

ACIT5900
MASTER THESIS

in

**Applied Computer and Information
Technology (ACIT)**
May 2021

Cloud-based Services and Operations

**The CAST-algorithm
bridging green energy with continuous testing.**

Ingvild Stølen

Department of Computer Science
Faculty of Technology, Art and Design

OSLOMET

Acknowledgements

I would like to thank my supervisor Kyrre Begnum for invaluable advice, help and motivation during the work with this thesis. This year has been a challenge with both work and school happening from home, but the support from my supervisor helped me keep my motivation throughout the semester. Both his technical advice and ideas and inspiration has been a great contribution during this whole process.

My gratitude also extends to OsloMet, for providing me with the necessary skills to undergo this work, and for facilitating learning and academic progress during this difficult last year of the Covid pandemic.

Finally, I need to thank my patient and caring boyfriend Marius Nilsen Kluftun for encouragement and support. Living with someone stuck behind the computer every waking hour has not been ideal during these months of lockdown, but he has kept rooting for me every step of the way.

Abstract

In society today, we are dependent on software in our daily lives. One key factor of success when creating this software is the use of automated testing. At the same time, we have a large challenge in reducing greenhouse gas emissions to prevent global warming. Every day, thousands of developers trigger automated tests, and each test uses some amount of energy.

The electricity used for running these tests can be produced in many ways, some of them greener than others. Green energy often comes from intermittent renewable energy (IRE) sources, such as wind and solar power plants. The intermittent nature of these power sources means that they cannot supply one area with electricity throughout the whole day.

This project explores the possibility of moving software testing jobs in time and geolocation (different data centers) to areas with green energy surplus, in order to minimize the greenhouse gas emissions caused by the tests. The results showed that it is possible to reduce the carbon footprint of automated tests by this method, but this requires sophisticated infrastructure along with a geography where data about the supply and demand of electricity and its production sources is available in sufficiently high resolution.

Table of contents

1.	Introduction	1
1.1	Problem statements:	3
2.	Background	5
2.1	The rise of DevOps	5
2.2	The importance of automated testing.....	6
2.3	The need for efficiency in automated testing	7
2.4	Computing and power consumption	8
2.5	The properties of electricity as a commodity	10
2.6	The role of renewables	11
2.7	Supply and demand imbalance.....	12
2.8	Can testing become a part of the solution?	14
3.	Approach.....	17
4.	Design.....	19
4.1	Geographical area for test data	19
4.2	The relationship between exchange cables, IRE surplus and energy prices.	21
4.2.1.	Introducing the CAST algorithm.....	23
4.3	Energy data sources	24
4.4	The cost of deploying a test server.....	25
	Decide allocation on demand and deploy on demand:.....	26
	Decide allocation preemptively and deploy on demand:.....	27
	Decide allocation and deploy preemptively:	27
4.5	The significance of the hour of the day	27

4.6	Data center capacity and pricing	29
4.7	Status and reservations of virtual machines in the cloud	30
4.8	Scenarios for test triggering.....	33
4.9	Test plan.....	33
4.10	Calculating greenhouse gas emissions from test runs	34
5.	Implementation	35
5.1	Tools.....	35
5.2	Data sources for production volumes and prices.....	35
5.2.1.	Time period selection	35
5.2.2.	Production volume data selection and transformation	36
5.2.3.	Price data selection and transformation	37
5.3	Test activity data.....	38
5.4	Constructing input data on delay tolerance	39
5.5	Test duration.....	39
5.6	Determining surplus areas and their price delta	40
5.7	Simulation script	41
5.8	Constructing test scenarios.....	44
5.8.1.	The baseline test scenario	44
5.8.2.	Other scenarios.....	45
5.9	Analyzing input data.	47
6.	Results/observations	49
6.1	Are some areas really better than others? – a closer look at the energy data.....	49
6.2	Test distribution.....	51
6.3	Approach for manual analysis.....	52

