloud provided clusters and local machines.

2.2.1 PySpark

PySpark is the Python API for Apache Spark, this provides a platform to use the Python programming language to interact with Apache Spark. It allows developers to write Spark programs who are not familiar with the Scala language. Python is easier to understand and has many powerful libraries available for Machine Learning, Natural Language Processing and data visualisation. PySpark also supports many data formats such as text files, JSON, and Parquet. This data can be easily analysed using some of the Python libraries such as NumPy, Matplotlib, Pandas and sklearn.

As PySpark can be run on a local machine, it allows effortless testing and experimentation on projects without having to deploy them on a costly cluster. The subprocess module in Python allows the creation of detached processes from a program. A detached process is a child process that runs independently of its parent process. From the point the process is detached, it operates on its own, while the parent process continues executing. The child process can communicate back to the parent process using interprocess communication mechanisms, this allows the detached process to communicate job status updates (Venkataraman and Jagadeesha, 2015).

2.2.2 Runtimes

Apache Sparks jobs will often involve vast amounts of data being processed, this results in runtimes ranging from seconds to minutes to hours. Thus estimating job runtimes is crucial for distributed computer systems, these predictions can be used to estimate costs to process the job, help determine the optimal number of nodes and in our case for scheduling actions (Al-Sayeh and Sattler, 2019). As previously mentioned the factors which affect a job's runtime are vast from the hardware capabilities and network constraints to the data itself – the format, the process it must go through, the complexity of the data and so on. The more these factors are interpreted and understood, the easier it is for a developer to employ a scheduling program with an objective such as reducing carbon emissions.

2.2.3 Spark on a Cluster

On a cluster, Apache Spark uses a master-worker architecture, this is when one machine is the master node and has multiple worker nodes. The master node maintains the Spark context objects, this communicates with the cluster manager in order to obtain resources to execute the jobs on. The worker nodes are responsible for completing the tasks assigned to them by the driver program, which is a process on the master node that manages the execution of the Spark application. The master node is also responsible for allocating the cluster's current configurations resources to each application, this includes the memory and CPU cores. As the tasks are being executed across the worker nodes they use shared resources, this is when Apache Spark utilises in-memory caching to enable worker nodes to share the data rapidly between one another.

In summary, multiple techniques are employed by Apache Spark to ensure tasks are executed efficiently and rapidly while also guaranteeing a low failure rate and mechanisms to prevent failures. This makes it the ideal platform to process big data and experiment with scheduling algorithms in a distributed computer environment. A job submitted to a Spark system is often referred to as a Spark Application in documentation, in summary this is a standalone program or set of programs that run on Apache Spark cluster to perform data processing tasks.

2.2.4 Google Cloud Platform Dataproc

Google Dataproc is a fully-managed cloud service for running big data processing jobs and analytical workloads using open-source frameworks such as Apache Hadoop, Apache Spark, and Apache Hive. It provides a fast, easy, and cost-effective way to create and manage Hadoop and Spark clusters in the cloud, allowing users to process large amounts of data rapidly. Fully-managed means that users do not have to deal with any of the burdens of setting up the hardware, configuring the virtual machines, maintenance, scaling or backups, this is all provided by the Google Cloud Platform (GCP). This fully-managed service means developers or small companies can focus on developing the tasks and performing analytics, without having to allocate any time to configuring the infrastructure involved in their cluster. The system has built-in monitoring tools which help troubleshoot any issues and also offers security features such as encryption of data being transmitted.

When creating a cluster on the Dataproc console, there are many ways to customise your cluster to your needs, this includes the hardware. There are numerous machine families to choose from including general purpose which are machines for common workloads designed to be cost-effective, compute-optimized for high performance, memory-optimized for memory-intensive workloads and GPU configurations. There are many web interfaces provided by the Google Dataproc API which are used to manage clusters and jobs on the platform, these include YARN ResourceManager, Spark History Server and JupyterLab among many more. These make testing implementations on the platform seamless, and the intuitive user interface makes it accessible for inexperienced users.

2.3 Related Work

In this field of computing, Google have been the pioneers in researching and integrating carbon-aware computing into their company infrastructure. They were one of the first companies to become carbon-neutral in 2007 and have been using only renewable energy sources since 2017 (Hölzle, 2022). There was a research paper by Radovanović et al. (2023) published which focuses on Google's Carbon-Intelligent Compute Management system, the aim of this system is to reduce the carbon footprint of their data centre's. Their method to do this is delaying the execution of temporally flexible workloads. Although this service is not available for user configured clusters on the Google Cloud Platform, only for internal applications at Google.

The paper does provide concrete evidence that the Google's Carbon-Intelligent Computing System is successful in its endeavourer in reducing the carbon footprint of data centres and highlights the importance of data centres in the future of decarbonisation of electricity grids. Although it would be interesting to compare how this strategy, compares to Google's existing data centre load shifting techniques which actually transport jobs to different locations which have lower carbon intensity than the current location. Overall, the paper suggests that the approach of delaying the execution of temporally flexible workloads in order to reduce overall carbon footprints is a promising approach in data centre operations.

2.4 Summary

In this section, the factors involved with carbon intensity forecasts have been discussed, the influence each one of these factors has on the overall values produced has been detailed. While also acknowledging other methods of carbon intensity measurement and how each of these have their weaknesses and strengths relevant to which environment they are being used in. The nature of big data jobs has been explored in order to give an understanding of the unpredictability and complexity involved with estimating the runtimes of such tasks. There is an introduction into how the Apache Spark engine operates and what the benefits are of using such a system over other methods, and the behaviour of Apache Spark on a distributed computer system has also been explained. In addition, it is vital to take into consideration the existing research that has been led in the relevant field, and to evaluate how the proposed implementation compares to previous work. This has provided an understanding of what can be expected from the project and identified any potential areas for improvement or future work.

3 Problem Analysis

The task presented by my supervisor Lauritz Thamsen, University of Glasgow lecturer, was to design a system which reduces carbon emissions produced by big data jobs being executed on cloud infrastructures. The requirements involved in developing a program which can do this were discussed and determined in the early stages of the project, these spanned from the initial research required to find a carbon intensity forecast API to the methods of experimenting on the product to evaluate the effectiveness of the system. Some requirements required alterations later into the development process, these were adjusted accordingly. Another strategy besides the recommendations from my supervisor for determining the requirements was investigating research papers on carbon aware systems to understand the components involved with such an application.

3.1 General Problem

The components which are required in order to create this scheduling system are listed here. Each of these requirements are a vital part in progressing from a simple program which determines the best place to allocate a timeframe for the lowest carbon emission score, to implementing the idea into a real Apache Spark environment with big data jobs. The system should allow the user to submit jobs to a database, with the credentials of the job. The scheduler will run alongside this command line job submission tool which processes these jobs and appropriately schedules them with respect to the carbon intensity forecast, in whichever timeframe would be predicted to have the lowest carbon footprint. Finally, submitting these jobs to a cluster on the cloud, once they reach their optimal execution time.

The functional requirements list below explains what the system must do in order to achieve the aforementioned goals. The non-functional requirements describe how well the system should perform to meet the expectation of reducing carbon emissions when compared to running the jobs at submission time. Achieving this ensures a functional system is created which meets the expectations of the proposal.

Functional Requirements:

- Develop a command line tool which allows the user to submit jobs to a database with parameters specific to a job: runtime, filename, arguments, and the deadline.
- Create a database architecture which will be used to communicate between the job submission tool and the scheduling algorithm, transferring the relevant information to the scheduler for processing.
- Analyse and observe carbon intensity forecasts and parse them into practical variables, which can be used to determine the optimal window to place a job in, constrained by the deadline. These predictions must be updated periodically to ensure updates to the prediction forecasts are not missed.
- Employ an algorithm which, given a runtime for a job and a deadline, can process this information alongside the current carbon intensity forecasts in order to output the most ideal execution time for this job, which minimizes emissions.

- Integrate a scheduling algorithm with the previously mentioned algorithm to intertwine this with actual jobs real in from a database file. The scheduler must dynamically adjust execution times in real-time based on updated carbon intensity forecasts.
- Configure the system such that it can be deployed on a cluster for experimenting, here information should be displayed to indicate when the jobs are scheduled to be executed.
- Enable inspection of Spark contexts, allowing job execution milestones to be obtained, so that the performance of the scheduler can be evaluated.

Non-Functional Requirements:

- The user should be able to input job details easily and monitor the system to see when the job started execution, while being able to monitor the optimised execution times before the job is executed.
- Overall, the interaction with the system should be seamless and not require much external documentation for a novice user to be able to quickly become familiar with how to operate the system.
- The system should be reliable, stable, and easily maintainable in the cluster environment, with reasonably consistent performance.
- The scheduling algorithm must reduce overall carbon emissions when compared to the job being executed at submission time if possible.

3.2 Accuracy of Carbon Intensity Forecasts

The accuracy of carbon intensity forecasts depends on many elements, these include the data used to produce these predictions, the modelling techniques employed and the time frame over which the predictions are being made. Generally it can be said that the further into the future predictions are made, the lower the certainty of these predictions. The accuracy of such predictions has improved in recent years due to improvements in data collection methods and modelling techniques (Ma et al., 2020). A way to measure the accuracy of these predictions is employing statistical metrics such as mean average error (MAE) or root mean squared error (RMSE). The MAE approach results in a value closer to the median when compared to the RMSE value, producing a more biased error calculation. This is optimal for this prediction dataset, as generally the large errors or outliers can be related to some specific event that happened that day or was during a lockdown period. Figure 3.1 illustrates that the vast majority of errors belong in a small domain, while the larger errors are sparse and uncommon.

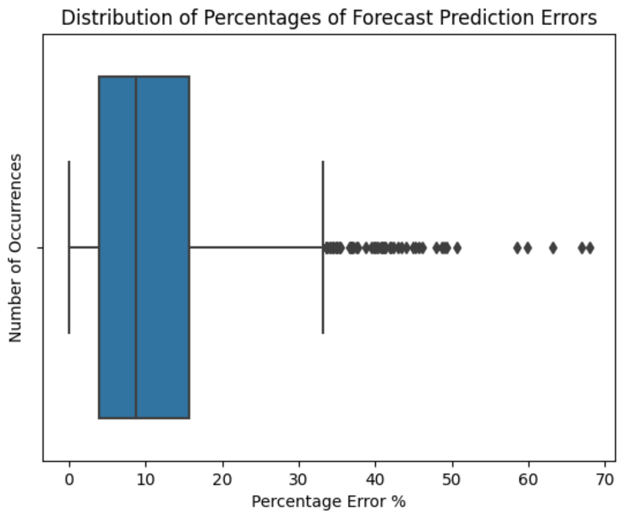


Figure 3.1: *Box plot representing the spread of percentage errors based on predictions and actual values for the National Grid Carbon Intensity API*

3.3 Assumptions

This section outlines the underlying assumptions that were made during this project. Assumptions are important as they give context for results and research questions in the future. They are underlying beliefs, ideas, or concepts that are taken for assumed and serve as the basis for the project. They help give an understanding of what the results produced mean and aid the reader in understanding how the project was carried out. Below, will outline a list of assumptions which have been made in this project:

- During this project the Google Cloud Platform was the cloud provider of choice, the assumption here is that there are infinite resources, the implication of this assumption is that the resources required for each job do not have to be considered. This means jobs can be submitted at the same time without any consideration of the implications of this on the system performance and how it could affect the runtime, because the resources available are assumed to be infinite.
- Another assumption is that jobs submitted have previous runtimes or the runtime of a similar job, this value is submitted by the user, and is used to determine the window that the job will be executed in. Also, this runtime provided must be from when the job was completed on this or another cluster with the same hardware configuration. Thus, the runtime is actually realistic to the environment and has a higher chance of being close to the real runtime on this run.
- Jobs being submitted utilise files which are already uploaded to the cluster's data bucket. The functionality to upload files during the submission of the job was decided to be out of the scope for this project because it is an unnecessary addition irrelevant to the aim.
- The CPU usage and memory optimisation throughout a job varies drastically throughout the processing of most big data jobs (Scheinert et al., 2021), at one point the job could be waiting for some complicated operation to finish, then could proceed with processing the dataset at this point at full speed, before the CPU usage would be low and would then spike after the waiting is finished. The assumption here is that the hardware utilisation is the same throughout the job, thus is not considered when optimising the execution time of the job to reduce carbon emissions.

- The power usage effectiveness (PUE) of a data centre is a measure of how energy efficient a data centre is, although the accuracy of the metric is disputed (Horner and Azevedo, 2016) it has become an industry standard. For the purpose of keeping the workload realistic and obtaining valid and meaningful results, the calculations in this project do not consider the energy efficiency metric. It is assumed that the system components are operating at maximum efficiency, without wasting any electricity.

