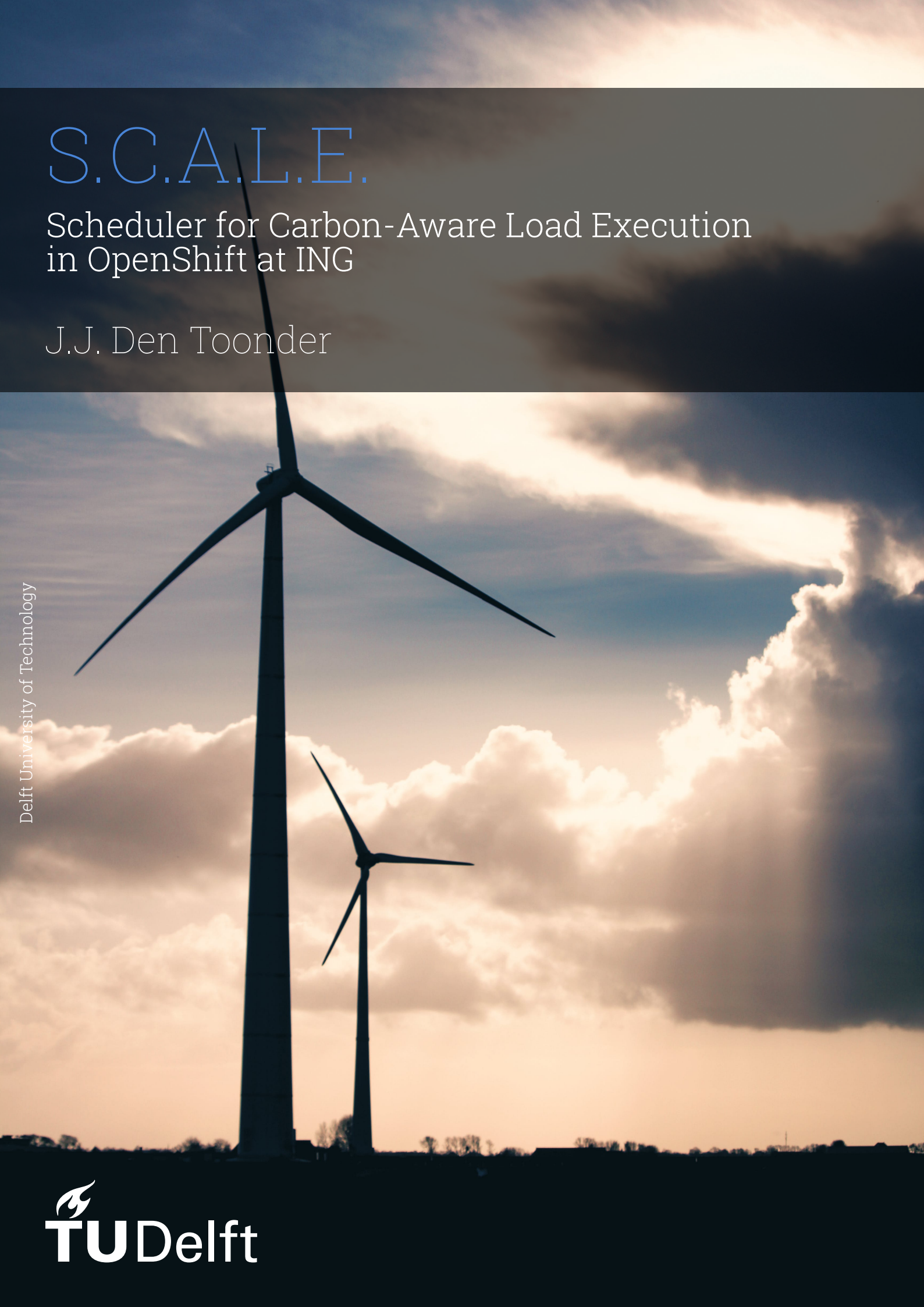
S.C.A.L.E.

Scheduler for Carbon-Aware Load Execution in OpenShift at ING

J.J. Den Toonder



Delft University of Technology

TU Delft

S.C.A.L.E.

Scheduler for Carbon-Aware Load Execution in OpenShift at ING

by

J.J. Den Toonder

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday, December 22, 2023, at 10:00 AM.

Student number	4431324	
Project duration	April 2023 - December 2023	
Thesis committee:	Prof.dr. A. van Deursen	TU Delft, committee chair
	Dr. T. Durieux	TU Delft, committee member
	Dr. M.A. Migut	TU Delft, committee member
Company supervisor:	P.G.P. Braakman	ING

An electronic version of this thesis is available at https://repository.tudelft.nl
Accompanying code for this thesis is available at https://github.com/Fastjur/S.C.A.L.E.

Cover: Silhouette Photo of Two Wind Mills during Golden Hour by Tom Swinnen

TU Delft

Preface

This thesis was written as part of the Software Engineering Research Group (SERG) at Delft University of Technology in combination with the squad trade and communications surveillance (TRCS) at ING. Throughout my research, especially at the beginning, I have often asked myself why there is so little focus on carbon-aware scheduling within the industry, even though the concept is theoretically quite simple. As I progressed with the work, I learned that while the concept may be simple, it does indeed pose quite a challenge to implement this, especially at such a complex multinational corporation like ING. I have learned many valuable skills that will apply to my work in the future, by implementing this scheduler at ING. For this, I am thankful.

Even though a thesis is a personal work, I was not alone during this period of researching and writing. Many people have helped me with this thesis, and I would like to extend my gratitude to them here.

I would like to begin by extending special thanks to my daily supervisor, Thomas. I can state, without a shred of doubt, that I would not have been able to do this without your help. Many times, I have felt wholly stuck as challenges arose, and many times, this was solved in our weekly meeting. In fact, usually, I felt invigorated again after we had discussed potential solutions to the problems in my work. I appreciate your help in all of this.

Additionally, I would like to thank Paul, my company supervisor within ING. Paul, thank you for the opportunity to create this impactful work within your team. You were always open to questions and aid and provided everything I needed to complete this work within ING. Thank you for the opportunity.

Lastly, I would like to thank the many friends who have aided me by reading parts of my thesis. Many parts of my thesis would probably be illegible without them, as writing has never been my strong point. Thank you all for your feedback.

Lastly, I would like to thank you, the reader, for taking the team to read this thesis report.

J.J. Den Toonder
Delft, December 2023

Summary

The global climate change crisis and the associated phenomenon of global warming have taken center stage in recent years. Greenhouse gas emissions due to electricity generation are a contributor to this problem. Internet Services running in data centers consume enormous amounts of energy and should be optimized to reduce their greenhouse gas emissions.

This thesis explores the possibility of intelligently scheduling resource-intensive batch data-processing jobs to green energy generation hours in the day. Green hours are hours within the day during which the amount of greenhouse gasses emitted per kilowatt-hour (kWh) is lower compared to other hours of the day. There is a variance in the amount of emissions due to the variability of renewable energy generation and grid demand.

The system "S.C.A.L.E. (Scheduler for Carbon-Aware Load Execution)" is proposed. It schedules compute jobs to periods of low-carbon-intense energy generation based on predictions of renewable energy generation and grid demand. The system was evaluated against a simulated data processing pipeline at ING; this pipeline is one of the larger consumers of the ING private cloud. The scheduler aims to reduce greenhouse gas emissions by intelligently predicting task running times and green hours for the next day and optimizing the times at which tasks are processed throughout the day.

Several main conclusions are drawn based on this research:

1. The accuracy of task load predictions regarding running times is crucial for effective scheduling. The research concludes that, with sufficient historical data, the scheduler can predict task running times with an acceptable margin of error (5-10%).
2. The research explores the scheduler's ability to predict periods of low carbon intensity and the resulting reduction in carbon emissions by implementing it. The research affirms the scheduler's accuracy in determining low-carbon-intensive energy generation periods and estimates a potential 20% reduction in greenhouse gas emissions.
3. The potential overhead introduced by implementing a carbon-aware scheduler is addressed. The research identifies that while the scheduling algorithm itself is lightweight, the concurrent processing of tasks introduces overhead. The tipping point, where the overhead outweighs the benefits, varies for each system and should be experimentally determined.

The thesis concludes by emphasizing the significance of implementing a carbon-aware scheduler to reduce the environmental impact of data centers. The proposed scheduler is a promising contribution to sustainable computing practices. Further, the research suggests the need for continued work and adoption of the scheduler into production environments, especially within the context of the ING data processing pipeline.

Contents

List of Figures

List of Tables

Acronyms

Symbols

gCO$_2$-eq grams of CO$_2$ equivalent emissions vi, viii, 21, 51–56, 60–62, *See glossary:* grams of CO$_2$ equivalent emissions
gCO$_2$-eq/kWh grams of CO$_2$ equivalent emissions per kWh iv, v, viii, 40, 41, 47–51, 53, 66, 70, *See glossary:* grams of CO$_2$ equivalent emissions per kWh

A

API application programming interface xiv
AWS Amazon Web Services xiii, 64

B

B/s bytes per second 31, 32, 45, 46
BB Bloomberg v, xiv, 6, 29, 38, 39

C

CRC CodeReady Containers *See glossary:* CodeReady Containers

E

ENTSO-E the European Network of Transmission System Operators for Electricity 16, 47, 49, 66, *See glossary:* the European Network of Transmission System Operators for Electricity

F

FERC Federal Energy Regulatory Commission xiii

G

GHz gigahertz *See glossary:* gigahertz

I

ICHP ING container hosting platform 8, 12, 64, 68, 70, *See glossary:* ING container hosting platform
IV Intelligent Voice *See glossary:* Intelligent Voice

K

KiB kibibyte 46, *See glossary:* kibibyte
KiB/s kibibyte per second vi, 76, 78
KPI Key Performance Indicator 10, *See glossary:* Key Performance Indicator
kWh kilowatt-hour ii, iv–vi, viii, ix, xii, 1, 2, 21, 35, 40, 41, 46–51, 53–56, 59, 64, 66, 68, 70

L

LCA Life-Cycle Analysis 40, 66, *See glossary:* Life-Cycle Analysis

M

MB megabyte 67, *See glossary:* megabyte
MISO Midcontinent Independent System Operator 14, *See glossary:* Midcontinent Independent System Operator
MS Microsoft xiv, 6, 29
MWh megawatt-hour 2

N

NLTK Natural Language Toolkit 38, 46, *See glossary:* Natural Language Toolkit

O

OLS ordinary least squares 47

R

RTO Regional Transmission Organization xii, *See glossary:* Regional Transmission Organization

S

S3 Simple Storage Service xiii

T

TRCS trade and communications surveillance i, iii, v, xii, 4, 6–9, 11, 12, 24, 34, 36, 39, 70
TSO transmission system operator xiv
TSV tab-separated values 37, 38
TWh terawatt-hour 1

V

VCC virtual capacity curves 14, 64, *See glossary:* virtual capacity curves
vCPU virtual centralized processing unit *See glossary:* virtual centralized processing unit
VM virtual machine xiv

Glossary

Symbols

R^2 The R-squared value tells us how well the independent variable(s) can explain or predict the dependent variable, with higher values meaning a better fit and stronger relationship between the variables. 2

CO_2 Chemical formula for carbon dioxide, a colourless and odourless gas consisting of one carbon atom and two oxygen atoms. It is naturally present in the Earth's atmosphere and contributes to climate change as a greenhouse gas iv–vi, viii, ix, xii, 1, 2, 4, 13, 21, 40, 41, 49, 51, 61, 64–66, 70

`Python` A high-level, interpreted programming language known for its readability and versatility. Developed by Guido van Rossum, Python features a clean syntax, dynamic typing, and an extensive standard library. Widely used for applications ranging from web development to scientific computing.[1] viii, xiii, 43, 46, 47

`TaskQueueExecutor` A `TaskQueueExecutor` is responsible for finding `TaskQueues` with files to be processed if the criteria for processing those files are met. A detailed explanation is given in Sections 4.3 and 5.1.5. 32, 41, 43, *see* `TaskQueue`

`TaskQueue` A `TaskQueue` contains tasks to be processed by the system at a certain time and in a certain order. A detailed explanation is given in Sections 4.3 and 5.1.5. xi, 32, 41, 43

C

CodeReady Containers CodeReady Containers by Red Hat is a lightweight, single-node development environment that enables developers to quickly set up and run a local OpenShift cluster for application development and testing purposes. ix

cron job A cron[2] job is a command or program that runs at a specified time or period. It's used for running scripts and commands at regular intervals, and at specific times and dates. 60, 63

D

derived file A file that is created by one of the processing steps in the data processing pipeline. A derived file is never a source file, as a derived file is always created by the pipeline itself during processing viii, xiii, 9, 24, 27, 29, 31, 32, 50, *See glossary:* source file

Django A free and open-source web framework written in Python, Django follows the model-template-views (MTV) architectural pattern. It is designed to facilitate rapid development of secure and maintainable websites 9, 11, 12, 34, 36, 38

docker A platform for developing, shipping and running applications in a containerised environment 40

E

[1] https://www.python.org/about/

[2] https://en.wikipedia.org/wiki/Cron

Electricity Maps Electricity Maps[3] is an online platform that provides electricity data for more than 160 regions [15]. It provides real-time information on the CO_2 emissions associated with electricity consumption. iv, viii, xii, 40, 41, 46, 47, 49, 66, 70

G

gigahertz A gigahertz is a unit of frequency equal to one billion hertz. It is commonly used to measure computer processing speed, alternating current, and electromagnetic (EM) frequencies[4]. When used in terms of computer processing speed, it is the measure of the processor's clock rate, which is the rate at which it generates pulses to synchronize all the operations of the computer. ix

grams of CO_2 equivalent emissions A measure of the amount of greenhouse gas emissions. Every type of greenhouse gas can be converted to its CO_2 equivalent in terms of global warming impact over 100 years.[5] ix, 21, 51, 61

grams of CO_2 equivalent emissions per kWh A measure of the amount of greenhouse gas emissions emitted per generated kWh. ix, 40, 66, 70, *see* grams of CO_2 equivalent emissions

I

ING container hosting platform An enterprise solution, based on OpenShift, that provides a platform for hosting containers. Developed by ING to make it more security hardened ix, 8, 64, 68, 70

Intelligent Voice A third-party application used by the business side of the trade and communications surveillance (TRCS) team ix

K

Key Performance Indicator A key performance indicator (KPI) is a high-level measure of system output, traffic or other usage, simplified for gathering and review on a weekly, monthly or quarterly basis.[6] Typical examples are bandwidth availability, transactions per second and calls per user ix, 10

kibibyte A kibibyte is 1,024 bytes. ix, 46

L

Life-Cycle Analysis Life-Cycle Analysis (LCA) is an assessment method by Electricity Maps that considers the environmental impact of a product or process from its initial resource extraction through manufacturing, operation, and eventual disposal [7]. In the context of carbon intensity factors, LCA evaluates emissions throughout the entire life cycle of a power plant, accounting for factors such as raw material extraction, fuel production, manufacturing, operation, and decommissioning. x, 40, 66

M

megabyte A megabyte is 1,000 bytes x, 67

Midcontinent Independent System Operator MISO is an Regional Transmission Organization (RTO) that operates the transmission grid and manages wholesale electricity markets in the central United States[8]. It is one of the largest RTOs in North America

[3] https://www.electricitymaps.com/
[4] https://www.analog.com/en/design-center/glossary/ghz.html
[5] https://www.electricitymaps.com/methodology
[6] https://www.gartner.com/en/information-technology/glossary/kpi-key-performance-indicator
[7] https://www.electricitymaps.com/methodology
[8] https://en.wikipedia.org/wiki/Midcontinent_Independent_System_Operator

and serves a diverse group of stakeholders, including utilities, power generators, and customers. x, 14

N

Natural Language Toolkit The Natural Language Toolkit[9], is a `Python` library that is used for working with human language data. It offers resources for text processing, such as tokenization and parsing, and it also provides access to over 50 corpora and lexical resources. x, 38

O

OpenShift A container application platform built on top of Kubernetes that provides additional features and functionality for building, deploying, and managing containerised applications. It includes integrated build and deployment pipelines, developer workflows, and security and compliance controls, and can be deployed on-premise or in the cloud iv, xii, 8, 11, 12, 34, 35, 40, 64, 83–85

P

pickle file The pickle module[10] implements binary protocols for serializing and de-serializing a Python object structure. *"Pickling"* is the process whereby a Python object hierarchy is converted into a byte stream, and *"unpickling"* is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy vi, 37, 75, 76, 78, 79

R

Regional Transmission Organization An RTO is an organisation formed with the approval of the Federal Energy Regulatory Commission (FERC)[11]. An RTO does not generate electricity, but rather manages the flow of electricity over the transmission lines. RTOs are responsible for ensuring the reliable and efficient operation of the transmission system, which includes coordinating and dispatching power generation resources, maintaining grid stability, and overseeing transmission planning and expansion x, xii

Relativity Third party eDiscovery and compliance software platform that provides organisations with the tools to manage large volumes of data, conduct investigations and ensure regulatory compliance v, 6, 7, 9, 24, 38

S

S3 bucket An Simple Storage Service (S3) bucket is a container for objects stored in S3, a service offered by Amazon Web Services (AWS)[12] or other 'S3 compatible' storage providers. It is an open distributed storage resource that is similar to a file folder, storing objects that consist of data and its descriptive metadata. 8, 12, 29, 43, *see* S3 & AWS

source file A file that arrives from one of the source systems. Source systems generally send these files at a regular interval. A source file can never be a derived file viii, xi, xiv, 9, 24, 27, 29, 31, 32, 50, *See glossary:* derived file & source system

[9] `https://www.nltk.org/`

[10] `https://docs.python.org/3/library/pickle.html`

[11] `https://en.wikipedia.org/wiki/Regional_transmission_organization_(North_America)`

[12] `https://aws.amazon.com/s3/`

source system A source system is one of the systems creating input data (source files) for the data processing pipeline. Examples of source systems are Microsoft (MS) Teams voice and Bloomberg (BB) chat. A detailed explanation can be found in Chapter 2 and Figure 2.1 xiii, 6, 9, 11, 12, 27, 29

T

the European Network of Transmission System Operators for Electricity The European Network of Transmission System Operators for Electricity is the association for the cooperation of the European transmission system operator (TSO). ENTSO-E manages a 'transparency platform', which provides application programming interface (API) access to electricity generation, transportation, and consumption information for the pan-European market. ix, 16, 47, 66

V

virtual capacity curves Hourly resource usage limits that serve to shape each cluster resource and power usage profile over the following day [19] x, 14, 64

virtual centralized processing unit A unit of processing power that is allocated to a virtual machine (VM) in a cloud computing environment. A vCPU is a portion of a physical CPU that is assigned to a VM and acts as if it were a separate physical CPU. The number of vCPUs assigned to a VM determines how much processing power the VM has available to run applications and perform tasks x

1 Introduction

The global climate change crisis and the associated phenomenon of global warming have taken center stage in recent years as an existential challenge facing humanity. The consequences of global warming are already noticeable. For example, recent studies show that it has a negative impact on crop production, threatening food security worldwide [14, 20].

Global warming has been significantly influenced by increased greenhouse gas emissions caused by human activities [29]. This includes the carbon emissions generated by Internet services, which account for a substantial portion of the overall emissions and are forecasted to only increase over time [13, 7]. These services operate primarily through large data centers, making data centers a considerable contributor to carbon emissions and global warming.

A study conducted in 2010 revealed that data centers already emitted as much CO_2 as the entire country of Argentina due to their high energy consumption [8]. In 2016, the world's data centers used more than 416.2 terawatt-hour (TWh) [2], which was higher than the total consumption of Britain at that time of around 300 TWh. The energy consumption and associated CO_2 emissions by data centers have only grown in recent years [2, 27, 26] and are expected to rise to 974 TWh in 2030 [1]. As such, there has been a growing interest in reducing carbon emissions, including emissions from data centers [13].

In addition to their high carbon emissions, data centers are generally inefficient and underutilized [8]. Server utilization rates within data centers barely exceed 6%, while facility utilization hovers around 50%. This inefficiency increases the environmental impact of data centers and highlights the need for improvements in their operation.

The research in this thesis focuses on a specific type of load in data centers: batch processing compute jobs. These compute jobs are characterized by their lack of strict requirements regarding the timing of their execution. In other words, they have a temporally flexible element, meaning their processing can be delayed to a later time of the day without causing issues for the system and without interrupting business as usual. A compute job consists of any processing step that does resource-intensive data processing. In this context, the resources refer to CPU and memory usage, which in turn have a direct correlation to electricity consumption [18, 4].

The work by [10] has shown that the amount of CO_2 emitted per kilowatt-hour (kWh) varies greatly depending on the location and time. In other words, there is a significant difference in the carbon intensity of electricity generation, depending on the time of day and the location.

However, significant differences in greenhouse gas emissions are only observable if green energy generation is a major source of a country's electricity generation. If more green energy sources are available in a country, the carbon intensity will vary more than in countries with a limited supply of green energy sources. Figure 1.1 shows this difference by comparing two countries with differing amounts of renewable energy generation. In this figure, by [10], two scatterplots are shown from two different countries, namely Bangladesh (A) and New Zealand (B). They plot the total generation in megawatt-hour (MWh) on the x-axis and the total CO_2-equivalent greenhouse gas emissions in tonnes on the y-axis. The top plot, visualizing this for Bangladesh, shows a significant relation between the total generation and the total carbon emissions, with an R^2 value of 0.9873. The bottom plot, visualizing this for New Zealand, shows a less significant relation between the total generation and carbon emissions, with an R^2 value of 0.2124.

The difference between these two countries is the availability of green energy sources. One of New Zealand's primary generation sources is hydroelectric power, providing abundant green energy. However, this is not always available when the energy load is high. In those cases, non-renewable energy sources such as coal and other fossil sources are used to a greater extent to meet the demand. In turn, the greenhouse gas emissions per MWh will be higher. Therefore, there is a much more significant variance in greenhouse gas emissions per generated kWh in New Zealand compared to Bangladesh.

There is an opportunity to intelligently schedule computing jobs to reduce the amount of greenhouse gasses emitted, provided these jobs have temporal flexibility. It becomes possible to optimize their energy consumption and reduce carbon emissions by leveraging the flexibility of compute jobs, the variability of greenhouse gasses emitted depending on the time, and the general under-utilization of data centers. The optimization can be done by scheduling these compute jobs to times at which the carbon load of the energy grid is low. Intelligently scheduling and redistributing compute jobs can align the demand for computing resources with periods of low-carbon energy generation, thereby minimizing the overall environmental impact of data centers.

Interestingly, the price of CPU time and memory consumption at cloud providers is rarely linked to the carbon footprint of their usage. The price does not scale according to the amount of green energy used, providing no incentive for companies to reduce the carbon footprint of their IT services in data centers. Therefore, creating and implementing a scheduler designed for this purpose is important if companies want to reduce their carbon footprint. This research investigates the possibility of creating and implementing a carbon-aware scheduler to minimize the carbon load of a real-world resource-intensive data processing pipeline at ING.

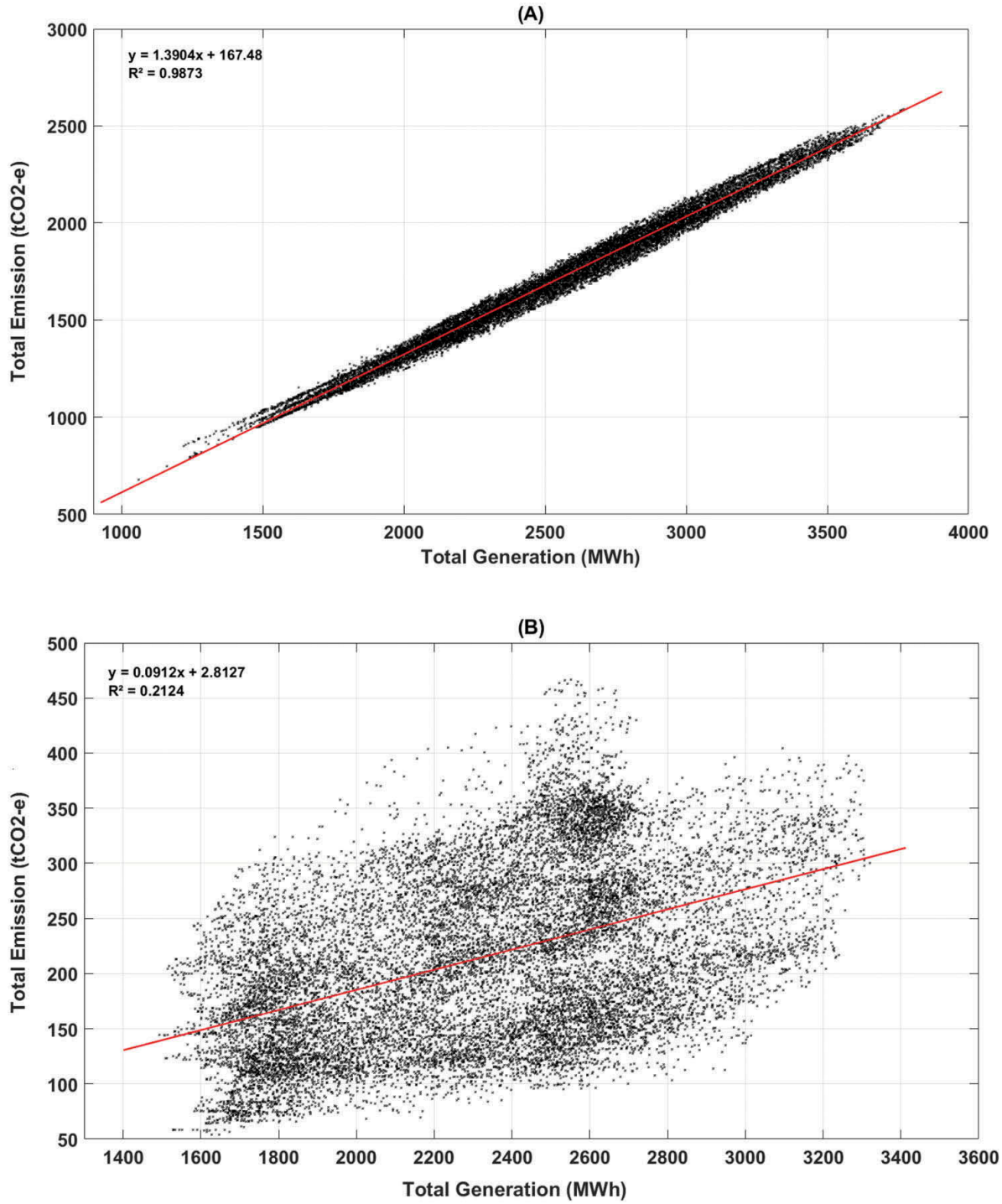


Figure 1.1: Correlations between generations and emissions for the electricity systems of (A) Bangladesh and (B) New Zealand in 2015, from [10].

1.1. Contributions

The objective of this master's thesis is to address the challenge of implementing a carbon-aware scheduler in a real-life corporate environment. This thesis is a collaborative effort between the Delft University of Technology and the trade and communications surveillance (TRCS) squad at ING. ING is a large multinational banking corporation headquartered in Amsterdam. It offers various financial services, including retail and commercial banking, and has a significant presence in many countries. The TRCS squad manages a data processing pipeline within ING that runs on the ING data centers. This pipeline has a high workload as it ingests and processes a large amount of data daily, consuming multiple virtual CPUs and much memory. The data in this pipeline has some temporal flexibility. Therefore, it allows scheduling the workload intelligently to reduce greenhouse gas emissions.

This thesis aims to demonstrate that implementing a carbon-aware scheduler within a company is an intricate process involving numerous requirements and challenges, even though the basic concept of a carbon-aware scheduler is theoretically relatively simple. The primary goal is to provide valuable insights for future teams in companies that are also interested in adopting or implementing a carbon-aware scheduler. To achieve this goal, the following problem statements have been formulated. They are presented chronologically to guide the reader through the research.

Task load prediction. Ideally, the scheduler can forecast the expected processing times of incoming tasks based on their foreknown properties. If the scheduler can predict this accurately, it can create a schedule without breaking the deadline constraints of incoming tasks. The process of determining, monitoring, and utilizing these features for task load predictions is explained in detail in section Section 4.1.

Acquiring information on energy generation. In order to accurately predict periods of low carbon load, a reliable source for green energy generation predictions must be selected, along with a calculation from energy consumption to CO_2 emissions. Interestingly, the conversion to absolute carbon emissions is opinionated; at the very least, there are vast differences in calculated carbon emissions depending on the source used for this calculation. The process of acquiring this information is described in Sections 4.2 and 6.3.

Scheduling. All gathered information on expected task load and energy production must be input to the scheduler. Some basic scheduling ideas are explored, such as splitting large tasks into smaller ones to run containers in parallel during non-carbon-intensive hours. It is crucial to closely monitor the cumulative carbon load associated with these various approaches to ensure that any additional overhead incurred does not lead to a net increase in overall carbon emissions. Details regarding the scheduler can be found in Section 4.3.

Implementation in OpenShift at ING. One of the main contributions of this thesis is implementing a carbon-aware scheduler in a practical setting at a company. The findings have been written down and summarised in this research to aid future teams wanting to implement a similar scheduler to reduce carbon emissions. Theoretically, the concept of a basic carbon-aware scheduler is not very difficult. However, when it comes to implementing it at a large company with business requirements, legacy systems, and authorization needs, there are inevitable challenges that need to be addressed. Chapter 5 provides an overview of implementing the carbon-aware scheduler in OpenShift at ING.

1.2. Research Questions

The following research questions are drafted to aid the research process and help it stay focused.

Research Question 1

"To what extent can task load be predicted regarding running times?"

Research Question 2

"How can predictions on solar and wind energy generation, as well as energy consumption, be utilized for implementing a carbon-aware scheduler?"

Research Question 3

"Is there any overhead introduced by implementing a carbon-aware scheduler?"

2 Background

In this chapter, the background of the research will be explained. Firstly, it explains the context in which this research was performed. Secondly, it will detail some generic and ING-specific requirements for the system that was made for this research.

2.1. Thesis context

This thesis partook within the team trade and communications surveillance (TRCS) at ING. ING is a large multinational banking corporation headquartered in Amsterdam. It offers various financial services, including retail and commercial banking, and has a significant presence in many countries. The primary objective of the TRCS team at ING is to detect and prevent fraudulent activity, such as insider trading, by its traders. To accomplish this, the team monitors these traders' trade and communications data to generate alerts of possible fraudulent activity that should be investigated. This data is delivered regularly and consistently in both text and audio formats. Some examples of data sources include email, Bloomberg (BB) chat, Reuters chat, and Microsoft (MS) Teams Voice logs.

The tool Relativity performs the alert generation, but the raw communication data must be processed before being sent to it. Relativity provides organizations with the tools to manage large volumes of data, conduct investigations, and ensure regulatory compliance. To process the data in preparation for Relativity, the two technical divisions of the team construct and maintain a pipeline through which the data 'flows'. These divisions are the capabilities and operations divisions; they are responsible for the pipeline. The capabilities division is responsible for constructing the necessary functionalities within the pipeline to enable data ingestion, cleaning, and transformation for compatibility with Relativity. Meanwhile, the operations division's role is to guarantee the continuous operation and maintenance of the pipeline. The business division of the team operates in Relativity, responding to alerts generated by the tool.

A conceptual overview of the entire pipeline can be seen in Figure 2.1. The flow of data goes from the left side to the right side. On the left side, source systems are displayed. These systems produce data in the form of emails, chat logs, and voice recordings of phone systems. The data from these systems is delivered in batches at regular intervals, about once daily, in a large encrypted zip file. Depending on the data type, it goes through various steps through the pipeline, as indicated by orange hexagons in the figure. Note that not all steps that exist are indicated in the figure.