6.4	Results of the baseline simulation	53
6.5	Results of the IRE_share scenarios	54
6.6	Results of different thresholds for keeping server open.....	55
6.6.1.	The cost of eliminating waste in a large project.....	56
6.7	The effects of different delay tolerances.....	57
6.8	The results with different test durations.....	58
6.9	Summary of cost / benefit analysis.....	59
6.10	Delay tolerance versus actual waiting time.....	60
7.	Discussion.....	65
7.1	How does these findings answer the problem statements?	65
7.1.1.	Problem statement 1	65
7.1.2.	Problem statement 2	66
7.2	Takeaways from the process	67
7.2.1.	The importance if interdisciplinary backgrounds for this type of projects	68
7.3	Deploy overhead.....	68
7.4	Data center availability	69
7.5	The future of automated software testing.....	69
7.6	Electricity production in the future	70
7.6.1.	More wind and solar power.....	70
7.6.2.	Thermal production sources - A comeback for nuclear power?	70
7.7	Other applications.....	71
7.8	Possible improvements and further work	72
7.8.1.	Parallelization and sectioning for advanced users	72
7.8.2.	Combine with test ordering algorithms.....	72

7.8.3.	Further examine the relationship between test frequency and waiting time.	73
7.8.4.	Apply nudging features to increase delay tolerance.	73
7.8.5.	Include data center pricing as a decision parameter.	73
7.8.6.	Build a prototype.	73
8.	Conclusion.....	75
9.	Litterature	76
10.	Appendices.....	80
	Appendix A Python files	80
	A1 globalVariables.py.....	80
	A2 simulate.py	80
	A3 decideServer.py	83
	A4 CAST.py	85
	A5 delayTolerances.py	87
	A6 heatmap.py	88
	A7 convert.py	90
	Appendix B – results	91
	B1 Simuation result data Python Algorithms.....	91
	B2 Simulation results MSC	94
	B3 Simulation results VSC	98

List of figures

Figure 4.1: How the proposed model will check for running or planned servers 31

Figure 5.1: illustration of the process of transforming production volume data..... 37

Figure 5.2: map from nordpoolspot.com showing exchange..... 41

Figure 5.3: the process of simulating test runs and recording the results. 42

Figure 6.1: heat map illustrating the energy situation in the Nordics..... 49

Figure 6.2: heat map showing surplus and deficit of energy for areas with a significant share of IRE from March 18th to March 21st 2021. 50

Figure 6.3 line diagram illustrating how many commits the project VS code has per day of the week, on average. 51

Figure 6.4: diagram showing how many percentages of the commits in each project are done in each hour of the day. 52

Figure 6.5: diagram illustrating how cost, emissions and actual waiting time develops as delay tolerance increases for the VSC project.. 62

Figure 6.6: diagram illustrating how cost, emissions and actual waiting time develops as delay tolerance increases for the MSC project.. 63

Figure 6.7: frequency diagram illustrating the frequency of different distances in minutes between one commit and the next. 64

Figure 7.1: Forecast of share of production from different energy sources in the Nordic countries 70

List of tables and listings

Table 4-1: the different grid area codes in the NordPool electricity trading system.	20
Listing 4-1: the CAST-algorithm.....	23
Table 4-2: all variations that will be explored in the analysis phase of the project.	33
Listing 4-2: Algorithm for finding booked servers	32
Table 5-1: example of production data..	36
Table 5-2: the first lines and rows of the csv-file with results from the simulation script run. ...	44
Table 5-3: the parameters for the different scenarios.	47
Table 6-1: the output of the simulation of the base scenario for all projects and seasons.	53
Table 6-2: selected values from spring and winter simulations of scenarios with different IRE threshold.	54
Table 6-3: results of the shutdown scenarios for all projects, summer week.	55
Table 6-4: selected results for the shutdown long and shutdown short scenarios for the VSC project.	56
Table 6-5: selected results of the delay tolerance scenario simulations..	57
Table 6-6: selected results from the simulations with different test durations.	58
Table 6-7: table showing the annual emissions in kilograms of CO2 and cost in number of servers started.	59
Table 6-8: statistic values for the waiting time during active working hours for the scenarios base, short delay tolerance and long delay tolerance.	61