3.4 Desired Limitations

It is important in research to choose limitations for projects such as this, as it allows increased focus on more distinct factors which are the aims of the project. Not to mention the importance of being transparent about the limitations of results and any potential weaknesses, it is crucial to be aware of these potential shortcomings as a researcher. This project aims to show how much carbon emissions can be saved from offsetting big data processing jobs on cloud infrastructures, based on carbon intensity forecasts. Limitations can also be as a result of elements which cannot be changed, in this case the timeframe for the project cannot be altered and there is no budget provided which could be used for extensive long-term testing. In this section, the chosen limitations for this project have been outlined:

- The system will not take into account the additional electricity that will be used to run the scheduler program versus a trivial implementation which executes the jobs at submission time. Although effort has been made to reduce the amount of CPU utilisation used to run the scheduler, this includes adding a delay between each iteration of the scheduler's main algorithm, this delay saves resources. This is an appropriate delay because it is not necessary to be updating the system absolutely continuously. This is because the carbon intensity forecasts are only updated hourly, and the only other reason to iterate is to check whether new jobs have been submitted.
- Calculating carbon emissions absolutely is complicated and presents many challenges which are not possible due to unknowns. For example, the cooling system efficiency of a data centre affects the performance of the machines and the weather affects the performance of the cooling system. Also, due to data centres often not publishing their location publicly, it is not possible to use regional data rather than national data for carbon intensity, which would lead to more accurate calculations. As a result of many complex factors, including the aforementioned, it is simply not realistic to be certain about carbon emission savings calculations with total certainty. Thus, percentage savings will be used for the purpose of evaluating performance of the scheduler.
- The timeframe for this Level 4 project is around seven months, most of this time cannot be used for testing as the system is still being developed. This constrains the testing stage to only consist of at maximum a few months, this means it is not possible to provide experimental data on how the seasons of the year affect the performance of the system. This sort of data is important when working with forecasts which rely heavily on weather forecasts and the impacts of national holidays.
- The lack of a budget constrains how long the system can be tested on a cloud cluster, this also means the jobs have to be kept relatively small otherwise the cost of running the tests will become unrealistic. Ideally, the system could be tested for months with a large variety of big data jobs. These jobs could have a large range of runtimes, and the deadline size could also be varied across weeks to find trends. Thus, extensive testing such as this is not possible, smaller jobs and short-term tests are still extremely valuable and insightful but in the ideal world more testing would be done.
- When optimising the execution window for a job based on its runtime, the hardware utilisation variance or trace over the execution of the job is not considered. This is relevant as at the points when the job has its highest CPU utilisation, it will of course use more power

at this point than at other points in the processing of the job. It is possible to understand how such a job runs based on factors such as memory usage patterns (Will et al., 2022) and workload traces. Optimal number of worker nodes is also another optimisation which can be made based on job profiles (Islam et al., 2020). This advanced job profiling was decided to be out of scope in this project as the timeframe is not realistic for such an implementation.

- This type of delaying method by optimisation is only designed to work with jobs which have an accurate runtime estimation, as mentioned in Section 3.3. This means jobs that have not been run before and have an unknown runtime or behaviour are not suitable for this type of solution.

3.5 Summary

In this section, the requirements for this project have been outlined and explained, these requirements have been formulated in order to create the desired functionalities. The general problem has been discussed, and an overall idea has been portrayed on how the system will need to be constructed and how components will need to communicate. The expectations for how the system should perform have been explained, these give a general idea of what was desired to be delivered. There is also some insight into the functionality of how the system will react to interactions from the user and updates in the carbon intensity forecasts. While also being conscious of the reliability of the forecasts and understanding how they are produced and updated over time. The appropriate assumptions for this project have been explored, in order to help the reader understand not only the context but understand the implications of the results of the project. While also being aware of the limitations which have been set out in order to focus on the factors specified in the aim and to keep the focus of research on reducing carbon emissions by offsetting big data jobs. An important clarification to make here in this dissertation is that submission time corresponds to the time when the user submitted the job to the database and thus the scheduler. The execution time is the time in which the job was executed on the cluster, i.e. the scheduler has submitted the job to the cluster at its optimised execution time and is finished scheduling this job.

4 Design

In this chapter, the design of the system will be broken down, initially with the full architecture of the system components and connections between them. Next, examining the characteristics and capabilities of every component and establishing their connection to the corresponding requirements they fulfil. Details about data structures and algorithms involved within each component will be designed and designed. Lastly, the chapter will explore the tools and technologies that were evaluated for the project, and the ones that were utilized during the implementation phase.

4.1 System Architecture

The system can be broken down into three parts namely, the job submission tool, the scheduler, and the cluster environment as in Figure 4.1. It is important to note that there must be communication channels established between each of these components. This scheduler is designed in such a way that it could be used as a generic scheduling framework, such that it can be reconfigured to work on any other big data cloud provider.

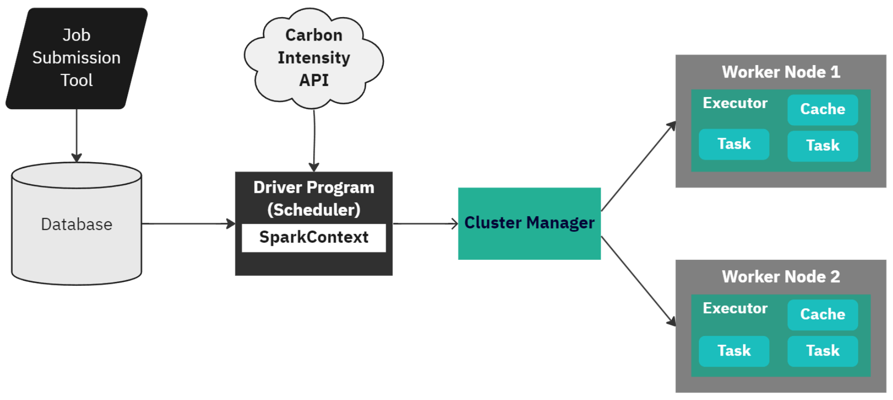


Figure 4.1: *System architecture, exhibiting how the components interact in order to maintain a stable environment with a job queue which has execution times which are dynamically updated in real time based on updated carbon intensity forecasts.*

The system was designed such that the scheduler can run continuously, receiving updated forecasts and checking for improvements in current optimisations based on updated forecasts. While concurrently there is a job submission tool running which is always ready to receive jobs from the user, validate the input then upload the information to the SQLite database file. Then, when the time comes that a job reaches the optimal execution time, the scheduler then utilises the Google Dataproc Spark Submit command to load the job files to the Spark context which begins the processing of the job assigning tasks to the worker nodes. This process is detached from the scheduler, i.e. once the Spark Submit command is transmitted, the scheduler continues normal operation. This means the scheduling program can continue running without waiting for the job to finish. Through this mechanism, the PySpark scheduling algorithm can enhance its concurrency and throughput, since it can carry out other tasks while the Spark jobs are executing. After the job is submitted, the scheduler will remove the job instance from the SQLite database file with the appropriate SQL command.

4.2 Command Line Job Submission Tool

The goal of this tool is to prompt the user for an input, validate the response, then parse the information and upload the entry to the SQLite database file with other information relevant to the entry. After committing the entry to the database, it must then loop and once again prompt the user for another job submission. This behaviour of prompting for a job, waiting for the response, processing the response then repeating the whole process means the program only uses system resources when processing a job entry and once this is complete will simply wait for the next one. Thus, the overall overhead of the program is minimal, and when running on a cluster would be insignificant.

Upon running, the program should prompt the user from a space separated list with information about the job they are submitting to the scheduler, these include the **'runtime'**, **'filename'**, **'args'** and the **'deadline'**. The runtime should be submitted in minutes for increased precision when optimising the window for execution. The filename should just be the python files involved in the job and any arguments that will be passed to the main method of your main class, if any (Spark, 2023). The deadline parameter should be submitted in hours, for example if a job was submitted at five o'clock in the evening and was required for the next morning at nine o'clock the user would simply specify '16' for this value to indicate sixteen hours from now.

A connection must be established to the database file and a cursor created as an attribute to this connection, the purpose of this cursor is a means to access and manipulate tables and rows within the database. This is done by creating SQL commands, the first command executed will remove any tables in the database, this is to ensure each time the Job Submission Tool is run the system starts with an empty database with no job submission entries.

After this, a table is created in the SQLite database called **'jobs'**, more on this in the Section 4.3. The input from the user is then validated to check whether there have been four entries submitted and that they have been submitted in the correct formats i.e. they are integer values for runtime and deadline, a string value for the filenames and multiple formats for the arguments.

This input is then formatted for insertion into the **'jobs'** table, the **'deadline'** is converted to a pandas date frame by adding the number of hours to the current time and the current time is used for the submission time to be associated with this entry. This information is then inserted into the **'jobs'** table and the changes are then uploaded to the SQLite database when the commit operation is called.

A commit in SQLite is a transactional operation that permanently saves any modifications made to the database since the last commit or rollback. By executing a commit, the changes made to the database are persisted, ensuring their availability to other processes or users accessing the database. For example, submitting a job entry to the database using a commit operation would upload the entry to the database file, allowing other components to recognize the changes made. Finally, the program should loop this whole process and once again prompt the user for another job submission.

4.3 Database Design

As briefly mentioned in the previous section, the database will include the columns **'submission-Time'**, **'id'**, **'runTime'**, **'fileName'**, **'args'** and **'deadLine'**. The primary key for this database is the 'id' column each time the Job Submission Tool parses a job submission successfully it will increment a counter, this counter is used for the 'id' parameter such that each entry has a unique identifier. This acts as a primary key for the table, a primary key is required for data integrity, it ensures each row has a unique identity and can be accessed without any ambiguity as it prevents duplicates. This also improves query performance when trying to access rows in the table, because of this unique attribute there is no need to search the table when looking to delete or read a particular row. Each row is assigned a submission time from the Pandas current time function, the purpose of this is so that the scheduler can calculate the deadline time for the job based on the specified number of hours given to execute the job. Table 4.1 shows a sample from the **'jobs'** table with a sample row:

submissionTime	id	runTime	filename	args	deadline
2023-03-01 12:18:15.329748	1	16	passengers.py	4	24

Table 4.1: *Sample from the 'jobs' table, in the SQLite database file*

4.4 Scheduler Design

In this section, the design of each of the schedulers components will be outlined, explaining the purpose and functionality of each one. Once each component has been described, the top level algorithm will be shown to give an understanding of how the system will run and sustain a job queue with or without jobs present in the connected SQLite database file.

4.4.1 Forecasts

The Carbon Intensity predictions are downloaded from the National Grid ESO API as a comma separated file, this should then be parsed into a pandas data frame. This data frame should then be processed such that any entries which have the actual values for the carbon intensity i.e. are in the past, are removed. Then the next 24 hours are to be selected so that the system has access to the next day's predicted carbon intensity at each 30 minute interval, as seen in Table 6.1.

datetime	forecast	actual	index
2023-03-01T00:00:00	233	218	high
2023-03-01T00:30:00	231	220	high
2023-03-01T01:00:00	230		high
2023-03-01T01:30:00	227		high
2023-03-01T02:00:00	208		high

Table 4.2: *Sample from the National Carbon Intensity Forecast for the GB electricity system (Datopian, 2023), missing entries in the actual column, indicating a timestamp which has not yet occurred or been updated by the API yet.*

The values used for the prediction of carbon emissions for a job will be the forecast value in the table for the corresponding time, this value represents the amount of CO_2 emissions produced per kilowatt-hour of electricity consumed. The 24-hour maximum window can be easily adjusted if the system were to be configured to schedule jobs over perhaps multiple days or weeks. The forecasts are updated hourly but not with routine timing, so the system will check for updates regularly. The index value is a measure that indicates the degree of carbon intensity in relation to other levels. The low granularity of data obtained here makes processing and calculations in the scheduler much more lightweight.

4.4.2 Job Queue Management

During the initialisation of the scheduler the first jobs will be read into the system from the database, such that they are stored locally into a Pandas data frame for processing, the SQLite database file will then be regularly checked for updates after this and any new jobs detected will be added accordingly. To do this, firstly there has to be a connection established to the database file, then a SQL query designed to select all the entries in the **'jobs'** table. Then make use of a Pandas function which reads the results of this query i.e. the table columns and rows, then processes these into an accessible data frame with the corresponding labels for each attribute.

This data frame will be made global so that the other components can manipulate and observe this list of jobs and process them appropriately. Besides the refreshing of the job queue from the database, the only other connection that must be made with the database is to delete jobs when the job has been submitted to the cluster, meaning it no longer must be processed for scheduling as it has reached the optimal execution time. This will be done by utilising the unique identifier **'id'** for the specific jobs' entry, then executing an SQL query where the job with the corresponding **'id'** is removed, these changes are then commited to the SQLite database file.

4.4.3 Calculating Target Execution Times

In order to decide the best time to execute a job, to have the lowest carbon emissions possible in the given time window, each position in the window must be considered. Where there is space to complete the job, i.e. the deadline minus the runtime would be the latest the job could be executed. The forecast provides a reading for every 30-minute window of what the carbon intensity should be in that window. To increase the effectiveness of the optimisation, these values will be processed into a list with 30 occurrences of each 30-minute window value. Thus, each minute will have its own value. Naively this may seem pointless, but when traversing every minute, there are occasions when starting a job at a certain point in one of these 30-minute windows and finishing in another window results in a significantly lower total carbon emission. When compared to simply working with these large 30-minute windows as options.

Hence, each time a job is processed a window is calculated based on the runtime and deadline values, then the algorithm transverses all the aforementioned minute points in chunks, these chunks are the length of the runtime. For example, a 40-minute job will calculate the sum of 40 values for carbon intensity for each minute it will be running then the algorithm will step one minute forward repeat the total calculation and compare to see if the total is smaller. More of this will be clarified in the implementation Section 5.3.4. Repeat this process until the time reaches the job deadline minus the runtime, i.e. there is no more space left for optimisation as if the job was run any later it would not be completed by the deadline.

The optimal time for execution is the result of this calculation and this is appended to the locally stored **'runtimes'** list explained in Section 4.4.4. This operation is run routinely by the scheduler such that when any updates are made to the Carbon Intensity predictions, if there is a better slot for the job available it will be reassigned to that updated execution time.

4.4.4 Runtime Management

The optimal execution time associated to each job will be stored locally, this is an effort to reduce the overhead of the program and means less connections have to be made to the SQLite database, than if these execution times were uploaded to the database after each update. This functionality is not required, as only the scheduler program needs to know this information. The optimal execution times will be stored in a list of tuples, each tuple containing the **'id'** attribute associated with the job and the optimal execution time which has been calculated for the job.