For example, take the BB chat source system. The data from this system is supplied as a zip

file, which a gpg key has encrypted. Assume that this file is named `chat-data.zip.gpg`. This file's first processing step is a decryption step, as it is encrypted. After decryption, a new file is created named `chat-data.zip`. This file can be unzipped in another step, creating many more files containing chat logs. Each of these files then needs to undergo more steps, for example, a deduplication step or some other data enrichment step, before finally being sent off to the Relativity tool.

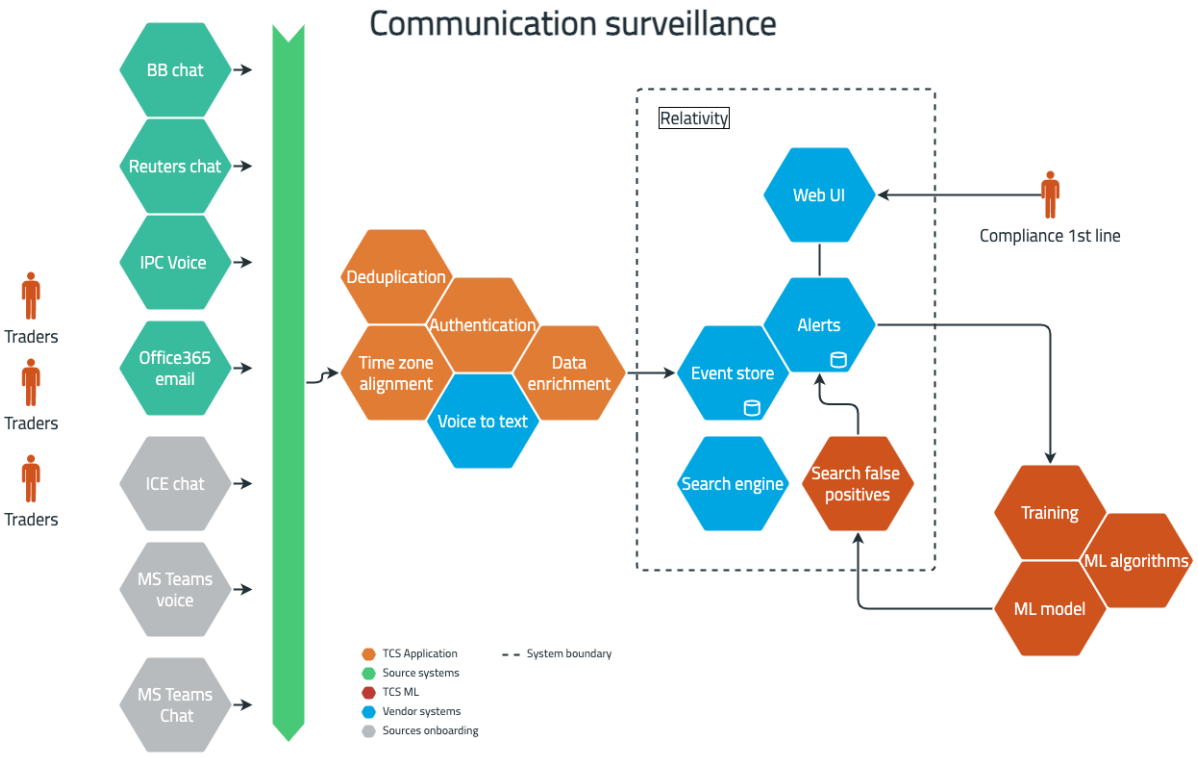


Figure 2.1: A conceptual overview of the communications surveillance pipeline from the TRCS team at ING. Source: ING.

2.2. Architecture of the data processing pipeline by the TRCS squad

The original data processing pipeline that the team uses is hard to manage, especially considering the sharing of files between steps and keeping track of the states of the files as they move through the pipeline. It also does not provide the team with a clear insight into possible backlogs of files that still need to be processed at any time. This uncertainty can be attributed to several factors, which will be discussed in more detail below.

First, the system lacks an overview of the current state of the files, including whether a file is being processed and the total number of files remaining for processing. It is hard for the team to verify that all the files the next step should process are actually processed because file states are not tracked and updated. By way of illustration, consider the scenario where a step in the pipeline processes 1,000 files but has to remove 100 as they are faulty or irrelevant. Thus, it outputs 900 files to be picked up by the following system. As the system does not keep track of the intermediate file states, it is hard to tell at the next step if it has received all files or if some of them were lost. It cannot know whether the 900 files it picked up are all, and no central entity keeps track of this.

Secondly, the pipeline also lacks a clear distinction between different processing steps, where

a single program handles one processing step. Instead, a file that arrives is ingested separately by a specific script or tool, depending on the data source of the file. This leads to the scenario where the team is unaware of the exact number of files processed at the various steps. They have to look through various logs of all the systems to get an idea of their throughput and identify possible sources of errors.

Lastly, there is no generalized method of sharing files between steps. Instead, the various processes output their files to multiple locations to be picked up by the next step. This meant that many files had to be copied back and forth multiple times, adding processing time and unnecessary complexity in keeping track of the state of the processing pipeline as a whole.

Because of this, the team opted to create a new system, solving all the issues mentioned above. The new system will have much more control over the data flow through the pipeline and be able to keep track of the state of files. The new system will also have a centralized solution for storing all files on an S3 bucket, reducing the number of file copies back and forth between systems.

The first step in creating this system is extracting every processing step that needs to be performed on the files to its own encapsulated program. Every program then has one specific task, relating much closer to the definitions of steps in the orange hexagons in Figure 2.1. These encapsulated programs will then be run in a docker container, allowing them to be deployed to ING container hosting platform (ICHP).

ICHP is an ING system, which is a custom version of OpenShift. OpenShift is a container application platform built on top of Kubernetes that provides additional features and functionality for building, deploying, and managing containerized applications. It includes integrated build and deployment pipelines, developer workflows, and security and compliance controls and can be deployed on-premise or in the cloud. It also has the added benefit of a fully featured user interface where the team's developers can interact with their deployed code. This allows them to verify that their code is running and working and check logs of all their deployed systems. ING is using a custom version of OpenShift to adhere to the more strict requirements that financial institutions need to adhere to by law.

Running the pipeline on a custom version of OpenShift provides the required data needed for the scheduler and the tooling to scale various parts of the data processing pipeline intelligently. OpenShift records metrics of all pods running on it, such as CPU usage metrics. It also provides the ability to scale individual deployments up and down through an API. Using the metrics and the ability to scale provided by OpenShift, it becomes possible to schedule the data processing pipeline intelligently.

All processed files will be stored using S3 buckets, provided by ING on-premise. A benefit of using S3 buckets is that they are centralized locations for file storage. Utilizing S3 buckets eliminates the need for multiple file-share locations. Instead, files are stored in a centralized known location, ready to be accessed by the subsequent stages of the processing pipeline. Only the S3 buckets need to be checked to determine what files are currently in the system. An additional benefit of using the S3 buckets is that it saves storage space because files produced by intermediate steps of the pipeline can be removed. It is possible to delete intermediate files knowing that the original file still exists on the S3 bucket. If processing needs to be redone later, the steps can be repeated, starting from the original file.

Two S3 buckets will be used: a `pending` bucket and a `processing` bucket. All newly arrived files produced by the *source systems* will be stored in the pending bucket, where the source systems are the ones described in Figure 2.1. Once the pipeline picks up a file to be processed,

it is moved to the processing bucket, where it will reside until its processing is done. If new files are created by one of the processing steps, they are also uploaded to the S3 bucket. Every processing container can access the processing bucket, which removes the need for multiple file-sharing locations to facilitate file exchange between different stages of the pipeline.

As seen in Figure 2.1, multiple source systems create input files. Most source systems upload a single file containing the data for a specific time period, like one day. This file is generally an encrypted and compressed `.zip.gpg` or `.tar.gz.gpg` file.

Figure 2.2 shows a conceptual overview of the flow of files through the various processing steps. Importantly, it shows that many more files may be created by the pipeline in the process of processing file A_1. As file A_1 arrives from one of the source systems, it is defined to be the 'source file', and every subsequent file $A_n, n > 1$ created by the pipeline is called a 'derived file'. Every processing step usually creates one or more derived files for the next processing step to pick up. In this figure, it can be seen that `Processing step 1` takes as input one file (A_1), but outputs three files (A_2, A_3, A_4) which all need to be processed in subsequent steps. It also shows that file A_6 never finishes processing, either due to an error or step 2 determining that the file would be unprocessable by any subsequent steps or because it is a duplicate.

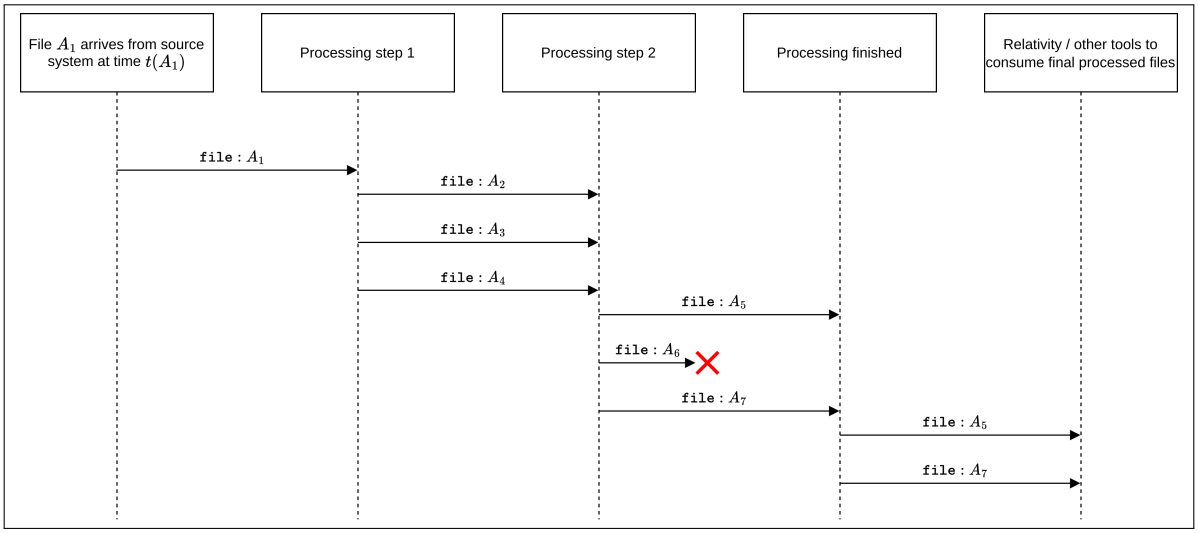


Figure 2.2: A conceptual overview of multiple processing steps from a source system to ingestion in Relativity.

The states of all the files are saved to a centralized database managed by the control panel. It is created based on the Django framework, which is an open-source web framework written in Python[1]. To enable processing steps to manage and update the state of the files within the pipeline, a REST API will be created in Django. Each processing step communicates using this API and registers any new files it uploads to the processing bucket. Every step also notifies the control panel when processing has started on the file to prevent multiple steps working on the same file. The advantage of using this centralized database and control panel is that it provides a clear overview of the processing pipeline and its files. It can quickly identify which files are pending processing, which are currently in process, which have completed processing, and which may have encountered errors during processing.

The control panel serves as the coordinator of the entire pipeline, with all the processing steps querying it for files and updating it whenever there are changes in file states. However, the

[1] https://www.python.org/

control panel currently lacks a scheduling mechanism to determine when files should be processed. Instead, it simply responds with the files currently awaiting processing whenever a processing stage requests for pending files. The systems decide when to query the control panel, and in the existing setup, they do this in a continuous cycle, with a brief pause of approximately 10 seconds between each query.

2.3. Requirements

This section will briefly explain the system requirements to implement a carbon-aware scheduler successfully. First, it will explain the generic requirements for systems to use such a scheduler effectively. Then, it will explain the requirements of the ING data processing pipeline, such that the relevance of the work of this research can be proven.

2.3.1. Generic requirements for a system to use a carbon-aware scheduler

A system must meet certain requirements before a carbon-aware scheduler can be implemented and affect the system's greenhouse gas emissions.

First of all, the system should be resource-intensive. It should mainly utilize resources that significantly influence energy consumption, such as CPU, memory, and GPU. If a system is not resource-intensive, the gains from using a carbon-aware scheduler may be less significant.

Secondly, the processing that is done by the system should possess some temporal flexibility. In other words, the tasks that run through the system should not be latency-sensitive. Ideally, they can be moved to be processed at any time throughout the day without interrupting business as usual for the system. As will be explained in Section 4.2, there is much variability in renewable energy generation throughout the day. As such, it should be possible for a task to be delayed up to 12 to 24 hours, such that energy from sources with a low carbon intensity can be used for most of the processing. Batch processing systems inherently align with this requirement, as they generally run periodically and are not necessarily latency-sensitive. An example of a system like this could be a resource-intensive job that calculates weekly reports for presenting metrics and Key Performance Indicators (KPIs).

A third requirement for the system is that the scheduling algorithm should be able to scale the system up and down when required. Ideally, the system runs in a containerized environment such as provided by OpenShift and Kubernetes, as these environments can start and stop multiple instances of pods and containers out of the box. This scaling must be programmatically possible for direct interface with the scheduler, ensuring task execution and processing control. The scaling can be implemented either at individual stages of the pipeline or the entire pipeline itself.

The fourth requirement is that the scheduler should have complete control over what task is processed by what instance to allow it to control when tasks are run. Therefore, it must be possible to set a system up to either request the scheduler for work or to wait until the scheduler assigns it work.

The fifth requirement is that it is assumed that the set of tasks to be processed is already known when determining the schedule for the next period of time. This means that the system must be capable of providing the scheduler with a list of all the tasks to be processed, along with their respective deadlines at the time of scheduling. If any tasks arrive after scheduling has already been performed for the current period, those tasks are moved to be processed at the next iteration. A more advanced scheduler could be implemented for more reactive scheduling during its set time period.

The sixth requirement for the system is that it should have the ability to either track CPU and memory usage to convert to energy consumption or monitor its own energy consumption directly. This is needed to continuously monitor the scheduler's effectiveness as it makes its scheduling decisions, as the energy consumption is required to calculate greenhouse gas emissions of the system. It must be known what hardware the system is running if CPU and memory consumption statistics are used to calculate the system's energy consumption, as the type of hardware influences the energy consumption.

To summarise, the following general requirements are drafted:

Generic requirements

1. The system should be resource-intensive.
2. The tasks processed by the system should have temporal flexibility.
3. The system should have the ability to scale up and down.
4. The scheduler should be able to have full control over task distribution over the system's instances.
5. The set of tasks to be processed should be known at schedule time.
6. The system must be able to either track resource consumption statistics, such as CPU and memory, to calculate energy consumption or to track energy consumption itself directly.

2.3.2. Requirements set by the ING pipeline

The ING system is a more specific implementation of a data processing pipeline that adheres to the generic requirements set in the previous section. The following paragraphs explain why the ING system adheres to the generic requirements stated earlier.

1. *The system should be resource-intensive.* The pipeline managed by TRCS is very resource intensive. The pipeline mainly consumes CPU and memory. In fact, the TRCS squad is one of the largest consumers of the ING private cloud. It is common for the pipeline to run for a few hours up to half a day. If the pipeline is halted for a few days, the backlog is generally too large to be processed on time.
2. *The tasks processed by the system should have temporal flexibility.* This is true for the tasks the TRCS pipeline processes. The arriving files must be processed within a few days per ING regulations. The deadline for arriving files is set to be processed before the next batch of files arrives to prevent a backlog of tasks.
3. *The system should have the ability to scale up and down.* As the system runs fully within OpenShift, it can scale up and down. In fact, every processing step is performed in its own encapsulated pod, thereby allowing separate parts of the system to be scaled up and down at will. The ability to scale pods up and down is available through an API offered by OpenShift.
4. *The scheduler should be able to have full control over task distribution over the system's instances.* Because every processing pod queries the central Django control panel for work, it allows the scheduler to control precisely which file is processed by which pod.
5. *The set of tasks to be processed should be known at schedule time.* The source systems presented in Figure 2.1 generate tasks for the system regularly. It is, therefore, feasible to run the scheduler at a predetermined interval to create and update the schedule for all currently known files in the system.
6. *The system must be able to either track resource consumption statistics, such as CPU*

and memory, to calculate energy consumption or to track energy consumption itself directly. As the TRCS pipeline is run on OpenShift, extracting resource consumption values such as CPU and memory statistics is possible. This is also provided through an API by OpenShift.

Apart from adhering to the general requirements, the ING system also has its own set of requirements.

ING Specific requirements

1. The data that is processed by the system is communications data, which requires processing in a predetermined sequential order. This data arrives from various source systems at a regular interval. Initially, the system only processes basic communications data like email and chat logs, but it should be easily extendible to other sources of data to be processed. Examples of new input data types are file attachments in emails and voice call recordings.
2. The data must be processed within a few days, as per ING requirements. However, as the amount of data input is large, it is ideally processed before the next batch of files arrives. Therefore, deadlines for arriving files are set to be processed before the arrival of the next batch. If there is a backlog of files, the system should allocate more resources to process those files to avoid further backlog.
3. The system should run on ICHP, an ING specific version of OpenShift. Any additions to the system should be able to run in this containerization platform.
4. The files are stored in a centralized location, in this case in various S3 buckets. This includes newly arrived files and any files produced during processing by any of the steps in the pipeline.
5. There is a centralized Django control panel, which is the only way to interact with the pipeline. The pipeline should be manageable from this control panel to aid the operations division of the TRCS squad. It should be the only source of truth of file states in the system and should provide a clear overview of those file states and any processing steps.
6. Some of the processing steps create a large number of files, up to thousands at a time. The system should be able to handle these files and keep track of all of them.

3

Related work

In this chapter, we review prior research and works related to the development of a carbon-aware scheduler, grouping them by topic.

3.1. Global warming, carbon emissions and the impact of internet services

This group delves into the broader environmental context surrounding global warming, carbon emissions, and the consequential impact of internet services. Studies in this category, such as those examining the environmental consequences of electricity generation and estimating global energy use of ICT networks, provide essential insights into the environmental footprint of information and communication technologies and data centers.

3.1.1. Climate Change and greenhouse gas emission sources

The issue of global warming and carbon emissions has gained significant attention in recent years. Yoro and Daramola [29] discuss CO2 emission sources, global warming, and climate change, focusing on environmental impacts. They provide insights into the impact of various industries and sectors, including coal-fired power plants and fossil fuels. Raza et al. [20] provide a review of the impact of climate change on agriculture, showing the adverse effects on crop adaptation and global food security. These works offer an understanding of the environmental impact of CO_2 emissions and are crucial for showing the relevance of the work in this thesis and other carbon-aware computing approaches.

3.1.2. Data center power consumption

Brown et al. [3] present a report to Congress on server and data center energy efficiency. The report discusses the energy consumption of data centers and provides recommendations for improving energy efficiency in these facilities. Their work highlights the importance of reducing carbon emissions from internet services.

Kaplan, Forrest, and Kindler [8] state the significant impact of data centers on carbon emissions, and thus, to that extent, the impact of services ran on them. The work shows that data centers are largely inefficient and underutilized. Their work provides insights into the strategies and technologies that can be employed to enhance the energy efficiency of data centers. It also shows that there is a large margin to work with regarding the maximum utilization of data centers. This margin can be used to postpone batch compute jobs to later times when the carbon load is lower, as apparently the capacity to do that exists in data centers.

Jones [7], Andrae [1] and Dayarathna, Wen, and Fan [4] addresses the energy consumption of data centers and the modeling thereof. They emphasize their significant environmental and financial costs. The articles highlight the need for increased efficiency and accountability in data centers to address the growing concerns related to energy consumption and environmental impact.

Katal, Dahiya, and Choudhury [9] provide a survey on software technologies for enhancing energy efficiency in cloud computing data centers. The study discusses software-based approaches at the virtualization, operating system, and application levels, aiming to reduce the energy consumption of data centers and address environmental concerns.

Radovanovic et al. [18] delve into accurately mapping data center resource usage to power consumption for Google data centers. They prove that there is a strong link between CPU and memory consumption in data centers and expected power consumption.

3.2. Variability of green energy and carbon emissions

Focused on the variability of green energy and carbon emissions, this group explores strategies for managing data center power efficiently and enhancing energy efficiency in cloud computing. The studies within this category contribute to the understanding of how green computing and renewable energy usage can be optimized to reduce the carbon footprint associated with information and communication technologies. The variability of green energy sources and their impact on carbon emissions is an important consideration in the transition towards a more sustainable energy system.

Khan [10] conducts a temporal carbon intensity analysis comparing renewable and fossil fuel-dominated electricity systems. The study reveals the temporal variability of carbon intensity, explores the interaction between electricity generation and emissions, and identifies peak carbon-intensive hours. In other words, the study investigates the carbon intensity of different electricity generation sources over time. It highlights the significant differences in carbon intensity at different hours and shows the importance of renewable energy integration in reducing carbon emissions.

Thind et al. [24] explore the environmental consequences of electricity generation. The study, focused on the Midcontinent Independent System Operator (MISO) region, reveals temporal trends in emission factors. They analyze the carbon intensity of different electricity generation methods by hour, day, month, and year. This provides valuable information for understanding the environmental impact of electricity consumption at different times throughout the day, generated by different sources. The relevance of this work to this thesis is that it provides a basis for the understanding of varying greenhouse gas emissions at different times throughout the day, which is the basis of the carbon-aware scheduler in this thesis.

3.3. Other carbon-aware (scheduling) solutions

This group contains diverse works displaying carbon-aware solutions beyond scheduling, including approaches to data center energy efficiency and the potential of smart grids. The studies highlight frameworks and strategies for optimizing energy consumption and carbon efficiency in various contexts, offering insights for developing a carbon-aware scheduler.

Radovanović et al. [19] present a system focussed on minimizing the electricity-based carbon footprint of Google data centers. By leveraging analytical pipelines, day-ahead demand prediction models, and risk-aware optimization, the system generates carbon-aware virtual capacity curvess (VCCs). The study demonstrates effective limitations on hourly capacity dur-

ing carbon-intensive periods, contributing to a more sustainable operation of data centers. It serves as a basis for the ideas implemented in this thesis.

Smale, Vliet, and Spaargaren [22] explore the shifts in goals concerning domestic energy uses in the context of smart grid transitions. The study analyzes how smart grids can be implemented to ensure that electricity grids can handle increased demands. Additionally, they argue that renewable energy consumption can be balanced more effectively by implementing smart grids. A carbon-aware scheduling system, which is the subject of this thesis, can serve as a part of such a smart grid.

In the context of internet services, Le et al. [13] propose an approach to capping the brown energy consumption of internet services, emphasizing the importance of renewable energy usage. They address the issue of reducing carbon emissions from data centers and internet infrastructure. Their approach focuses on reducing the carbon footprint of internet services by leveraging renewable energy sources and optimizing the allocation of workloads across data centers in different geographical regions.

4

Contributions

This chapter presents a high-level overview of the various improvements and new systems developed for this master's thesis. It explains the key contributions made to the new system to enable the carbon-aware scheduler to work, such as predicting task running time and gathering resource consumption metrics. Additionally, it explains how energy generation predictions are used to determine optimal times to schedule. Lastly, it briefly details the simple scheduling algorithm that is used. A more concrete explanation of these systems will be given in Chapter 5.

Figure 4.1 shows a conceptual overview of the contributions. In the middle, the scheduler is shown. It gathers data from various sources, such as the the European Network of Transmission System Operators for Electricity (ENTSO-E) Transparency Platform and historical runs of the system. It combines these data points to create a schedule aiming to reduce the system's carbon emissions. Once it has determined a schedule, it communicates this to the container orchestration platform, which in turn starts up pods to process at the right times. The parts in this figure will be highlighted in the next few sections.

4.1. Task load prediction

The first step to determining a schedule is predicting the running time of arriving tasks. A brief overview of how the scheduler makes predictions is provided in the following sections, and this describes the contributions in the blue box at the left side of Figure 4.1. The implementation of this at ING is explained in more detail in Section 5.1.

The scheduler needs predictions on task running times for two reasons. First of all, the scheduler must ensure that it does not break deadline constraints when scheduling tasks. Secondly, as the scheduler's ultimate goal is to reduce carbon emissions, it attempts to shift processing to periods of low carbon intensity. For this to be effective, the scheduler must be able to estimate the running time of all the tasks to be processed. This is to ensure that the scheduler knows when tasks need to start such that they utilize mostly low carbon-intensive energy, as will be explained in more detail in Chapter 5.

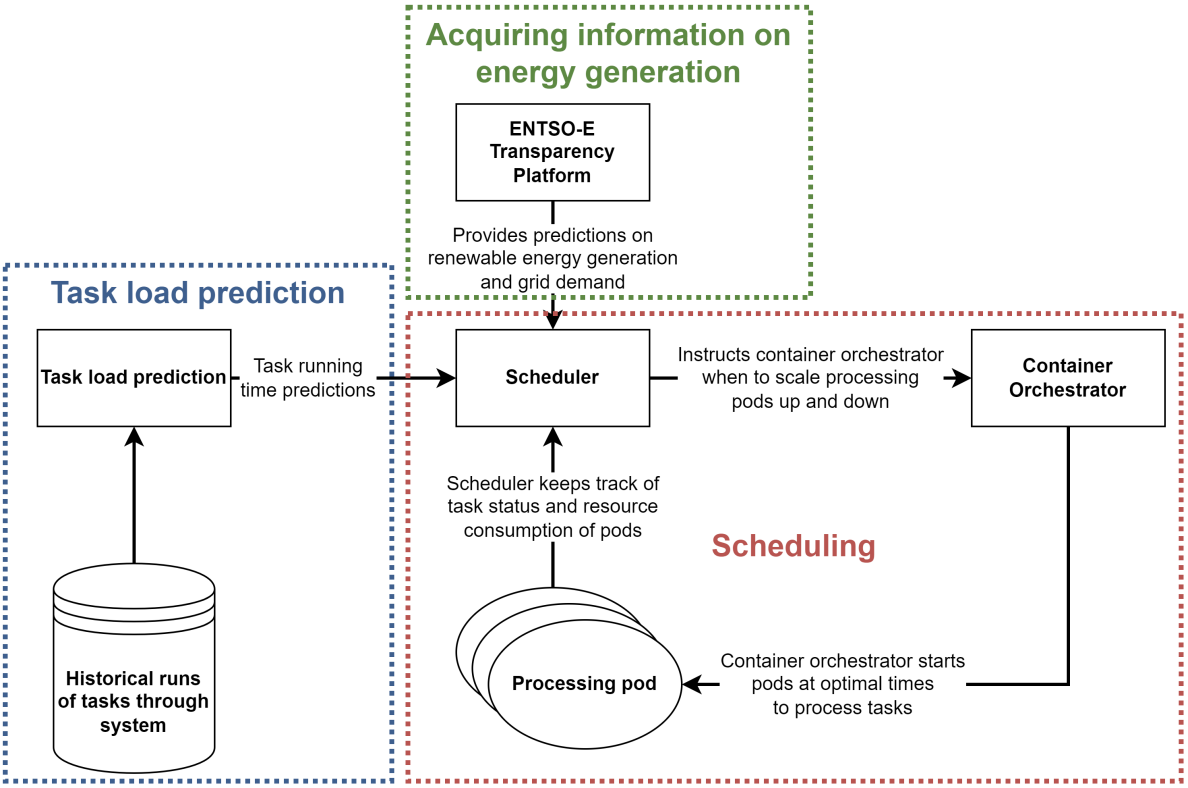


Figure 4.1: A conceptual overview of the contributions from this thesis.

4.1.1. Collecting metrics

Task running time predictions are based on data collected from historical runs of tasks through the system, which can be grouped roughly into two categories. The first category contains properties of tasks that can be determined before they have been scheduled or processed. This category, therefore, contains 'foreknown properties' of tasks, such as input file size and task type. The second category contains metrics regarding the processing times of the tasks. Examples of data points in this category are the start and end times of tasks.

For every task that is processed by the system, a metric is saved to the database. This metric contains all the data points from both categories mentioned previously. Therefore, there is a one-to-one relationship between a task and a metric.

4.1.2. Using collected metrics to determine task load

When a new set of tasks arrives, the scheduler predicts how long each task will take to process based on the collected metrics of previous tasks in the system. The scheduler does this by determining the processing speeds of previous tasks in the system and formalizes this based on their foreknown properties. It synthesizes new data points based on the data points in the first and second categories.

The scheduler refines predictions as more data arrives to adapt to changes in tasks and changes in the processing pipeline. This is implemented using a time-based sliding window. By implementing a sliding window, average task running times can be calculated over a set timeframe, automatically discarding the oldest tasks.

Using a time-based sliding window has the advantage that changes in the tasks and the processing system are automatically updated in the task load predictions. For example, consider the scenario in which the amount of input data that is delivered slowly increases over time. If the scheduler would not discard older metrics, it would be much slower to respond to these increasing file changes. In fact, it would never converge to the real task processing times as the older, slower tasks are still taken into account. By using a sliding window, metrics implicitly have a timeframe for which they are valid, after which they are not taken into account anymore.

4.1.3. Determining the latest feasible starting time

It is important for the scheduler to prevent deadline incursions when scheduling files. To prevent deadline incursions, the concept of a 'latest feasible starting time' is introduced.

When a scheduler assigns tasks, it calculates the 'latest feasible starting time' based on the task's predicted running time and its deadline. This calculation involves subtracting the predicted running time from the deadline to determine the latest time at which the task can start and still be completed before the deadline. By doing this, the scheduler ensures that every task is processed within the given deadline.

When scheduling a set of new tasks, the scheduler collects all the corresponding files and puts them into a single queue. The scheduler then determines the optimal starting time for this queue. The tasks within the queue are then prioritized based on their latest feasible starting times. The files with the earliest start times are processed first. Additionally, any tasks at risk of missing their deadline are moved forward to ensure they are completed on time, even if this means that they are not processed during the optimal time.

4.2. Acquiring information on energy generation

The primary objective of the scheduler is to minimize carbon emissions whilst maintaining the same processing capacity as when the scheduler is not used. In order to achieve this, the

scheduler focuses on scheduling compute jobs during green energy hours, which are periods when renewable energy sources are actively generating electricity. Additionally, it takes into account the forecasted total load on the electricity grid. By leveraging green energy hours, the scheduler aims to optimize the allocation of compute jobs that still need to be executed in order to use more green energy on average, thus reducing carbon load. The following sections describe the contributions in the green box at the top of figure Figure 4.1.

The scheduler does not perform the actual conversion of predictions into carbon emissions. In fact, it does not actively use predicted carbon emissions in its scheduling decisions. The scheduler primarily relies on two types of predictions: green energy production and total grid consumption. By considering these two sets of data, the scheduler can generate a reasonably accurate estimate of when the carbon load will be at its lowest. After the fact, the energy generation metrics can be used to determine if the scheduling decisions were effective by calculating the greenhouse gas emissions by the system. The purpose of this research is to show that this is enough for improved scheduling of compute jobs simply by aligning them roughly with periods of high renewable energy generation and low consumption. The connection between the scheduler's operations and carbon emissions will be explored further in Chapter 6. This section delves into the process of finding a robust prediction method and explains the data on which the scheduler ultimately makes its decisions.

Variability of green energy production

Figure 4.2 displays the average renewable energy generation in the Netherlands by the time of day for every season in 2022. The figure shows that, on average, the amount of renewable energy generation is highest between 12:00 and 14:00 in the afternoon in the Netherlands, depending on the season. The most significant contributor to this peak is the amount of solar generation, which peaks between 12:00 and 14:00.

The variability in renewable energy generation is the primary reason for implementing a carbon-aware scheduler in this thesis. Clearly, there are significant differences in renewable energy availability throughout the day in the Netherlands. Therefore, attempting to align resource-intensive processing to these predictions is expected to have an impact on the system's total greenhouse gas emissions.

Although renewable energy peaks around noon on average due to solar generation, other sources should be considered. Figure 4.3 displays the energy generation statistics in the Netherlands on November 13th, 2023. These graphs clearly show that wind energy should also be taken into consideration when scheduling tasks on a day-to-day basis, as the peak production of renewable energy occurred around midnight on November 13th, 2023.