1. Introduction

Today, most communities use digital products like bus ticketing apps, digital maps, newspapers, traffic control systems and digital communication in everyday life. Digital products are everywhere and provide us with access to services like education, health care and other human rights, as well as less critical but widely used services like social media and entertainment. We have made ourselves dependent on software because it increases our efficiency compared to analog methods. It is expected that the demand for digital products will increase as more domains become digitized and larger parts of the world's population gain access to the internet.

Software lives in an ever-changing environment, with new requirements, hardware, laws and regulations and increasing user's expectations. Therefore, the software must undergo rapid changes, and it is of great importance that it still behaves as expected when changes are introduced. Unplanned outages and unexpected behavior can have very costly consequences. One example is the opening of a new terminal at Heathrow airport in 2008, where a software bug caused thousands of bags to be left behind while their flights took off. Over 500 flights had to be cancelled, and the costs have been estimated to be around 50\$ million. Another example is the Knight Capital trading glitch in 2012, where unexpected behavior in newly installed trading software caused the loss of \$440 million.

The antidote to new software failing, is to have a rigorous testing scheme in place. It is not uncommon for testing activities to account for 50% of the development costs in a software project, and for projects where the consequences of a failure have massive impact, it can be even more (Dudekula Mohammad, Katam Reddy Kiran et al. 2012).

A widely used way to ensure the consistency and quality of the software is to run a collection of automated tests, either after a change or at scheduled intervals. The size of the test collection can be as large or even larger than the production code.

Running automated tests is necessary, but it has a cost. The most obvious cost is developing and maintaining the tests themselves, but there is also a cost associated with running the tests,

as they require hardware and energy to execute. Getting small pieces of code to production often is a key metric for successful software businesses, which means that small changes are done frequently, which triggers at least one run of the test suite each time.

The resource usage associated with testing can become significant, and there is no reason to believe it will decrease in the future. Data centers that offer infrastructure as a service charge for the resources used and running tests can be a cost driver for software businesses. In addition, there are indirect costs associated with using computing power. Every time a computational calculation takes place, some amount of electric energy is used. In addition to monetary costs, the production of electric energy often releases carbon emissions that contribute to global warming. This cost is not necessarily reflected in software projects literature but is forwarded to those who experience the most dramatic effect of climate changes.

Data centers have become more energy efficient in recent years, which helps stem their current impact on energy consumption. Virtualization has also contributed to reduced energy spending, as more VMs now run on each physical machine. Still, it is expected that energy usage from data centers and computing in general will increase in the coming years due to an increasing demand (Masanet, Shehabi et al. 2020).

The amount of energy used is not the only relevant factor when in reducing the negative impact a test suite run will have on the climate. Electric energy is produced in numerous different ways, some causing large emissions while others have a small carbon footprint. Wind turbines and solar power plants are considered green production methods, but they also have varying production throughout the day of time and year. Hence, a data center placed near a wind turbine park will not always use green energy.

Moving testing workloads dynamically between data centers based on the availability of green energy would require the combination of the complex and dynamic conditions for local energy production across a large region with the most advanced automated framework in software engineering - the continuous integration pipeline. Software projects have different constraints and requirements for their testing process. Still, they all would have to adapt to the same

conditions of the availability of green energy, which in turn can only be predicted for one day ahead. The aim of this thesis, in broad terms, is to build a bridge between these two concepts.

1.1 Problem statements:

1: Investigate the development of a model which attempts to optimize the organization of tests in an automated test suite with the objective of least energy greenhouse gas emissions.

2: Evaluate whether the model is successful in making a significant reduction in the usage of non-green energy.