The purpose of this list is to keep track of when jobs are to be executed and is used when verifying if the job has reached its runtime and should be executed. The list will be routinely updated by the scheduler if there are any new jobs are submitted, or as a result of forecast updates, the optimal execution times will be updated accordingly. After a job is executed, the corresponding entry in the list will be removed subsequently.

4.4.5 Job Execution

Each job must be routinely checked to see whether it has reached the execution time, then submitted to the cluster with the appropriate command containing the filename and any associated arguments. This component will iterate through the list of jobs, checking this condition if any jobs are present in the queue, of course. In the case, that a job has reached its optimal execution time, the program should submit a command line statement to the cluster with all the relevant information for running the job.

After submitting a job, the scheduler needs to keep running to handle other jobs. It will not wait for the current job to complete before proceeding with other tasks. To achieve this, a detached process is necessary, which separates from the parent process, this child process then submits the jobs details to the cluster. Subsequently, once the job has been submitted, the corresponding entry for the job in the **runtimes** list will be removed and the SQLite database will be updated to remove the job entry.

4.4.6 Core Routine

The main method will initially call the method responsible for fetching the jobs in the SQLite database, then if any are present begin the loop for scheduling the jobs. If there are no jobs present in the database, the program will wait then keep checking routinely to see if any jobs have been submitted for scheduling, if so, begin the scheduling loop.

Figure 4.2 is a flow diagram which describes the behaviour of the scheduler and how it reacts to jobs presence and ensures a continuous cycle of improvements and maintenance. The diagram primitively outlines the roles and responsibilities of each component and demonstrates how the scheduler operates to maintain a dynamic and adaptive environment by processing job submissions, reacting to changes in forecasts or new jobs being submitted to the database. While waiting between each iteration of the loop in order to keep the overhead of the program to a minimum.

4.5 Google Cloud Configuration

This section will describe the environment necessary for such a system to function and a platform which is suitable for testing and experimentation. These details will allow the scheduler to be tested in the real-world, not simply simulated on a local virtual machine like other research papers on this topic (Zhao and Zhou, 2022). Real-world testing allows for the collection of actual results from real-world situations, which provides a more accurate and comprehensive understanding of the effectiveness of the program. This configuration also helps identify unforeseen issues and

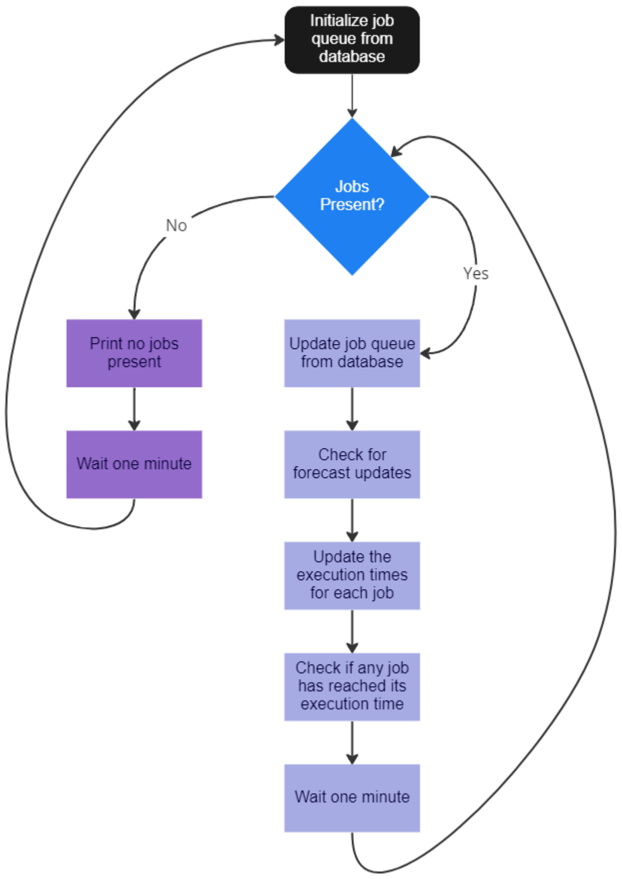

Initialize job queue from database
Jobs Present?
No
Yes
Print no jobs present
Wait one minute
Update job queue from database
Check for forecast updates
Update the execution times for each job
Check if any job has reached its execution time
Wait one minute

Figure 4.2: *Flow chart diagram of the scheduler behaviour based on job presence*

challenges which may have not been anticipated or encountered when testing on a local machine or simulation. Overall, it will delve into the cluster configuration information, the cloud storage techniques employed and the web interfaces which will be utilised for experiments and testing.

4.5.1 Cluster Configuration

For the experiment results to be valid and meaningful, the cluster used must be located in a region which the National Grid Carbon Intensity forecasts covers, i.e. in the UK. Thus, the cluster will be assigned a location within the UK, namely **’europe-west2’** which resides in London (Cloud, 2023).

Then comes the hardware configuration for this, a master and worker architecture has to be employed for Apache Spark to work effectively, for the purpose of testing and keeping costs to a minimum, this will involve one master node and two worker nodes. Each node will be a quad-core CPU with 4 GB of RAM per core and 50 GB of disk space per node, this will be more than enough processing power for this implementation (Kaewkasi and Srisuruk, 2014) and to generate valid results.

Google Cloud Dataproc service automatically installs many useful observatory web interfaces for the cluster including YARN ResourceManager, MapReduce Job History and Spark History Server. For running the scheduler with ease, the Jupyter interface will also be installed to improve the overall usability of the cluster.

4.5.2 Bucket

A bucket will be used to store the scheduler, job submission tool, the job files and datasets the jobs will be processing. On the Google Cloud Platform (GCP), a bucket is a container used to store data objects such as files, images, videos, and other unstructured data. The purpose of a bucket in the GCP is to provide a highly scalable and reliable way to store and manage data, while also providing granular access control and management features. The bucket will always hold the data whether the scheduler or job submission tool is running or not, so in the case of a crash, all data will not be lost. One bucket's data can also be accessed by multiple clusters, this opens the door for various cloud configurations or structures that may be appropriate for different scenarios.

4.5.3 Web Interfaces

Having web interfaces openly available on the cluster is extremely valuable for testing and analysing performance. The Spark History Server is a simple tool which gives information about when a job started and finished, as well as an event log, which contains information about the job's metadata and information like the CPU usage and memory usage which would be relevant for calculating power usage. Jupyter web interface provides a powerful, scalable, and cost-effective platform for collaborative data science work, with access to a wide range of big data technologies.

4.6 Summary

In summary, the section outlined how a carbon aware scheduling system can be assembled, it goes into detail explaining each of the component's functionalities and dependencies. A command line job submission tool has been described and the method of communication between this and a scheduling program. The intricacies of the scheduler are explained in a way such that it can be understood how the system actually determines the optimal execution time for a given job. Also, the dynamic behaviour of the system where it adapts to changes in forecast updates and new job submissions, this behaviour is illustrated by Figure 4.2 which gives an overview of how the system maintains a stable environment. Finally, a cluster configuration that is appropriate for this system has been described, which has the potential to yield real-world outcomes through experimentation.

5 | Implementation

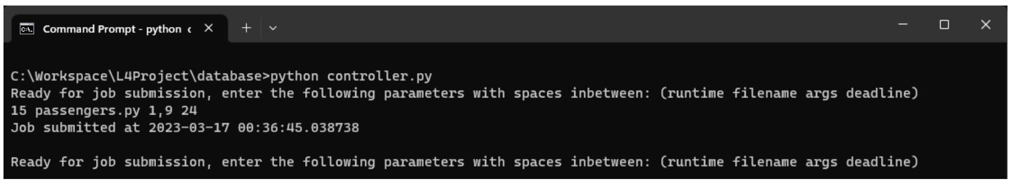
In this chapter, the details of how the program, which was previously outlined in earlier chapters, has been implemented. It will explore the various steps that were taken to ensure the successful implementation of the program and the technical choices that were made during this process. This chapter aims to provide a comprehensive understanding of the implementation process, including the challenges faced, and the solutions employed to overcome them. By the end of this chapter, there will be a clear picture of the various intricacies involved in turning an idea into a functioning system.

5.1 User Interface

This section will briefly highlight the user interfaces which will be available to the user when using the system. These include the job submission tool, monitoring the scheduler and the various web interfaces available for tracking and monitoring the Spark contexts.

5.1.1 Job Submission Tool

The job submission tool displays a prompt to the user asking for their submission when the program is run. This prompt asks the user for **runtime**, **filename**, **args** and **deadline** in a list separated by spaces as seen in Figure 5.1. This input is then validated to check whether the four values have been entered correctly and if not will display an error message, then prompt the user again to retry the input as in Figure 5.2. If the input is parsed successfully and in the correct format, then the data will be passed into the database and a success message indicating the job has been uploaded to the database will be displayed with the submission time, followed by a prompt for another job.

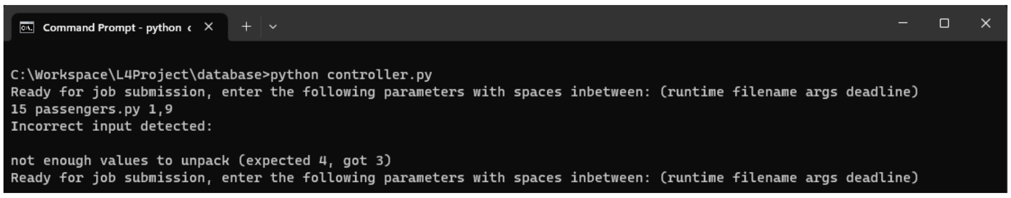


```
C:\Workspace\L4Project\database>python controller.py
Ready for job submission, enter the following parameters with spaces inbetween: (runtime filename args deadline)
15 passengers.py 1,9 24
Job submitted at 2023-03-17 00:36:45.038738

Ready for job submission, enter the following parameters with spaces inbetween: (runtime filename args deadline)
```

Figure 5.1: *Screenshot showing the Job Submission Tool being started and a prompt being displayed to the user, followed by job parameters being entered by the user. Subsequently, the success message is displayed as the input has been validated, and the submission time being displayed*

5.1.2 Scheduler Program

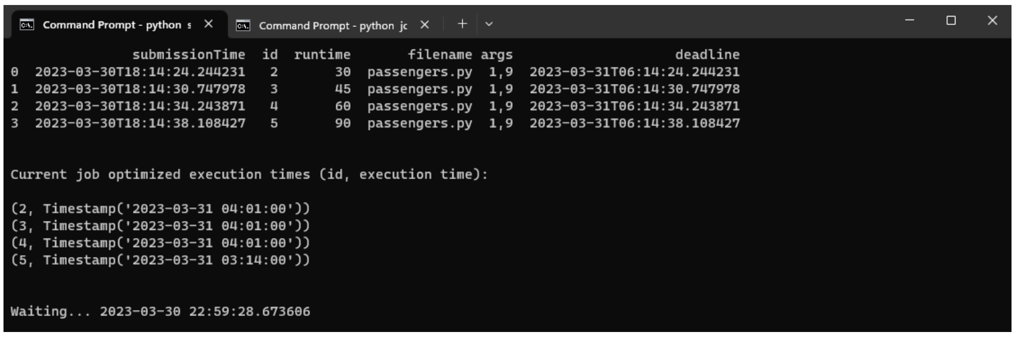
Upon starting the scheduler, the system will give one of two outputs depending on whether jobs are present in the database for scheduling. If jobs are present, the system will display the current jobs in the queue, then once the initial scheduling, display each of the job's optimised execution time. Then, display that the system is waiting for the next iteration to occur, all of this is shown



```
C:\Workspace\L4Project\database>python controller.py
Ready for job submission, enter the following parameters with spaces inbetween: (runtime filename args deadline)
15 passengers.py 1,9
Incorrect input detected:

not enough values to unpack (expected 4, got 3)
Ready for job submission, enter the following parameters with spaces inbetween: (runtime filename args deadline)
```

Figure 5.2: *Screenshot showing the Job Submission Tool being started and a prompt being displayed to the user, followed by job parameters being inputted incorrectly, with a missing parameter for the 'deadline' value. Thus, an error message explaining the mistake is displayed*

in Figure 5.3. If jobs are not present, then a message will be displayed stating so and that the system is waiting for jobs to be submitted to the database. Upon a job reaching its execution time, it will be sent to the cluster and a message will be displayed stating it has been removed from the database with an execution time.