Considering grid demand

Relying solely on renewable generation predictions is not sufficient. The scheduler should take into account the expected load on the electricity grid, too. Consider a scenario in which there is a lot of renewable energy but also an expected peak in grid load. In this scenario, other non-renewable sources must be employed to meet the demand, reducing the clean energy available for the scheduler. The scheduler should take into account the expected load of the grid as there may be better options throughout the day, at which the total renewable generation is slightly lower, but the predicted demand is significantly lower. To that end, the scheduler calculates a "renewable energy percentage", using both the renewable energy generation predictions and the predicted demand on the electricity grid. The renewable energy percentage is the main source of information the scheduler uses for determining periods of low-carbon-intensive energy generation.

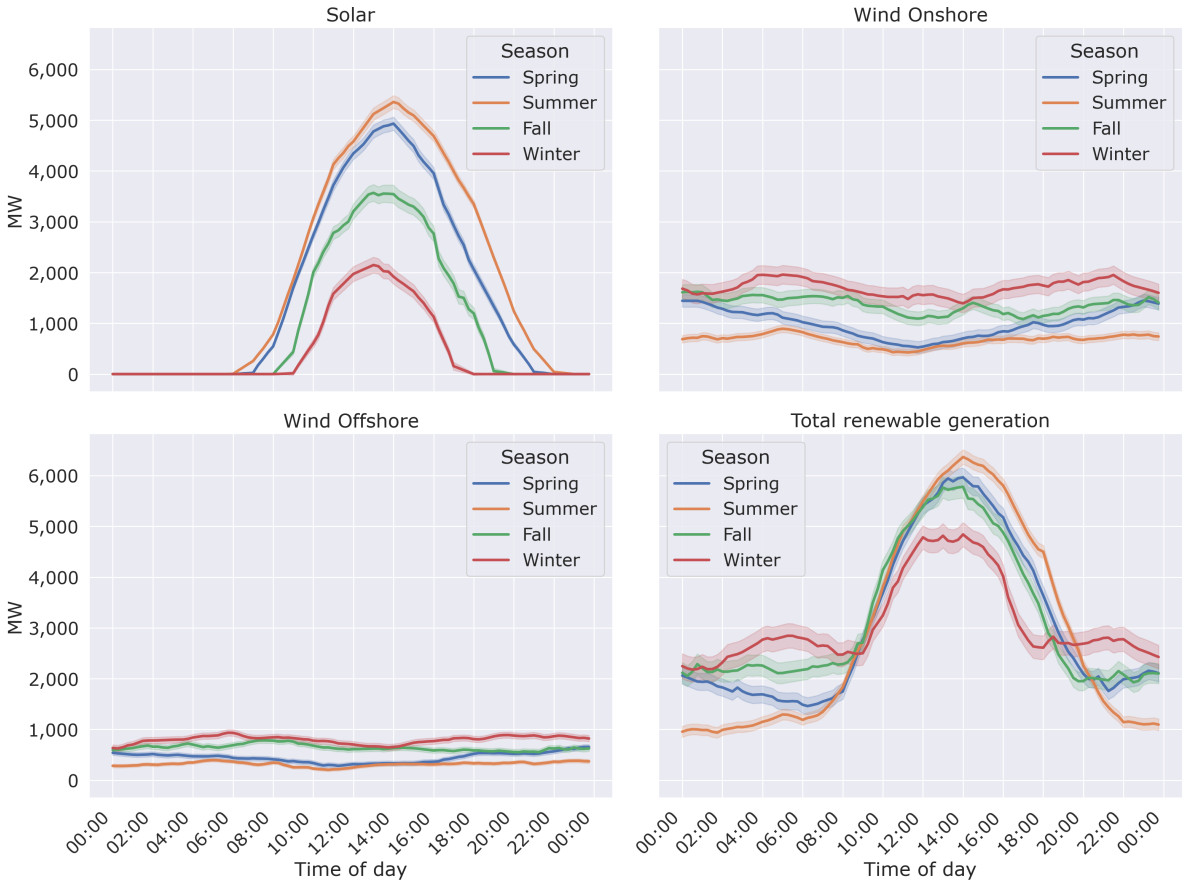


Figure 4.2: Renewable generation predictions per season in 2022 per time of day

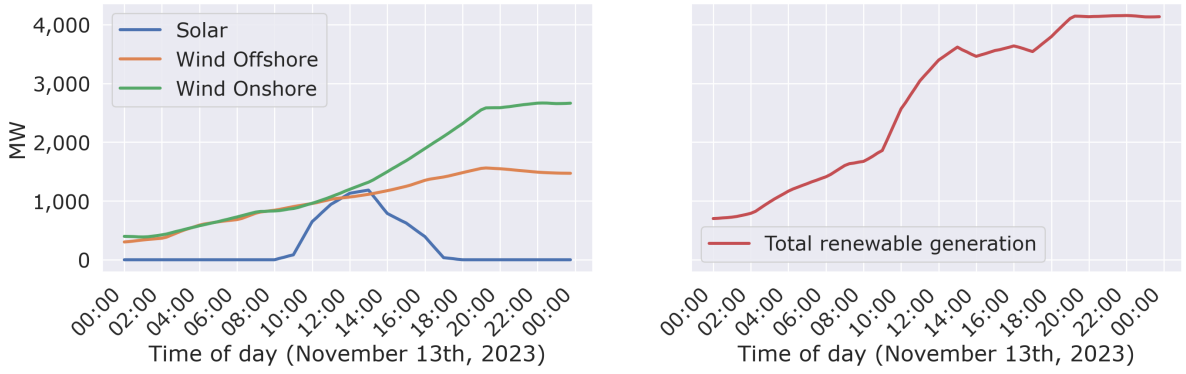


Figure 4.3: Renewable energy generation predictions in the Netherlands for November 13th, 2023, with an increasing amount of wind energy. The total amount of renewable energy available does not peak during noon but during the night.

The renewable energy percentage is simply calculated by summing the renewable energy predictions for solar and wind and dividing them by the forecasted demand.

$$\texttt{renewable energy percentage} = \frac{\texttt{solar} + \texttt{wind offshore} + \texttt{wind onshore}}{\texttt{grid demand}} \tag{4.1}$$

The average renewable energy percentages over the time of day can then be plotted per season. This is shown in Figure 4.4. At the top left, the forecasted renewable energy generation is plotted. The top right shows the forecasted load in The Netherlands. The combination of these values is then plotted in the bottom graph. As can be seen, on average, renewable energy generation is highest around noon every day, and the forecasted demand is lowest at these times. Therefore, it is expected that the energy generation will be, on average, the cleanest around noon in terms of grams of CO_2 equivalent emissions (gCO_2-eq) per generated kilowatt-hour (kWh).

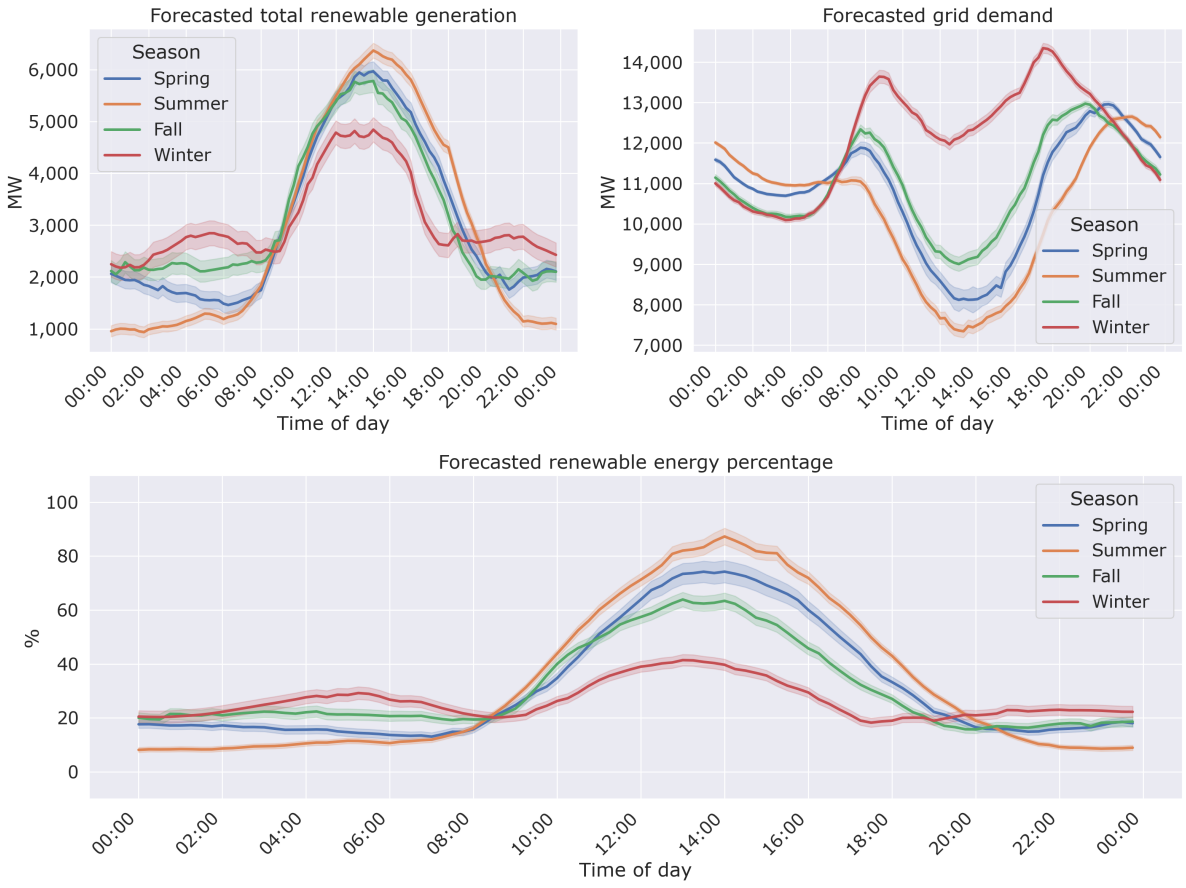


Figure 4.4: Average renewable energy percentage forecasts for the four seasons in 2022, based on the predicted renewable energy generation and grid demand in The Netherlands.

Two conclusions can be drawn from these calculated averages. First, the graphs show that the largest difference is between noon and midnight. Therefore, a 24-hour forecast should be sufficient to find clean periods of energy generation in terms of greenhouse gas emissions. Second, even though there is a difference between seasons, on average, the optimal time to schedule is around noon.

4.3. Scheduling

The tactic of the scheduler is to aim for the predicted highest renewable energy percentage in the next 24 hours. Once it has determined the optimal time, the scheduling methodology is simplistic; it aims to process all tasks symmetrically around this point. In other words, tasks are batched together as one processing job to be processed symmetrically around this peak in low-carbon-intensive energy generation. To do so, it predicts a processing time for all tasks and sums these times up. Once this has been calculated, the scheduler knows the expected total running time of the tasks to be processed. Using this information, it can determine when the set of tasks should start to process to maximize the usage of low-carbon-intensive energy. The following paragraphs describe the contributions from the red box, shown in the middle and bottom right parts of Figure 4.1.

Additionally, the scheduler takes concurrent processing into account. A maximum concurrency setting is configurable in the system; the scheduler considers this value when predicting the total running time of a batch of tasks. The main purpose of adding concurrency is to maximize the utility of low-carbon-intensive energy.

Figure 4.5 shows this concept visually. It plots the average renewable percentage over the time of day in summer. As can be seen, the peak in renewable energy percentage is at 14:00; the scheduler aims to process tasks symmetrically around this time. Assume for this scenario that the scheduler has predicted the total task running time to take 4 hours without any concurrency. If the scheduler would get this as input, it would start processing at 12:00, as it expects to finish around 16:00. However, if the maximum concurrency is set to 4, the scheduler expects the running time to be approximately 1 hour. Therefore, it would delay processing until 13:30, with an expected end time of around 14:30. From this figure, it should be clear that adding concurrency maximizes the utility of renewable energy generation, on average.

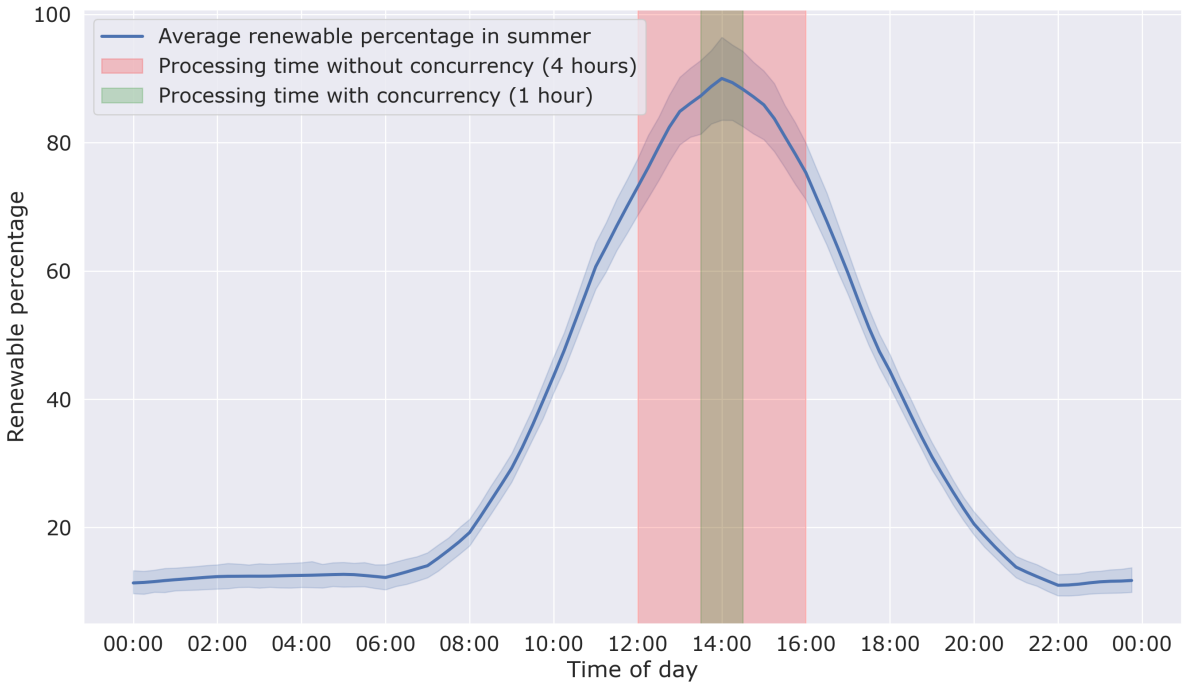


Figure 4.5: Example of scheduling decisions without and with concurrency, based on the average renewable energy percentage in the summer of 2022.

Concurrent processing is implemented by the scheduler using horizontal scaling. Horizontal

scaling refers to a method of increasing the capacity of a system by adding more instances of the same type of resource. In this case, by adding more processing pods. Conversely, vertical scaling increases the capacity of existing resources by allocating more resources to them. Vertical scaling could be implemented in this context by allocating more CPU time or memory to one pod.

5

Implementation in OpenShift at ING

The previous chapters have explained the context of this research, its relation to the currently existing work, and the main contributions that this work aims to achieve by implementing a carbon-aware scheduler. To show the relevance of the work and to test the effectiveness of the scheduler, it was implemented into a data processing pipeline managed by the trade and communications surveillance (TRCS) squad at ING. The scheduler can be evaluated at a later stage using this implementation. The implementation is shown conceptually in Figure 5.1.

5.1. Task load prediction

In this section, the methodology of task load prediction is explained, along with how this is specifically implemented at ING. It describes the contributions from the blue box, shown in the middle of Figure 5.1.

5.1.1. Defining a task

The first step in task load prediction is precisely defining what a task is. As explained in Chapter 2, the refactored system keeps track of every individual file through the various processing steps. Additionally, Figure 2.2 shows how various steps may produce multiple files from one single input file. In other words, if a source file (A_1) arrives at processing step 1, three new derived files (A_2, A_3, A_4) may be produced, all of which need processing in step 2. Due to this, there are two options for defining a task:

Option A: A task is defined as a single file needing processing in a single step. Thus, processing file A_1 in step 1 and files A_2, A_3 and A_4 in step 2, respectively, are four separate task definitions, all with their own deadlines and measurements. There would be four separate tasks defined when using this option. This option is shown in Figure 5.2a.

Option B: A task is defined as the processing of one source file (A_1) up to the arrival of the final files A_5 and A_7 at the destination system, such as Relativity. In other words, any files that are created in subsequent steps after the arrival of a source file are not counted as separate tasks. Instead, a task is linked to the arrival of the first source file until all processing steps are done. Only a single task is defined, which is linked to the file A_1, when using this option. This option is shown in Figure 5.2b.

Both options are valid and have their advantages and disadvantages. Option A allows the scheduler more granular control over the task execution times. Its main advantage, therefore, is that it is a more fine-grained and detailed definition of a task, allowing for metrics to be

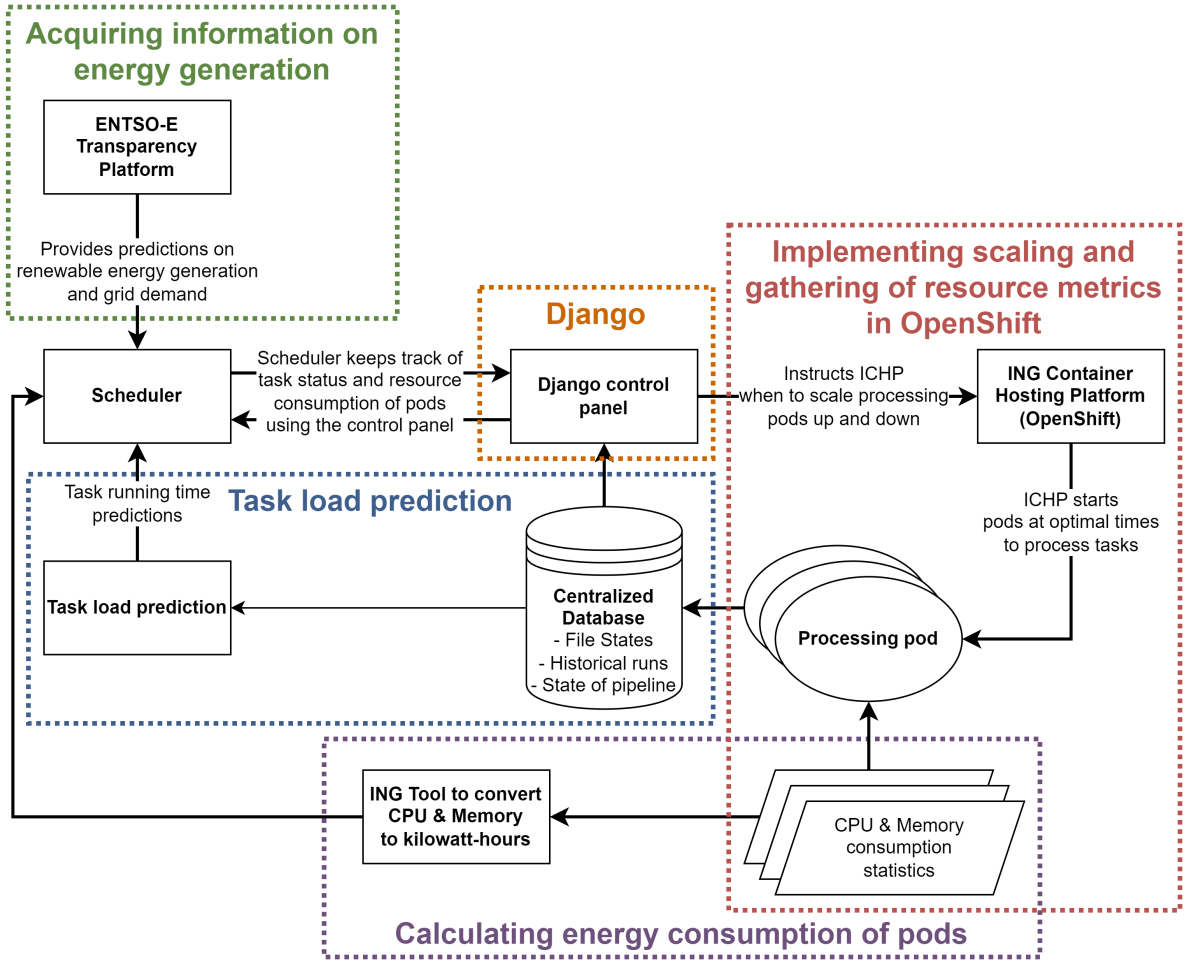
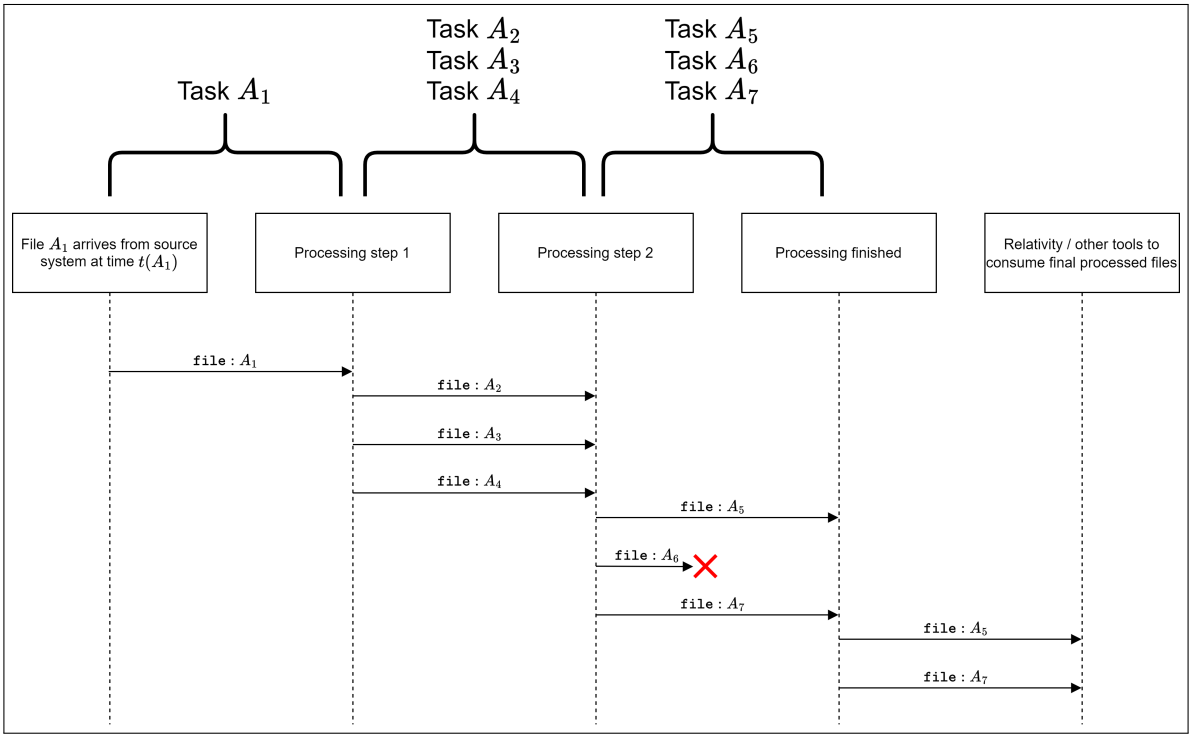
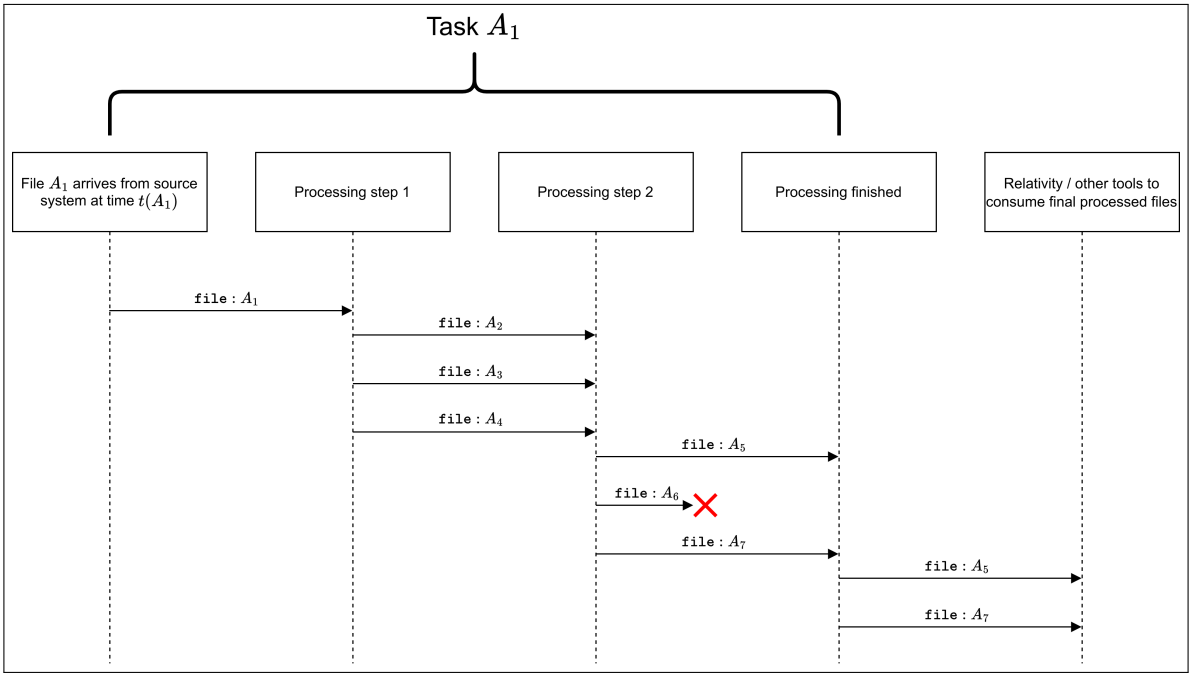


Figure 5.1: A conceptual overview of the implementation of the carbon-aware scheduler at ING.



(a) Task definition visualized for option A, in which every file created has its own linked task.



(b) Task definition visualized for option B, in which a task is linked only to the incoming source file and is defined as done at the point in time when the final files are consumed.

Figure 5.2: Conceptual overview of the two possible definitions of a task.

determined per processing step. The main disadvantage of option A is the complexity of the scheduling. The reason for this is the fact that the files are processed in sequential order through the various processing steps. It becomes hard to predict the total running time of the task and its associated resource consumption when considering the fact that processing steps may create or remove intermediate files.

The main advantage of the second option is the fact that it allows for a much simpler implementation of the scheduler. The obvious disadvantage of the second option is the lack of granular control over the various processing steps of the pipeline.

For this thesis, option B is chosen. The purpose of this research is to show that even a simple scheduling algorithm on a basic definition of a task is able to have a significant impact. If option A is chosen, the scheduling algorithm becomes a complex multimodal scheduling problem for which advanced scheduling algorithms are required. This thesis primarily aims to set up a basic scheduler at ING and demonstrate its benefits. By doing so, it aims to establish a strong foundation for the creation of a more complex scheduler in the future.

5.1.2. Linking derived files to source files

Upon the arrival of a source file in the pending bucket, a new entry is created in the database. This entry registers the new source file. Any derived files that are created need to be linked to its original source file. This information helps the scheduler to distinguish the files that belong to the same task.

Figure 5.3 displays a UML diagram of the data structure of the scheduling app and how its models are defined and linked. This figure represents how files are stored in the database. It is interesting to note that there is no distinction between how source files and derived files are saved in the database. This is because, inherently, there are no differences between these files. The difference only matters to the definition of a task, but the files are stored the same way in the database.

Figure 5.3 also shows how a derived file is linked to a source file by means of the `source_file` property of the `File` class. This property is not required, and setting it defines a link from a derived file to a source file. If the property is set, it means that the file is a derived file, in which case the `File.source_file` property links to the original incoming source file arriving from a source system. If the property is not set, the file is defined as a source file.

File	`id`	`source_file`	Implicit file type
A_1	1	`null`	source file
A_2	2	1	derived file
A_3	3	1	derived file
⋮	⋮	⋮	⋮
A_7	7	1	derived file

Table 5.1: Database Representation of source files and derived files, given the example source file from Figure 5.2b

To make this distinction between files in the database more concrete, take another look at Figure 5.2b, which displays an example source file arriving. In this case, file A_1 is a source file, and any subsequent files A_n with $2 \leq n \leq 7$ are derived files. Therefore, the files would be stored in the database as shown in Table 5.1.

5.1.3. Determining task deadlines

Upon arrival of a source file from a source system, a deadline is determined for that file. By default, the deadline is set to 24 hours after the arrival time of the file. This default is chosen specifically to align with the ING system, as files arrive in daily batches. It is tweakable using

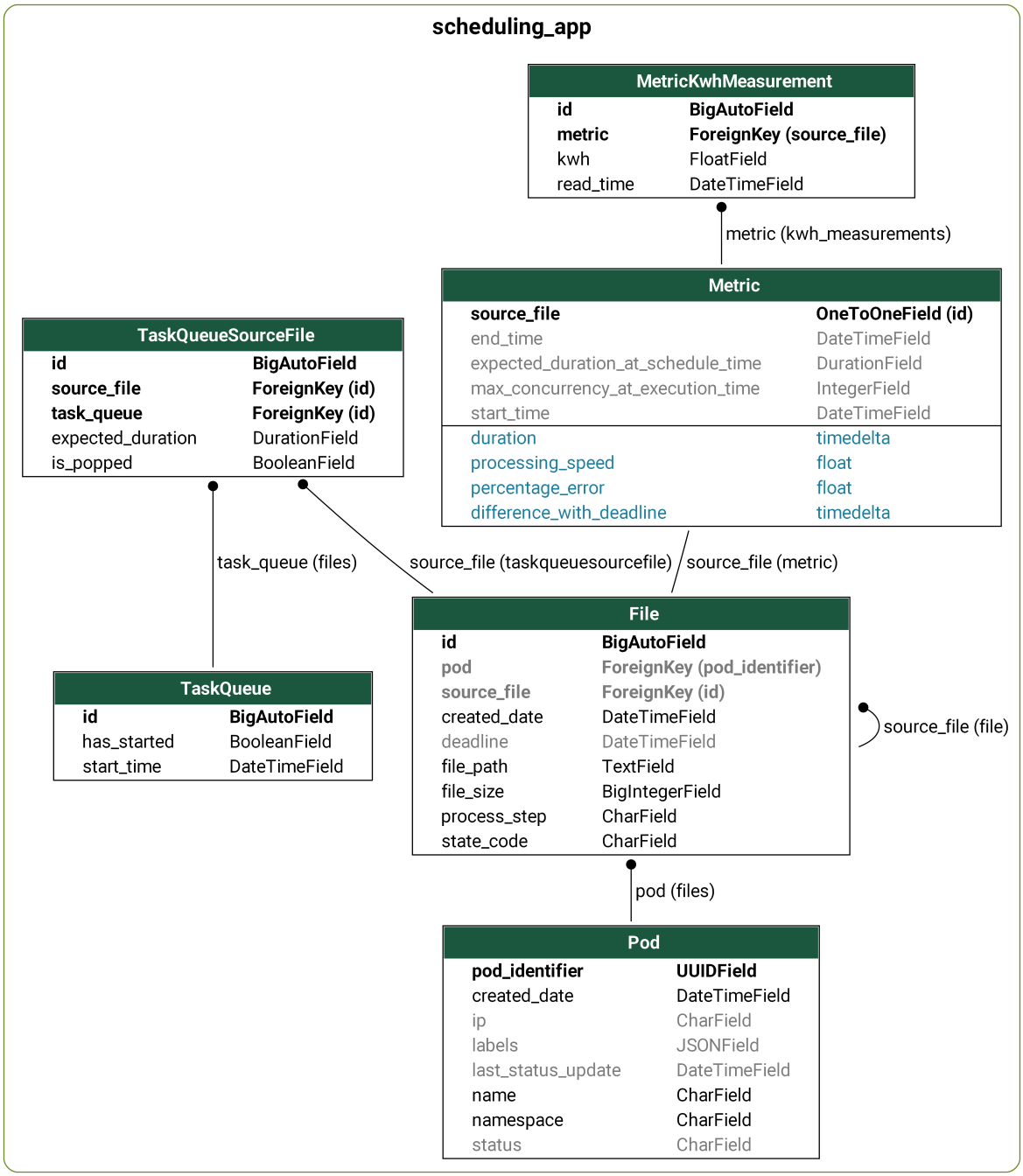


Figure 5.3: UML diagram displaying the data structure of the scheduling app and how its models are defined and linked.

an environment variable for development and testing purposes. Any subsequent derived files will have the same deadline as the source file.