2. Background

Imagine you just bought a new phone. You tear open the wrapping and ogle this shiny and expensive new companion which you will carry with you for years to come. You feel the heft of it, the cold metal and glossy screen which radiates performance. Oh, how you look forward to trying out all its new features! Fingerprint recognition, facial authentication and no more typing with human-sized fingers on mouse-sized keyboards. This new phone is *progress*. Then it happens. After all the apps are installed and your data is transferred, you want to log in to your banking app using your smirking, smiling, I-just-got-a-new-phone face. What? No support for the fingerprint reader? Not the other app either? Slowly, the smile turns sour.

Users expect their apps and services to keep up with the development of new technology. Companies who manage to change their product as new requirements arise without introducing defects have a significant competitive advantage in today's market. The ability to constantly deliver improvements and new features without the customers noticing instability is the key factor to make software that drive business value (Forsgren, Humble, & Kim, 2018).

2.1 The rise of DevOps

The term DevOps came to light in 2009 after a presentation named “10+ Deploys per Day: Dev and Ops Cooperation at Flickr” was held at the O'Reilly Velocity conference. The first DevOps days were held in Belgium the same year, and in 2013 the novel *The Phoenix project* was published. The book tells a story about an IT manager who uses ideas from lean manufacturing to break down the walls between development and operations to successfully deliver his seemingly hopeless project. This book is still used as a resource for understanding the DevOps and Continuous Integration (CI) methods, and at the time of writing it is ranked number three on the Amazon list of best sellers in Business production and Operations (Butgereit, 2019).

The more traditional way of thinking, where code is delivered in bulk from the developer team to the operations team, is known to slow down delivery frequency as well as causing collaboration issues and finger-pointing between the teams. DevOps methods are designed to

turn the code a developer is writing into something that has value for the end user fast. This is done by streamlining and automating the processes build, test, and deploy in a Continuous Integration/Deploy pipeline.

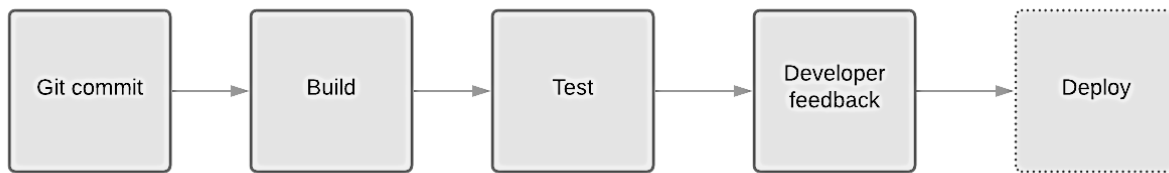


Figure 2.1: a continuous integration pipeline with deploy.

When a developer writes a piece of code and commits it to the version control system, this action triggers a build software where the code is compiled and turned into running software on a server. Next, a series of automated tests are triggered, and the developer is notified as to whether the tests went well or not. If all tests pass, the new version of the software will deploy automatically, and can now be used by the customer.

This process is followed through from end to end, even for the smallest changes. A developer typically wants to commit their work quite often, in order to have many possible points to restore from, so something like fixing spelling errors or changes to indentations can be run through the cycle.

2.2 The importance of automated testing

CI methods are widely used by successful businesses to deliver software fast and efficient (Haghighatkah, Mäntylä, Oivo, & Kuvaja, 2018), but ensuring quality in a frequently changing code base is a challenge that only grows bigger with the rate of change, and it is complicated to design an appropriate testing scheme (Murugan Tanggiah, 2016). The automation of commit and deploy has been in focus since the beginning of DevOps, but fitting tests and quality assurance into the pipeline has received less focus. The deployment is quite similar for most projects, while tests suite collections are unique.

The book Software Engineering by Ian Sommerville is used in educating software engineers and has been around in numerous editions since it first came out in 1982. Still, the topic of software

testing only gets a 23 pages long slot in the 755-page book, and automated testing is not a part of it. A 2017 study found that software engineering graduates had little or no knowledge of automated testing, and many professionals have stopped expecting graduates to know anything about it (Pham, Kiesling, Singer, & Schneider, 2017).