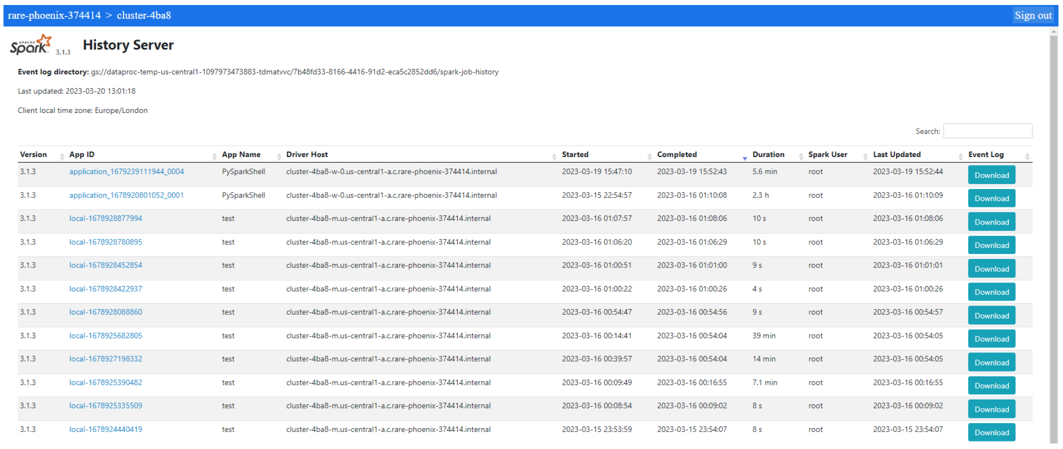


```
              submissionTime  id  runtime       filename args                   deadline
0  2023-03-30T18:14:24.244231   2       30  passengers.py  1,9  2023-03-31T06:14:24.244231
1  2023-03-30T18:14:30.747978   3       45  passengers.py  1,9  2023-03-31T06:14:30.747978
2  2023-03-30T18:14:34.243871   4       60  passengers.py  1,9  2023-03-31T06:14:34.243871
3  2023-03-30T18:14:38.108427   5       90  passengers.py  1,9  2023-03-31T06:14:38.108427


Current job optimized execution times (id, execution time):

(2, Timestamp('2023-03-31 04:01:00'))
(3, Timestamp('2023-03-31 04:01:00'))
(4, Timestamp('2023-03-31 04:01:00'))
(5, Timestamp('2023-03-31 03:14:00'))


Waiting... 2023-03-30 22:59:28.673606
```

Figure 5.3: *Screenshot of the Scheduler showing the current jobs in the queue, waiting for to be executed. Followed by the current optimised execution time for the jobs in the queue with their corresponding 'id' number. Then a message indicating the time and that the scheduler is waiting to loop again.*

5.1.3 Spark Job History Server

The cluster was configured such that the Spark context created when a job is submitted to the cluster can be monitored to observe the execution of the application. This user interface gives an in-depth analysis of how the job has run on the cluster (Nnk, 2020), breaking down how the job has utilised the systems' hardware. These metrics would be helpful in the future for improving the way a job is run on the cluster to better use resources more effectively and using less energy (Ganesan et al., 2020). In Figure 5.4 can see a list of Spark Contexts which have been created during instances of jobs being submitted on the Google Cloud Platform cluster, these individual contexts can be investigated further to reveal further information. For instance, when the driver has been added and the worker nodes as displayed in Figure 5.5. These executors can be individually analysed to find out information about how much memory is assigned per node and task success or failures. This can be observed in Figure 5.6. This information was used to improve the cluster hardware configuration for more effective and more cost-effective testing.



History Server

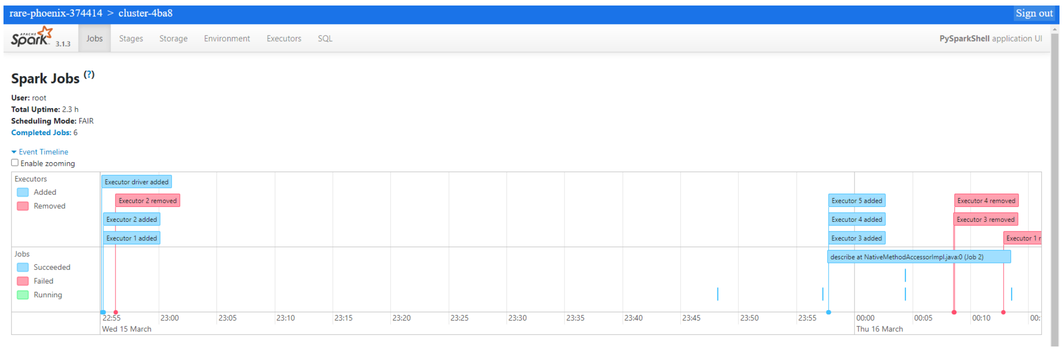
Event log directory: gs://dataproc-temp-us-central1-1097973473883-tdmatvvc/7b48fd33-8166-4416-91d2-eca5c2852dd6/spark-job-history

Last updated: 2023-03-20 13:01:18

Client local time zone: Europe/London

Version	App ID	App Name	Driver Host	Started	Completed	Duration	Spark User	Last Updated	Event Log
3.1.3	application_1679239111944_0004	PySparkShell	cluster-4ba8-w-0.us-central1-a.c.rare-phoenix-374414.internal	2023-03-19 15:47:10	2023-03-19 15:52:43	5.6 min	root	2023-03-19 15:52:44	Download
3.1.3	application_1678920801052_0001	PySparkShell	cluster-4ba8-w-0.us-central1-a.c.rare-phoenix-374414.internal	2023-03-15 22:54:57	2023-03-16 01:10:08	2.3 h	root	2023-03-16 01:10:09	Download
3.1.3	local-1678928877994	test	cluster-4ba8-m.us-central1-a.c.rare-phoenix-374414.internal	2023-03-16 01:07:57	2023-03-16 01:08:06	10 s	root	2023-03-16 01:08:06	Download
3.1.3	local-1678928780895	test	cluster-4ba8-m.us-central1-a.c.rare-phoenix-374414.internal	2023-03-16 01:06:20	2023-03-16 01:06:29	10 s	root	2023-03-16 01:06:29	Download
3.1.3	local-1678928452854	test	cluster-4ba8-m.us-central1-a.c.rare-phoenix-374414.internal	2023-03-16 01:00:51	2023-03-16 01:01:00	9 s	root	2023-03-16 01:01:01	Download
3.1.3	local-1678928422937	test	cluster-4ba8-m.us-central1-a.c.rare-phoenix-374414.internal	2023-03-16 01:00:22	2023-03-16 01:00:26	4 s	root	2023-03-16 01:00:26	Download
3.1.3	local-1678928088860	test	cluster-4ba8-m.us-central1-a.c.rare-phoenix-374414.internal	2023-03-16 00:54:47	2023-03-16 00:54:56	9 s	root	2023-03-16 00:54:57	Download
3.1.3	local-1678925682805	test	cluster-4ba8-m.us-central1-a.c.rare-phoenix-374414.internal	2023-03-16 00:14:41	2023-03-16 00:54:04	39 min	root	2023-03-16 00:54:05	Download
3.1.3	local-1678927198332	test	cluster-4ba8-m.us-central1-a.c.rare-phoenix-374414.internal	2023-03-16 00:39:57	2023-03-16 00:54:04	14 min	root	2023-03-16 00:54:05	Download
3.1.3	local-1678925390482	test	cluster-4ba8-m.us-central1-a.c.rare-phoenix-374414.internal	2023-03-16 00:09:49	2023-03-16 00:16:55	7.1 min	root	2023-03-16 00:16:55	Download
3.1.3	local-1678925335509	test	cluster-4ba8-m.us-central1-a.c.rare-phoenix-374414.internal	2023-03-16 00:08:54	2023-03-16 00:09:02	8 s	root	2023-03-16 00:09:02	Download
3.1.3	local-1678924440419	test	cluster-4ba8-m.us-central1-a.c.rare-phoenix-374414.internal	2023-03-15 23:53:59	2023-03-15 23:54:07	8 s	root	2023-03-15 23:54:07	Download

Figure 5.4: *A page displaying a table on the Spark History Server, showing entries which list Spark Contexts which has been created by the cluster*



Spark Jobs (?)

User: root
Total Uptime: 2.3 h
Scheduling Mode: FAIR
Completed Jobs: 6

Figure 5.5: *A page displaying a time chart showing when executors have been added to a Spark Context*

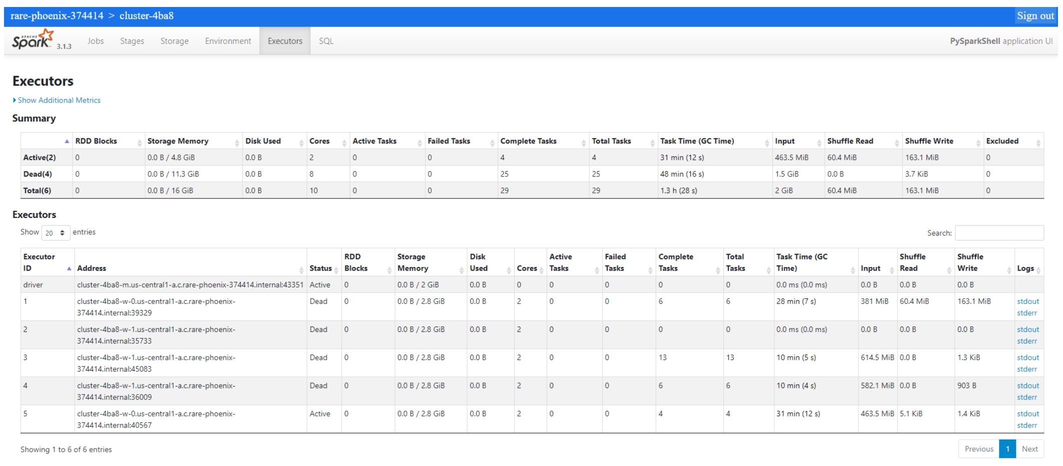
5.2 Back-end

5.2.1 Database Management and Maintenance

The database controller program maintains a job queue in an SQLite database, allowing job submissions with the job specific parameters which are required for the scheduling algorithm to make its optimisations. Each job submitted is stored in the **'jobs'** table, each entry with its corresponding submission time, runtime, filename, arguments, and deadline. The programs utilizes the Pandas library to determine the deadline time by adding the specified number of hours provided by the user as the final parameter to the current time, which sets the time limit for the job's runtime optimization. The input taken from the user is processed as a space separated list, this allows the user to list multiple arguments for their job as seen in Figure 5.1.

Each column in the table is assigned an SQL data type, this is used to validate the input and if the input is in the wrong format the error will be caught and the user will be prompted again with a formatted error message. For the scenario where the user has inputted an incorrect number of parameters, a **'ValueError'** is raised during the process, and if there are any issues with inserting the values into the database, such as detecting a wrong data type or any other submission error, a **'sqlite3.Error'** is also caught.

The interactions between the database and the scheduler must be reliable and fast to ensure updates and deletions are not missed by the scheduler, this problem is common in such a Producer–



Executors

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Excluded
Active(2)	0	0.0 B / 4.8 GiB	0.0 B	2	0	0	4	4	31 min (12 s)	463.5 MiB	60.4 MiB	163.1 MiB	0
Dead(4)	0	0.0 B / 11.3 GiB	0.0 B	8	0	0	25	25	48 min (16 s)	1.5 GiB	0.0 B	3.7 KiB	0
Total(6)	0	0.0 B / 16 GiB	0.0 B	10	0	0	29	29	1.3 h (28 s)	2 GiB	60.4 MiB	163.1 MiB	0

Executors

Show 20 entries

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs
driver	cluster-4ba8-m.us-central1-a.c.rare-phoenix-374414.internal:43351	Active	0	0.0 B / 2 GiB	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	
1	cluster-4ba8-w-0.us-central1-a.c.rare-phoenix-374414.internal:39329	Dead	0	0.0 B / 2.8 GiB	0.0 B	2	0	0	6	6	28 min (7 s)	381 MiB	60.4 MiB	163.1 MiB	stdout stderr
2	cluster-4ba8-w-1.us-central1-a.c.rare-phoenix-374414.internal:35733	Dead	0	0.0 B / 2.8 GiB	0.0 B	2	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr
3	cluster-4ba8-w-1.us-central1-a.c.rare-phoenix-374414.internal:45083	Dead	0	0.0 B / 2.8 GiB	0.0 B	2	0	0	13	13	10 min (5 s)	614.5 MiB	0.0 B	1.3 KiB	stdout stderr
4	cluster-4ba8-w-1.us-central1-a.c.rare-phoenix-374414.internal:36009	Dead	0	0.0 B / 2.8 GiB	0.0 B	2	0	0	6	6	10 min (4 s)	582.1 MiB	0.0 B	903 B	stdout stderr
5	cluster-4ba8-w-0.us-central1-a.c.rare-phoenix-374414.internal:40567	Active	0	0.0 B / 2.8 GiB	0.0 B	2	0	0	4	4	31 min (12 s)	463.5 MiB	5.1 KiB	1.4 KiB	stdout stderr

Showing 1 to 6 of 6 entries

Figure 5.6: *A page displaying an in-depth break-down of each executor involved in this Spark Context Instance*

Consumer model and must be considered when developing such communication channels (Revathy and Priya, 2020). This requires the communications with the database to be consistent and reliable in a way that ensures any data committed to the database is valid. In SQL database frameworks like SQLite, there is a methodology to ensure this is the case. This is an **ACID** transaction, which is a data transaction, which that must be completed following certain rules. This approach ensures that the scheduler can process the data from the database without encountering any errors, enabling it to make informed decisions based on the data. An **ACID** transaction has the following properties:

- Atomicity: Each transaction is interpreted as a whole, indivisible operation. Meaning all operations are completed successfully and if not the changes are not committed to the database.
- Consistency: The transaction's operations must leave the database in a valid state after the changes are submitted, after every transaction.
- Isolation: The operations being performed in these transactions are not yet visible in the database, they are isolated. To ensure data is not falsely interpreted, hence errors are avoided.
- Durability: Transactions successfully committed are permanent in the database and will be sustained if the system fails. In this case, if the Command Line Job Submission tool was terminated, the database would survive and the scheduler would still be able to function correctly with the remaining job queue.

The system has been designed to comply with these standards, which guarantees smooth and error-free access and utilization of the database for all transactions. One way this has been employed is input validation strategies, these have been implemented to prevent any data that is not in the appropriate format from being uploaded to the database. This is done before any changes are committed to the database, ensuring only valid, true data can be uploaded. All of this means less fatal exceptions, meaning less time wasted or tests invalidated.

5.3 Carbon-Aware Scheduling Algorithm

5.3.1 Entry Point

The entry point of the program is the **'main'** method, this is responsible for creating an initial environment by reading any jobs from the database. If jobs are found, the method will then proceed to interact with the other components in order to queue the jobs with the appropriate execution delay. This routine method is also responsible for calling the method which checks whether jobs have reached their optimal runtime, i.e. are pending jobs. The other responsibilities of this method are calling the method responsible for checking for forecast updates and for calling the method responsible for making the runtime optimisations.