5.1.4. Data collection for task load prediction

The collected data can be broadly categorized into two groups, which will be explained in the next few sections. The first category contains information that can be extracted from tasks before their execution, such as task type and source file sizes. This category is named the 'foreknown information of tasks'. The second category contains data about the task's execution time, which could influence energy production and subsequent greenhouse gas emissions. These factors include the time of day, the day of the week, and the month in which the task is scheduled. It also stores the starting and ending times of tasks so that the total running time and processing speed can be calculated. This data can be leveraged at a later stage to identify optimal scheduling times aimed at minimizing greenhouse gas emissions associated with job execution. It is also used to retroactively determine the emitted carbon emissions, as will be explained in Section 4.2.

Both data categories are collected and linked together in a `Metric`. Refer back to Figure 5.3, which also presents a UML diagram of the `Metric` class. In this figure, the definition of a `Metric` is given, defining how this is stored in the database and how this relates to a `File`. As can be seen, there is a one-to-one relation between a metric and a file. However, this relation is not always defined. It is only defined for source files. In other words, a metric is only relevant for source files and does not apply to derived files. This is because tasks are associated directly with source files, as explained in the previous sections.

Collecting foreknown information about tasks

There are a couple of properties that can be determined from a task before it is executed. These properties include all the information which are known the moment a new source file arrives in the pending bucket.

The first property that can be determined is the task type. Depending on the source system, the input data type may be different and thus may require different processing steps. For example, the source file that arrives from the Microsoft (MS) Teams voice system contains audio recording data, whereas the data from Bloomberg (BB) Chat contains textual data. The pipeline needs to process these different types of data through different steps. Therefore, the processing time will vary depending on the source of the data. As every task is linked to a single source file, the task type changes depending on the system the source file arrived from. The second property that can be immediately extracted is the file size of the input source file, as this can be retrieved from the pending S3 bucket.

Listing 1 presents a code snippet from the `TaskQueueExecutor` class. This class is responsible for executing all tasks that are queued when their starting times have passed. These are the first actions that are taken when a task starts processing. Therefore, lines 14-17 update the metric start time to the current time. They also update the metric with the expected duration that this task should take. In other words, the expected total running time for this task is saved in the metric to allow for an evaluation of the predicted running time after the task has finished processing.

Collecting information about the task's execution time

The second category of data collected is regarding the execution time of a task.

The first two properties that are collected and stored are the start and end times of a task. Thus, whenever a task starts processing at its scheduled start time, the actual starting time is

```python
def _execute_thread(self, lock: threading.Lock):
    while self._has_unpopped_files(lock):
        task_queue_source_file = self._task_queue.pop()
        lock.release()
        source_file = task_queue_source_file.source_file
        logger.info(
            "Processing source_file %s from task queue",
            source_file,
        )

        # GFC
        gfc_files = File.objects.filter(file_path=source_file.file_path)
        for file in gfc_files:
            file.metric.start_time = now()
            file.metric.expected_duration_at_schedule_time = (
                task_queue_source_file.expected_duration
            )
            file.metric.save()
        self._process_files(gfc_files)

        ...

        unzip_all_finished = all(
            file.process_step == FileProcessStep.FINISHED.code
            for file in unzip_files_from_source_file
        )
        unpickle_all_finished = all(
            file.process_step == FileProcessStep.FINISHED.code
            for file in unpickle_files_from_source_file
        )

        if unzip_all_finished and unpickle_all_finished:
            for file in gfc_files:
                file.metric.end_time = now()
                file.metric.save()
```

Listing 1: Snippet from the `TaskQueueExecutor` class, responsible for executing all the queued tasks.

saved to the `Metric`. Once the task has completed processing the source file and any of its derived files fully, the end time is updated to the metric as well.

The second set of properties collected are regarding the time of execution of a task. Namely, the time of day, the day of the week, and the month are saved. These factors are expected to have an impact on the amount of greenhouse gasses emitted, as there may be differences between renewable energy available depending on the season.

It must be noted that the simple scheduler implemented does not take these expected emissions into account directly. A more advanced scheduler could take into account these data points to optimize the scheduling decisions even more, as will be explained in Section 4.3. However, these data points are used to retroactively calculate the amount of emitted greenhouse gasses, such that the simple scheduler can be evaluated on its effectiveness.

Data collection regarding the execution system

The system on which tasks are executed may have an impact on the processing times and carbon emissions. Therefore, relevant data regarding the execution system is saved for every task. This data includes factors such as allocated (virtual-)CPUs, allocated memory, and number of concurrent jobs allowed. These factors may not change drastically over time; nonetheless, they are expected to have a significant impact on task execution times. The system must be able to take changes in the execution system into account. Therefore, this data is saved and used to calculate the expected running times of new tasks.

5.1.5. Predicting task running time and latest feasible starting times

The scheduler must be able to predict the running time of their associated tasks, including any subsequent processing steps on derived files, to adhere to the deadline requirements of the incoming source files. Once it has determined the estimated running time, it is able to determine the 'latest available starting time' for that file. The latest feasible starting time is the time at which the first step of processing should start the latest, such that the last files are processed before the deadline.

There are two factors that influence the latest feasible starting time, which are the deadline of a task and the expected processing time of a task. The definition of task deadlines has been explained in Section 5.1.3 above. The expected processing time is influenced by some of the fore-known properties of the task and the maximum concurrency setting of the system. The maximum concurrency setting determines how many containers can be running at the same time at any time during the processing of all tasks and files. These will be discussed in the next few sections.

Using the foreknown properties of a task to determine expected running time

The scheduler determines the expected running time of a new task based on the foreknown properties of a task. In the case of the ING system, that is the task type and the file size of the accompanying file. As stated previously, a `Metric` keeps track of the start and end times of tasks, as well as the accompanying files and file sizes. From this, the scheduler can calculate for every historic task its processing speed in bytes per second (B/s).

$$\texttt{processing\_speed} = \frac{\texttt{metric.end\_time} - \texttt{metric.start\_time}}{\texttt{metric.file.file\_size}} \tag{5.1}$$

The scheduler then considers all tasks of the last 30 days that have been processed successfully and calculates their median processing speed. It then has an expected processing speed in B/s.

Once a file arrives, the expected processing time of that file is calculated by dividing the task file size by the median processing speed:

$$\texttt{expected\_duration} = \frac{\texttt{task.file.file\_size}}{\texttt{median\_processing\_speed}} \tag{5.2}$$

Influence of the maximum concurrency setting on the expected running time
Figure 5.4 shows two scenarios with different maximum concurrency settings. In Figure 5.4a, the maximum concurrency setting is set to 3, whereas in Figure 5.4b, this is set to 2. These figures demonstrate the relationship between the maximum concurrency and the expected task execution time. Both figures conceptually show the timeline of processing a task and its related files. At the left, the point in time at which a file A_1 arrives is marked with $t(A_1)$. It shows that three more derived files are created, A_2, A_3, and A_4 by processing file A_1 From the arrival time of file A_1, a deadline is calculated; in the case of the ING system, this is 24 hours later. This time is marked with $D(A_n)$, as this deadline is for all files, including derived files that may be created in the processing of file A_1. In the middle, both figures conceptually show the timeline of processing all files. Whereas the top figure has a maximum concurrency of 3, the bottom figure has a maximum concurrency of 2. This shows the influence of the maximum concurrency on the expected processing time in both scenarios. As is clear, the pipeline in the top figure with more concurrency is able to process the derived file in step 2 all at the same time, whereas the bottom figure needs to postpone processing of one of the files.

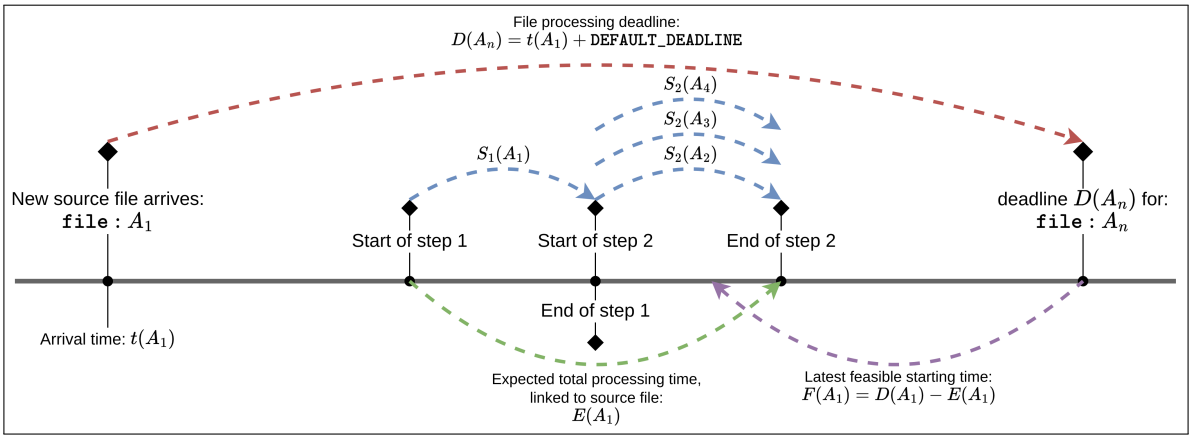
It should be clear that the maximum concurrency setting has a large influence on the expected processing times of tasks. Thereby, it also has an influence on the calculated latest feasible start time of a task, which should be taken into account to prevent deadline incursions.

To execute tasks concurrently, the `TaskQueue` and a `TaskQueueExecutor` classes are defined and implemented. These are responsible for executing tasks at the right time, possibly concurrently. A `TaskQueue` contains a batch of tasks to be processed and a starting time. A `TaskQueueExecutor` executes the files in a `TaskQueue`, one by one, by starting up the necessary pods sequentially to process a file.

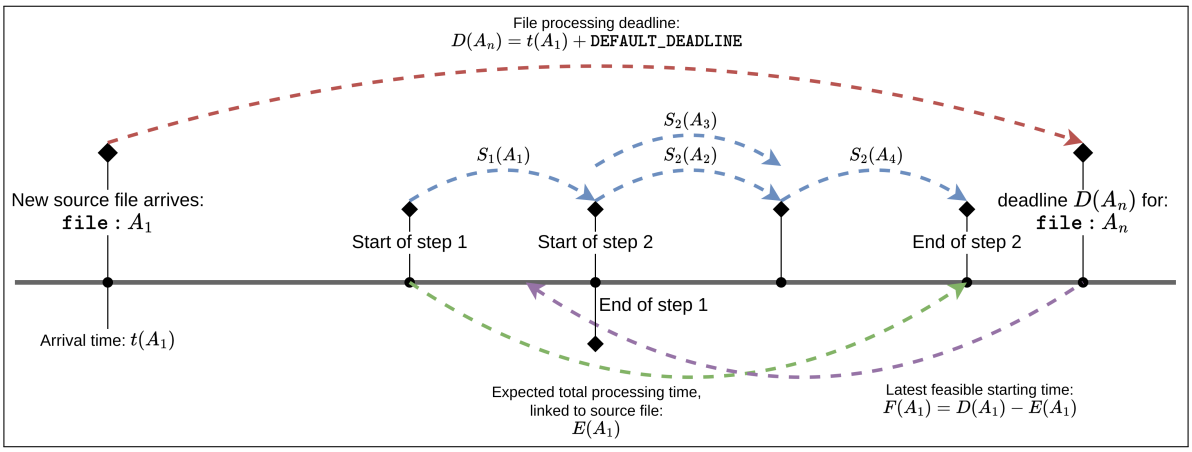
Concurrency is implemented by starting two or more `TaskQueueExecutors` in separate threads, where every executor is responsible for processing files from start to finish. The maximum concurrency is always directly controlled by the number of `TaskQueueExecutors` that are started. As every `TaskQueueExecutor` runs one pod at a time at maximum, there can be no more pods running than there are `TaskQueueExecutors` running.

Unfortunately, the scheduler is unaware of all the files that will appear from processing an arriving source file. In other words, when considering the conceptual scenario of Figure 5.4, only file A_1 is known at schedule-time. It is unknown how many more files will need to be processed by the subsequent steps. Therefore, the scheduler simply calculates the expected running time of the whole process, as that is how a task is defined anyway. Thus, from historical runs, it knows their average processing speeds in B/s. The scheduler simply divides the expected running time of a task by the maximum concurrency to get to an expected running time of the task through the whole pipeline. This does introduce some inaccuracies.

The first inaccuracy is that not all steps can process files concurrently. Specifically, the ING system has two steps at the beginning of the pipeline that deal with just one file. Those two steps cannot process files concurrently. However, as the calculated processing times are calculated from the total task processing times, this is taken into account when calculating the average processing speeds. Therefore, the system eventually converges on the true process-



(a) Scenario in which the maximum concurrency is set to 3



(b) Scenario in which the maximum concurrency is set to 2

Figure 5.4: Conceptual overview of task deadlines, expected total processing times, and latest feasible start times with different maximum concurrency settings.

ing speed of a single task through the pipeline and is able to predict the expected running time of a task accurately.

A second inaccuracy occurs when only a single task needs to be processed by the pipeline, containing a single file in all steps. In this scenario, the scheduler still divides the expected processing time by the maximum concurrency. However, not all steps will run concurrently, as there is only 1 task to process with one file in it. The system would, therefore, underestimate the running time of this task. This is not an issue for the ING system, as this is an unrealistic scenario. Tasks that are processed by the system process hundreds to thousands of files a day. Therefore, concurrent processing is still feasible.

5.2. Django

One of the requirements of the ING system is that the pipeline is managed by a centralized control panel, implemented in Django. Ideally, the scheduler is implemented as part of this control panel. Django promotes a modular design by means of stand-alone apps that are part of a bigger system that interface with each other and use the same database. Therefore, the scheduler is implemented as a stand-alone app that is able to interface with the Django control panel. Any additional models that are added to the system by the scheduler are also fully manageable from the Django administration control panel. In doing so, it is possible for the operations division of the TRCS squad to interact with the state of the scheduler. This part of the implementation is shown conceptually in the orange box in the middle of Figure 5.1.

The control panel serves as an API to interact with the pipeline and the files within it. Every processing step communicates with the control panel about files that need to be processed. Therefore, the scheduler is implemented as an app that communicates with the control panel itself but does not directly communicate with processing pods. Instead, the control panel gets extended to enable scheduling functionality. For example, files must be distributed over multiple processing pods when implementing concurrency. The control panel must be aware of this to communicate it correctly once processing pods are started and are ready to process files.

An additional requirement of the control panel is that it should be the single source of truth for all file states in the system. Therefore, the scheduler does not implement its own additional layer of file tracking and instead fully utilizes the control panel to keep track of file states, including possible additional states added by the scheduler. Any additional information regarding scheduling information is saved to the database and manageable by the control panel to keep the control panel the single source of truth.

5.3. Implementing scaling and gathering of resource metrics in OpenShift

The scheduler requires access to certain parts of the OpenShift environment to perform its primary duties. These parts are shown conceptually in the red box at the right of Figure 5.1. It requires this to scale deployments of processing pods, as well as to acquire resource consumption metrics from the running pods. This is handled by directly interfacing with the OpenShift API. This API provides direct access to the OpenShift environment, given that the correct rights are assigned to the pod. Interfacing with the OpenShift API is handled in the `scheduler/scheduling_app/openshift` package.

The scheduling pod needs to be assigned certain rights, named roles in OpenShift, to be

able to interface with the API. These rights are assigned by means of a `ServiceAccount`[1]. A `ServiceAccount` can be granted certain `Roles` by means of a `RoleBinding` and is defined in a `yaml` configuration file. This configuration file is shown in Listing 2 in Appendix C.

Similarly, gathering metrics on CPU and memory consumption by pods can be implemented by calling the OpenShift metrics API[2]. Again, the scheduling pod needs to be assigned certain rights by means of a `ServiceAccount`, assigned using a `yaml` configuration file, as shown in Listing 2 in Appendix C. Managing metrics measurements and querying them is fully managed by OpenShift and therefore requires little work to implement for the carbon-aware scheduler. This API exposes endpoints that allow the scheduler to retrieve CPU and memory percentages for every pod it manages.

With the `ServiceAccount` definition, the scheduler is able to perform its primary duties. It can scale up and down deployments in the OpenShift environment to start and stop processing pods, and it can call the OpenShift metrics endpoint to query all metrics for all the pods in the same namespace.

5.4. Calculating energy consumption of pods

To be able to calculate the greenhouse gas emissions of the pipeline, data on kilowatt-hour (kWh) consumption is required for every processing pod. Once the total kWh consumption for a pod is known, it can be converted into carbon emissions, as will be explained in Section 6.3. CPU and memory usage directly influence the electricity consumption of pods and are therefore used for the calculation of a pod's kWh consumption.

The process of acquiring kWh measurements requires two main parts. First of all, for every pod that is started, metrics are collected on its CPU and memory consumption. Secondly, an ING tool converts these metrics into the expected kWh consumption. This is encapsulated in the package found in the repository at `scheduler/energy_calculator`. The process is explained in the next few paragraphs and describes the parts shown in the purple box at the bottom of Figure 5.1.

OpenShift keeps track of CPU and memory consumption statistics for every pod in the system and provides this data to external tools using an API. The scheduler must interface with this metrics API to gather the CPU and memory consumption statistics of pods. It requires certain rights to be able to do this, as explained in the previous section. A conversion to kWh consumption can be performed based on these metrics.

The conversion to kWh uses an ING-provided tool. This tool converts CPU and memory percentage measurements to kWh consumption. It determines the expected energy consumption based on measurements within the ING private cloud data centers. In other words, based on real-world measurements in the ING data centers, it is able to calculate kWh consumption based on CPU and memory statistics. The tool is configurable based on platform and hardware configuration. It takes into account the average minimum and maximum watts consumed by a platform, along with the specific hardware configuration on which the pods run.

The scheduler measures the kWh consumption roughly every second, which introduces a lot of data points. Therefore, every thread that collects these measurements saves them to memory and waits until a pod has completed processing. The measurements are batched together as one database call once a pod has stopped to prevent overloading the database with calls.

[1] https://docs.openshift.com/container-platform/4.14/authentication/understanding-and-creating-service-accounts.html

[2] https://docs.openshift.com/container-platform/4.14/monitoring/managing-metrics.html

6

Methodology, results, and evaluation

This chapter presents the methodology, results, and evaluation of the carbon-aware scheduler implemented in this thesis. It begins by explaining the creation of a simulated environment with synthetic processing steps to mimic the ING system's workload, as the system from ING incurred significant delays in the refactor.

The evaluation section, outlined in Section 6.4, elaborates on the methodology utilized to evaluate the carbon-aware scheduler's performance. The evaluation contains various tests on the simulated data pipeline related to the accuracy of task load predictions, the quantification of carbon emission reductions, and expected overhead by implementing a carbon-aware scheduler. This chapter provides an overview of the simulation process and the subsequent evaluation of the scheduler's effectiveness in meeting the research objectives.

6.1. Simulating the ING processing pipeline

Initially, this research aimed to create a scheduler that would optimize an ING system by integrating it into the data processing pipeline of the trade and communications surveillance (TRCS) squad. At the start of this research, the TRCS squad had already started refactoring to the improved system, as explained in Section 2.2. Due to various delays, the team could not finish the refactoring as they had planned before the research evaluation phase for this research started. As a result, it was necessary to simulate the ING system, enabling the scheduler to execute its tasks and collect metrics about them.

It was initially believed that the refactored system would still be available on time to perform final evaluations in this real-world system. Therefore, the simulation of the ING system had to be as life-like as possible to ensure it could be easily adapted to the real system once it was ready. Additionally, if the simulation is very life-like to the ING system, it would adhere to the generic requirements set out in Section 2.3.

Due to the system not being fully operational, it was determined that it would be best for this research to create a similar system with simple synthetic processing steps. These synthetic processing steps would take as input some form of textual communications data on which they had to perform some resource-intensive processing steps to simulate the workload. Additionally, the simulated system was to be run in a containerized environment, with every processing step having its dedicated processing pod and a single Django control panel pod that orchestrated the simulated pipeline.

6.1.1. Gathering synthetic data

The first step of simulating the system involved creating synthetic data inputs. The first attempt of creating synthetic batch data from these pickle files was unsuccessful, as is explained in Appendix A. The correct methodology will be explained in the next few paragraphs.

As the ING pipeline processed communications data, the synthetic data had to be structurally similar to actual communications data. Therefore, the dataset chosen is a Twitch[1] chat dataset. This dataset is a collection of chat logs of 2,162 Twitch streams by 52 streamers, ranging from April 24th, 2018, to June 24th, 2018 [11]. It was chosen because it contains textual chat logs in which multiple people are conversing with each other. Therefore, it accurately represents one of the significant types of input data used by the ING system. Additionally, it contains chat logs over a long period of time, thereby providing a large variance in the number of chat logs per day. It contains 52 pickle files, a binary file format enabling high compression on various data types. The file sizes of the pickle files vary greatly, ranging from 23.0 MB up to 1.7 GB in compressed format.

Column name	Description	Python data type
`body`	Actual text for user chat	`str`
`channel_id`	Channel identifier	`int`
`commenter_id`	User identifier	`int`
`commenter_type`	User type	`str`
`created_at`	Time of when chat was entered	`datetime`
`fragments`	Chat text including parsing information of Twitch emotes	`dict`
`offset`	Time difference between the start time of video the stream and the time of when chat was entered	`float`
`updated_at`	Time of when chat was edited	`datetime`
`video_id`	Video identifier	`int`

Table 6.1: Overview of columns in the Twitch Chat dataset [11]

This data is then transformed into batches, with every batch representing a synthetic day's worth of data input. These batches are created according to the following steps.

1. All the pickle files are unpickled, returning the raw chat data in the format shown by Table 6.1. Some unused columns are dropped, and the remaining columns are: `body`, `channel_id`, `commenter_id`, `created_at`, `video_id`. The other columns are removed to reduce the size of the files.
2. Every single chat file is then grouped by date, based on the `created_at` column. Specifically, they are grouped by year, month, and date, but not on time.
3. Then, for every day of chat data, files are split into smaller chunks. Every chunk contains 40 to 100 chats at random. The randomly sized chunks are then saved as tab-separated values (TSV) files.
4. Lastly, the resulting TSV files are zipped into four equally sized zip files. These then represent one day's worth of batched input.

[1] https://twitch.tv

6.1.2. Creating synthetic data processing steps

For the sake of the simulation, only a few processing steps were simulated to generate enough synthetic workload for evaluating the scheduler. Therefore, the simulation that was created is based on the first three data processing steps performed on the Bloomberg (BB) chat data in the real system. These files undergo the following processing steps sequentially:

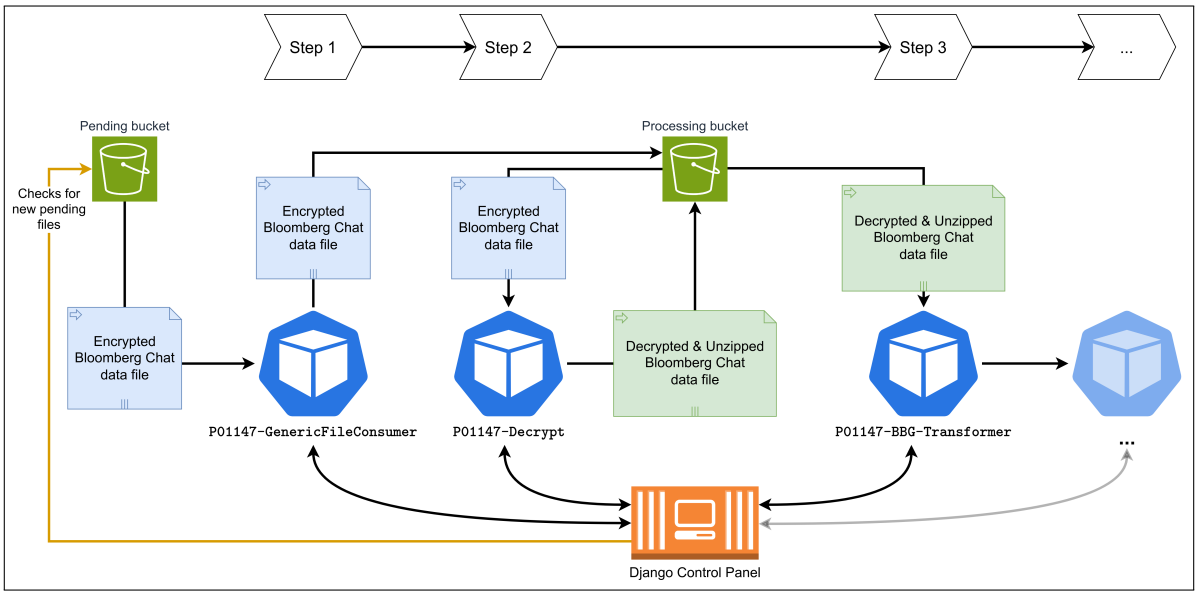
1. The first processing step is performed by the `P01147-GenericFileConsumer` pod, which is a processing step that simply moves files from the `pending` bucket to the `processing` bucket after performing a data integrity check. It then registers them to the Django control panel as new files to be processed.
2. The second step is a decryption step, performed by the `P01147-Decrypt` pod. As the name suggests, this step decrypts the file such that it can be read out by the next processing steps. Additionally, if the input file is a `tar.gz` file after decryption, it performs an untar operation on the file and uploads all the resulting files to the `processing` bucket.
3. The third and last step considered is performed by the `P01147-BBG-Transformer` pod. This step extracts conversations from the XML files presented by the BB Chat export files. It then transforms them to a data structure such that it eventually can be parsed by the Relativity tool to perform alert generation on the chat logs. This is the first step that works on the chat log data in an uncompressed format.

The following synthetic simulations are created to simulate this workload using the Twitch chat dataset.

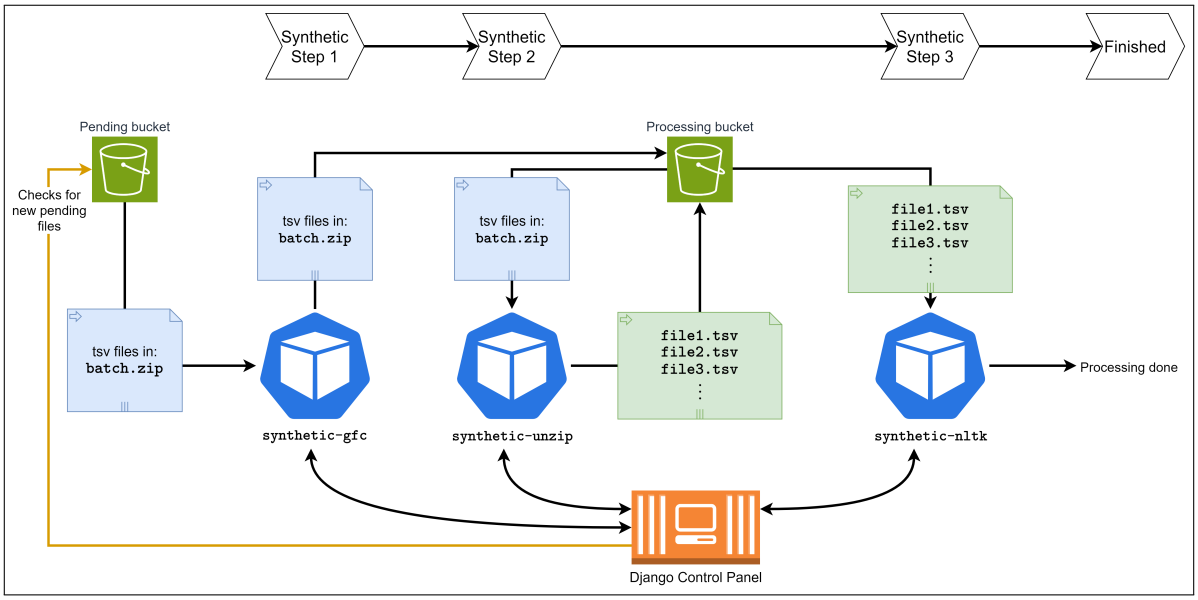
1. First of all, to simulate the `P01147-GenericFileConsumer` step, a similar `synthetic-gfc` step is created. This synthetic GFC step does exactly the same as the original processing step; it moves a file from the `pending` bucket to the `processing` bucket. However, it does not perform a data integrity check.
2. The next step is an unzipping step, in which all the input files that have been moved from the pending bucket to the processing bucket are unzipped. The synthetic data is not encrypted, but it does not need to be for the second step to be heavy enough to simulate some workload. Once the files are decompressed, the resulting TSV files are again uploaded to the `processing` bucket.
3. The last synthetic step is performing some natural language processing steps on the chats in the various TSV files. In this step, every chat log is analyzed using a natural language model provided by the Natural Language Toolkit (NLTK). For every chat, the words are analyzed and compared to each other. Using this comparison, the word that matches the other words in the chat the least is chosen. Additionally, on every chat, a sentiment analysis is performed. This sentiment analysis returns a dictionary containing the calculated positive, negative, and neutral scores for the given chat.

Figure 6.1 displays a graphic representation of the real pipeline at ING and its synthetic counterpart used to evaluate the real pipeline. As can be seen, the data flows through the pipeline in a similar way, according to the descriptions given previously.

The largest difference between the real and simulated pipeline is that the real pipeline has more than three steps to perform on the BB Chat files, but these are not simulated. Simulating these steps is not required. The simulated workload could be easily increased by feeding the pipeline more files if a larger workload was needed for a simulation.



(a) TRCS pipeline.



(b) Synthetic pipeline.

Figure 6.1: Graphic representation of the processing steps and data flow through the pipeline for the BB Chat source files and its synthetic counterpart used to simulate the workload of the real pipeline.

6.2. Calculating energy consumption of containers in the synthetic pipeline

CPU and memory consumption statistics for every container are required to calculate their kilowatt-hour (kWh) consumption. Initially, these could be retrieved from the OpenShift API, as explained in Sections 5.3 and 5.4. Unfortunately, the synthetic pipeline does not run in OpenShift but in a separate docker environment. Therefore, these metrics could not be extracted by calling the OpenShift API. Instead, they needed to be collected by directly collecting CPU and memory statistics from the docker containers in the synthetic pipeline. Even though docker provides an API for metrics collection, it did not provide CPU and memory statistics percentage values. Therefore, these had to be calculated manually. Luckily, the API does provide counts of CPU cycles consumed by the containers and the system, as well as memory usage values. From these numbers, the percentage values for CPU and memory consumption could be calculated to be used as input for the ING tool to calculate kWh consumption for every pod. The calculation of CPU and memory percentages can be found in the repository at `scheduler/scheduling_app/openshift/docker_client.py`.

The measurements are taken by polling the docker metrics API, which returns a stream of these metrics. For every pod needing monitoring, the scheduler starts a separate thread to monitor them individually. The stream updates roughly every second for every pod, including timing information.