“Automated testing is an entirely new concept to most new hires. High-level test suite design and real-world experience is universally lacking.”

– survey respondent (Pham, Kiesling, Singer, & Schneider, 2017)

Automated testing, it seems, is not on the curriculum in most software engineer degree programs, but there are some highlights. Delft University has launched an online course in automated testing and several other universities have testing courses or DevOps courses where automated testing is a component. Knowledge and skill in automated testing are in high demand by the software industry, but it has gotten little attention at universities so far.

2.3 The need for efficiency in automated testing

Running automated tests with each iteration of the software is a natural part of any CI-pipeline. However, for the largest projects it can be too resource-consuming to run all tests every time changes are made. This would require far too much time, which will delay feedback to the developer and require significant computational resources. Fast feedback is useful, as a developer will spend less time fixing a bug in code that she recently worked on (Saff & Ernst, 2003). Reducing computational resource usage is important, both for saving direct costs and reducing carbon emissions associated with energy consumption. Choosing which tests to run and how to order them are the key factors to testing performance, and some interesting research has been done on these issues. Most research before 2012 has been focused on code coverage-based techniques for bug discovery (Catal & Mishra, 2012), but in later years research on model-based techniques like Test Suite Minimization, Tests Case Selection and Test Case Prioritization have gotten more attention.

Previous findings indicate that the order in which the tests are run is of importance as to how fast a failure is detected. This is important because one can stop the test suite from proceeding further once a test has failed, saving time and costs. There are two main methods of determining the order; historically based test case prioritization (HBTCP) and diversity-based test case prioritization (DBTCP).

HBTCP is based on the idea that tests that have failed before are more likely to fail again and should therefore be prioritized to provoke test failures faster. This hypothesis is supported by research that show significant improvements in running time for test suites that uses HBTCP in continuous integration environments (Hematti, Fang, Mäntylä, & Adams, 2016). Diversity based test prioritization is based upon the idea that if one runs tests that are the most different from each other first, one can find bugs faster.

2.4 Computing and power consumption

Getting feedback fast is one motivation behind making the tests run as efficiently as possible, cost savings is another. Most cloud vendors bill for virtual machines on an hourly basis, so from a cost perspective it is preferable to use as little as possible. In addition, there is the environmental aspect. The fact that computing causes greenhouse gas emissions has gotten more attention during recent years, and headings like “How *thank you* emails are polluting the planet” and “The dark side of cloud computing: soaring carbon emissions” have been observed in newspapers and magazines (World Economic Forum, 2019) (Schmidt, 2010).

Today, roughly 1% of the electric energy produced is consumed by data centers. A study from 2018 estimated that the ICT industry will contribute between 7 and 14% of the total global greenhouse gas emissions by 2035, whereas 44% will come from data centers (Belkhir & Elmeligi, 2018). Other projections are more optimistic, and the industry has done a lot to increase efficiency over the later years. Placing data centers in cooler places to reduce the need for heating, more efficient technology and smart virtualization software makes it possible to do more with less electricity. In fact, data center energy usage has remained almost the same for the last ten years, even though their workflow has increased twelvefold. Still, the International

Energy Agency expects a high increase in demand for data center services in the coming years and state that it is still much needed to keep the focus on data center energy efficiency.

Strong government and industry efforts on energy efficiency, renewables procurement, and RD&D are necessary to limit growth in energy demand and emissions over the next decade. - (International Energy Agency, 2020)