These actions are performed periodically in a loop when jobs are present, this loop waits 60 seconds between each iteration. As aforementioned, this waiting between iterations is to reduce the schedulers overhead and does not affect the performance of the algorithm.

The flow diagram in Figure 4.2 displays how the method loop interacts with each of the program's components in order to maintain a stable state. When jobs are present, they are held in an optimized queue until they reach the optimized execution time. In the Figure 4.2 a list of components can be seen which have separate responsibilities within the scheduler, these will be explained further in this section.

This technique of periodic updates and calculations, guarantee that the optimal execution time for a job is updated in real time, meaning once updated forecasts are received these will be checked to see whether there is a better time slot for the job to be run in. This technique is employed in order to further reduce carbon emissions, and execute the job as close as possible to the absolute perfect time slot to achieve the lowest carbon footprint.

5.3.2 Updates from the Database

The method **'updateQueue'** is responsible for updating the global ***'jobs'*** (not to be confused with the 'jobs' table) variable in the program with any updates from the database, these include jobs being added, removed or perhaps the database being reset. Firstly, a connection is established with the database and then a cursor created in order to access the data by performing SQL queries. Then the data will be read from the **'jobs'** table, processed into the ***'jobs'*** variable in the program.

This process of translating the data from a SQLite database to an accessible and indexable format in the stack is completed using the Pandas packages function for reading SQL tables. This function takes two mandatory arguments, namely the SQL query, in this case a SELECT statement which selects all from the table and the connection variable created earlier in the method. The returned data frame contains a row for each record in the result set, with column names derived from the SQLite database, these column names are used for selecting data specific to an entry (job) within the program. Finally, if any jobs have been read from the database, they are printed out in a readable table format for the user to observed from the command line interface, alongside their optimised execution times.

5.3.3 Processing of Carbon intensity Forecasts

The **'updatePrediction'** method is responsible for Carbon Intensity forecast data, it retrieves the latest carbon intensity forecast data from the National Grid API and extracts the forecast values for the next 24 hours. This is done through a few conditions, namely removing all entries without prediction values i.e. ones too far in the future and all entries timestamps must be within the next 24 hours. This data when retrieved is in a comma separated file format, a Pandas function is used in order to process this data in a Pandas data frame which can be accessed and used in carbon emission calculations later in the program. The **'forecast'** column is then extracted from the data frame and converted to a list of integer values representing the Carbon Intensity value for each 30-minute block.

The blocks in the forecast are 30-minute intervals, but simply multiplying each block by 30 to represent the next 24 hours is not possible. For example, if the current time is 14:20, the first block used for the forecast prediction will be the 14:00-14:30 block. Multiplying this block by 30 would be incorrect because 20 minutes of the block have already passed, which would offset all the other Carbon Intensity values incorrectly and result in a misalignment with the actual time.

To address this issue, a formula is used, which involves taking the remainder of the minutes divided by 30. For instance, in the case of 14:20, the number of minutes is 20, so the formula would be 20%30. This value is then used to remove the Carbon Intensity prediction values that are in the past. Specifically, the first (minutes past the hour % 30) elements in the list of Carbon Intensity values for the next 24 hours, are removed. This is done after the blocks have been multiplied by 30 to represent each minute in the 24-hour window.

5.3.4 Optimising Runtimes for Lower Carbon Emissions

Once jobs are processed into the scheduler, they have to be assigned an execution time. This mechanism is the responsibility of the **'updateTargets'** method, here each job in the queue is allocated an execution time, this time is calculated by determining which time slot in the current Carbon Intensity forecasts would produce the lowest carbon emissions, based on the job's runtime. This window for optimisation is constrained by the submission time and the deadline which is associated with the job.

Firstly, the method will check whether each job exists in the list of **'runTimes'**, if not the job will be assigned its deadline minus the runtime of the job, as its execution time for now. Then each job is passed into a function with its corresponding runtime, this function namely **'lowestCarbon'** returns the optimal runtime for this job based on the current Carbon Intensity forecasts, this function is discussed later. This optimal execution time is then assigned to the job in question and its current entry in the list of **'runTimes'** is replaced with this new updated one.

Note before this update attempt the method checks whether the job is within 5 minutes of its currently assigned execution time, if so it will not attempt to optimise the timing, thus letting the job be executed as here it is unlikely that the forecasts are going to increase accuracy any more.

The **'lowestCarbon'** function is implemented such that it will find the best slot in a list of numbers, in this case representing Carbon Intensity predictions. It does this by summing blocks which are the size of the runtime. For instance, if the job took 4 minutes to run, the function would sum four blocks in each step. The function then steps forward one minute over the list of values until the tail of the section reaches the end of the list, meaning there is no more room for the job to run after this time. Figure 5.7 illustrates this method, the blue represents the blocks which are being summed to calculate the overall carbon emissions the job would produce, this technique finds the slot which has the lowest value. More namely, the most optimal timing possible for the lowest carbon emissions for that specified runtime.

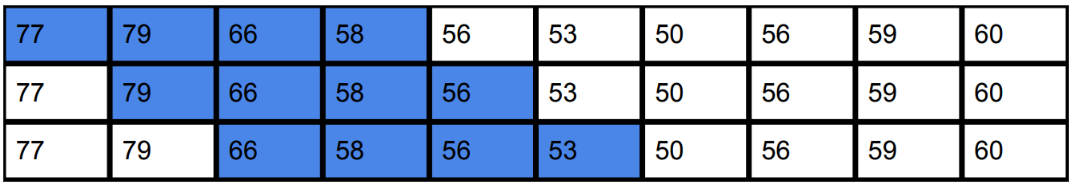


Figure 5.7: This is a visual representation of how the algorithm finds the minimum in the list of values, each block is assigned a different value to better understand how the comparisons are made each step.

Once this optimal slot is found the starting position of the slot is taken, this represents the minute when the job should be started. The function then converts this value into a timestamp which corresponds to the optimal time when the job submitted should be executed. This value is returned to the **'updateTargets'** function, which as previously mentioned then updated the list of 'runTimes' with the new updated optimal timing.

5.3.5 Submitting Jobs to the Cluster

Besides maintaining a stable environment and optimised job queue, the program must also submit these jobs to the cluster once they reach their optimal execution times. For this, the **'executeDeadlines'** method is part of the routine procedure, the purpose of this method is to submit pending jobs to the cluster. It does this by firstly iterating through the current list of jobs and checking whether their execution time has occurred yet or not.

If this is the case, the job must be submitted to the cluster and the method should continue to check the other jobs if any present and if not, terminate. The method submits the jobs to the GCP cluster using the **'spark-submit'** command, within this command the filename and arguments corresponding to the job are passed to the cluster. This command is executed in the cluster terminal outside the program, this is done by using a subprocess which detaches the action of submitting the job from the program's parent process, this is illustrated in Figure 5.8. This procedure is necessary to allow the parent process i.e. the scheduler to keep running while jobs are being processed on the cluster.

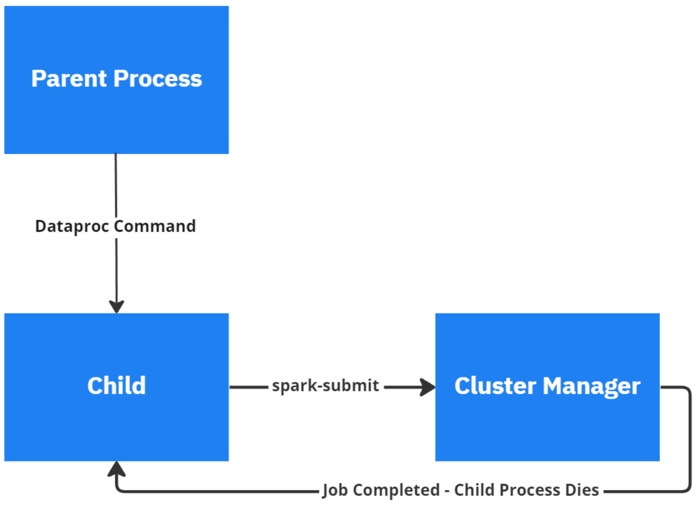


Figure 5.8: *This figure illustrates how the parent process passes the job submission details to a new child process, which then submits the job to the Cluster Manager. This child process waits for the job to be completed, then is terminated. During this time, the parent process has been running unaffected.*

Once this command has been transmitted, the method then deletes the job instance from the global variable **'jobs'** and also removes its corresponding execution time from the **'runTimes'** list. Finally, the method calls a function which is responsible for removing the job from the SQLite database, the unique identifier is passed for this job to the function. The function **'deleteJob'** then proceeds to establish a connection with the database, create a cursor to run SQL queries and removes the entry from the table.

Then will subsequently print out a message stating the time the job was submitted and when it was executed on the cluster. After a job is successfully executed and removed, the method will recursively call itself in order to reset the index in the loop such that there is no misinterpretation now that the **'jobs'** variable is one smaller.

5.4 Integration of Scheduling Algorithm with Spark on GCP

In order for the job files and datasets to be accessible to the system, they were uploaded to a bucket on the Google Cloud Platform, these can be accessed via an address which fetches the file from the cloud location. The scheduler, database, and job submission tool were also in this bucket, this is where the cluster will access these components. Separate command line interfaces are opened on the cluster, in one of them the job submission tool is started, then job submissions are made to the database from this terminal.

From the second command line terminal the scheduler program is started, this will begin to then schedule any jobs in the database and execute each job at their corresponding optimal time. Jupyter provides these terminal window interfaces for completing these actions and monitoring the status of the system.

Each job submitted has its own **spark-submit** string, which is responsible for starting the job in Spark on the cluster. The format of this string contains information about the current cluster being used, the location of the cluster and the address of the corresponding bucket where the job files are located as seen here:

```
"gcloud dataproc jobs submit pyspark
–cluster=cluster-4ba8
–region europe-west2
gs://gti-bucket1-dataproc/notebooks/jobs/{0}
– {1}"
```

where {0} represents the filename and {1} represents the list of arguments.

This command sends an HTTP request to the Dataproc API, with the relevant information about the PySpark job, this request is then authenticated and the parameters verified. Lastly, the Dataproc master node processes the job submission details and begins to schedule the job on available worker nodes for processing.

This command is the only part of the program which is unique to the Google Cloud Dataproc Platform. Other cloud platforms also use this type of command to submit PySpark jobs to a cluster environment. This results in the program being easily transferable into many environments and cloud providers, by simply changing one command.

Assuming this command is altered, the operation of the scheduler will run the same, this does not affect the performance of the algorithm. This is a consequence of ensuring when designing the scheduler the finished solution is not strongly blended or integrated into one environment or cloud service, this results in a more versatile and maintainable implementation.

5.4.1 Fault Tolerance and Scalability

In a distributed systems environment, fault tolerance is an indispensable element that must be given high priority, since overlooking it can have severe consequences. Reliability is crucial as in such systems there are regularly numerous components which are prone to failure, these failures could not only waste time but produce unnecessary carbon emissions if the task must be reattempted. Fault tolerance also affects the availability and scalability of a system, which is vital for the overall performance and user experience.

Hence, Apace Spark was selected for the framework in this implementation as it offers many mechanisms to address this factor. Spark utilises Replication, this is a technique where data is replicated across multiple nodes which are available in the cluster, this ensures data is available even if one of the nodes fail. Spark can also replicate RDD partitions across multiple nodes in order to provide backup in the event of a node being unsuccessful.

Several of these mechanisms also contribute to increasing the scalability of the system. For instance, the replication of data across nodes allows the system to handle larger workloads without compromising reliability or availability. The assumption of infinite resources is of course a fallacy, but due to the nature of the schedulers simple interaction methods with the Cloud Service. It could easily be configured to make use of multiple clusters, which could be used for a more sophisticated solution to allocating resources for different jobs.

5.5 Development Practises

In research, it is common for the technologies used, methodologies, or the timeline to evolve as new data is collected or as new insights are obtained. **Agile** is a software engineering methodology used to assert an iterative and incremental development pattern which focusses on continuous prototyping and time-bound iteration cycles (Abrahamsson et al., 2017). Compared to other development patterns like Scrum or Waterfall, Agile allows for more flexility and adaptability when adjusting requirements and strategies.

This is particularly important in research projects like this, where new information or developments can be made which significantly change the researchers' perspective of the proposal. Requirements must be concise and strictly adhered to in order to produce a result which is relevant and answers questions which have been asked. Hence, it is important to revisit these requirements throughout the research and development process in a project, as a means to ensure they are appropriate and progress is being made towards them.

Once the first working prototype was developed, many features still had to be implemented, such as the actual scheduling components of the program and the methods of communication between the Job Submission Tool and Scheduler. In order to ensure this progress was not lost, if say down the line the program had an unsuccessful development cycle. For instance, complicated faults or obstacles could be encountered when there is coupling of more components, complexing the implementation.

Feature Branching is a method that was used, this is a technique where new features are developed on a separate branch in the version control system, **GitHub** in this case. This technique allowed for changes to be reviewed and easily identified when compared to previous iterations. The risk of merge conflicts and errors are also reduced, as separate components of the implementation can be worked on independently in their own features branches.

When these branches are merged with the main branch it is important they do not disrupt the other components in the system, it must be ensured that the feature branch is not bringing any changes which could have these implications. The method employed to prevent this was **Unit Testing**, this can be used to test isolated units of code to ensure they produce expected outputs. This is crucial for other components which rely on this unit of code to produce reliable and valid data. Rapid identification of mistakes, which could have been missed, through unit testing prevents these errors from developing into more complex issues later in the development process.