Saving the timing information alongside the CPU and memory utility values is crucial as this is required to calculate a kWh consumption value. This timing information is required because the CPU and memory percentages can be converted into a wattage, a measure of power flow or energy consumption rate. However, a measure of total energy used is required to convert into greenhouse gas emissions, such as a kWh. Greenhouse gas emissions are always calculated from the total energy consumption at a specific time, not from some measure of power such as watts or kilowatts.

6.3. Acquiring information on greenhouse gas emissions per kWh

Data is required on greenhouse-gas emissions per consumed kWh to evaluate the predictions of low-carbon energy generation periods, as well as the expected reduction in greenhouse-gas emissions by using the carbon-aware scheduler. The data source that provides data on greenhouse gas emissions is Electricity Maps.

Electricity Maps provides data on the environmental impact of energy generation in a specific region at a given time. The degree of cleanliness in electricity generation indicates the amount of greenhouse gases emitted per kWh of electricity generated and consumed. This metric is quantified in grams of CO_2 equivalent emissions per kWh (gCO_2-eq/kWh). The platform factors in various greenhouse gases, converting their emissions to an equivalent CO_2 value based on their global warming impact over a 100-year period [15].

The calculation of grams of CO_2 equivalent emissions per kWh (gCO_2-eq/kWh) on Electricity Maps involves a Life-Cycle Analysis (LCA) approach. LCA is a methodology that assesses the environmental impact of a product or process throughout its entire life cycle, from raw material extraction to production, distribution, use, and disposal. In the context of electricity generation, this means considering emissions not only from the direct combustion of fuels during electricity production but also from activities such as fuel extraction, transportation, and plant construction. For example, the analysis takes into account the extraction of metals like iron for manufacturing steel used in windmills, as well as the anticipated greenhouse gas emissions associated with both the construction and dismantling of the windmill.

The data provided by Electricity Maps contains gCO_2-eq/kWh emissions per hour for 2021 and 2022 in The Netherlands.

Column name	Example value
Datetime (UTC)	2021-01-01 00:00:00
Country	Netherlands
Zone Name	Netherlands
Zone Id	NL
Carbon Intensity gCO eq/kWh (direct)	385.02
Carbon Intensity gCO eq/kWh (LCA)	480.26
Low Carbon Percentage	28.56
Renewable Percentage	21.88
Data Source	Electricity Maps Estimation
Data Estimated	true
Data Estimation Method	ESTIMATED_MODE_BREAKDOWN

Table 6.2: Table representing the data structure of gCO_2-eq/kWh emissions per hour provided by Electricity Maps.

6.4. Evaluating the scheduler

This section describes the methods to answer the research questions and evaluate the scheduler. It briefly describes the various tests performed using the simulated data pipeline. Before the various tests are explained, an explanation of some basic shared functionality is given.

Not all tests require delaying the execution of tasks until the 'optimal' time. For instance, a test that checks whether the scheduler converges to the system's processing speed does not require tasks to be executed during green hours. This test runs tasks through the system sequentially to gather data about task execution times and the median processing speed determined by the system. Its sole purpose is to gather data about task running time predictions.

A modification was made to the `TaskQueueExecutor` class to increase the speed of the tests and to reduce the amount of idling during tests. The `TaskQueueExecutor` class is detailed in Section 4.3. The modification adds an optional argument to the `execute_task_queues` function of the `TaskQueueExecutor`. This new argument is named `instantly`; by default, its value is `False`. Whenever this argument is set to `True`, the executors' filter is modified slightly to always consider `TaskQueues` that have pending files to be processed, even when their starting times have not passed yet. An obvious side-effect of using this is that files are not processed during optimal times in terms of CO_2 emissions. It is, therefore, not used during tests that gather information regarding the effectiveness of reducing CO_2 emissions and should never be used except to evaluate the scheduler.

The evaluation aims to answer the following research questions.

Research Question 1

"To what extent can task load be predicted regarding running times?"

Research Question 2

"How can predictions on solar and wind energy generation, as well as energy consumption, be utilized for implementing a carbon-aware scheduler?"
a) "Can the scheduler predict periods of low carbon intensity effectively?"
b) "Compared to the existing system, how much reduction in carbon emissions can be achieved by using a carbon-aware scheduler?"

Research Question 1

"Is there any overhead introduced by implementing a carbon-aware scheduler?"

6.4.1. Determining the accuracy of task running time predictions

A comparison should be made between predicted times and actual processing times of tasks to determine the accuracy of task processing time predictions. The accuracy of the predictions is calculated by determining the difference between the expected duration prediction and the actual duration. When a task is scheduled, the `expected_duration_at_schedule_time` property is saved to its accompanying `Metric` entity. This value represents the expected duration of the task at schedule-time based on previous runs and task properties. Detailed information on how this value is calculated is explained in Section 5.1.

The `duration` property is a calculated value based on the `start_time` and `end_time` properties. Logically, the value of the `duration` field is simply the difference between the start and end time:

$$\texttt{duration} = \texttt{end\_time} - \texttt{start\_time} \tag{6.1}$$

This value can be calculated the moment a task has finished processing.

Once a task has finished processing, the `percentage_error` property can be calculated using the `expected_duration_at_schedule_time` and `duration` properties. Both these properties are stored and available in the `Metric` class once a task has finished processing. The `percentage_error` property represents how much the scheduler has overestimated or underestimated the running time of the task. It is calculated using the following formula and then rounded to two decimals:

$$\texttt{percentage\_error} = 100 \times \frac{\texttt{expected\_duration\_at\_schedule\_time} - \texttt{duration}}{\texttt{duration}} \tag{6.2}$$

The percentage error is a positive value whenever the scheduler has overestimated the running time of a task. Thus, if the scheduler predicted a task to run for 10 seconds, but in reality, it took 5 seconds, the resulting percentage error value is $100 \times \frac{10-5}{5} = 100\%$. In other words, it overestimated the running time of the task by 100%. Conversely, the value is negative if the scheduler has underestimated the running time of a task and it took longer in reality. In an ideal scenario, this value is as close to 0 as possible.

Another valuable metric is the median processing speed of the system at the moment a task's running time is predicted. It is especially interesting to gather the speeds over time whilst more tasks are run through the system to see if the scheduler converges to the median processing speed. The calculation of the median processing speed on which the scheduler makes decisions is explained in Section 5.1.

Tests

The following steps are performed at a high level to determine if the running time can be predicted accurately:

1. Assume that the database is completely empty and that there are no pending files in the `pending` S3 bucket.
2. For every synthetic test file:
   (a) Clean the `pending` and `processing` S3 bucket to save disk space during the tests. This step removes any files that previous runs of this test have left. It also ensures that there aren't any residual files left, which the scheduler may incorrectly pick up in its schedule.
   (b) Upload the test file to the `pending` S3 bucket.
   (c) Call the scheduler to determine a schedule. This will result in it creating one `TaskQueue` containing the single test file.
   (d) Create one `TaskQueueExecutor`, let it process the file immediately by setting the `instantly` flag to `True`.
3. After all files have finished processing sequentially:
4. Extract all `Metrics` from the system and save them to the file system for analysis.

All the random files are processed sequentially, one after the other. Due to this, every next file that is processed by the system has more historic runs to determine the median processing speed of the system. The processing speed predictions should increase in accuracy as more files are processed by the test over time.

The extracted data is stored in a pandas `DataFrame` [23] and saved to a `data.csv` file in `scheduler/synthetic/results/get-processing-speed-converging`. Table 6.3 shows the data columns in this file, with their respective `Python` data types and an example value.

Column name	Data type	Example value
source_file_id	int	142937
start_time	datetime	2023-11-15 23:19:11.863008+00:00
end_time	datetime	2023-11-15 23:19:30.903780+00:00
expected_duration_at_schedule_time	timedelta	0 days 00:00:20.015413
max_concurrency_at_execution_time	int	1
duration	timedelta	0 days 00:00:19.040772
processing_speed	float	33354.80655089834
percentage_error	float	5.12
difference_with_deadline	timedelta	-1 days +23:58:19.100683
file_path	str	pending/2018-03-07_1.zip
file_size	int	1077073
total_kwh_used	float	9.601510e-05

Table 6.3: Column names, `Python` data types, and example values for the results saved by the `get_processing_speed_converging.py` test.

Results

A total of 468 tasks were processed by the system, where every task contained one synthetic test file as described in Section 6.1.1. Of those tasks, 467 were completed successfully. The maximum concurrency during this test was set to 1; therefore, at all times, only one task containing one synthetic test file was processed at a time. A full overview of all numeric values is plotted in a scatterplot grid in Appendix B.

Figure 6.2 shows the evaluation of the processing speed predictions over time. The top graph displays the tests grouped into 10 bins. Every bin, therefore, contains 46-47 tests grouped together based on their starting time. From left to right, the graph shows the performance of the predictions over time, with the first bin containing the oldest tests and the last bin containing the most recent tests. The y-axis shows the percentage error of the predictions, as calculated by Equation (6.2). The bottom graph shows a plot of the percentage errors of the tests as a line, also ordered by their starting times from left to right. The most recent tests have more historical data available to them and, therefore, take more tasks into account when calculating the expected processing speed.

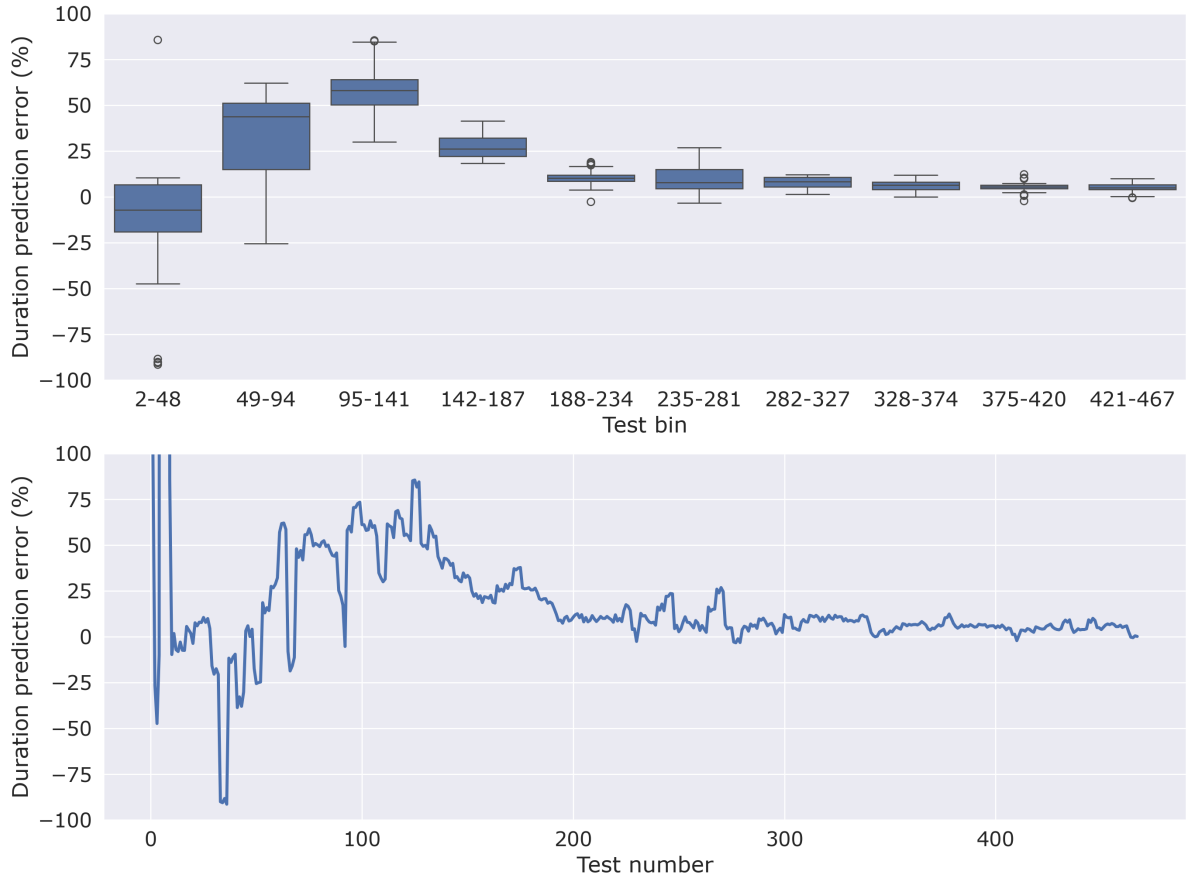


Figure 6.2: Percentage errors of tests over time. The top plot shows a boxplot in which tests are divided into 10 equally sized bins. The bottom plot displays the percentage errors of all tests over time. Tests are ordered by their starting time from left to right.

The graphs in Figure 6.2 clearly show that the scheduler's predictions improve over time as more tasks have been processed by the system. The first few tests heavily overestimate and underestimate the running times of new tasks, explaining the very jittery lines for the first few tests. After roughly 100 tests, the system starts to converge to a lower error rate. Eventually,

the system accurately predicts tasks with an error percentage between 5 and 10%. To put this into perspective, a 5 to 10% error results in an error of 12 to 24 minutes if the expected running time of a day's tasks is 4 hours.

Table 6.4 shows the data from the first 12 tests that were performed. Note the rows marked in red. These rows indicate that test numbers 5 through 8 overestimated the running times up to 2540% but quickly subdued back to a much smaller percentage error from row 9 onwards. A percentage error of 2540% indicates a very large overestimation and requires investigating.

	Durations		
Test	**Prediction**	**Actual**	**Percentage error**
1	00:00:27.862000	00:00:13.498787	106%
2	00:00:09.704282	00:00:13.258699	-27%
3	00:00:07.049457	00:00:13.420734	-47%
4	00:00:12.020205	00:00:13.354103	-10%
5	00:15:31.698940	00:00:35.290904	2540%
6	00:14:22.433027	00:00:35.317646	2342%
7	00:11:35.915645	00:00:35.317486	1870%
8	00:10:04.647016	00:00:35.828439	1588%
9	00:00:55.264734	00:00:29.790370	86%
10	00:00:28.216634	00:00:31.297615	-10%
11	00:00:31.185316	00:00:30.686416	2%
12	00:00:28.154662	00:00:30.334998	-7%

				Processing speeds	
Test	**Processing speed**	**Percentage error**	**File size (KiB)**	**Mean**	**Median**
1	2.064	106%	27,2	2.064	2.064
2	1.511	27%	19,6	1.787	1.787
3	939	47%	12,3	1.505	1.511
4	1.360	10%	17,7	1.468	1.435
5	37.892	2540%	1305,9	8.753	1.511
6	36.890	2342%	1272,3	13.443	1.787
7	35.219	1870%	1214,7	16.554	2.064
8	34.833	1588%	1218,8	18.838	18.449
9	34.224	86%	995,7	20.548	34.224
10	30.855	-10%	943,1	21.579	32.540
11	33.069	2%	991,0	22.623	33.069
12	30.692	-7%	909,2	23.296	31.962

Table 6.4: Data from first 12 tests of the test to determine if the processing speed converges.

The source of these overestimations can be found in the bottom table when looking at the `File size (KiB)` column. When starting test number 5, the scheduler only had information from tests 1 through 4 available to determine the median processing speed. As can be seen, tests 1-4 contained much smaller files as input. Tests 1-4 had a processing speed between 939 and 2.064 bytes per second (B/s), whereas tests 5-9 had a much higher processing speed of around 35.000 B/s.

This can be explained by the overhead of starting up a container for every step in the process-

ing pipeline. For example, the third step performs some natural language processing on the chat data. For this step, the NLTK `Python` package needs to download some datasets into the container. For every step, this happens once when the pod starts up before processing all files of that test. This adds some overhead to the test, as it needs to download this tooling into the docker container before being able to process all the chats of that task. This overhead reduces the processing speed of tests with very small input files, as the preparation steps of the task require more time proportionally than the actual processing compared to tests with larger input files. However, as can be seen in the bottom table, over time, the mean and median processing speeds actually recover to normal values. Therefore, the median processing speed will eventually take into consideration the processing speeds of large as well as small input files.

Figure 6.3 displays a scatterplot of task processing speeds over their file sizes. The x-axis represents the file size in kibibyte (KiB), and the y-axis represents the processing speed in bytes per second (B/s). The relatively large overhead for smaller files is clearly visible in this plot. As the input file size increases, the processing speed initially increases quickly but then levels off as the file size increases further. This relationship roughly follows a logarithmic curve, which is expected since the relative overhead of a task's setup time decreases as the file size increases.

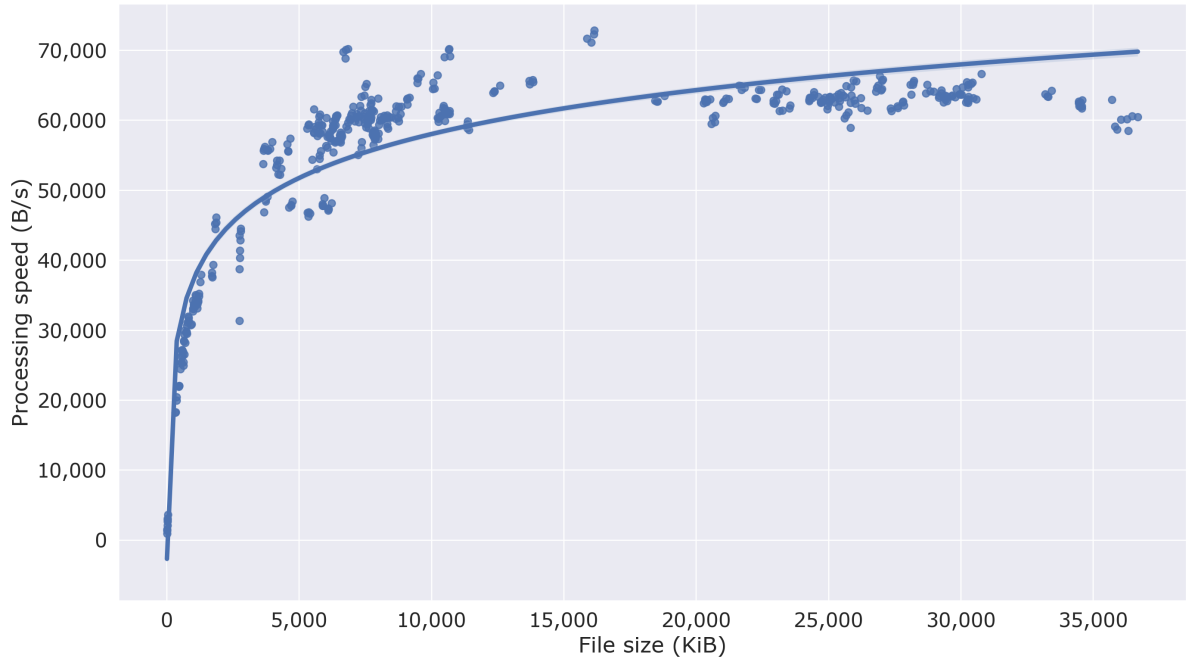


Figure 6.3: Scatterplot of processing speed over task file size.

6.4.2. Determining the accuracy of predictions of low-carbon energy generation periods

The scheduler aims to schedule tasks at periods during which the energy generation has a low carbon intensity. It uses predictions on solar and wind data for this, as is described in Section 4.3. These predictions must be evaluated for their accuracy.

Data from Electricity Maps is used to evaluate the predictions. This platform provides historical data on greenhouse gas emissions per generated kWh, as explained in Section 6.3. This data can be used to show a correlation between renewable energy generation predictions and

actual greenhouse gas emissions.

The scheduler predicts a 'renewable energy percentage' value to determine optimal times to schedule processing jobs. This value represents how much of the total energy consumption in The Netherlands is generated by renewable sources, such as solar and wind energy. A more in-depth explanation of this value is given in Section 4.2. It should be tested whether there is a correlation between this value and the actual gCO_2-eq/kWh emissions.

Figure 6.4 plots the relationship between predicted 'renewable energy percentage' and actual gCO_2-eq/kWh emissions for 2021, per season. The x-axes of these graphs are the predictions on renewable energy percentage available for every 15 minutes in 2021, provided by the European Network of Transmission System Operators for Electricity (ENTSO-E) transparency platform. The y-axes of these graphs are the actual gCO_2-eq/kWh emissions at those same times, gathered from Electricity Maps. The resolution of energy generation predictions is 15 minutes, but the resolution of gCO_2-eq/kWh emissions is hourly. Therefore, these values have been interpolated to align with the 15-minute intervals of the energy generation predictions. Figure 6.4 appears to show a logarithmic correlation between the two values.

To prove this correlation, a regression fit can be performed on the data. In this case, the independent variable is set to the renewable percentage calculated by the scheduler, and the dependent variable to the measured gCO_2-eq/kWh by Electricity Maps. An ordinary least squares (OLS) regression method is performed for every season on these data points.

The line that is fitted by the OLS method for one independent and one dependent variable can be described by the following equation:

$$Y = \beta_0 + \beta_1 \ln X_1 + \varepsilon, \tag{6.3}$$

where Y is the dependent variable, β_0 is the intercept term, X_1 is the independent variable, β_1 represents the coefficient for X_1, and ε is the error term. In this scenario, X_1 is the value of the 'Renewable percentage' prediction by the scheduler, and Y is the accompanying expected gCO_2-eq/kWh emissions. To prove there is a correlation, we set the null hypothesis to the scenario in which the renewable percentage has no influence on the measured expected greenhouse gas emissions, meaning it does not correlate. In other words, the β_1 coefficient of the 'Renewable percentage' value (X_1) should be 0. We can then define the hypothesis as follows:

$$H_0 : \beta_1 = 0 \tag{6.4}$$
$$H_1 : \beta_1 \neq 0 \tag{6.5}$$

An OLS linear regression can be performed, and the relevant values can be calculated to determine if the null hypothesis can be rejected by using the `statsmodels` [21] `Python` package. The results of this test for all four seasons are shown in Table 6.5. For every season, the β_0 and β_1 values are shown, along with their standard errors. Next to that, the t-statistic is shown, with its accompanying p-value. We define the significance level to be $\alpha = 0.05$. If the p-value is less than the significance level α, the null hypothesis is rejected. As becomes clear from Table 6.5, we can conclude that the renewable percentage is a good indicator to use for predicting periods of low carbon load in all seasons. For all seasons, the p-value is less than 0.0001, thereby being less than the significance level of $\alpha = 0.05$. This means that the null hypothesis of $H_0 : \beta_1 = 0$ can be rejected for all seasons, implying there is a correlation between the predicted renewable percentage by the scheduler and the actual carbon load of electricity generation at that time. Therefore, it can be concluded that the 'renewable energy

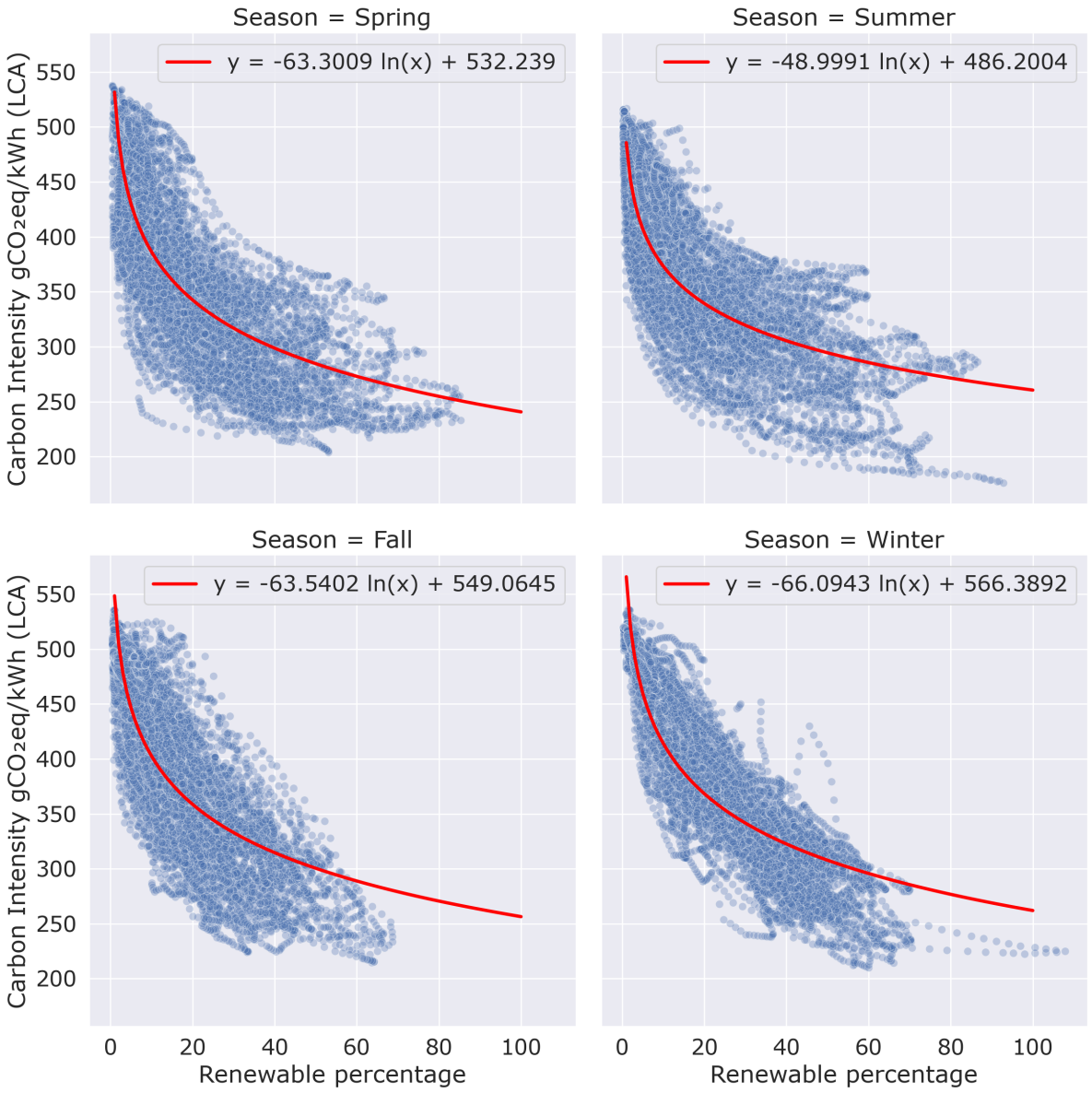


Figure 6.4: Scatterplot of carbon intensity of electricity generation in gCO_2-eq/kWh, over renewable energy percentage available per season.

percentage' prediction is an effective value for determining optimal times to schedule tasks in terms of carbon intensity.

Season	Parameter	Coefficient	Standard Error	t	$P > \|t\|$
Spring	β_0	532.2390	1.289	412.769	0.000
	β_1	-63.3009	0.463	-136.641	0.000
Summer	β_0	486.2004	1.086	447.653	0.000
	β_1	-48.9991	0.393	-124.730	0.000
Fall	β_0	549.0645	1.329	413.100	0.000
	β_1	-63.5402	0.489	-130.066	0.000
Winter	β_0	566.3892	1.304	434.509	0.000
	β_1	-66.0943	0.418	-158.114	0.000

Table 6.5: Results of the hypothesis tests to determine if 'renewable percentage' predictions by the scheduler correlate with the gCO_2-eq/kWh emissions as calculated by Electricity Maps.

6.4.3. Determining reduction of greenhouse gas emissions

Arguably, the most important evaluation to perform on the carbon-aware scheduler is evaluating the amount of reduction in CO_2 emissions it can achieve, if any at all. An extensive full-system test was set up to evaluate the scheduler's performance, which will be explained in this section. First, an explanation of why season averages were used is given. Then, the tests that were performed are explained, followed by their results.

Using season averages for expected carbon emission reductions

The tests to determine the expected CO_2 reductions use season averages for electricity generation statistics and carbon intensity. There are two main reasons for this. Firstly, it is the purpose of the tests to estimate the expected reduction of CO_2 emissions throughout the four seasons of the year. The weather at the time of conducting the tests should not impact the results. Secondly, it allows for a predictable input to the scheduler. This allows for testing the system with and without the scheduler, using exactly the same input into the scheduler regarding electricity generation statistics. The averages of renewable energy generation statistics, such as predicted wind and solar generation, are based on data by ENTSO-E. Section 4.2 provides an explanation of how this data is gathered and shows the averages per season in 2022.

To determine the average emissions per kWh generation in The Netherlands, the data from Electricity Maps [15] is used. This is a platform that provides predictions on gCO_2-eq/kWh emissions, as well as historical emissions per kWh for a certain bidding zone or country. The latter data was used to calculate the average emissions per season per time of day in The Netherlands. Whilst the predictions on emissions are part of a paid plan, the historical data is provided for free for research purposes. More details about the data that Electricity Maps provides is given in Section 6.3.

Tests

All the gathered synthetic data was processed by the system whilst measuring container kWh consumption to determine the expected reduction in greenhouse gas emissions. The differences in greenhouse gas emissions can be calculated using the season averages of gCO_2-eq/kWh emissions per consumed kWh. Section 6.1.1 explains the creation of the synthetic data used in this test.

The test data is actually provided by the same test runs as explained in Section 6.4.1. Briefly explained, the test simply ran every synthetic zip of batched test files through the system sequentially.

The system also calculated the consumed kWh for every pod. To do so, for every pod that is started, a separate thread is started by the scheduler, polling the docker API for CPU and memory consumption statistics. The total kWh consumption can be calculated for every pod from the CPU and memory consumption statistics using the same methodology as explained in Section 5.4. These values are calculated roughly every second and are saved separately to the file system after the test has been performed. The kWh measurements are also aggregated in the `Metric` class for every synthetic input file. In other words, the kWh measurements from every container that performed processing on the arriving synthetic source file, as well as any containers that processed any derived files from that source file are aggregated into a single metric.

It is essential to save the individual kWh measurements for each container, not just aggregates per metric. The main reason is that long-running tasks may take hours to process. Therefore, the expected gCO_2-eq/kWh emissions may vary throughout the processing time of that task. The synthetic tasks, however, did not have running times spanning hours. Figure 6.5 shows a histogram of the processing times of all the tasks during the tests, grouped into 10 bins based on their total duration. As can be seen, many tasks have running times of less than 200 seconds or between 400 and 500 seconds. The longest tasks ran for approximately 10 minutes. Therefore, the aggregated values in the `Metric` class can be used instead of using individual measurements to calculate the greenhouse gas emissions.

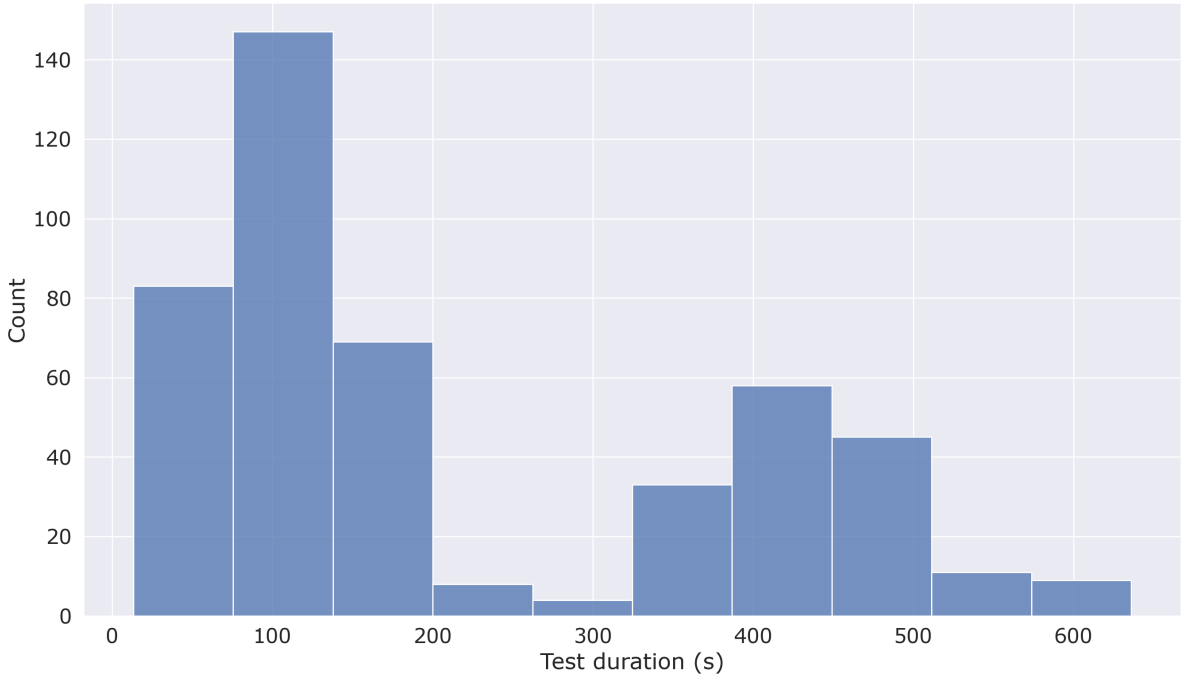


Figure 6.5: Histogram of the test durations in seconds.