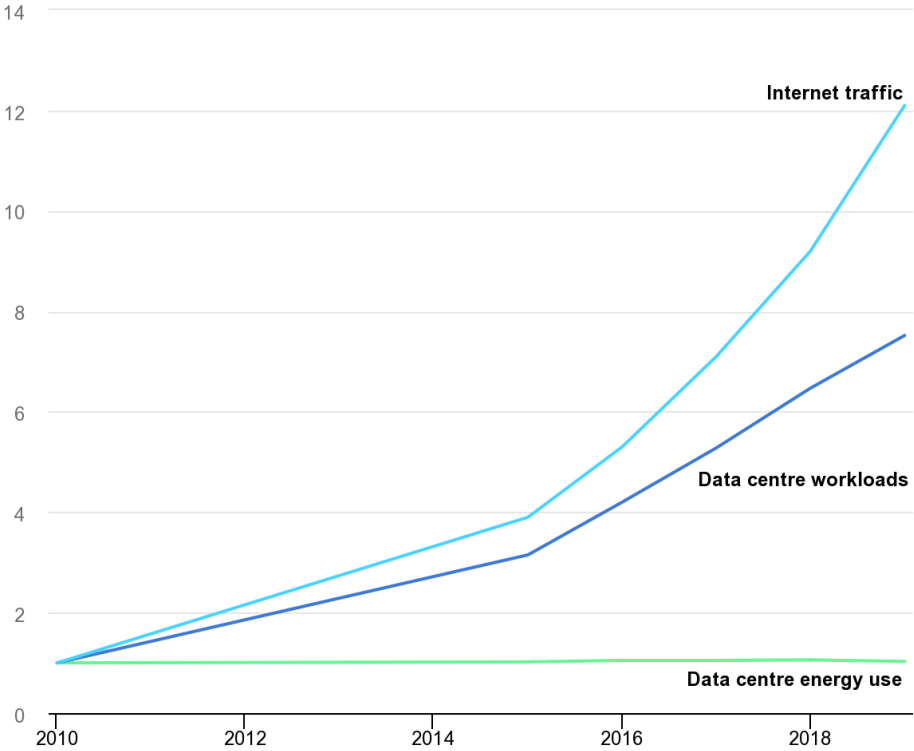


Figure 2.2: IEA, Global trends in internet traffic, data centre workloads and data centre energy use, 2010-2019, IEA, Paris <https://www.iea.org/data-and-statistics/charts/global-trends-in-internet-traffic-data-centre-workloads-and-data-centre-energy-us>

Many of the larger cloud service providers, like Google and Microsoft buy green certificates to ensure the energy they use comes from green sources. Some of them, like Amazon Web Services also invest heavily in green energy projects like wind and solar power plants (Amazon, 2020). Even so, they cannot guarantee that the electricity they use at any given time comes from a green source, as the production of green energy is variable and does not always match the demand of the data centers.

2.5 The properties of electricity as a commodity

Electricity is the energy carrier used to power the data centers, and it has some special properties compared with other commodities. Unlike oil, corn, or water, it cannot be stored in larger quantities for later consumption. There are batteries, but with today's battery technology they lack the capacity to store large quantities of energy in a cost-efficient manner. Hydropower can also be stored in dams (impounded hydropower), but this requires special geographical characteristics. Electricity, as a rule of thumb, must be produced and consumed at the same time.

In addition to the production and the consumption happening simultaneously, there are also constraints in the power grid systems which limit where electricity can be used. Exports from one grid area to another are made possible with high-capacity power lines, but only up to a certain capacity. This means that a region that produces a lot of electricity must consume all of its surplus which cannot be exported. Conversely, a region must always produce whatever electricity it needs that exceeds the import capacity.



Figure 2.3: map showing the exchange capacities in the Nordic region. Source: nordpoolgroup.com

In countries with mature energy markets, the current energy price is a product of supply and demand. The supply of electricity is composed of a multitude of different production sources, which is commonly referred to as the power systems energy mix. Some of the production sources deliver steadily at all hours of the day and time of year (constant energy sources), such as thermal power plants which use coal or nuclear power sources. Ramping production up or slowing it down takes a long time with these technologies, and they are usually most effective when producing at a particular level. Other sources are more unreliable, like wind and solar based plants where the production output varies with the weather (intermittent renewable energy, IRE).

We also have some production types that can regulate their output on short notice, such as gas-powered and impounded hydropower plants (controllable energy sources). Impounded hydropower depends on the presence of mountains and lakes, which are not available in most parts of the globe. Gas-powered plants are therefore often used to regulate production on short notice.