6 | Evaluation

As discussed in Chapter 1, there are many obstacles to be overcome when trying to implement a successful carbon aware scheduling algorithm. The limitations and assumptions involved with this solution have been discussed, it is always important to portray and understand the implications of these. Whether it be for future research or to further enhance the algorithm's performance and applicability in real-world scenarios. For example, analyse the algorithm's scalability, robustness to ability to handle varying workloads and unexpected events.

In this chapter, the way these challenges have been tackled will be discussed and analysed, in order to determine not only if they were successful but also if any insights have been uncovered in the process. The performance of the algorithm will also be measured in an experimental environment on a real Google Cloud Platform cluster, this will be interesting as most papers on carbon aware computing generally only use simulations, not real experiments. The objective here is to show how well the solution has addressed the general problem, how these results should be interpreted, all while providing evidence to support this.

6.1 Experimental Setup

The experiments in this project were performed on a GCP Dataproc cluster, the location of this hardware was in London. The location of the cluster had to be inside the UK, this is because the Carbon Intensity Forecast API solely provides data for the UK. Meaning, the electricity used by this distributed system is actually relevant to the forecasts being interpreted for the job delays.

The hardware configuration used for testing, as previously mentioned in Section 4.5.1 was a master node and two worker nodes. In order to monitor the time the jobs were being executed i.e. the optimised time. A text log file was created, in here information for each job was appended, each time a job was executed on the cluster. This log file would contain all the parameters involved in the job submission and the time which the job was sent to the child process to be submitted to the Dataproc API and then the cluster. The cluster was kept running for the entirety of an experiment, this was until every job was executed, and the queue was empty, no downtime whatsoever.

To gauge the performance of the algorithm, many factors were considered to get a variance of test results. Workload characterisation was the key here, using jobs with different runtimes was key to finding the relationship between runtimes and amount of carbon emissions saved due to delaying workloads.

Different deadline windows for the jobs were used to simulate results that would be more like a real world scenario. For example, a job with a 12-hour deadline could be a developer submitting a job at the end of the working day which they required the results for the next morning. Similarly, a job with a 4-hour window could be a developer submitting a job in the morning and requiring the job before the end of the working day, the aim here is to show with alternating windows carbon footprint reductions can still be made.

Job submission time was also varied across testing, to further imitate a real-world scenario. In summary, the variances of job submission profiles includes the variance of the runtimes in minutes, deadline window in hours and finally the time of day for the submission.

To determine the actual carbon footprint savings, the total carbon emissions used by the job at the optimised time is calculated by multiplying the runtime of the job by the carbon intensity value for the corresponding time block it was executed in. If the job was executed in more than one block, this calculation is changed into a vector calculation which involves the percentage of the job which was executed in each block and this percentage multiplied accordingly with the runtime and carbon intensity of that block.

For example, a job runtime could be comprised of two blocks where 75 percent of the work is done in one block and the rest in the succeeding block, as in Figure 6.1. Here the calculation would simply be the first block's carbon intensity value multiplied by 75 percent of the runtime, and the same for the second block with its carbon intensity value and 25 percent of the runtime.

This value is a representation of the scale of the carbon emissions, it does take into to account the actual amount of electricity used by the hardware, this is not relevant. As the percentage saved is all that is required to interpret the savings. To calculate the total amount of carbon saved, the difference between the amount of carbon produced between the job being executed at submission time versus the optimised time, is calculated and converted into a percentage.

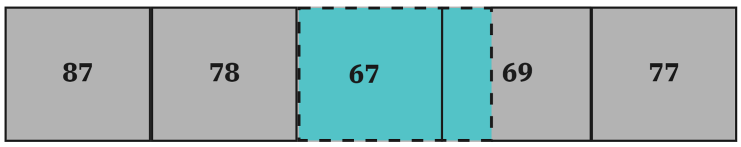


Figure 6.1: *This figure illustrates carbon intensity values in 30-minute blocks, the blue box represents a job runtime overlapping between two blocks. 75 percent of the job is in the first block and the other 25 percent is in the final block.*

An assumption here is that the runtime submitted with the job is absolutely accurate, thus will be used for these calculations.

6.1.1 Testing Schedule

This dissertation has consistently emphasized the multitude of factors that are involved in producing carbon intensity forecasts, as a result of this, the trend of this value can be susceptible to infrequent anomalies. To ensure valid and effective testing, this has been considered throughout the testing period.

As previously mentioned, jobs have been submitted with many runtimes and deadline windows. Tests were completed over a number of weeks to generate as many data points as possible in the time period available and with the lack of budget still in mind. Each day there were three 24-hour deadline window jobs submitted with different runtimes, this guaranteed there was some data on every day from the start of testing to the end.

Three times a week on random days, there were numerous jobs submitted with variating deadline windows and runtimes, these heavy testing days aimed to gather lots of data with different job submissions in a short period of time on the cluster. All job submissions were made during a variety of times during the days, ranging from the morning till late in the evening. The cost of keeping the cluster online was determined after the first week, and this gave information about if more or less jobs could be tested the following week based on the budget of free credits on the GCP available.

Although ideally for this type of implementation, which relies on forecasts which are heavily impacted by seasonal variation (Khan et al., 2018). Especially for nations like the UK, which rely heavily on renewable energy, the correlation between carbon intensity and electricity demand is weak. Longer term testing would be required to fully understand how the scheduler performs overall. The experimental results cannot be considered a definitive basis for the schedulers' ability to reduce carbon emissions at this magnitude throughout the year.

6.2 Evaluation of Carbon-Aware Scheduler

6.2.1 Gathering of Data

Once the scheduler program was functional and successfully offsetting job's execution time based on forecasts, a method had to be deployed of tracking these delays and submission times, for analysing the performance of the algorithm. For this, a logging strategy was created, this action was completed in the *'deleteJob'* method which is the last part of the scheduler which interacts with a job. Here a text entry is appended to a logs file, including all the details for the executed job, this includes the submission time, and the execution time appended to the end of this entry, such that this can be tracked.

This logging text file is located on the cloud and can be easily reset by completing a write command instead of an append command in Python. This log was processed into an Excel sheet which represented all relevant information about the lifetime of a job on the scheduler, used for calculating the carbon intensity savings. The reason these calculations couldn't be implemented into the program and then appended to the logs file, is that these calculations use the *'actual'* value for the forecasts not the *'forecast'* value for carbon intensity, this is so that the carbon emissions savings are the real savings not just the predicted ones. This of course increases the validity of the data produced, the actual data is not uploaded to the National Grid Carbon Intensity API until at least over an hour has passed, this delay is illustrated in Figure 6.2.

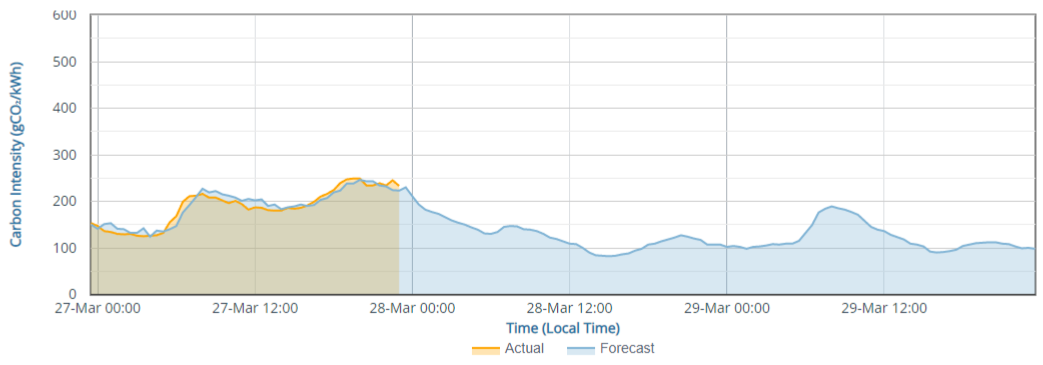


Figure 6.2: *This figure illustrates the carbon intensity forecast on a graph with the actual carbon intensity on the same graph, with a visible delay for times which are in the future or soon to be calculated and processed.*

Any outliers due to unforeseen circumstances, for instance the scheduler being run with jobs from days ago which are past their deadline, thus are run immediately. Exceptions such as these have been removed from datasets used for the actual data analysis and disregarded.

6.2.2 Results

The carbon emission reductions for each job analysed in the experiments are illustrated in the scatter plot in Figure 6.3. The vast majority of jobs saw a reduction in their carbon footprint when compared to the job being executed at submission time. The magnitude of these savings varies, as this is due to the jobs represented in this graph having a variety of runtimes and deadline windows, more of this in Section 6.2.3.

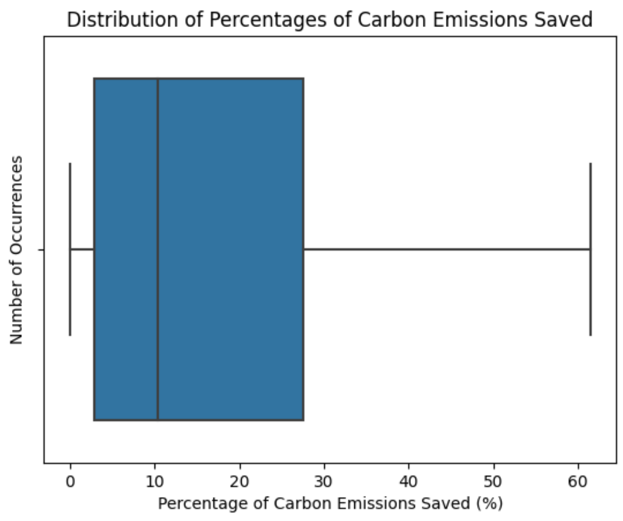


Figure 6.3: *This box plot illustrates the distribution of the carbon emission percentage savings based associated with all the jobs executed during testing.*

To evaluate the implementation's performance in a statistical manner, the savings percentages can be compiled into values which can tell the story.

Firstly, the average carbon footprint reduction percentage was 17.4%, this value is generally what is expected from this kind of workload shifting techniques (Wiesner et al., 2021). By considering the magnitude of this value, it can be inferred as a potential to serve as a realistic estimate of the amount that can be saved through these methods. Any drastically higher value, would question the validity of the experimental methods.

Secondly, the median value was 11.5%, this value disregards outliers which can heavily affect the average value, thus this is generally the expected value for a job submitted to the scheduler with a deadline between 4 and 24 hours and a runtime between 15 and 100 minutes.

Lastly, the standard deviation of these values was 10.124, this indicates there is a deviation present between results, but this is mainly due to forecasts being consistent during a particular day or rising slowly meaning large savings are impossible for the time period, giving low values. But the opposite case can happen, which produces large savings, producing this high standard deviation value.

It is important to understand that the performance of the scheduler is generally at the will of the spread of values in the current carbon intensity forecasts, more namely the presence of significantly lower carbon intensity value periods in the window for the job's execution, for the scheduler to work with or not.

6.2.3 Data Analysis

This subsection will investigate the trends involved between carbon emission savings, and the deadline window or runtimes associated with the jobs. Graphs will be displayed to illustrate the trends which are produced when these variables are varied. This is in order to understand how such conditions affect the performance of the scheduler.

Deadline Window Trends

The deadline window represents the amount of time the scheduler has to complete the job, thus the window here represents the space for optimisation. For the purpose of this project, the hours specified for the jobs were 4-hour, 12-hour and 24-hour windows for the workload shifting to occur.

The box plot in Figure 6.4, represents box plots for each of the deadline windows, respectively, showing the distribution of the carbon emission percentage savings.

When the deadline window is low, the carbon emissions that can be saved generally are low, if present at all, this is due to the small amount of time available to shift the workload in this small amount of time. The carbon intensity value generally does not change as significantly as it can over a longer period of time.

Conversely, for larger deadline windows more substantial carbon emissions savings can be achieved and this is generally the case. This benefit is due to the larger number of forecasts available in the future, which often means better execution windows can be found for the specific job runtime.

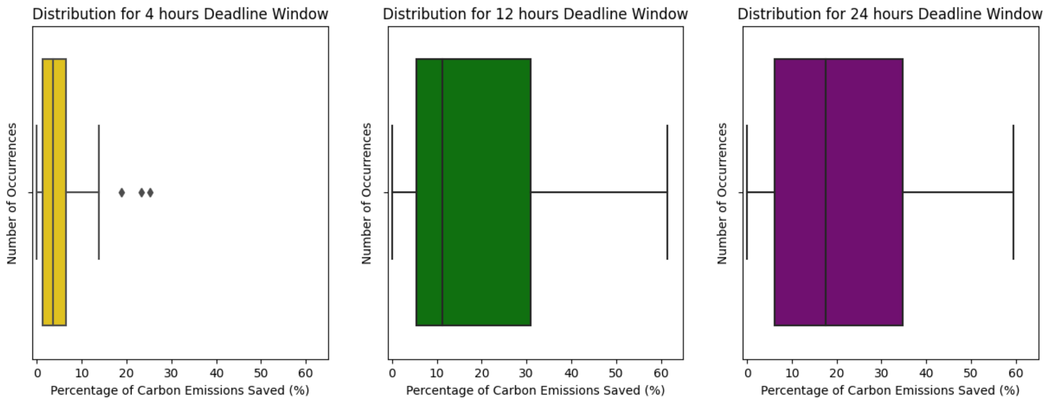


Figure 6.4: *This figure contains three box plots corresponding to each of the deadline windows, Carbon Emission percentage savings. These each display the distribution of values present in the results.*

In summary, the average amount of carbon emissions saved increases with the number of hours allocated for the workload to be shifted in. This trend can be seen in Figure 6.5, which clearly indicates the upward trend in savings when the deadline window is increased. Although it should still be noted that if the job is submitted exactly when the carbon intensity is at its lowest for the next 24 hours it is unlikely that many savings can be made, although this is an uncommon case for larger deadline windows.