Two scenarios are simulated to evaluate the expected reduction in greenhouse gas emissions by implementing the carbon-aware scheduler. The first scenario represents a system before the scheduler optimizes it, and the second scenario represents a system that uses the scheduler.

The first scenario is one in which the files arrive randomly throughout the day, with a uniform probability for every time of the day. In this scenario, a task is immediately processed upon arrival, as is the case in the real ING system. A histogram of the arrival times is plotted in Figure 6.6. A uniform probability for the arrival times is chosen in this scenario to reduce the influence of arrival times on the expected reduction in emissions. The arrival time of input data has a significant impact when comparing situations with and without the scheduler, as will be discussed in Section 7.3.

The second scenario chooses the optimal time of the day to "schedule" the processing of a task, and it simulates as if the scheduler delayed the execution of the task to this time. The optimal time of the day is the point at which the gCO_2-eq/kWh emissions are lowest. As shown previously, the scheduler can accurately predict those points in time. Utilizing this optimal value in this test provides an upper bound on the expected reductions by the carbon-aware scheduler and simulates a best-case scenario of implementing it.

Both simulations are performed for every season of the year, using the season averages of gCO_2-eq/kWh emissions in 2021. For both scenarios, cumulative emissions are calculated and ordered by the synthetic test date.

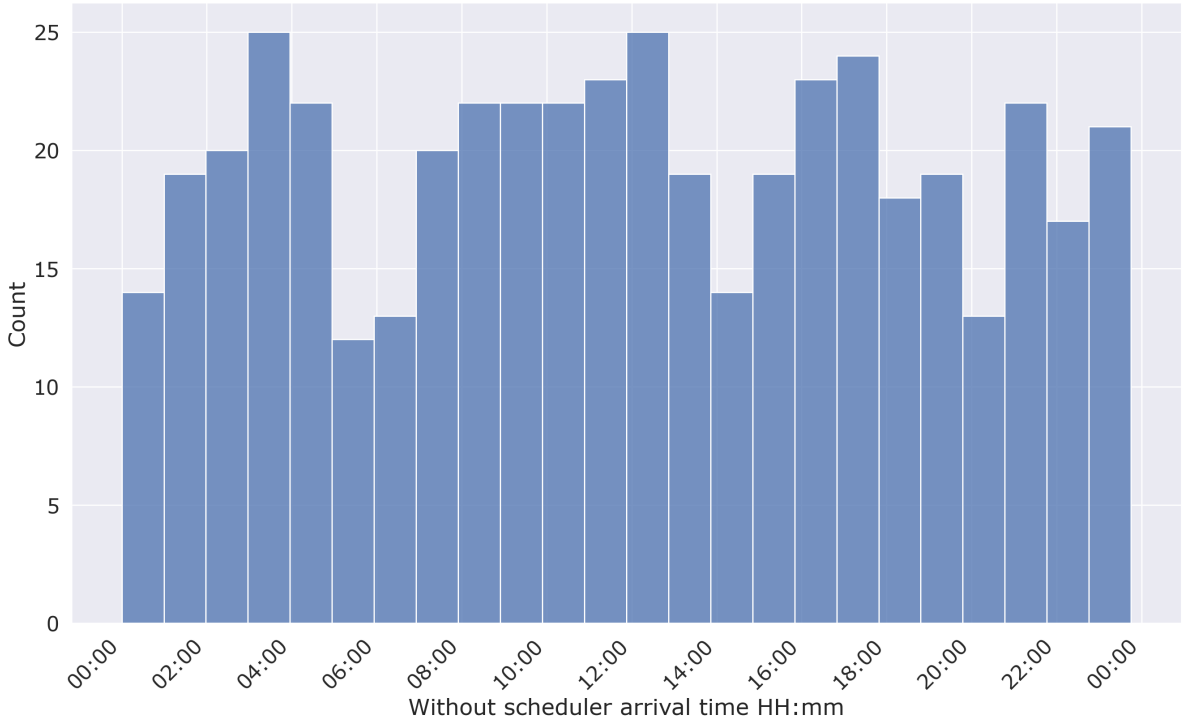


Figure 6.6: Histogram chart depicting counts of arrival and processing times in hourly intervals. The x-axis represents the time in 24-hour format (HH:mm), while the y-axis represents the count of arrivals.

Results

Figure 6.7 shows a side-by-side comparison of the cumulative grams of CO_2 equivalent emissions (gCO_2-eq) emissions by the system in both scenarios. The left graph shows the scenario without the carbon-aware scheduler, and the right graph shows the scenario using the scheduler. The x-axis displays the synthetic tasks, ordered by their date. The y-axis displays the cumulative gCO_2-eq emissions for both scenarios. The graphs show no discernable differences between seasons. It also shows that the system using the carbon-aware scheduler performs better than the system without it in terms of greenhouse gas emissions.

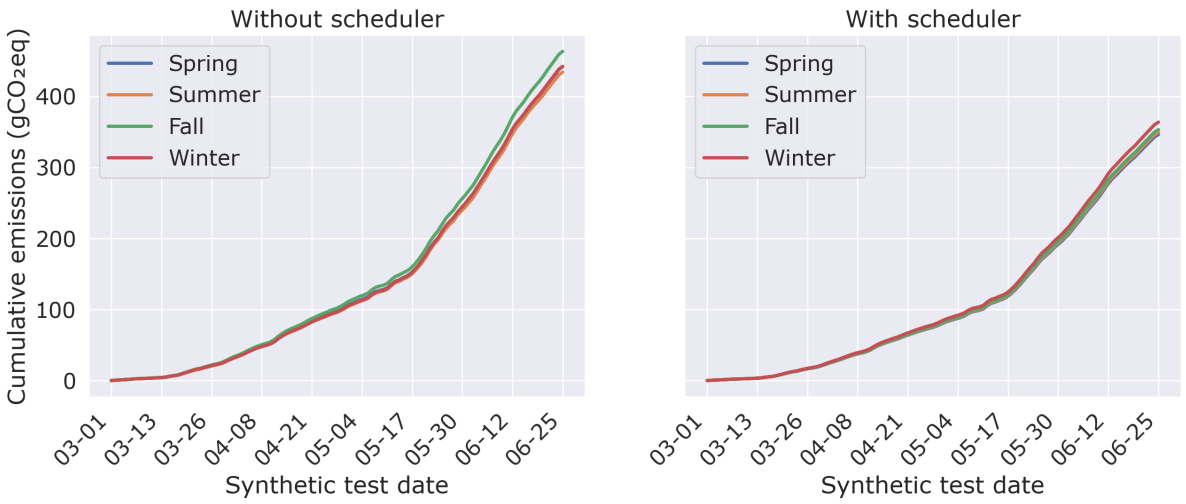


Figure 6.7: Side-by-side comparison of cumulative gCO_2-eq emissions by the system in scenarios without and with the carbon-aware scheduler. The left graph shows the scenario without the carbon-aware scheduler, and the right graph shows the scenario using the scheduler. The x-axis displays the synthetic tasks, ordered by their date. The y-axis displays the cumulative gCO_2-eq emissions for both scenarios.

Table 6.6 shows the calculated total emissions in gCO_2-eq for both scenarios, for all seasons. As can be seen, the scheduler is expected to save about 20% emissions compared to the scenario in which files arrive randomly throughout the day. However, it is not very usual for files to arrive completely random throughout the day for batch processing pipelines. Therefore, other scenarios for the arrival times are sketched in Section 7.3.

Season	Without Scheduler (gCO_2-eq)	With Scheduler (gCO_2-eq)	Difference (gCO_2-eq)	%
Spring	442.76	346.91	-95.85	-21.65
Summer	435.06	350.15	-84.91	-19.52
Fall	464.43	353.92	-110.51	-23.79
Winter	443.25	364.41	-78.84	-17.79

Table 6.6: Comparison of calculated total emissions in gCO_2-eq for the system using the carbon-aware scheduler and the system without. Results are separately calculated for the four different seasons.

The previously shown results only consider sequential processing. In other words, the batch data for a synthetic day is processed by one pod at a maximum at a time. However, with long-running tasks spanning multiple hours, the gain could be potentially increased by running tasks concurrently at the optimal times. Depending on the system configuration and the task size, there will be a tipping point at which starting up more containers introduces more overhead than gain. This will be considered in the next section, Section 6.5.

6.5. Regarding task processing overhead

The expected reduction in carbon emissions from Section 6.4.3 only considers a scenario in which the tasks are processed sequentially by one pod. However, as is explained in Section 4.3, the potential gain could be increased by scheduling and processing tasks sequentially. Theoretically, the optimal solution for the scheduler to minimize greenhouse gas emissions would be to start up an infinite number of containers. In that scenario, all the tasks would be instantly processed at the optimal time for the lowest greenhouse gas emissions. However,

overhead will be introduced as more processing is done simultaneously. Eventually, there will be a tipping point at which adding more concurrency does not reduce the amount of emissions, as the overhead introduced will be too large. This tipping point is influenced by various factors, such as the task size, the systems hardware specifications and kWh consumption, and the overhead introduced depending on the task.

The first potential overhead to consider is that of the scheduling algorithm itself. However, the scheduling decisions are very simple. Therefore, the added overhead by the scheduler is negligent. The scheduler takes less than a second to determine the optimal schedule for a full day's worth of tasks to process; it is, therefore, inconsequential to the overall processing time and resource consumption.

Secondly, the overhead introduced by parallel processing should be considered. Whenever any system introduces parallel processing or multi-threading, there is an overhead associated with it. The source of this overhead depends on the system and the processing done. For example, there needs to be communication between different parts of the system to ensure synchronization of the processing and prevent deadlocks. At some point, the overhead added by parallel processing will outweigh the benefits of the parallel processing itself.

Processing of the synthetic data has also been tested using concurrency. In this case, the maximum concurrency was set to 4, meaning that at any point, four pods could be running simultaneously to process a single day's worth of synthetic test data. The introduced overhead has already been shown in Section 6.4.1. Specifically, Figure 6.3 clearly shows that the introduced overhead is proportionally much higher for small tasks. The added overhead of this concurrency increased the total amount of kWh consumed by the system.

Figure 6.8 shows the total kWh consumption of the system without and with concurrency. As can be seen, the introduced overhead of concurrency increases the total amount of kWh consumed. On these two runs of the system, the same test is performed as in Section 6.4.3. In other words, two scenarios are simulated, one without concurrency and without a carbon-aware scheduler and one with both. In the first scenario, the files again arrive randomly throughout the day, with a uniform probability for every time of the day. In this scenario, a task is immediately processed upon arrival, as is the case in the real ING system. The second scenario chooses the optimal time of the day to "schedule" the processing of a task. It, therefore, simulates as if the scheduler delayed the execution of the task to the optimal time. It also processes the tasks concurrently, resulting in the increased kWh consumption as shown in Figure 6.8. This is performed for every season, using the season averages of gCO_2-eq/kWh emissions in 2021. For both scenarios, cumulative emissions are calculated and ordered by the synthetic test date.

Figure 6.9 shows a side-by-side comparison of the cumulative gCO_2-eq emissions by the system both scenarios. The left graph shows the scenario without the carbon-aware scheduler and without concurrency. The right graph shows the scenario using the scheduler and using a maximum concurrency of 4. The x-axis displays the synthetic tasks, ordered by their date. The y-axis displays the cumulative gCO_2-eq emissions for both scenarios. The graphs show no discernable differences between seasons. As can be seen, the graphs appear to be very similar, and in fact, the amount of emitted greenhouse gasses is slightly higher in the right graph.

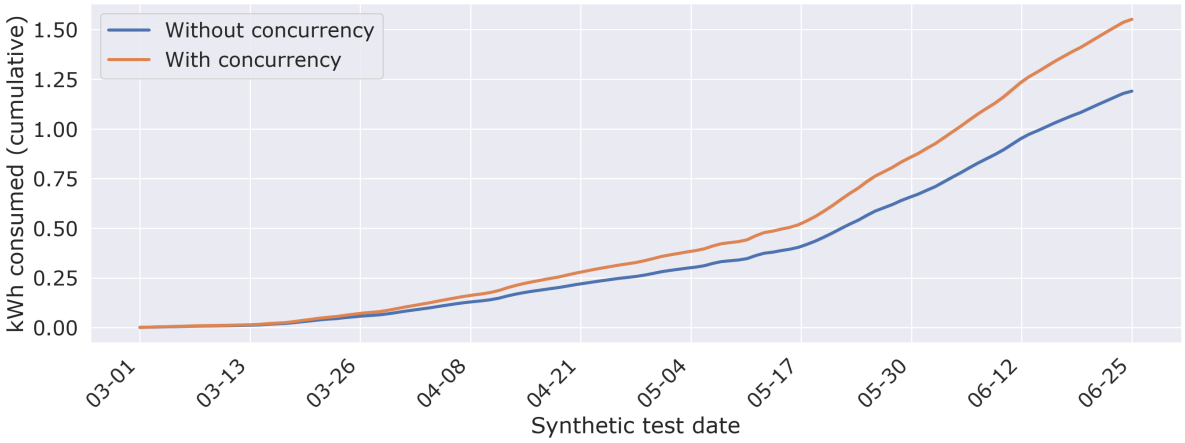


Figure 6.8: Cumulative kWh consumption of system without and with concurrency.

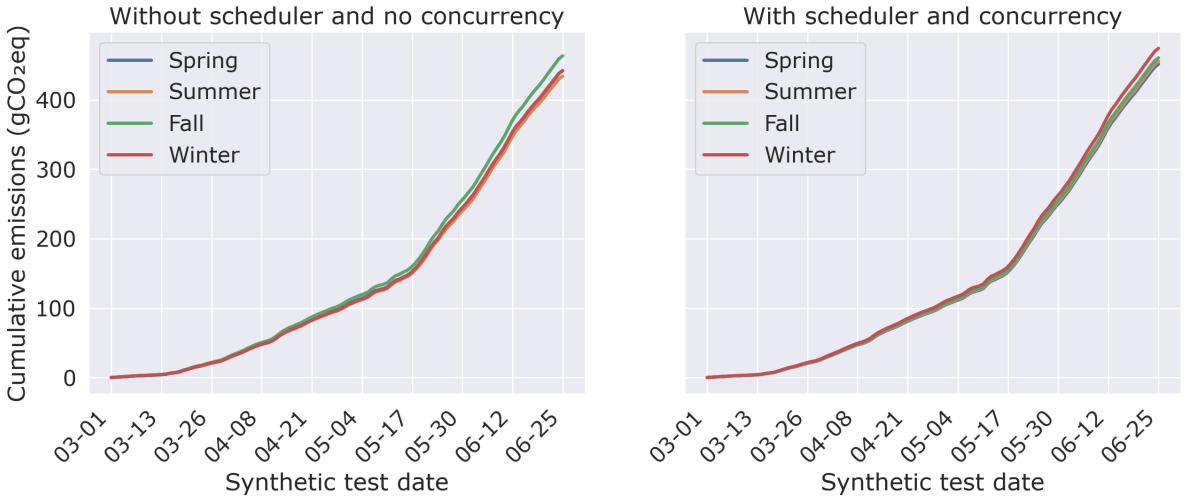


Figure 6.9: Side-by-side comparison of cumulative gCO_2-eq emissions by the system in scenarios without and with the carbon-aware scheduler and without and with concurrency. The left graph shows the scenario without the carbon-aware scheduler and without concurrency. The right graph shows the scenario using the scheduler and using a maximum concurrency of 4. The x-axis displays the synthetic tasks, ordered by their date. The y-axis displays the cumulative gCO_2-eq emissions for both scenarios.

Table 6.7 shows the calculated total emissions in gCO_2-eq for both scenarios, for all seasons. The table shows that the added kWh introduced by the overhead of processing tasks concurrently outweighs the benefit of scheduling tasks concurrently in this scenario. As can be seen, for all seasons except fall, the system actually emitted more greenhouse gasses. In fall, there is a very light reduction but only about half a percent.

Season	Without Scheduler (gCO_2-eq)	With Scheduler (gCO_2-eq)	Difference (gCO_2-eq)	%
Spring	442.76	452.38	+9.62	+2.17
Summer	435.06	456.61	+21.55	+4.95
Fall	464.43	461.52	-2.91	-0.63
Winter	443.25	475.19	+31.95	+7.21

Table 6.7: Comparison of calculated total emissions in gCO_2-eq for the system in scenarios without and with the carbon-aware scheduler and without and with concurrency. Results are separately calculated for the four different seasons.

Clearly, the added overhead outweighed the benefits in this case. However, it should be noted that the tipping point of this is influenced by task size and system specifications. In the synthetic scenario, processing tasks concurrently does not add any significant gain, as the tasks only run for a maximum of 10 minutes. Therefore, the added gain by processing in roughly a quarter of the time is very small. However, as task sizes start to increase, the expected gain will increase, as the scheduler is then able to run more tasks at peak times of renewable energy availability. The tipping point should be determined experimentally for every system that introduces a carbon-aware scheduler.

6.6. Conclusions

In the following section, conclusions will be drawn based on the previously explained tests and their results.

Research Question 1 - Conclusions

"To what extent can task load be predicted regarding running times?"

This research question is answered by the tests in Section 6.4.1 and Figure 6.2. The figure clearly shows that the error in task duration prediction stabilizes over time as the system processes more tasks. Eventually, the prediction errors settle down to between 5 and 10%. Therefore, the scheduler will have a prediction error of 12 to 24 minutes if a day's batch of tasks requires 4 hours of processing time. This is an acceptable margin of error.

Therefore, it can be concluded that a task's running time can be predicted accurately, given enough historical data has been processed by the system.

Research Question 2 - Conclusions

"How can predictions on solar and wind energy generation, as well as energy consumption, be utilized for implementing a carbon-aware scheduler?"

a) "Can the scheduler predict periods of low carbon intensity effectively?"
b) "Compared to the existing system, how much reduction in carbon emissions can be achieved by using a carbon-aware scheduler?"

This research question is answered by the tests in Sections 6.4.2 and 6.4.3.
Subquestion A is answered by analyzing the accuracy of the 'renewable energy percentage' calculation. It is evaluated if this value is a good predictor for periods of green energy generation hours. Clearly, as Figure 6.4 shows, the calculated 'renewable energy percentage' is a good indicator of the expected carbon intensity. This is proven using a statistical test shown in Table 6.5. It can be concluded that the scheduler is accurate in determining periods of low-carbon-intensive energy generation.
Subquestion B is answered by simulating two scenarios, one without and one with the carbon-aware scheduler. Table 6.6 shows that the expected reduction in greenhouse gasses is roughly 20%, depending on the season. The reduction is highest in summer, when there is a larger difference in renewable energy generation throughout the day.
Both subquestions can be used to answer the main research question. Clearly, an effective carbon-aware scheduler can be implemented based on energy generation statistics.

Research Question 3 - Conclusions

"Is there any overhead introduced by implementing a carbon-aware scheduler?"

This research question is answered by Section 6.5. The tests show that the scheduling algorithm itself is very lightweight and, therefore, does not add any significant resource consumption. However, overhead is expected when adding concurrent processing of tasks. Figure 6.8 shows that the kWh consumption increased due to this overhead. This led to an increase in expected gCO_2-eq emissions and outweighed the benefits of implementing the carbon-aware scheduler. It is shown that, at some point, the overhead outweighed the benefits of implementing concurrency and a carbon-aware scheduler. In this scenario, this tipping point is reached at a concurrency of 4. However, this is expected to vary depending on the processing tasks and the system they run on. Therefore, each system and task set should experimentally determine this tipping point.

7 Discussion

7.1. Optimisation of scheduling algorithm

The scheduler's algorithm effectively reduces greenhouse gas emissions, though it is simple and could be improved in future work. In this section, some scenarios are sketched in which the simple scheduling algorithm may not suffice.

7.1.1. Scheduling symmetrically around the highest peak of renewable energy

By predicting renewable energy generation and system consumption, the scheduler finds the time at which electricity generation has the lowest carbon load. This is explained in Section 4.3. Once this time has been determined, the scheduler aims to process all tasks centered around this single point in time. This works as intended for the average scenario in which there is a single large peak of renewable energy available around noon. However, it may work less optimally in cases where there is no such peak.

Consider the scenario shown by Figure 7.1. This figure shows the energy generation and consumption predictions, as well as the predicted renewable energy percentage on November 18th, 2023. In this case, the highest percentage of renewable energy available is at the end of the day, on November 19th at 00:00. This may cause the scheduler to opt for a suboptimal scheduling decision.

The issue with the scenario from Figure 7.1 is that the scheduler will attempt to delay tasks to the end of the day. If those tasks are expected to take a couple of hours, they may be at risk of missing their deadline the next day. The scheduler will move those tasks to earlier times to circumvent this and reduce the risk of missing their deadlines. However, those earlier times may not be the optimal time anymore. The right graph in Figure 7.1 shows this issue graphically. Consider the scenario where a task arrives on November 18th at 02:00 in the morning. The deadline for this task is then set to November 19th, at 02:00, as the default deadline given by the system is 24 hours after arriving. Assume that the task is expected to take 8 hours to process. The latest feasible starting time for it would be on November 18th at 18:00, without considering any safety margin to ensure the task is processed on time. However, this may not be the optimal time to process the task. It could have been better to process the task around noon in this scenario, for example, between 12:00 and 18:00.

To possibly solve this issue, the scheduler could take into account multiple maxima of the renewable energy generation graph combined with the total area underneath the graph. For example, it could find the three highest peaks of renewable energy generation and calculate

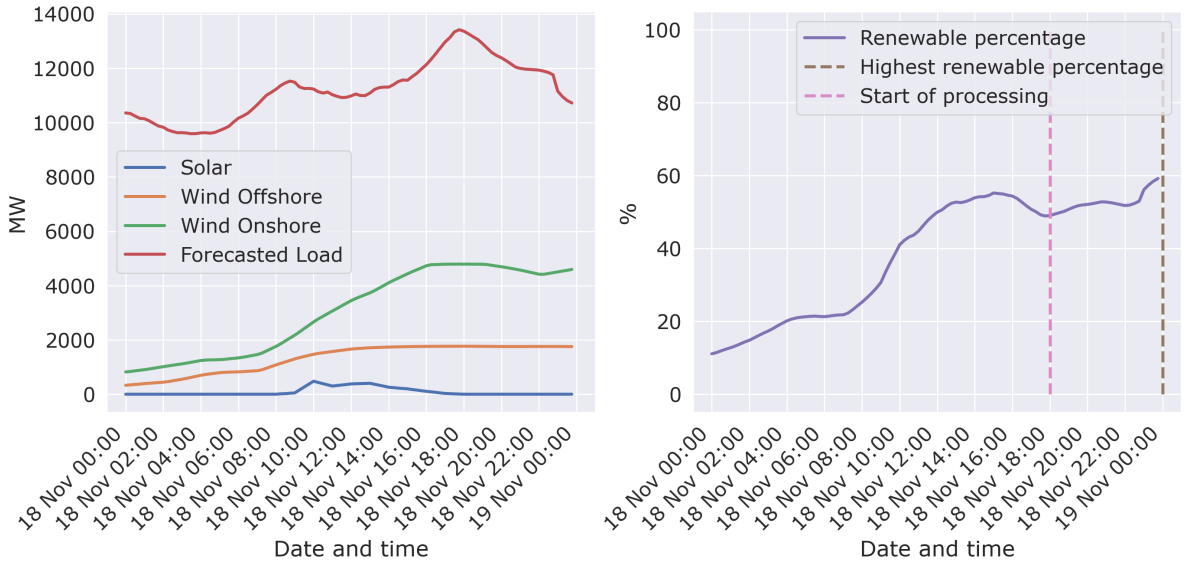


Figure 7.1: Renewable energy generation predictions in the Netherlands for November 18th, 2023, with an increasing amount of wind energy. The total amount of renewable energy available does not peak during noon but during the night.

the areas around them. Based on this information, the scheduler could then optimize for the highest value of this area underneath the graph instead of simply aiming for the highest peak.

The simple algorithm schedules all the tasks of a day to be processed sequentially, one after the other. A more advanced algorithm could split up the processing of tasks to multiple points throughout the day, aiming for multiple maxima in the renewable percentage.

The purpose of this thesis is to prove that creating a carbon-aware scheduler is sensible, not to create the best possible scheduler. However, how the scheduler's algorithm could be optimized should be investigated in future work, as this thesis proves that the concept works and that a carbon-aware scheduler is expected to reduce carbon emissions.

7.1.2. Overhead of container setup relatively large for very small input files

As explained in Section 6.4.1, small input files will be underestimated by the scheduler. Due to the overhead of processing steps, there will always be a larger percentage error when input files are relatively small. This cannot be circumvented when using a simple mean or median processing speed to determine the expected processing time of a task. Consider the scenario in which there have been many larger input files, and the next batch of files is very small. In this case, the overhead of containers is relatively small compared to the actual processing time of the container. When the next batch of small files arrives, the system has a relatively fast median processing speed compared to the actual processing speed of the new small file. The overhead becomes relatively smaller as the input file sizes increase.

A simple solution for this could be to split up the median processing speed calculation into two or more groups based on the file sizes of input files. In other words, the historical data could be split up into three bins based on their file sizes: a bin with small files, a bin with medium files, and a bin with large files. When a new task arrives to be scheduled by the system, the scheduler could consider what bin of files to use based on the new task's file size instead of predicting the running time on all the historic tasks run through the system.

7.2. Optimizing for other external requirements

The scheduler prioritizes reducing greenhouse gas emissions but can be optimized for other external requirements. These requirements are contextual and set by the owners of the systems that should be optimized. The purpose of the scheduler in this thesis was to reduce greenhouse gas emissions by one of the largest consumers of the ING private cloud. However, other external requirements could be taken into account by the scheduler. Some examples will be discussed in this section.

7.2.1. Total system load

A probable external requirement is taking into account the total system load. For example, the scheduler could take into account the predicted total system load to ensure that temporally flexible processing jobs do not interfere with latency-sensitive real-time jobs.

Consider the case of a large video streaming platform as an example. Processing jobs for this platform could be categorized into temporally flexible and inflexible jobs. An example of a flexible job could be processing newly uploaded video files to the platform. An inflexible job could be the live transcoding and streaming of a video file to one or more users. If the system load is very high, the scheduler could postpone the processing of newly uploaded video files to times at which there are fewer users watching. Processing a newly uploaded video file may not necessarily be time-sensitive, and it could be more important for the platform that users are able to stream videos.

7.2.2. Reducing load on the electricity grid

The electricity grid in the Netherlands is approaching its maximum capacity due to an ever-increasing demand [5, 25, 12]. In fact, power grid operators are warning of possible shortages if no actions are taken [16]. Therefore, more research is done to investigate solutions for making the electricity grid 'smart' [22]. A solution could be to integrate large consumers of electricity into the local electricity grid's smart system to reduce the grid's peak demand.

Figure 7.2 shows the average forecasted load per season and time of day in 2022 in The Netherlands. As can be seen, the system load varies greatly by time of day, no matter the season. Differences in system load differ up to 100%. Large consumers of electricity, such as ING or other data centers, could take these predicted loads into account to reduce the grid's peak demand.

Incentivizing large consumers of electricity to consume energy at off-peak hours could be incentivized using dynamic pricing. Dynamic pricing is the concept of publishing upcoming electricity costs based on the predicted load of the system. For example, an energy provider could publish their energy prices per kilowatt-hour (kWh) a day ahead, as is the case for some providers in The Netherlands.

By publishing electricity costs a day ahead, based on the predicted system load, large consumers of electricity could be incentivized to move the bulk of their electricity consumption to those periods at which the costs are lower. However, as [17] argues, implementing dynamic prices may not have the assumed linear effect as is expected. By publishing these predicted prices beforehand, the actual load of the system may shift significantly. Nevertheless, as the electricity grid cannot keep up with the growing demand, solutions must be implemented on the consumer side to balance this.

The accuracy of forecasted load predictions of the electricity grid can be improved by integrating systems that do a lot of resource-heavy processing using a scheduler similar to the one created in this thesis. This is not only limited to companies that do daily processing of batched

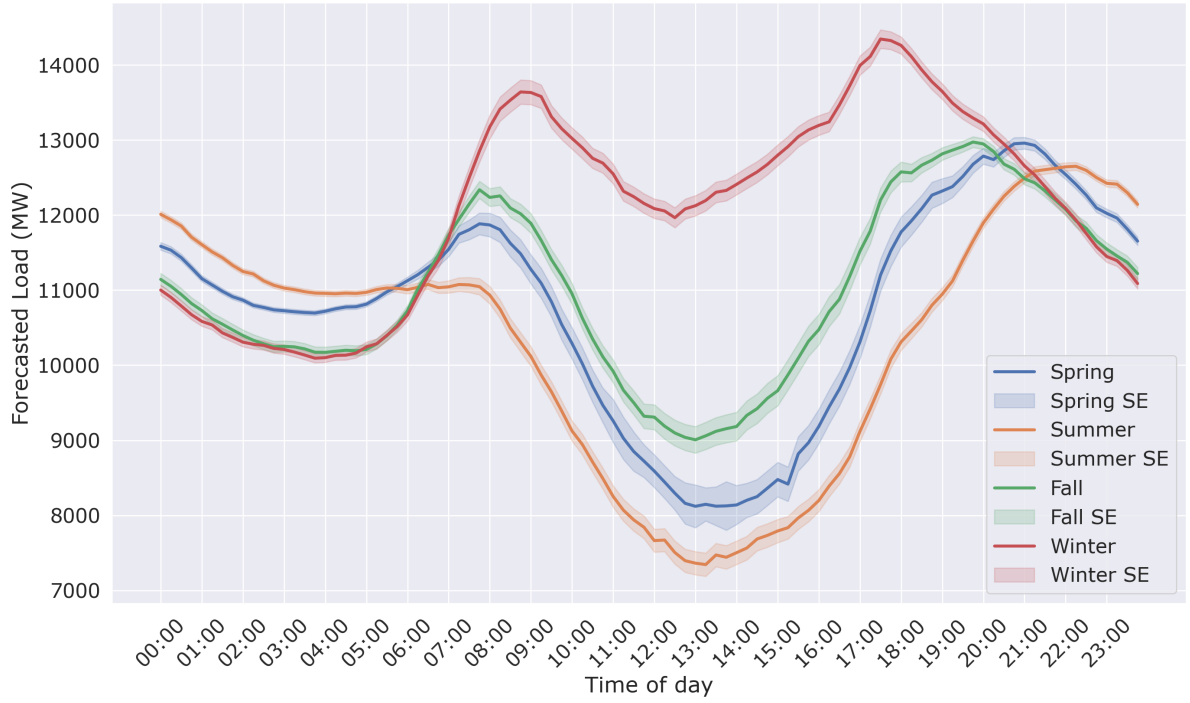


Figure 7.2: Average forecasted load per season and time of day in 2022 in The Netherlands

data. A similar scheduler could be implemented at a district level for electricity consumption, such as charging electric vehicles or heating homes.