On the demand side, there are also large variations which are affected by factors such as temperature, light and daily consumption patterns. A typical day has a peak in demand at around 0730 when people take a shower and cook breakfast, and a second and slightly larger spike at around 1630 when people are preparing dinner. This pattern is often referred to as a devil-heads curve.

2.6 The role of renewables

Renewable energy is becoming increasingly popular and with wind turbines and solar panels being installed at a rapid pace in Europe and the US. From 2000 to 2020, the production from wind turbines in the U.S. increased from 6 to 338 TWh (U. S. Energy Information Administration, 2021). This is still a small part of the global energy mix, 8.4% of total production in 2020, but it has already surpassed hydropower as the largest renewable energy source in the United States. The numbers are similar in Europe where wind production was 311TWh in 2017, which amounts to 11% of the total production.

Electricity production from solar and wind is expected to increase in the years to come. In the United States they have introduced a production tax credit for these types of production. In the European Union, the target is to have 32% of production from renewable sources by 2030 (European commission, 2021).

2.7 Supply and demand imbalance

An increased share of renewables in the energy production mix is beneficial for reducing CO₂-emissions that come from fossil-fuel powered energy production, but it also brings certain challenges. The wind does not necessarily blow at the same time we need electricity to cook dinner. This means that there is a mismatch between supply and demand, and unlike what we do with other commodities, storing the commodity until demand catches up is not an option. Too little production will cause a black or brown-out, and too high production will cause overheating of the power grid. A study done on the power grid of the Hokkaido region of Japan concluded that when the production of IRE amounts to more than 20% of the production in an area, the need for balancing measures increases significantly (Otsuki, Komiyama, & Fuji, 2017).

Traditionally, there are five types of solutions proposed for this mismatch:

1: Regulate demand by price differentiation. By making electricity more expensive when production is low, consumers have an incentive to move their consumption to a particular time. This can nudge people to charge electric cars at nighttime and turn down the heat at certain times of the day. Still, little can be done about the fact that most people shower in the morning and make dinner in the afternoon, so this can only solve parts of the problem.

2: Regulate supply by ramping up and down production. This can be done by regulating the water flow in an impounded hydropower plant, or more commonly by ramping up the production from a natural gas-powered plant. As natural gas is not a renewable energy source, it is preferable to minimize the use of this possibility. It is also possible to regulate production from thermal plants like coal and nuclear, but the regulation is slow, and the plants normally only operate at full efficiency at certain levels.

3: Expand transmission capacity. Due to various climate and geographical conditions, energy production from renewable sources will differ from area to area. By moving the energy to where it is most needed at any given time, mismatches between supply and demand can be reduced. This is the most widely used solution, and most energy systems have some sort of exchange towards their neighbors.

An example of what can happen without exchange was seen in Texas, USA, in February 2021. For political reasons, Texas is not connected to the national power grids, they run an isolated system without exchange. When unusually cold weather caused their production units to fail, they could not get power from their neighbors. The result was a massive blackout, leaving 4.5 million homes and businesses without electricity for several days. Prices for those who could access electricity rose to a level that would be impossible to pay for the middleclass household.

Transmission capacity is effective, but it cannot remedy the problem completely. Sending large amounts of electricity across multiple time zones is not viable with today's technology, both due to high building costs and energy loss during transportation.

4: Move demand to where the production is. This is not a realistic path when it comes to household consumption, but it is a good fit for energy-intensive industries. An example is Iceland, where they have easy access to geothermal energy that can be used for electricity production. This electricity cannot be exported from Iceland directly, but it can be used to produce aluminum, which is a process that requires large amounts of energy. Exporting aluminum becomes a way of exporting energy. This is, however, most useful where there are power sources that can produce a steady load, like geothermal or hydro. Data centers could also be placed near such sources of renewable energy, but there are other considerations to take as well when placing a data center, like infrastructure, climate, regulations, and security.