For jobs with a 24-hour deadline window, the average savings were 22% and the median 17.4%, respectively. This is a large improvement on the average value for all the jobs mentioned in subsection 6.2.2.

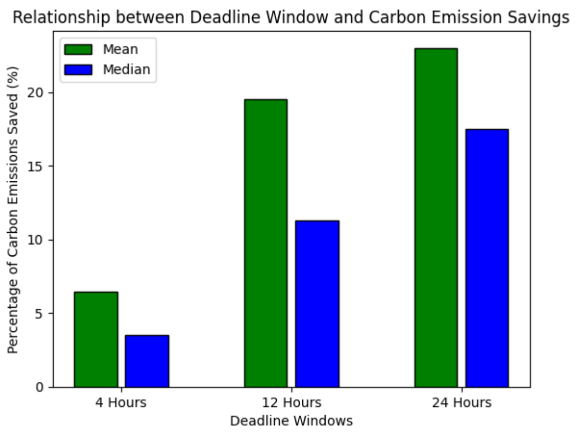


Figure 6.5: This figure is a bar chart which portrays the relationship between deadline windows and Carbon Emission reduction percentages.

Runtimes Trends

Each job submitted has an associated runtime based on previous runtimes on the same cluster or one with the same configuration. As aforementioned, this value has been varied during testing to see if any trends can be identified between the runtime and percentage carbon emission savings. These values will be split into brackets of 15–30 minutes, 31–60 minutes, and over 60 minutes. The reasoning for this composition is that the carbon intensity forecasts are in 30 minute blocks, thus these different brackets will represent jobs which can utilise a different number of blocks when optimising their execution times.

Figure 6.6 below represents a scatter plot for each of these brackets respectively, these contain the carbon emission percentage savings for each of the jobs, each in their corresponding bracket. The data in the tables are generally skewed between 5% and 15% this is where most of the savings percentages belong.

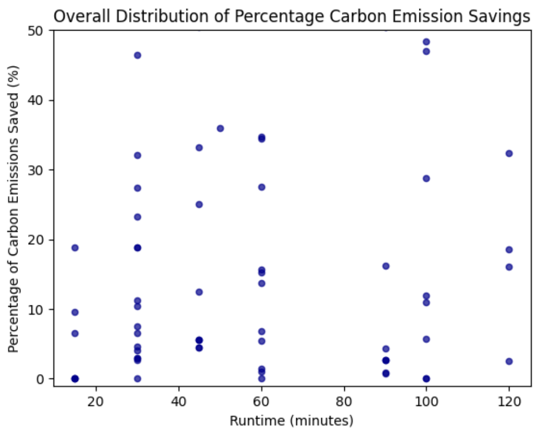


Figure 6.6: This figure is a scatter plot which represents the distribution of Carbon Emission reduction percentages for jobs with different runtimes.

Having access to more blocks for optimisation can be a benefit, but can also become a hindrance, depending on the circumstances. For instance, if the job has a large runtime but a small deadline window to optimise the jobs, there are significantly fewer positions for the scheduler to run the job. This smaller window and longer runtime instance has been observed as a common exception to the larger runtime, larger emissions reductions trend.

In summary, it can be said naively, that the percentage of carbon emissions that are saved by the scheduler marginally increases when the runtime of the jobs increases as is illustrated in Figure 6.7. This trend is dependent on the ratio between the job runtime and the deadline window available, larger jobs tend to require larger deadline windows to be influenced by the aforementioned trend. From this it can also be concluded that if the carbon intensity forecasts blocks were more frequent, for instance they were for every 15 minutes, it could lead to greater carbon emission savings with the scheduler.

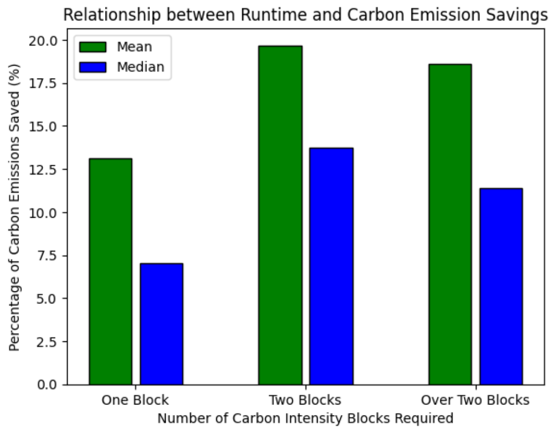


Figure 6.7: *This figure is a bar chart which portrays the relationship between job runtimes and Carbon Emission reduction percentages.*

6.2.4 Optimisation Method

The aim of the optimisation strategy in this implementation was to reliably offset job execution times based on carbon intensity forecast data, with the rationale being to reduce the overall carbon footprint of a job. This was done by deconstructing the forecasts into minute blocks and finding a minimum value when summing each minute required (this value is equal to the runtime of the job) to finish the job. This was done over the entire deadline window to find the slot with the lowest carbon emissions for the job to be executed within.

From the results produced in Section 6.2.2, it can be concluded that the scheduling algorithm successfully does what it is designed to do. While still being aware that this algorithm can be improved, for example the scheduler doesn't take into account the percentage errors associated with the forecasts, these could be used to make decisions to wait longer to see improved forecasts before making final decisions for execution times.

6.2.5 Code Coverage

Code coverage is a software testing strategy which measures the percentage of code in a script which has been executed during testing. This metric can give an idea of the quality of the source code to ensure it is functioning as it was designed to do. The test keeps track of each line of code executed and traces each branch taken between methods and which functions are called.

These strategies are employed by Google, who use coverage analytics on millions of lines of code a day. When implemented in realistic test suites that simulate real-world conditions, the metric provides a reliable indication of the quality of a program, but it must be acknowledged that if implemented incorrectly with unrealistic test suites the metric is less valuable (Ivanković et al., 2019). This is generally when the score reaches 100% for a coverage in a test, it would be likely that the test has been designed in a certain way to manipulate the coverage statistics. Generally, a score of around 80% shows a great quality of code, anything above is a further improvement, while conscious of the previous statement.

Program	Coverage
Job Submission Tool	89%
Scheduler	91%

Table 6.1: *Table showing the code coverage scores for the Job Submission Tool and Scheduler in percentages*

Most of the lines indicated as missing, i.e. not ran at any point during the test, were crash prevention methods which are designed to catch errors like incorrect data formats or connections failure. These methods print out a readable statement explaining what has gone wrong rather than the traditional error message. There is research that indicates that code coverage has a correlation with the number of faults within an implementation, indicating that code coverage can serve as an indicator of the presence of faults in source code (Hemmati, 2015).

6.3 Discussion of Technologies Used

This section will evaluate the technologies chosen throughout the project, each technology will be analysed, giving an explanation of the decision to use it and in what manner it has affected the implementation. The functional and non-functional requirements will be referenced to portray how the technology ensures the implementation meets its specification.

6.3.1 Google Cloud Platform

Google Cloud Platform (GCP) was the chosen cloud provider for the project, it offers many services for data storage and analytics on clusters which can be configured in virtually any hardware configuration. One of the main reasons for selection, was the Extensive Partner Ecosystem offered by the platform, many third-party vendors offer services which integrate with GCP, this made it easier to manage the running of multiple components simultaneously on the cluster. The GCP also offered $300 of free credits to use on their platform, which was vital for doing a significant amount of experiments while still being able to test early prototypes.

6.3.2 Apache Spark and PySpark

Apache Spark as a framework for the scheduling of big data jobs, interaction with Apache Spark has been intuitive and streamlined throughout the project. The simple commands in the API, made it easy to schedule jobs on a cluster in real-time. The fault-tolerant attribute of the framework has ensured the experimenting process has run smoothly, and fortunately if a node fails, Apache Spark automatically recomputes the information on another node without losing any data. This means experiments are not wasted due to small failures like this, resulting in less credits being wasted and more importantly course unnecessary carbon emissions.

For developers with no experience in Apache Spark or distributed systems, working with PySpark means an intuitive experience which allows for understanding of how big data processing is carried out in this environment. Thanks to PySpark the language used for the scheduler program was Python, this high level language allowed for the creation of effective solutions without many lines of code. Also, the number of libraries and modules available made many interactions with components and datasets like the National Grid Carbon Intensity API seamless. This highly flexible language can also be executed on a variety of hardware, operating systems and environments, which was ideal for testing in different configurations on the cluster.

6.3.3 National Grid Carbon Intensity API

The API offers predictions for the next 96 or more hours for the carbon intensity of the electricity grid in the UK, these predictions are updated hourly and improve the closer the current time gets to the predictions time. This was required to meet the functional requirement of making workload shifting decisions based on carbon intensity forecasts. There was also a non-functional requirement related, that the system should be reliable and stable. Overall, the API has been effective at doing its job of providing the predictions in a timely and reliable manner.

Although there are occasions where the API has not been updated correctly, the entries incorrectly filled with NaN (not a number) values, which of course leads to a fatal crash in the system. This has only occurred once in the duration of the project, but did last multiple days. When the national carbon intensity dataset is misbehaving like this, an alternative solution is to use the country-specific dataset and selecting England from it, as this is where the data centre is located. However, since this dataset does not provide the actual values once the time has passed, it can only be used for scheduling but not for the evaluation of performance, which uses the actual carbon intensity values for savings calculations.

6.3.4 SQLite

SQLite was selected as it is a lightweight, easy to use, and relational database framework which can be effortlessly implemented into a Python program. For the purpose of this project, SQLite has done exactly what the framework was required to do. Which was creating a relational database which communicates between the Job Submission Tool and the scheduler program, while also being stable and reliable throughout testing.

In retrospect, SQLite is not the most appropriate choice for a scheduling algorithm for several reasons. Lack of scalability, this is because SQLite is only designed for small applications which do not work with large volumes of data or complex operations. Of course, this project does work with big data, but this component is separate from the job database, this would only become an issue on a large-scale infrastructure with vast amounts of job submissions. Also, the lack of concurrency in SQLite means that write operations cannot be completed concurrently, as a result of this, if the scheduler was being run on multiple clusters or virtual machines it could lead to miscommunications.

6.4 Summary

In summary, this chapter has investigated how the performance of the scheduler has been measured and what implications have been drawn from the data gathered. Firstly, the experimental setup was outlined to give an understanding of the methods used to generate the results. Then this data was analysed in order to tell the story of how the scheduler performs based on different constraints and conditions. The methods used for making the optimisation decisions were also evaluated, based on what was deduced in the preceding data analysis sections. The overall quality of the implementation was also visited and analysed using the code coverage metric. The decisions made to use certain technologies were also discussed in this chapter, giving an insight on how they satisfy the requirements of the project. Throughout this chapter, there was a consciousness of any hindrances present in the implementation, and these were acknowledged via critical analysis. Additionally, technologies that were deemed inappropriate have been identified and elucidated upon.

7 Conclusion

This dissertation presented an approach to creating a carbon aware big data job scheduler, which aims to reduce the carbon footprint of big data jobs by delaying their execution time based on carbon intensity forecasts. These delays are updated in real-time based on more accurate forecasts being available over time. Once the jobs reach their optimal execution time, they are then submitted to a cluster for processing, while the scheduler continues to schedule any remaining jobs.

The design of the system was presented, which utilises many components interacting with each other with the common goal of reducing the carbon footprint of a big data job. The design outlined how calculations will be made to determine the optimal delays for a job based on carbon intensity forecasts in the deadline window. The scheduling system allowed jobs to be submitted when the scheduler was already running with jobs already present in the queue. For each job submitted, an entry was appended to the local database and the job table was displayed in the user interface, this was updated periodically every minute. Another desired implication of this local database, is that in the unlikely case of the scheduler having a failure, the program can be restarted without any data or progress being lost. The design has also allowed for the system to be highly agile and deployable in a variety of environments, and can be easily reconfigured to work with different cloud providers than Google Cloud Platform (GCP), with the change of one command.

The implementation collaborated the ideas presented in the design with the cohesion of big data technologies, such as Apache Spark and a cloud provider, namely GCP. The interactions between the scheduler and the other components were explained here, the communication channels formulated here had reliability and stability in mind when being developed. Because of this, a stable scheduler was created which can maintain an optimised job queue for the cloud provided cluster, while also being able to actively and safely respond to updates from the carbon intensity forecast API and job database. Throughout the implementation process, good software engineering practising were followed to ensure constant progress and effective version control.

The solution was integrated on a GCP Dataproc cluster for testing, in order to quantify the performance of the scheduler. The scheduler successfully reduced the carbon footprints of the vast majority of jobs submitted and would never increase this footprint. The relationship between deadline windows and runtimes with the overall carbon footprint reductions has also been explored, with trends identified between the factors and the overall carbon emission savings. Overall, the median value for carbon emission percentage reductions for a job with a 24-hour runtime was 17.4%, this gives an indication of how much the carbon footprint of a job should be reduced by the scheduler.

7.1 Future Work

Throughout this dissertation, there has been a consciousness to the limitations of this implementation, and the assumptions required to make it possible. Many of these can be adjusted or implemented to open the door to more research findings. Good candidates for this will be briefly explained in this section.

The first improvement that could have been made was long term and more intensive testing. This could produce more insightful results on how the time of the year affects the performance of the algorithm and draw more powerful trends of the relationships between the constraints and the data.

The efficiency of the hardware being used could also be considered in future work. The factors involved in this are vast, such as the amount of power consumed by the cooling systems in the data centre or power inefficiency of hardware (Horri et al., 2014). These calculations would produce interesting results which could be used to improve the algorithm to further reduce carbon emissions.