7.3. Measuring the effect of arrival times on expected greenhouse gas emission reductions

The expected reduction in greenhouse gas emissions is roughly 20%, as shown in Section 6.4.3. However, those tests used a random arrival time for the scenario without a scheduler. This is unrealistic, especially for batch processing jobs dependent on daily uploads of files from external systems, as is the case for the ING system. Therefore, this section will investigate the effect of other arrival times.

The test methodology is the same as in Section 6.4.3, with one difference: the arrival times in the scenario without the scheduler will not be randomized for all times in the day, but will be set to a specific time in the day. This simulates a daily upload job from an external party or a cron job with a specific time set. For all these simulations, the expected greenhouse gas emissions are plotted and shown in a table, according to the same methodology as in Section 6.4.3.

The results of these simulations are visually shown in Figure 7.3. This figure shows the expected greenhouse gas emissions for systems with and without the scheduler for varying arrival times of the batched data. Clearly, the expected reduction of greenhouse gasses depends on the arrival time of the batched data. As can be seen, for the scenarios in which the input data arrives during the night, the expected reduction is higher than in cases in which the data arrives during the day. In fact, if data usually arrives around noon during the day, the expected reduction of greenhouse gasses is very small.

Table 7.1 shows the calculated total emissions in gCO_2-eq for scenarios without and with the carbon-aware scheduler for all different simulated arrival times and seasons. As can be seen, the expected reduction in emissions depends highly on the arrival times of the synthetic data.

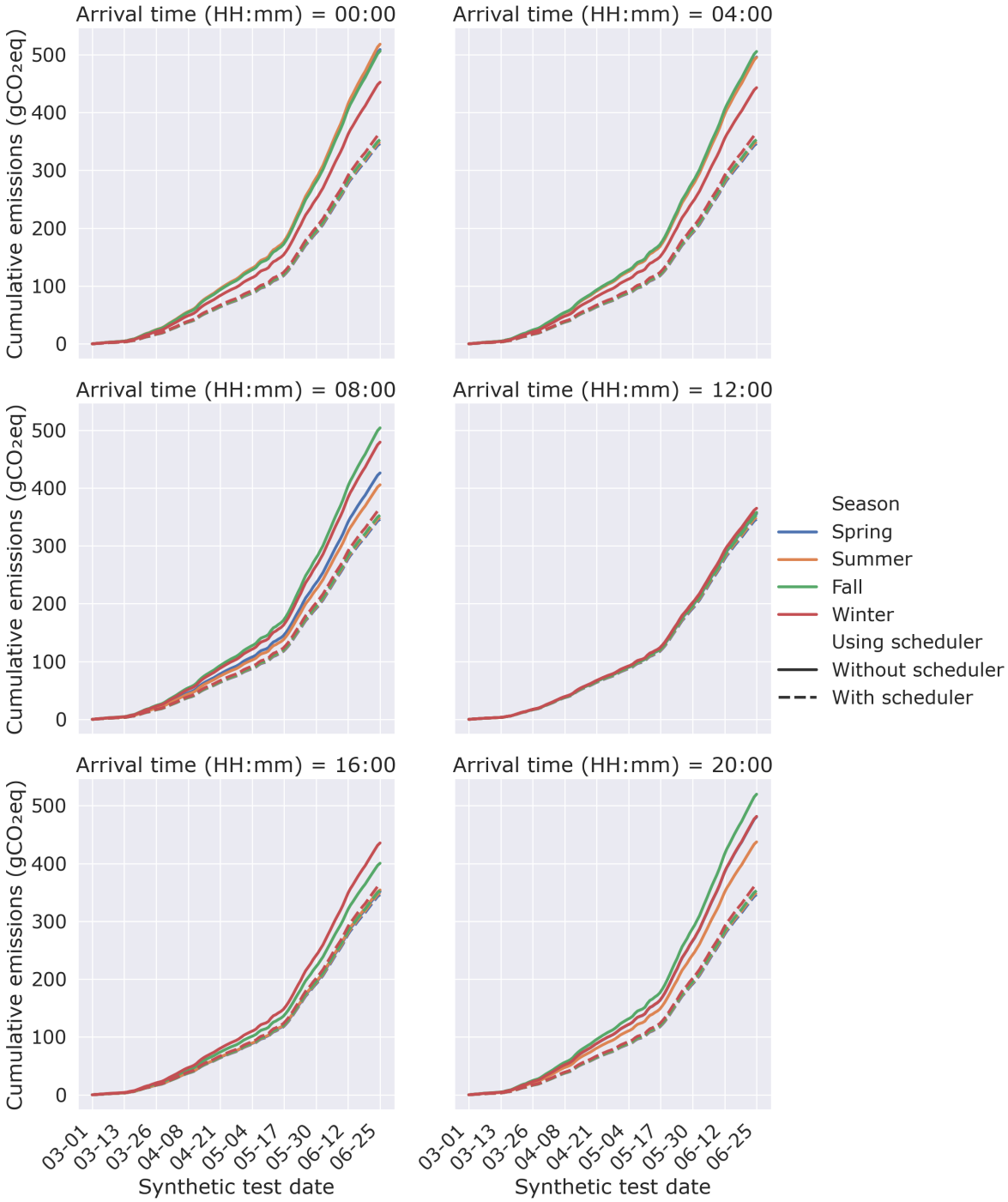


Figure 7.3: Comparison of calculated total emissions in grams of CO_2 equivalent emissions (gCO_2-eq) for the system in scenarios without and with the carbon-aware scheduler for varying arrival times of the synthetic data. Results are separately calculated for the four different seasons. The x-axis represents the synthetic test date, and the y-axis represents the cumulative gCO_2-eq emissions.

For example, systems that process daily data that arrives nightly around 00:00 are expected to reduce emissions by as much as 32.5%. Additionally, the differences between seasons become clearer. The expected reduction in summer for files arriving at 00:00 is 32.5%, but in winter, the reduction is only 19.6%.

Arrival time	Season	Without Scheduler (gCO_2-eq)	With Scheduler (gCO_2-eq)	Difference (gCO_2-eq)	%
00:00	Spring	510.03	346.91	-163.12	-31.98
	Summer	519.32	350.15	-169.17	-32.58
	Fall	507.34	353.92	-153.42	-30.24
	Winter	453.44	364.41	-89.03	-19.64
04:00	Spring	497.31	346.91	-150.40	-30.24
	Summer	496.44	350.15	-146.28	-29.47
	Fall	506.70	353.92	-152.78	-30.15
	Winter	444.00	364.41	-79.60	-17.93
08:00	Spring	427.03	346.91	-80.12	-18.76
	Summer	406.48	350.15	-56.33	-13.86
	Fall	505.13	353.92	-151.21	-29.94
	Winter	480.37	364.41	-115.96	-24.14
12:00	Spring	356.50	346.91	-9.60	-2.69
	Summer	358.25	350.15	-8.10	-2.26
	Fall	358.64	353.92	-4.72	-1.32
	Winter	365.80	364.41	-1.39	-0.38
16:00	Spring	352.32	346.91	-5.41	-1.53
	Summer	355.05	350.15	-4.90	-1.38
	Fall	401.34	353.92	-47.42	-11.82
	Winter	436.51	364.41	-72.10	-16.52
20:00	Spring	481.54	346.91	-134.63	-27.96
	Summer	438.31	350.15	-88.16	-20.11
	Fall	520.97	353.92	-167.05	-32.07
	Winter	482.53	364.41	-118.13	-24.48

Table 7.1: Comparison of calculated total emissions in gCO_2-eq for the system in scenarios without and with the carbon-aware scheduler for varying arrival times of the synthetic data. Results are separately calculated for the four different seasons.

From these results, it should become clear that implementing a carbon-aware scheduler is effective in most cases. However, it should be noted that systems that process files around noon under normal operations without a scheduler should expect less of a change. Additionally, it should be considered that these results are valid for The Netherlands. In The Netherlands, the renewable percentage peaks, on average, around noon for all seasons. The results may differ in other countries, where other sources of renewable energy generation and different weather patterns are present. These findings lead to another possibility of implementing a simplified scheduler, as will be discussed in Section 7.4

7.4. Using a simplified non-predicting scheduler

As shown in Section 7.3, the arrival times of data influence the reduction in greenhouse gas emissions. If the data arrives around noon under normal circumstances, the expected reduction in carbon emissions is insignificant. However, this opens up the possibility of implementing a simplified scheduling algorithm.

Consider a scenario in which it is known that data always arrives around 04:00 in the morning and that, under normal circumstances, processing this data takes roughly 2 hours. If this is known, a simplified scheduler could be implemented. The simplified scheduler would not consider any predictions on renewable energy, nor would it keep track of historical runs of the system to calculate an expected processing speed. Instead, the system could be adapted to delay the processing to 11:30 in winter and 12:30 in summer.

There are both advantages and disadvantages to this implementation. A significant advantage is that the system does not need to implement a complicated metrics collection system and scheduler algorithm. The system could be left as is without implementing anything other than a delay function to processing. Many systems may already be based on a cron job, indicating a starting time to process files. This scenario could require nothing more than a one-line change in a cron job entry.

There are two main disadvantages to this situation. Firstly, the system loses any reactive ability to non-average weather days. In other words, if high winds are expected at night and the day is heavily clouded, the carbon load may be low in the evening instead of noon. Consider the scenario as shown in Figure 4.3. In this case, the simplified scheduler could process its tasks at a suboptimal time. Nevertheless, as is shown by the average renewable energy predictions in Section 4.2, the system would predict correctly on average. Secondly, the system loses any reactive ability to varying task sizes. An added benefit of implementing the carbon-aware scheduler is that the system can accurately predict the running times of a batch of data. If the system suddenly receives an abnormal amount of data or has been down for a couple of days for maintenance, it will not consider the increased task size. In this scenario, the system risks missing task deadlines, as it may start processing tasks after their latest feasible starting time.

Nevertheless, it is expected that a simple tweak to a system may be sufficient in many cases. If nothing else, this research may inspire other teams and companies to consider the energy generation statistics of their local areas in determining when to schedule batch processing jobs throughout the day.

7.5. Peak loads and system idle time

Implementing concurrent processing has some advantages and some disadvantages. A main advantage is that concurrency allows the scheduler to utilize periods of low-carbon-intensity energy as much as possible by implementing concurrent processing at optimal times. This is discussed in Sections 4.3 and 6.4.3. A main disadvantage is that the system's capacity must be able to handle the peak load of processing multiple tasks simultaneously. If a system wants to decrease greenhouse gas emissions even further by implementing concurrency, it needs to be able to handle the increased peak workload. Systems that are already limited in processing capacity will only be able to move tasks around to other times throughout the day in an attempt to utilize more low-carbon-intensive energy. Additionally, the system idle time increases as processing is compressed to a smaller timeframe. Consider a scenario in which a system using a carbon-aware scheduler always delays the processing of tasks to noon for a total of 2 hours using concurrency. The system will then be idle during the other 22 hours of the day, and this may be considered wasteful.

It is expected that many data centers can handle the increase in peak demand, as they are generally underutilized [8]. Server utilization rates within data centers barely exceed 6%, while facility utilization hovers around 50%. Therefore, it is expected that introducing a higher peak demand at low-carbon-intensive hours should be possible in many cases. As server utilization rates barely exceed 6% at times, it should be feasible to implement carbon-aware scheduling in many data centers. The feasibility of this has already been shown by research done on Google data centers [19]. The authors show that by implementing virtual capacity curves (VCC), the peak energy consumption during high-carbon-intensive energy generation hours can be reduced by as much as 8% without introducing significant delays for batch processing jobs.

7.6. Native energy monitoring in OpenShift

The scheduling system in this thesis calculates the kWh consumption of pods itself. These calculations are performed by gathering CPU and memory consumption percentages. The total kWh consumption is calculated for every container, as if it were running on ING container hosting platform (ICHP), using the CPU and memory percentages.

More tools are being created that aid developers in minimizing their carbon footprint as the attention to data center power consumption and its impact on greenhouse gas emissions increases. One of these tools is being implemented by RedHat's OpenShift itself. At the time of writing, a developer preview is available in OpenShift for native energy monitoring[1]. The tool is very much in its early stages of development. It would be a viable option for future teams to use once it has been released.

The tool would simplify parts of this research's scheduling and monitoring system. When native energy monitoring is implemented in OpenShift, manually calculating kWh consumption by pods by the system would no longer be required. Instead, the system could simply monitor and verify its scheduling decisions based on the energy monitoring system provided by OpenShift. Additionally, the fact that this tool is being implemented indicates that carbon-aware processing is rising.

7.7. An argument for carbon-aware pricing research at cloud providers

Greenhouse gas emissions by internet services and data centers account for a substantial portion of the overall emissions and are forecasted to only increase over time [13, 7]. The research in this thesis has shown a clear connection between processing times throughout the day and their respective resulting greenhouse gas emissions. Yet at the time of writing, the three largest cloud providers (Google Cloud[2], Amazon Web Services (AWS)[3] and Microsoft Azure[4]) do not incentivize their consumers to use green energy by implementing carbon-aware pricing. While all three providers present tools for consumers to calculate their carbon footprints, they offer no monetary incentive to schedule workloads to low-carbon-intensive energy consumption periods. Instead, all three providers provide pricing based on a flat fee system or hardware utility such as CPU time and memory.

A monetary incentive for cloud consumers could be implemented as a CO_2-tax or another form of carbon-aware pricing. For example, the pricing of cloud resource consumption could

[1] https://cloud.redhat.com/blog/introducing-developer-preview-of-kepler-power-monitoring-for-red-hat-openshift

[2] https://cloud.google.com/

[3] https://aws.amazon.com/

[4] https://azure.microsoft.com/

be linked to dynamic energy prices in the region in which the data center resides. These dynamic prices could reflect the carbon intensity of energy generation in said region, where the price decreases if more renewable energy is used.

Recent research has shown that implementing a carbon tax on a country level has a positive effect in the long run [28]. The authors show this by linking implemented CO_2-taxes to renewable energy consumption in 39 economies. Though the effect is positive in the long run, the authors show that implementing a carbon tax has negative effects in the short term. Therefore, this thesis argues that future work should investigate the effects of implementing a carbon-based pricing model at cloud providers. It should be investigated whether these positive effects can be expected when implementing carbon-aware pricing at cloud providers or whether the short-term negative effects outweigh them in the long run.

7.8. Adaptability of carbon-aware scheduler

Section 7.4 argues that a simple scheduling algorithm can be implemented in certain scenarios whilst maintaining some level of improvement. Such an algorithm could be as simple as updating a scheduled time at which a system runs its tasks without any predictions on running times or optimal times to schedule. This will work in the average scenario, but such a system loses a lot of the adaptability the carbon-aware scheduler provides.

There are two main benefits to spending the effort to implement a proper carbon-aware scheduler, like the one implemented for this thesis. Firstly, it automatically adapts to changes in the processing system and tasks. Secondly, it is flexible to short-term changes in renewable energy generation, like changing weather, and long-term changes in renewable energy generation due to the diversification of renewable energy sources. These will be discussed here shortly.

The scheduler is designed to automatically adjust when there are changes in the tasks it has to process or changes in the system that processes these tasks. For instance, if the size of a task suddenly decreases, the expected running time will decrease as well. When the running time of a task decreases, it is necessary to reschedule it to make use of the highest peak in renewable energy generation again fully. Additionally, if the system that processes the tasks changes, for example, when its capacity increases, the processing times will change with it. Again, this means that the timing of when to start processing tasks changes. Both these scenarios are handled by the carbon-aware scheduler implemented for this thesis. It will automatically discard older tasks that have become irrelevant, as it uses a time-based sliding window to select the most recent runs of tasks through the system.

An additional benefit of using the carbon-aware scheduler is that it will adapt to changes in renewable energy generation, both short-term and long-term changes. Short-term changes are deviations from the average shape of energy generation throughout the day, such as shown in Figure 7.1. An example of this in the Netherlands is on a very cloudy day, with high winds in the evening. In such a scenario, the optimal time to schedule almost certainly is not noon, as the total renewable energy generation peaks around the evening. If the weather deviates from the average of the season, the scheduler will move the processing of tasks to the highest peak in renewable energy, thereby adapting to these short-term changes. Long-term changes are classified as changes in renewable energy generation over the years due to the diversification of renewable energy sources. To ensure a sustainable energy supply in the future, the sources of renewable energy generation are diversified [6]. This will almost certainly change the shape of renewable energy generation throughout the day. Once again, the scheduler will simply follow the optimal times throughout the day when scheduling tasks.

7.9. Using the predicted grams of CO_2 equivalent emissions per kWh (gCO_2-eq/kWh) from Electricity Maps directly

Part of the work in this research has gone into predicting optimal times to schedule workloads. This prediction is based on a calculated renewable energy percentage using data from the free transparency platform API by the European Network of Transmission System Operators for Electricity (ENTSO-E). It was then evaluated to be a good predictor for periods of low-carbon-intensive energy generation in Section 6.4.2, using historical data from Electricity Maps. However, Electricity Maps provides API access to 24-hour forecasts of gCO_2-eq/kWh. These predictions can be utilized instead of the calculated energy generation predictions as a basis for the scheduling decisions. Additionally, using these predictions directly removes the need to evaluate the scheduling decisions after the fact, as data on greenhouse gas emissions is available at schedule-time.

The 24-hour forecasted gCO_2-eq/kWh predictions are part of a paid plan in Electricity Maps; at the time of writing, it costs €500/month per zone. This thesis has shown that acquiring this data is not necessarily required, but it may be lucrative for a company to consider it anyway. A €500 per month investment may not be much for many medium-sized and larger companies, and it may be a worthwhile investment to acquire this data from Electricity Maps directly instead of dedicating software developer resources to implementing it using ENTSO-E data.

Even though Electricity Maps provides convenient access to 24-hour forecasts of gCO_2-eq/kWh, one of their main data sources is the freely available transparency platform provided by ENTSO-E. Therefore, companies looking to implement a carbon-aware scheduler for free can use the data provided by ENTSO-E and should still be able to predict periods of low-carbon intensity relatively accurately.

Even though this data is freely available from ENTSO-E, it must be noted that Electricity Maps performs a much more extensive Life-Cycle Analysis (LCA) on the electricity generation sources. Therefore, this data source is expected to be more reliable in the long term, considering changing energy generation demands and carbon emission regulations. A company opting for the paid enterprise plans from Electricity Maps mainly pays for the added convenience and additional estimations done by the platform itself. However, the basis of the 24-hour forecasted emissions relies on the free platform by ENTSO-E. Therefore, companies wanting to implement a carbon-aware scheduler for free could consider the transparency platform by ENTSO-E.

7.10. Continuation of work within ING

A team within ING will continue this thesis's work, based on the positive results from this research. The team's purpose is to set up a carbon-aware scheduler at a broader scale within ING using the learnings from this thesis. The team will attempt to centralize the implementation of a carbon-aware scheduler, allowing other divisions within the company to register their batch-processing jobs to this centralized scheduler. The scheduler will then distribute all the tasks it has received from other divisions over the totality of the ING private cloud capacity. It will perform a similar scheduling methodology based on the energy generation predictions and accompanying expected greenhouse gas emissions.

The formation of this team, with its accompanying budget allocation required, is a clear indicator of the relevance of the work in this thesis. It shows that the work is considered valuable and relevant for companies wanting to reduce the carbon footprint of their data centers and cloud infrastructure.

8

Threat to Validity

8.1. Synthetic data

A possible threat to the validity of this work is the structure and type of the synthetic data, as well as the synthetic processing steps, compared to the real system at ING. There are two factors to consider.

The first factor is the differences between the contents of the data in the ING system and the synthetic system. The synthetic data is extracted from Twitch[1] chat logs, whereas the data in the ING system comes from various sources such as email and direct messaging apps. An effort was made to make the synthetic data realistic by considering the data type. The ING system processes textual chat data, which are conversations between two or more persons. Conversely, the synthetic data also contains textual chat data of conversations between multiple people.

Additionally, the size of the synthetic data is less than that of the real system. The synthetic tests processed roughly three months of 'synthetic test days', where each day processed data varying from 1 megabyte (MB) to 100 MBs in size. The real system, on the other hand, processes much more data than this on a day-to-day basis. However, the test data was required to be smaller so that multiple days could be simulated in a short timeframe. Multiple days needed to be simulated to have a realistic representation of the pipeline at ING and to show the effectiveness of the carbon-aware scheduler over time. Therefore, the synthetic data is made to vary on a day-to-day basis, as is the case in the ING system. For example, the number of chats per day is expected to vary in the ING system, as there will be more conversations between employees during the week than during the weekend. Similarly, the number of chats varies per day in the synthetic system.

The second factor to consider is the differences in the processing steps between the real and simulated systems. The synthetic steps are chosen to be representative of text processing steps like the ING system performs. In this case, the synthetic steps are partially copies of the real ING system and partially different. The first two steps align closely with the real steps in the ING system, but the third step deviates as this step is very dependent on the contents of the data. In this case, the third step performs some natural language processing on the chat data, representing the fraud detection system that ING runs.

Even though there are some differences between the real and the simulated systems, both

[1] https://www.twitch.tv/

have some resource-intensive processing steps on textual chat log data. The synthetic system is considered to be a good representation of the real system at ING. The expected results are, therefore, assumed to be representative of the possible results in the ING system.

8.2. Conversion from CPU and memory consumption statistics to kilowatt-hour (kWh) and greenhouse gas emissions

Sections 5.4, 6.2 and 6.4.3 explain how the kWh consumption of pods is calculated and how this is converted to expected greenhouse gas emissions. The calculation from CPU and memory statistics to kWh consumption is done using an ING-provided tool. This tool is based on real-world measurements within the ING private cloud. Therefore, the conversion from CPU and memory statistics to kWh consumption assumes that a pod runs within ING container hosting platform (ICHP). However, the tests were not performed on pods running in ICHP and were instead performed on containers running in a docker environment, as the ING processing pipeline was not completed on time. Therefore, the actual consumed kWh during tests likely differs from the expected consumption within ICHP.

This does not pose a significant threat to evaluating the results for two reasons. First of all, the results show a relative difference in greenhouse gas emissions in scenarios with and without the carbon-aware scheduler. Therefore, the relative difference between them is still relevant in determining the effect of implementing a carbon-aware scheduler. Secondly, both scenarios are tested using exactly the same methodology of calculating kWh consumption. Therefore, any error within the calculated kWh consumption appears in both scenarios and should not significantly impact the relative difference. An additional point to consider is the fact that CPU and memory percentages may differ between the synthetic pipeline and the pipeline within ICHP. Again, this does not pose a significant threat to evaluating the results for the same reasons mentioned previously.

8.3. Possibility of bugs

It is essential to consider potential issues that could impact the reliability of the results. One concern is the possibility of unnoticed mistakes or bugs in the code of the carbon-aware scheduler despite the efforts made in testing and validating the scheduler. Additionally, the simulation of the ING pipeline, crucial for validating the scheduler, introduces another layer of potential uncertainty. Potential bugs in the code could affect the scheduler's performance and, consequently, influence the evaluation outcomes.

Care was taken to reduce the possibility of bugs influencing the results by testing crucial parts of the scheduler's code and the simulated pipeline. However, it is necessary to acknowledge that possible issues in the code may influence the expected results shown in this thesis.

9

Conclusion

This thesis has set out to investigate if greenhouse gas emissions can be reduced by intelligently scheduling batch processing jobs, thereby reducing the impact of data centers on global warming. A carbon-aware scheduler has been developed to optimize a resource-intensive data processing pipeline at ING. The pipeline managed by ING processes daily batches of communications data, and it is one of the larger consumers of the ING private cloud. An investigation was performed to determine if the carbon-aware scheduler can reduce greenhouse gas emissions in such a pipeline.

The carbon-aware scheduler reduces emissions by scheduling resource-intensive processing jobs to green energy hours. Green energy hours are times of the day when there is more renewable energy available, and less energy is generated from other sources. The scheduler relies on the variability of green energy production to reduce carbon emissions. By using 24-hour forecasts on solar and wind generation and 24-hour forecasts on grid load, the scheduler determines optimal times to schedule. Once an optimal time is found, it attempts to schedule as many tasks around this time as possible to use as much renewable energy as possible.

The proposed scheduler, named `S.C.A.L.E` (Scheduler for Carbon-Aware Load Execution), comprises three modules. The first module predicts task running times based on previous runs of tasks through the system. This module gathers data from previous system runs to calculate an expected processing speed, which the scheduler uses to predict the running times of newly arriving tasks. Accurate task running time predictions are required so the scheduler can create a schedule for tasks yet to be processed. The second module provides the scheduler with predictions regarding renewable energy generation and electricity grid demand. The scheduler uses information from this module to determine optimal times to process tasks to reduce greenhouse gas emissions. The third module interacts with the processing pipeline. It ensures that tasks are processed at the right time by interacting with the container orchestrator.

The task running time predictions must be accurate to prevent task deadline incursions. Therefore, the accuracy of the predictions is validated. Section 6.4.1 and Figure 6.2 show that the scheduler can accurately predict the running times of tasks. This is validated by sequentially processing data through the pipeline and keeping track of prediction errors over time. The error reduces over time and eventually converges to an error between 5 and 10%. A 5 to 10% error means that the scheduler will predict 4 hours of tasks with an error between 12 and 24 minutes, which is an acceptable margin of error.

The scheduler's predictions of green energy hours are validated using known greenhouse gas emissions per generated kWh, using data from Electricity Maps. This validation is performed by letting the scheduler calculate a 'renewable energy percentage' value, which is then compared to the known grams of CO_2 equivalent emissions per kWh (gCO_2-eq/kWh) values at that exact point in time. Section 6.4.2 and Figure 6.4 show that the scheduler's predictions are accurate in determining periods of low-carbon-intensive energy generation.

The scheduler now has two important pieces of information to determine its schedule. Firstly, it knows the expected running time of the tasks that need to be processed and their deadlines. Secondly, it has information on when it expects the carbon load of energy generation to be low in the coming day. Using this information, it schedules tasks around the peak of renewable energy generation in an attempt to reduce the carbon emissions of the system.

Unfortunately, the ING pipeline was not finished in time for the work in this thesis; the trade and communications surveillance (TRCS) squad incurred significant delays in refactoring the pipeline. As a result, the carbon-aware scheduler could not be tested against the ING pipeline. Therefore, the ING system was simulated using a synthetic data processing pipeline that processed similar textual chat data. This system was set up in accordance with the plans of the real system, thereby having a very similar structure. The key differences between the two systems are the specific processing performed on the data and the total processing time required. Additionally, the synthetic pipeline did not run within ING container hosting platform (ICHP) but in a dockerized environment mimicking the structure of the real system. However, as is shown, the synthetic pipeline is a realistic simulation of the ING pipeline and could, therefore, be used to evaluate the expected reduction in greenhouse gas emissions.

Two simulations are performed on the synthetic pipeline to determine the expected reduction in greenhouse gasses by implementing the carbon-aware scheduler. During these simulations, the pipeline processed all the synthetic data while recording the kWh consumption of all pods. The expected greenhouse gas emissions are calculated from all pods' measured kWh consumptions. The emissions are calculated using the seasonal averages of gCO_2-eq/kWh in 2022. The results in Section 6.4.3 show that the scheduler is expected to reduce emissions by 20%.

The expected reduction in greenhouse gas emissions will differ depending on the season and the expected arrival time of the batched input data. It is shown that, no matter the season, the scheduler reduces carbon emissions. Therefore, it is recommended to use the scheduler in all seasons of the year. However, the tests show that the expected reduction varies significantly depending on the input data's arrival and processing times. When input data usually arrives around noon, the expected reduction by the carbon-aware scheduler is less significant because the data is then processed around noon, when the carbon intensity is already low.

The limitations of a carbon-aware scheduler are investigated as a last test. It has been shown that there is an expected reduction in greenhouse gas emissions by implementing the carbon-aware scheduler. Adding concurrency to the system is expected to improve the effects of the scheduler further, as more green energy hours can be utilized if tasks are processed in a shorter timeframe. However, adding concurrency introduces overhead. Eventually, the benefits of using the carbon-aware scheduler are outweighed by the overhead introduced by implementing more concurrency. In this case, a simulation was performed using a concurrency of 4. With a concurrency of 4, the increased kWh consumption by the system was past the tipping point, negating the benefits of scheduling tasks at optimal times. The tipping point at which the concurrency overhead will outweigh the benefits differs for every system and task set. Therefore, any team wanting to implement a carbon-aware scheduler should be aware of

this and investigate experimentally in their system when this point is reached.

In conclusion, the work in this thesis has set out to investigate the feasibility of implementing a carbon-aware scheduler to reduce greenhouse gas emissions by resource-intensive data processing pipelines. It shows that there are definite benefits to implementing a carbon-aware scheduler and that the variability of green energy production can be utilized to reduce the carbon footprint of a data center by as much as 20%. Therefore, it is recommended that this work be continued within ING and that the carbon-aware scheduler be adopted into the production environment once it is ready.

References

[1] Anders Andrae. “Comparison of Several Simplistic High-Level Approaches for Estimating the Global Energy and Electricity Use of ICT Networks and Data Centers”. In: *International Journal of Green Technology* 5.1 (Dec. 2019), pp. 50–63. ISSN: 24142077. DOI: 10.30634/2414-2077.2019.05.06. URL: https://ijgtech.com/ijgtv5a6/.

[2] Tom Bawden. “Global warming: Data centres to consume three times as much energy in next decade, experts warn”. In: *Independent* (Jan. 23, 2016). URL: https://www.independent.co.uk/climate-change/news/global-warming-data-centres-to-consume-three-times-as-much-energy-in-next-decade-experts-warn-a6830086.html (visited on 10/29/2023).

[3] Richard Brown et al. *Report to Congress on Server and Data Center Energy Efficiency: Public Law 109-431*. Tech. rep. Aug. 2007.

[4] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. “Data Center Energy Consumption Modeling: A Survey”. In: *IEEE Communications Surveys & Tutorials* 18.1 (2016), pp. 732–794. DOI: 10.1109/COMST.2015.2481183.

[5] DutchNews. *Watts up: Dutch electric grid is at capacity*. Sept. 2021. URL: https://www.dutchnews.nl/2021/09/watts-up-dutch-electric-grid-is-at-capacity/ (visited on 12/11/2023).

[6] Lazar Gitelman, Mikhail Kozhevnikov, and Yana Visotskaya. “Diversification as a Method of Ensuring the Sustainability of Energy Supply within the Energy Transition”. In: *Resources* 12.2 (2023). ISSN: 2079-9276. DOI: 10.3390/resources12020019. URL: https://www.mdpi.com/2079-9276/12/2/19.

[7] Nicola Jones. “The Information Factories”. In: *Nature* 561 (Sept. 2018), pp. 163–166.

[8] James M Kaplan, William Forrest, and Noah Kindler. *Revolutionizing Data Center Energy Efficiency*. Tech. rep. McKinsey&Company, July 2008.

[9] Avita Katal, Susheela Dahiya, and Tanupriya Choudhury. “Energy efficiency in cloud computing data centers: a survey on software technologies”. In: *Cluster Computing* 26.3 (June 2023), pp. 1845–1875. ISSN: 15737543. DOI: 10.1007/s10586-022-03713-0.

[10] Imran Khan. “Temporal carbon intensity analysis: renewable versus fossil fuel dominated electricity systems”. In: *Energy Sources, Part A: Recovery, Utilization, and Environmental Effects* 41.3 (Sept. 2018), pp. 1–15. ISSN: 1556-7036. DOI: 10.1080/15567036.2018.1516013. URL: https://www.tandfonline.com/doi/full/10.1080/15567036.2018.1516013.

[11] Jeongmin Kim. *Twitch.tv Chat Log Data*. Version V2. Harvard Dataverse, 2019. DOI: 10.7910/DVN/VE0IVQ. URL: https://doi.org/10.7910/DVN/VE0IVQ.