Perhaps the most interesting work that could be done would be considering big data jobs execution profiles when making calculations for carbon aware scheduling. The use of workload execution traces gives an idea of when the job is using the most resources (i.e. memory or CPU utilisation) (Scheinert et al., 2021; Will et al., 2022). Therefore, when the job is using the most electricity. If this factor was taken into account when looking for the optimal timeslot to offset the job to, possibly with a weighting value for different times in the jobs execution trace. It could lead to improved carbon footprint reductions for delays placed on jobs which have distinct, substantial periods where they are using higher power when compared to another time in the job's execution trace.

7.2 Personal Reflections

This Level 4 Individual Project has surpassed any challenge I have undertaken during my undergraduate degree while at the University of Glasgow. When I started this project I had no experience or prior knowledge at all about distributed systems, carbon intensity or even big data jobs. I have embraced each hurdle which I have come across during this year, beginning with the understanding of what the functionalities of a job scheduler is and what a 'cloud infrastructure' actually constitutes. These challenges have improved my ability to use technologies which I am unfamiliar with and improve my understanding of frameworks in computer systems as a whole. I have also begun to understand architecture of systems and how important the communication channel methods are between components. Finally, being able to implement a carbon aware scheduler that can be deployed on a cluster, which successfully reduces the carbon footprint of jobs submitted, has given me a sense of pride, I am exceedingly proud of the work which I have completed. This was also the first dissertation I have written, this style of writing is extremely different to the writing I have endeavoured in before. So it has been an interesting challenge writing in a research style manner, a challenge I have enjoyed thoroughly. I have no doubt that the skills and knowledge I have acquired during this project, will be an invaluable aid to my professional career.

A | Appendices

A.1 Public Repository

The source code has been made available in a public repository, so that any reader of this dissertation or researcher interested in the field was interested. The full system is available with a readme file which give instructions of how to start the Job Submission Tool and subsequently start the scheduler.

Repository Link Here

A.2 Further Data Analysis

In the figures below, there are the distributions of the percentage errors, for different deadline windows produced during the experiments.

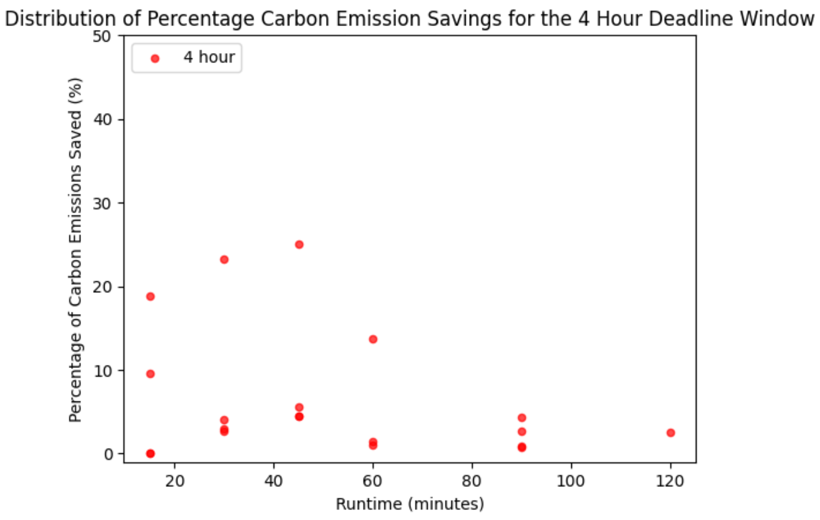


Figure A.1: This figure is a scatter plot which represents the distribution of Carbon Emission reduction percentages for jobs submitted with a 4-hour deadline window.

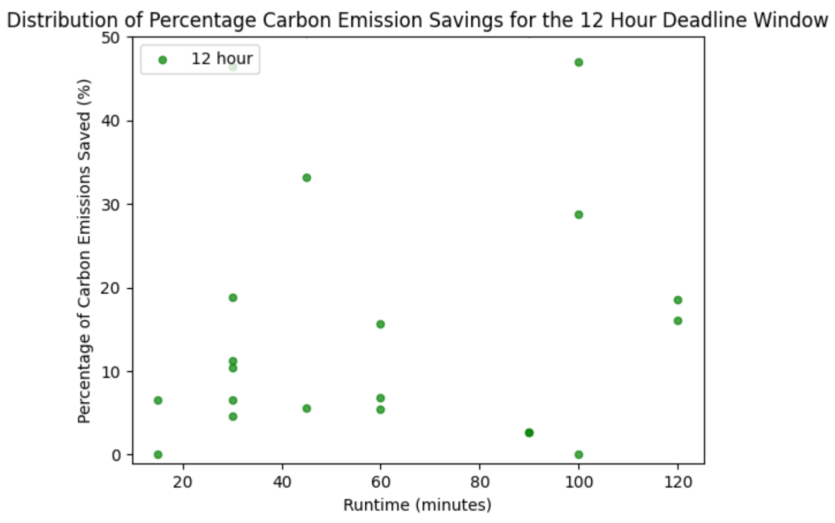


Figure A.2: *This figure is a scatter plot which represents the distribution of Carbon Emission reduction percentages for jobs submitted with a 12-hour deadline window.*

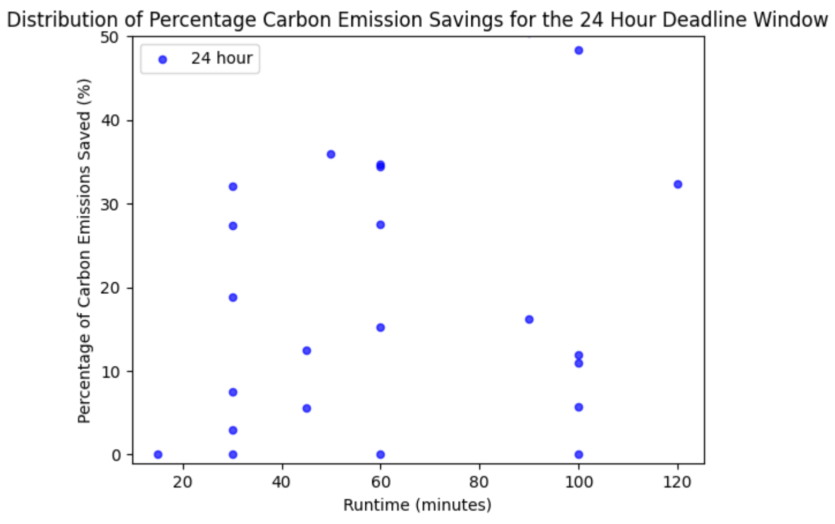


Figure A.3: *This figure is a scatter plot which represents the distribution of Carbon Emission reduction percentages for jobs submitted with a 24-hour deadline window.*

7 Bibliography

Abrahamsson, P., Salo, O., Ronkainen, J. and Warsta, J. (2017), 'Agile software development methods: Review and analysis.', *arXiv preprint arXiv:1709.08439* .
URL: *https://doi.org/10.48550/arXiv.1709.08439*

Al-Sayeh, H. and Sattler, K.-U. (2019), Gray box modeling methodology for runtime prediction of apache spark jobs., *in* '2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)', pp. 117–124.
URL: *https://doi.org/10.1109/ICDEW.2019.00-23*

Cheng, K.-W., Bian, Y., Shi, Y. and Chen, Y. (2022), Carbon-aware ev charging., *in* '2022 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)', pp. 186–192.
URL: *https://doi.org/10.1109/SmartGridComm52983.2022.9960988*

Cloud, G. (2023), 'Regions and zones | compute engine documentation.', `https://cloud.google.com/compute/docs/regions-zones`.

Corradi, O. (2022), 'Marginal vs average: Which one to use in practice?'.
URL: *https://www.electricitymaps.com/blog/marginal-vs-average-real-time-decision-making*

Datopian (2023), 'National carbon intensity forecast.'.
URL: *https://data.nationalgrideso.com/carbon-intensity1/national-carbon-intensity-forecast/r/national%5Fcarbon%5Fintensity%5Fforecast*

ESO, N. G. (2023), 'Carbon intensity api.', `https://carbonintensity.org.uk/`.

Ganesan, M., Kor, A.-L., Pattinson, C. and Rondeau, E. (2020), 'Green cloud software engineering for big data processing.', *Sustainability* **12**(21).
URL: *https://www.mdpi.com/2071-1050/12/21/9255*

Hemmati, H. (2015), How effective are code coverage criteria?, *in* '2015 IEEE International Conference on Software Quality, Reliability and Security', pp. 151–156.
URL: *https://doi.org/10.1109/QRS.2015.30*

Horner, N. and Azevedo, I. (2016), 'Power usage effectiveness in data centers: overloaded and underachieving.', *The Electricity Journal* **29**(4), 61–69.
URL: *https://www.sciencedirect.com/science/article/pii/S1040619016300446*

Horri, A., Mozafari, M. S. and Dastghaibyfard, G. (2014), 'Novel resource allocation algorithms to performance and energy efficiency in cloud computing.', *The Journal of Supercomputing* **69**, 1445–1461.

Hölzle, U. (2022), 'Google achieves four consecutive years of 100% renewable energy.', `https://cloud.google.com/blog/topics/sustainability/google-achieves-four-consecutive-years-of-100-percent-renewable-energy`.

Islam, M. T., Srirama, S. N., Karunasekera, S. and Buyya, R. (2020), 'Cost-efficient dynamic scheduling of big data applications in apache spark on cloud.', *Journal of Systems and Software* **162**, 110515.
URL: *https://www.sciencedirect.com/science/article/pii/S0164121219302894*

Ivanković, M., Petrović, G., Just, R. and Fraser, G. (2019), Code coverage at google., *in* 'Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering', ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, p. 955–963.
URL: *https://doi.org/10.1145/3338906.3340459*

Kaewkasi, C. and Srisuruk, W. (2014), A study of big data processing constraints on a low-power hadoop cluster., *in* '2014 International Computer Science and Engineering Conference (ICSEC)', pp. 267–272.
URL: *https://doi.org/10.1109/ICSEC.2014.6978206*

Khan, I., Jack, M. W. and Stephenson, J. (2018), 'Analysis of greenhouse gas emissions in electricity systems using time-varying carbon intensity.', *Journal of Cleaner Production* **184**, 1091–1101.
URL: *https://www.sciencedirect.com/science/article/pii/S0959652618306474*

Ma, X., Jiang, P. and Jiang, Q. (2020), 'Research and application of association rule algorithm and an optimized grey model in carbon emissions forecasting.', *Technological Forecasting and Social Change* **158**, 120159.
URL: *https://www.sciencedirect.com/science/article/pii/S0040162520309859*

MacMillan, F. (2021), 'Carbon intensity of balancing actions.'.
URL: *https://data.nationalgrideso.com/carbon-intensity1/carbon-intensity-of-balancing-actions*

Masanet, E., Shehabi, A., Lei, N., Smith, S. and Koomey, J. (2020), 'Recalibrating global data center energy-use estimates.', *Science* **367**(6481), 984–986.
URL: *https://www.science.org/doi/abs/10.1126/science.aba3758*

Nnk (2020), 'Spark history server to monitor applications.'.
URL: *https://sparkbyexamples.com/spark/spark-history-server-to-monitor-applications/*

Radovanović, A., Koningstein, R., Schneider, I., Chen, B., Duarte, A., Roy, B., Xiao, D., Haridasan, M., Hung, P., Care, N., Talukdar, S., Mullen, E., Smith, K., Cottman, M. and Cirne, W. (2023), 'Carbon-aware computing for datacenters.', *IEEE Transactions on Power Systems* **38**(2), 1270–1280.

Revathy, S. and Priya, S. S. (2020), Blockchain based producer-consumer model for farmers., *in* '2020 4th International Conference on Computer, Communication and Signal Processing (ICCCSP)', pp. 1–5.
URL: *https://doi.org/10.1109/ICCCSP49186.2020.9315214*

Scheinert, D., Alamgiralem, A., Bader, J., Will, J., Wittkopp, T. and Thamsen, L. (2021), On the potential of execution traces for batch processing workload optimization in public clouds., *in* '2021 IEEE International Conference on Big Data (Big Data)', IEEE, pp. 3113–3118.
URL: *https://doi.org/10.1109/BigData52589.2021.9671275*

Spark, A. (2023), 'Submitting applications.', `https://spark.apache.org/docs/latest/submitting-applications.html`.

Trihinas, D., Thamsen, L., Beilharz, J. and Symeonides, M. (2022), Towards energy consumption and carbon footprint testing for ai-driven iot services., *in* '2022 IEEE International Conference on Cloud Engineering (IC2E)', pp. 29–35.
URL: *https://doi.org/10.1109/IC2E55432.2022.00011*

Venkataraman, A. and Jagadeesha, K. K. (2015), 'Evaluation of inter-process communication mechanisms.', *Architecture* **86**, 64.

Wiesner, P., Behnke, I., Scheinert, D., Gontarska, K. and Thamsen, L. (2021), Let's wait awhile: How temporal workload shifting can reduce carbon emissions in the cloud., *in* 'Proceedings of the 22nd International Middleware Conference', Middleware '21, Association for Computing Machinery, New York, NY, USA, p. 260–272.
URL: *https://doi.org/10.1145/3464298.3493399*

Will, J., Thamsen, L., Bader, J., Scheinert, D. and Kao, O. (2022), Ruya: Memory-aware iterative optimization of cluster configurations for big data processing., *in* '2022 IEEE International Conference on Big Data (Big Data)', IEEE.
URL: *https://doi.org/10.1109%2Fbigdata55660.2022.10020295*

Zhao, D. and Zhou, J. (2022), 'An energy and carbon-aware algorithm for renewable energy usage maximization in distributed cloud data centers.', *Journal of Parallel and Distributed Computing* **165**, 156–166.
URL: *https://www.sciencedirect.com/science/article/pii/S074373152200079X*