[12] Yusuf Latief. *Gridlock: how the Netherlands hit capacity*. Feb. 2023. URL: https://www.enlit.world/smart-grids/grid-management-monitoring/gridlock-how-the-netherlands-hit-capacity/ (visited on 12/11/2023).

[13] Kien Le et al. “Capping the brown energy consumption of Internet services at low cost”. In: *International Conference on Green Computing*. IEEE, Aug. 2010, pp. 3–14. ISBN: 978-1-4244-7612-1. DOI: 10.1109/GREENCOMP.2010.5598305. URL: http://ieeexplore.ieee.org/document/5598305/.

[14] Gurdeep Singh Malhi, Manpreet Kaur, and Prashant Kaushik. *Impact of climate change on agriculture and its mitigation strategies: A review*. Feb. 2021. DOI: 10.3390/su13031318.

[15] Electricity Maps. *Electricity Maps*. 2023. URL: https://www.electricitymaps.com/methodology (visited on 11/28/2023).

[16] Bart Meijer and Mark Potter. *Dutch power grid operator TenneT warns of shortages by 2030*. Jan. 2023. URL: https://www.reuters.com/business/energy/dutch-power-grid-operator-tennet-warns-shortages-by-2030-2023-01-12/ (visited on 12/11/2023).

[17] Joeri Naus et al. “Smart grids, information flows and emerging domestic energy practices”. In: *Energy Policy* 68 (2014), pp. 436–446. ISSN: 03014215. DOI: 10.1016/j.enpol.2014.01.038.

[18] Ana Radovanovic et al. “Power Modeling for Effective Datacenter Planning and Compute Management”. In: *IEEE Transactions on Smart Grid* 13.2 (Mar. 2022), pp. 1611–1621. ISSN: 1949-3053. DOI: 10.1109/TSG.2021.3125275. URL: https://ieeexplore.ieee.org/document/9600450/.

[19] Ana Radovanović et al. “Carbon-Aware Computing for Datacenters”. In: *IEEE Transactions on Power Systems* 38.2 (Mar. 2023), pp. 1270–1280. ISSN: 0885-8950. DOI: 10.1109/TPWRS.2022.3173250. URL: https://ieeexplore.ieee.org/document/9770383/.

[20] Ali Raza et al. *Impact of climate change on crops adaptation and strategies to tackle its outcome: A review*. Feb. 2019. DOI: 10.3390/plants8020034.

[21] Skipper Seabold and Josef Perktold. “statsmodels: Econometric and statistical modeling with python”. In: *9th Python in Science Conference*. 2010.

[22] Robin Smale, Bas van Vliet, and Gert Spaargaren. “When social practices meet smart grids: Flexibility, grid management, and domestic consumption in The Netherlands”. In: *Energy Research and Social Science* 34 (Dec. 2017), pp. 132–140. ISSN: 22146296. DOI: 10.1016/j.erss.2017.06.037.

[23] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Nov. 2023. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[24] Maninder P. S. Thind et al. “Marginal Emissions Factors for Electricity Generation in the Midcontinent ISO”. In: *Environmental Science & Technology* 51.24 (Dec. 2017), pp. 14445–14452. ISSN: 0013-936X. DOI: 10.1021/acs.est.7b03047. URL: https://pubs.acs.org/doi/10.1021/acs.est.7b03047.

[25] NL Times. *Dutch power grid overloaded in more places; No new connections possible*. Aug. 2022. URL: https://nltimes.nl/2022/08/04/dutch-power-grid-overloaded-places-new-connections-possible (visited on 12/11/2023).

[26] Charlotte Trueman. “What impact are data centres having on climate change?” In: *Computerworld* (Aug. 9, 2019). URL: https://www.computerworld.com/article/3431148/why-data-centres-are-the-new-frontier-in-the-fight-against-climate-change.html (visited on 10/28/2023).

[27] John Vidal. "'Tsunami of data' could consume one fifth of global electricity by 2025". In: *Climate Home News* (Dec. 12, 2017). URL: https://www.climatechangenews.com/2017/12/11/tsunami-data-consume-one-fifth-global-electricity-2025/ (visited on 10/29/2023).

[28] Hua-Tang Yin et al. "Carbon tax: Catalyst or hindrance for renewable energy use in climate change mitigation?" In: *Energy Strategy Reviews* 51 (Jan. 2024), p. 101273. ISSN: 2211467X. DOI: 10.1016/j.esr.2023.101273. URL: https://linkinghub.elsevier.com/retrieve/pii/S2211467X23002237.

[29] Kelvin O. Yoro and Michael O. Daramola. "CO2 emission sources, greenhouse gases, and the global warming effect". In: *Advances in Carbon Capture*. Elsevier, Jan. 2020, pp. 3–28. ISBN: 9780128196571. DOI: 10.1016/B978-0-12-819657-1.00001-3. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780128196571000013.

A

Wrong method of generating synthetic data

The first method of creating synthetic data from the set of 52 pickle files was wrong. Initially, synthetic data was generated from this set by randomly selecting one to three pickle files. Then, the chosen pickle files were added to a `.zip` archive and then compressed into a `.7z` file. This allowed for a great variation of input files; the total number of random input files possible is:

$$\begin{aligned} \frac{52!}{(52-3)!3!} + \frac{52!}{(52-2)!2!} + \frac{52!}{(52-1)!1!} = \\ \frac{52!}{49!3!} + \frac{52!}{50!2!} + \frac{52!}{51!} = \\ 22100 + 1326 + 52 = 23478 \text{ possible random files} \end{aligned} \tag{A.1}$$

Initially, creating input data using this methodology seemed sensible, as there was a large variance between random input file sizes. Figure A.1 displays the variance in file sizes when generating synthetic data using this methodology. As can be seen, the input file size increases depending on the number of pickle files that were zipped. Additionally, the files with more pickle files were larger on average, but the input file sizes had a large overall variance. This seemed a sensible choice for randomizing the creation of synthetic input data, as the input data of the real system by ING also varies daily in file size.

This method of generating randomized input data was wrong. After analyzing the results from tests run on these files, it became clear that the data did not simulate the ING data properly. Additionally, there was too much reliance on the number of files zipped into a random file and what specific pickle file was zipped into the random file. This problem showed up when plotting the prediction error percentage of the system over time as the system processed more and more tasks. It was expected that the scheduler would be able to improve the accuracy of running time predictions over time as more tasks were processed through the system. A scatterplot was made of these values to determine if this was the case.

Figure A.2 is a scatterplot of task running time prediction errors over time. The x-axis represents individual tests, ordered by their starting time from left to right. The y-axis plots the percentage error of the running time prediction by the scheduler for the task in that test. Clearly, the scheduler did not improve its running time predictions over time as the system processed

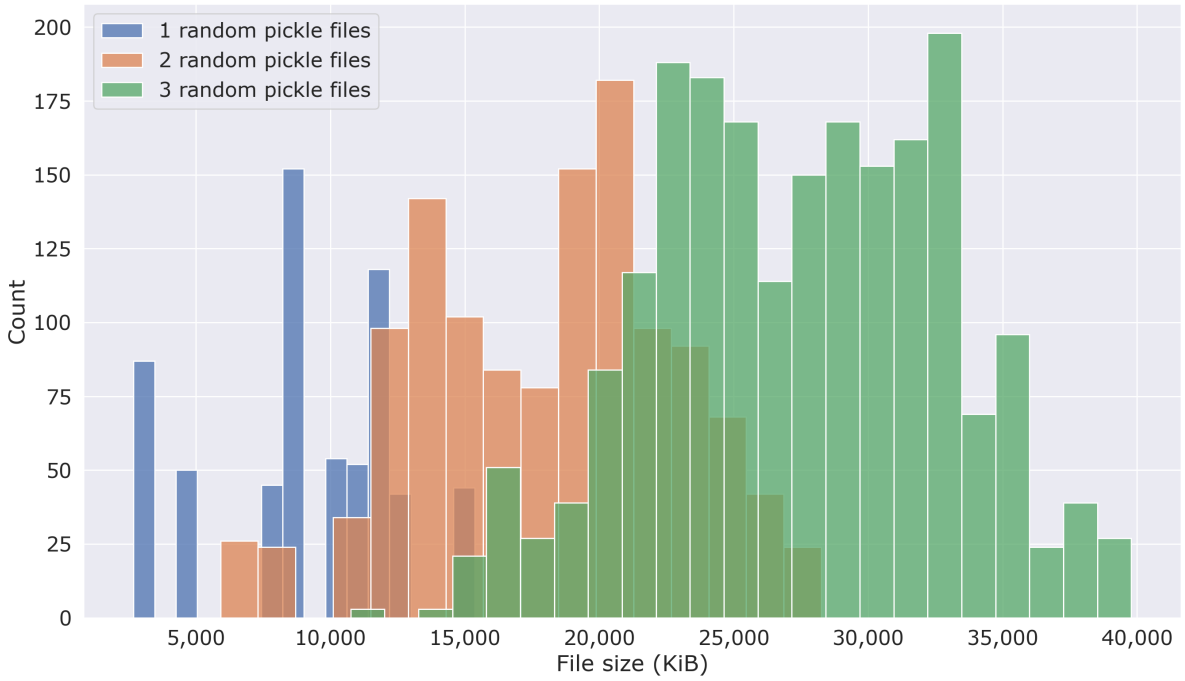


Figure A.1: A histogram of the file sizes of random input files generated by the wrong methodology.

more tasks. Ideally, the scheduler could predict the running times of tasks with greater accuracy as more tasks had been processed. This was not the case; the scheduler did not improve at all. However, it needed investigated whether this was a test data limitation or a limitation of the scheduler itself.

The first clue that the synthetic data was wrong was that the median processing speed converged almost immediately to a rather consistent value. However, many tests were still over and underestimating their running times up to 70%. Figure A.3 shows this issue visually; it displays two charts with the “Median processing speed before test (KiB/s)” and “Percentage error of predicted duration versus actual duration.” The x-axis represents the “Test number” and shows the tests ordered from left to right by their starting times. The top chart displays the median processing speed calculated by the system before a test, and the bottom chart shows the percentage error of the scheduler’s prediction of the running time of that test. As can be seen, the system quickly converged to a processing speed of around 1,250 kibibyte per second (KiB/s), but the prediction error percentage did not decrease.

The processing speeds of the individual tasks were investigated to determine the source of the prediction errors. Figure A.4 shows all tasks’ processing speeds. As can be seen, the tasks varied greatly in their processing speeds, but those processing speeds are relatively centered around the median processing speed. This led to the next investigation, as the system was pretty accurate in determining the average processing speed, but the speeds did not correlate with the input file size.

The input file compositions were investigated as the next step in the investigation. When a task is scheduled, only its file size is known, but after the tests have been run, it is possible to investigate the file’s contents. Specifically, it was possible to determine exactly what test file contained which pickle files. Using this information, it could be investigated if any pickle files were the source of the high errors. To determine this, tests were categorized based on their prediction error percentages. Tests that had a prediction error of more than 25% were

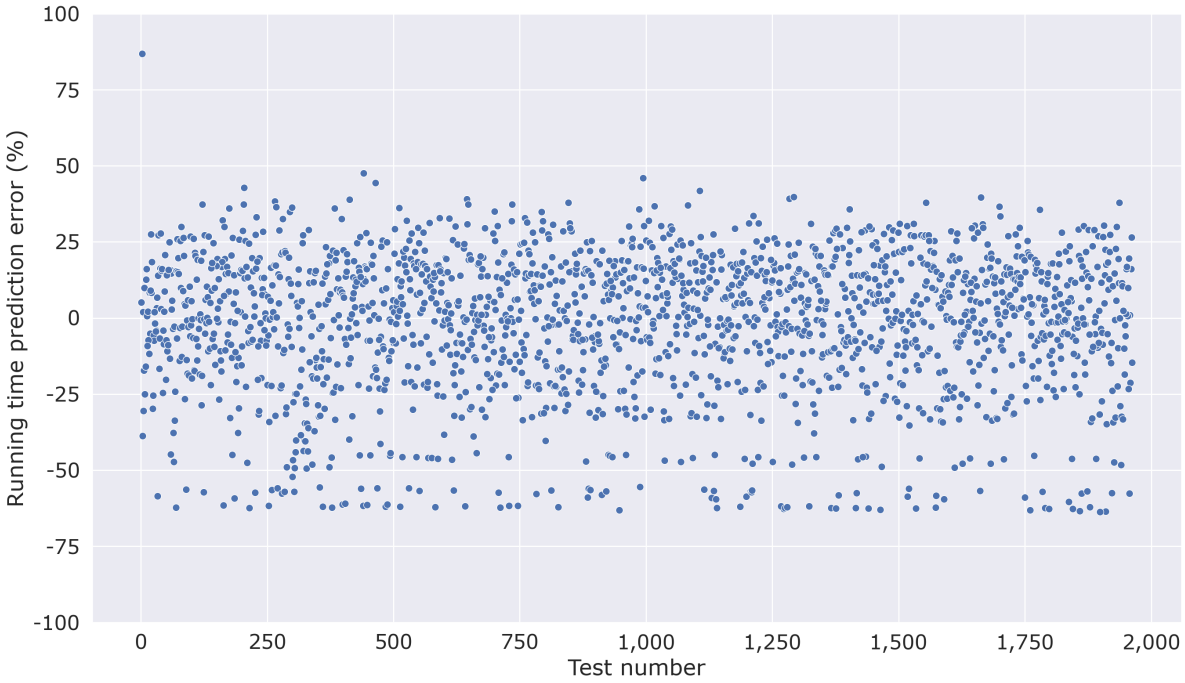


Figure A.2: A scatterplot of the percentage error of running time predictions over time. The x-axis represents individual tests, ordered by their starting time from left to right. The y-axis plots the percentage error of the running time prediction by the scheduler for the task in that test.

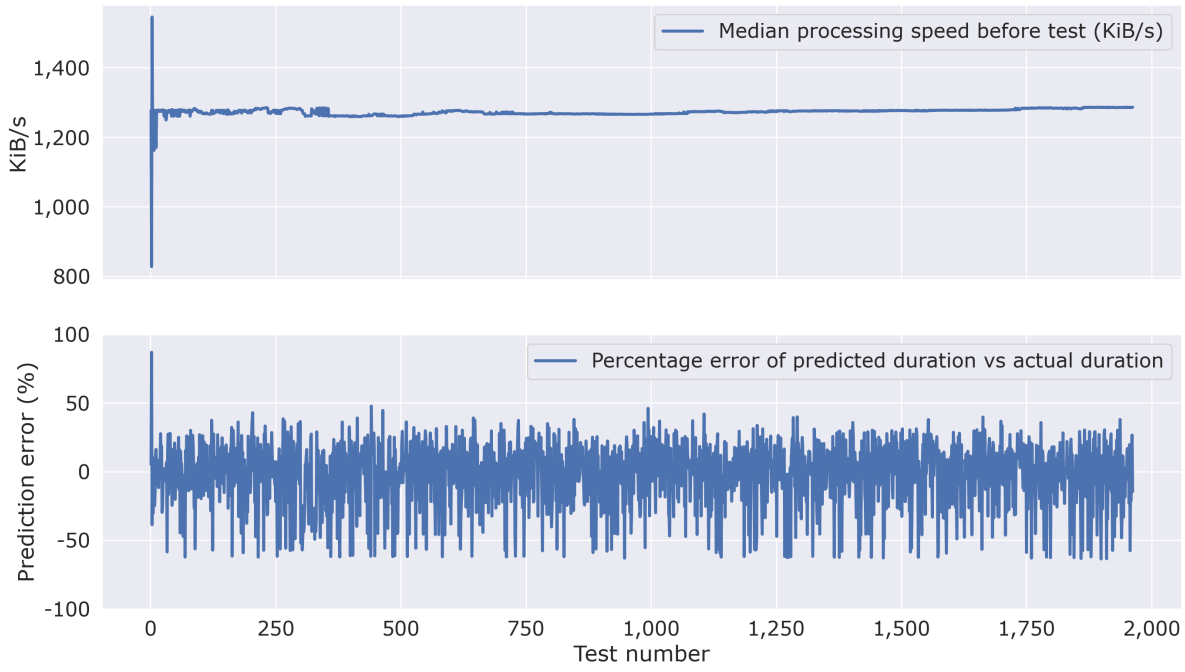


Figure A.3: Two charts displaying the “Median processing speed before test (KiB/s)” and “Percentage error of predicted duration versus actual duration.” The x-axis represents the “Test number” and shows the tests ordered from left to right by their starting times. The top chart displays the median processing speed calculated by the system before a test, and the bottom chart shows the percentage error of the scheduler’s prediction of the running time of that test.

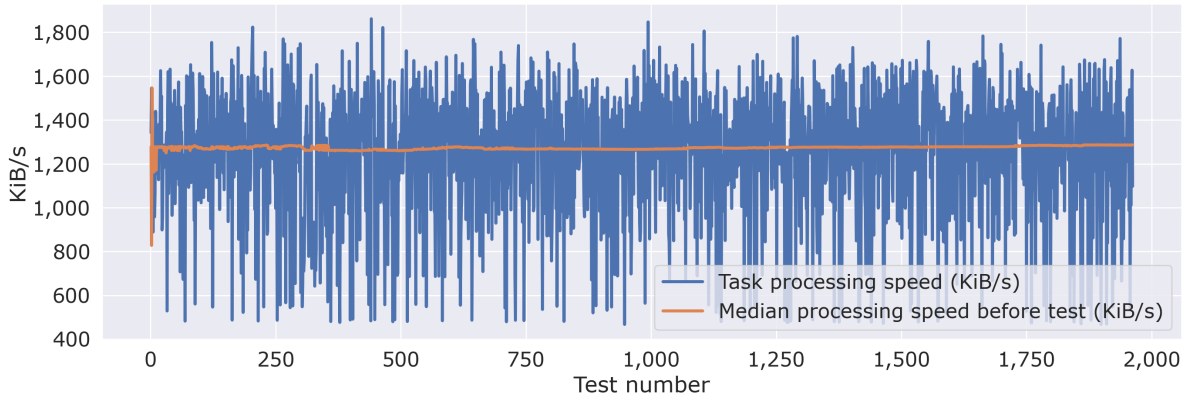


Figure A.4: A chart displaying task processing speed in KiB/s and the median task processing speed before a test started from tests using the wrong synthetic data as input. The x-axis represents the test number, ordered from left to right by the test starting time. The y-axis represents the processing speed.

categorized as “high error tests”. Then, for all tests, and all tests with a high error rate, it was counted how many times a pickle file appeared in the tests. In other words, for every pickle file, it was counted in how many tests said pickle file appeared. Using these counts, the proportional appearance of that pickle file was calculated and compared to the proportional appearance in the high error tests. Figure A.5 shows these values. On the x-axis, various pickle files are shown. The top chart displays the proportion of appearances of those pickle files in all tests and tests with a high error rate. The bottom chart displays the relative change in proportion between these two categories. As can be seen, the pickle file `kinggothalion.pkl` appears in more than 15% of the tests, compared to only about 7.5% in all tests. This is a relative increase of more than 100% in comparison. From Figure A.5, it was observed that certain pickle files appeared more frequently in tests with a high error rate compared to all tests. This suggests that the processing time of a test was dependent on the specific pickle files being processed and not necessarily on the file size of that file.

After it was found that specific pickle files caused significantly more prediction errors, the tests containing the pickle files that caused the most errors were removed from consideration. Then, the same analysis was done to investigate whether the processing speed did converge, disregarding the tests with the highest error rates. However, this did not solve the problem, and the system still performed reasonably on average but had many outliers in prediction errors.

A second analysis was performed to determine the cause of the prediction errors. This analysis compared the running time prediction errors for all tests, grouped by the number of pickle files that were zipped into the test input. Thus, for every test, it was determined whether the input file contained 1, 2, or 3 pickle files. Figure A.6 displays the result of this analysis visually. It shows clearly the issue with the synthetic data. The graphs again plot the tests ordered by their starting time from left to right, but they are grouped by the number of pickle files in the input file. The left graph displays all tests with one pickle file; the middle displays tests with two pickle files; and the right shows all tests with three pickle files. As can be seen, the scheduler systematically underestimated the running time of tasks containing only one pickle file and overestimated tests containing three. Also, the middle graph shows that the system accurately predicted tasks with two pickle files.

From the above figures, it can be determined that the method of creating synthetic data was incorrect. First, the running time was influenced too much by the specific pickle files that were chosen, as the synthetic input files only contained 1 to 3. But more importantly, the scheduler

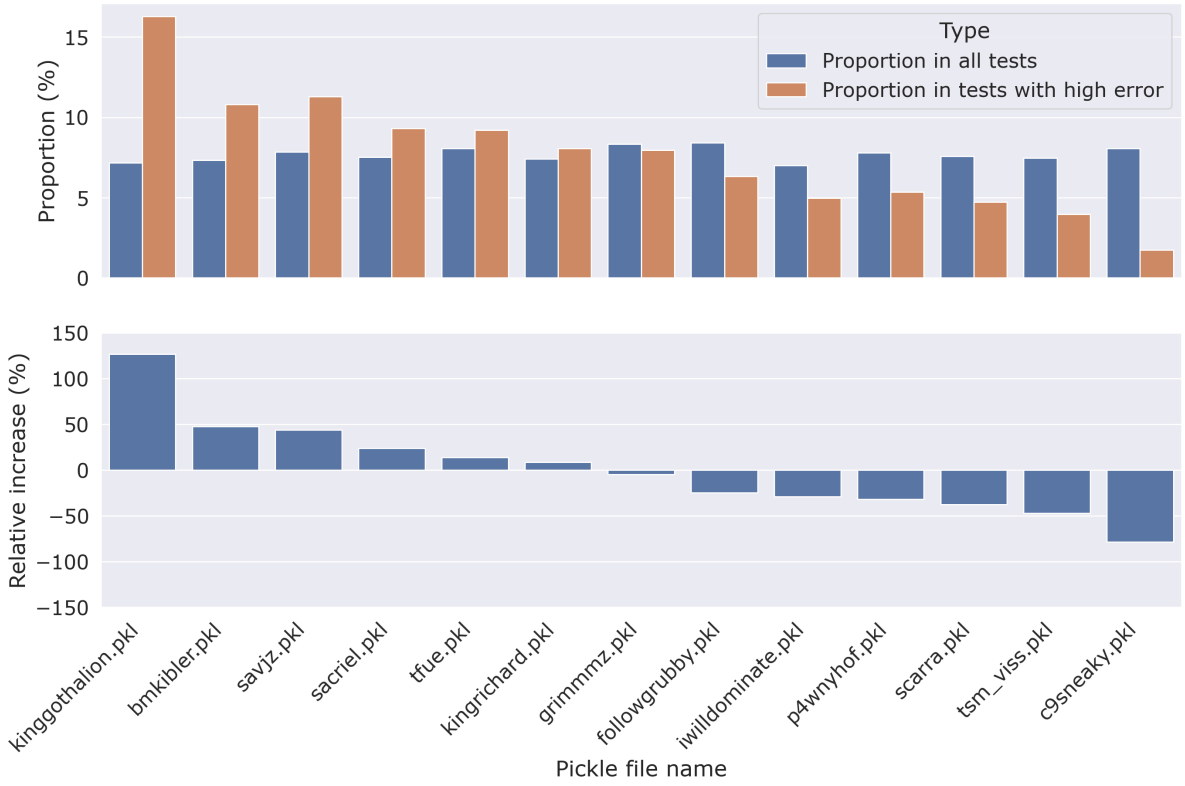


Figure A.5: Chart displaying relative appearances of pickle files in all tests and tests with a high prediction error. The bottom chart shows the relative increase or decrease in the proportion of appearances in high-error tests compared to the proportion of appearances in all tests.

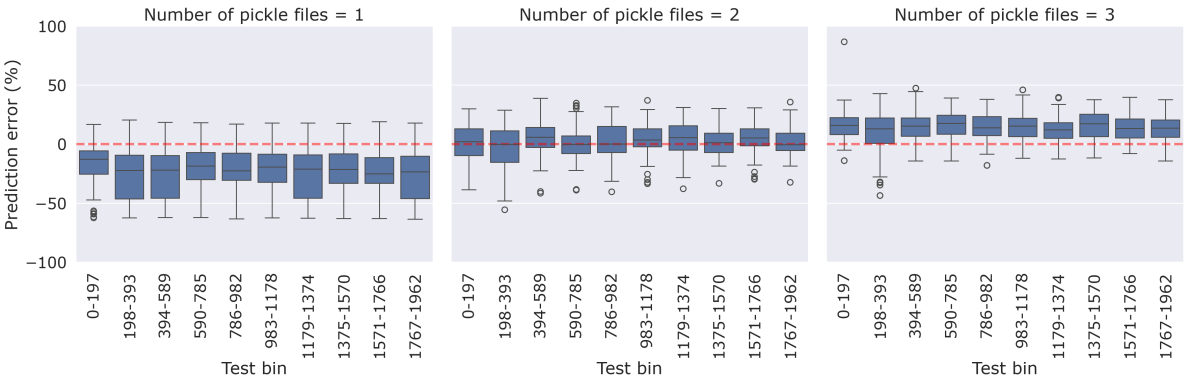


Figure A.6: Prediction error percentage of tests over time, where the tests are grouped based on the number of pickle files the synthetic input file contained. The tests are ordered from left to right based on their starting time.

did not converge to a processing speed that could be calculated based on the input file size. Instead, it converged to the “average number of pickle files in a synthetic input file”, namely 2. Figure A.6 shows this clearly. As can be seen, the prediction errors for tests containing two pickle files had means around 0%. Secondly, in all three graphs, the whiskers and outliers of the boxplots indicate a large variance in the data. This can be traced back to the differences in errors depending on the specific pickle files in the input file. Due to these issues, another method of creating synthetic data was set up. This method is explained in Section 6.1.1.

B

Scatterplot grid of all numeric values gathered in the 'running time prediction accuracy test'

Figure B.1 shows all the numeric values in a grid of scatterplots from the test that was ran to determine the accuracy of the running time predictions by the scheduler. These values are from the test described in Section 6.4.1. From these scatterplots, the 4 tests that contained very high errors are dropped to prevent the scatterplots from being unreadable in the '*Duration prediction error (%)*' row and column.

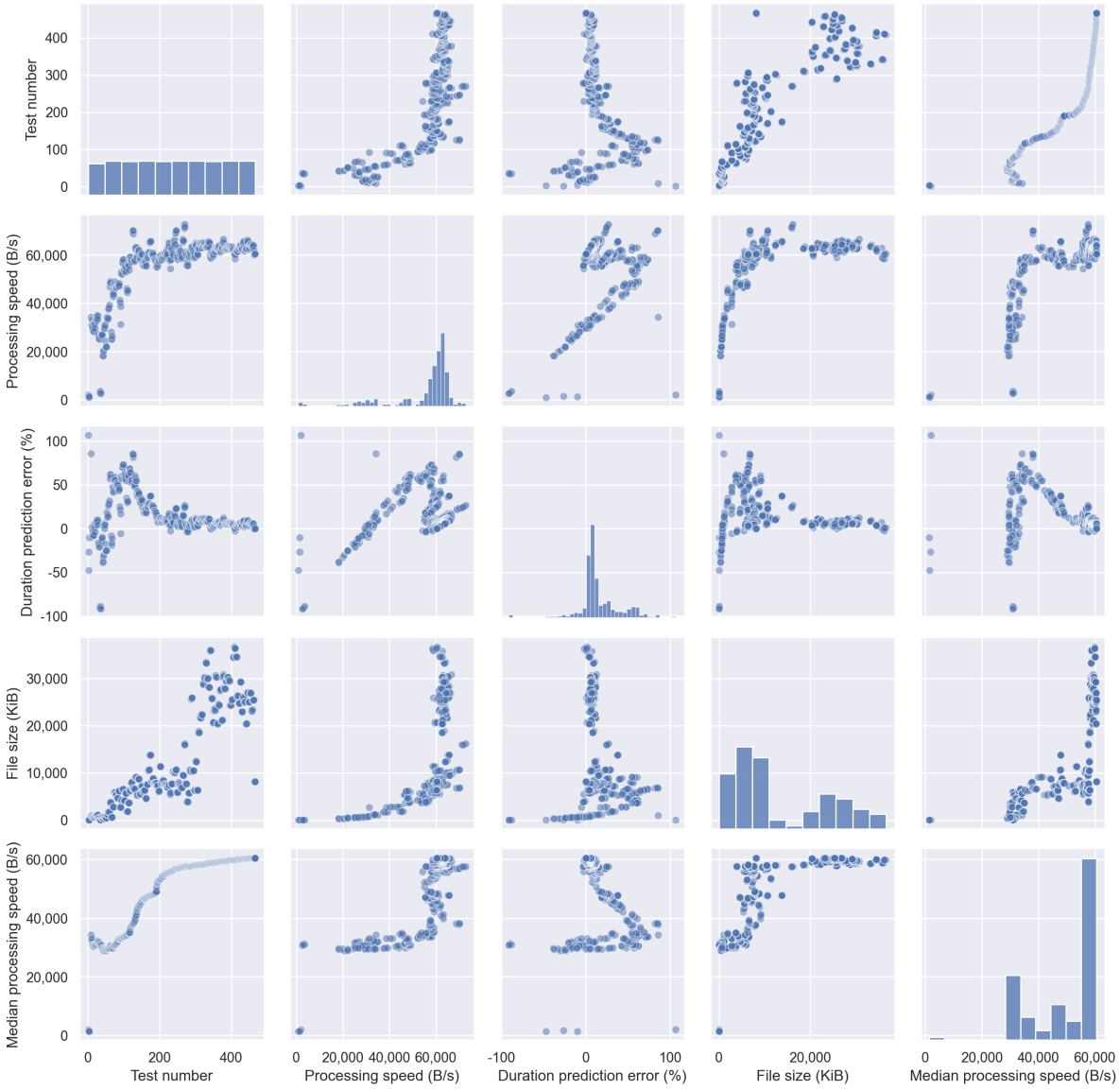


Figure B.1: A scatterplot grid depicting all the numeric values collected during the ‘running time prediction accuracy test’, as described in Section 6.4.1.

C

Service account to grant scheduling pod API access for scaling and metrics collection in OpenShift

For the scheduler to be able to access the OpenShift API it needs to be assigned the correct rights. The scheduler requires these rights to be able to scale up and down deployments, as well as extract pod resource consumption metris. This can be defined in a `yaml` configuration file. This configuration file is shown in Listing 2.

Once the `ServiceAccount` is defined, it needs to be added to the deployment configuration of the scheduling pod. The deployment `yaml` configuration of the scheduling pod is shown in Listing 3. This is not the full configuration file, it only shows relevant parts for binding the `ServiceAccount` to give the pod proper rights to interface with the OpenShift API. The highlighted line designates the `ServiceAccount` to the scheduling pod, effectively granting it the same rights and roles assigned to that `ServiceAccount`.

```yaml
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: scheduler-sa
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: scheduler-sa-role
rules:
  - apiGroups:
      - ''
    resources:
      - pods
    verbs:
      - get
      - watch
      - list
  - apiGroups:
      - 'apps'
    resources:
      - deployments/scale
    verbs:
      - patch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: scheduler-sa-rolebinding
subjects:
  - kind: ServiceAccount
    name: scheduler-sa
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: scheduler-sa-role
```

Listing 2: `yaml` file defining the `ServiceAccount` in OpenShift containing the correct rights to get information on running pods, and scale deployments up and down using the OpenShift API.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    name: scheduler
  name: scheduler
spec:
  replicas: 1
  selector:
    matchLabels:
      name: scheduler
    spec:
      hostNetwork: false
      serviceAccountName: scheduler-sa
      containers:
        ...
```

Listing 3: `yaml` file defining the deployment of the scheduling pod in OpenShift. This is not the full configuration, it only shows relevant parts for binding the `ServiceAccount` to give the pod proper rights to interface with the OpenShift API. The highlighted line designates the `ServiceAccount` to the scheduling pod, effectively granting it the same rights and roles assigned to that `ServiceAccount`.