5: Produce where the demand is. De-centralizing production by putting solar panels on the roofs of homes, offices and industrial buildings have been done for some time and this development is still ongoing. This is especially popular in sunny areas with high energy costs like California, US. Some also build small scale hydropower plants near farms or production facilities

where this is a possibility. This kind of production can help with the geographical distribution, but it does not solve the problem of supply and demand imbalance over the hours of the day.

All in all, there are myriads of different causes and remedies for supply and demand imbalance, and we know that one of the causes, intermittent renewable energy (IRE), will grow larger in the coming years. Hence, it is necessary to increase the efforts on mitigating actions in order to successfully reach the goals of a higher share of green energy in the years to come.

2.8 Can testing become a part of the solution?

As there are financial and practical gains to getting fast feedback from tests to the developers, it is not advised to schedule the tests to times with a high supply of green energy. However, with today's vendors providing multiple datacenters at different locations, it becomes possible to increase demand in areas with a currently high production by running tests in data centers located in these areas. This ability to dynamically shift the geographical location of the demand of a commodity is quite unique to the IT industry. Like what is done with aluminum production in Iceland, re-locating the workload of software testing becomes a way of exporting energy, but much more dynamic.

Automated software testing is particularly well suited to be moved around following green energy production. The tests are independent in nature, one test does not depend on another. They also do a considerable workload, so there is some energy usage involved. The software to test must be built new for each test, making it easier to move it around than other parts of the IT system like a mail server or a database. Because of the need for fast feedback, tests cannot be shuffled around in time like one can with maintenance batch jobs either, so a movement across geographical areas is a better candidate for ensuring green energy usage.

Some research has been done on the topic of data center allocation and energy savings. A 2012 study showed possible financial savings of 15% for a cloud provider that could forward requests from its end users to the data center with the lowest energy price at the time (T. Sakamoto, Yamada, Horie, & Kono, 2012). Other studies have looked at how a cloud provider can "follow the green energy" by routing traffic to the areas with the highest production of IRE.

One study used numerical experiments on data from real traffic to data centers and renewable energy production in the US to investigate how data centers could be powered with as much green energy as possible without large-scale storage. They found that wind was more useful than solar power for this purpose because wind production has low correlation across geographical locations and is available at all hours of the day. They also found that geographical load balancing could significantly reduce the required capacity of renewable energy to power the data centers (Liu, Wierman, Ling, & Low, 2011).

Another study designed a prototype for applying geographical load balancing to web application requests using the available renewable power and estimated electricity price at each data center. The study used real meteorological data and realistic workloads from logs of web requests to Wikipedia. Their simulations showed reduced use of gray energy, under the assumption that the data centers produced their own green energy from local installed production capacity (Toosi, Qu, Assunuco, & Buyya, 2017).

A study from 2011 found that it was possible to use 95% energy from green sources without delaying processes or jobs by geographical load balancing of incoming requests. The study used the data center power consumption ratio to the effective wind production as the allocation criteria. (Gao, Zeng, Liu, & Kumar, 2013).

3. Approach

To investigate the effect allocating test runs could have on greenhouse gas emissions, an exploratory approach will be used. Experiments with different variations of a model will be done before analysis of the results. Testing the same scenario with different inputs and threshold variables can also be done to reveal how different factors affect the outcome.

To make discoveries that could be relevant in a business setting, an imagined case scenario will be used to guide the design work. The case can be described as envisioning being a DevOps engineer given the task of making the test pipeline as green as possible with minimal delay to the development process. This will require that the engineer balances many considerations at once.

One must consider the energy usage and greenhouse gas emission from testing. At the same time, one must consider the operational part of it. As previously discussed, rapid deployments are a key success factor for an IT business. Neither colleagues nor management would accept slower processes and code being delayed on its way to the customer. Lastly, the cost perspective is important. One cannot ignore the fact that some ways of doing things are more expensive than others. An efficient solution for climate footprint reduction will probably not be acceptable if the costs are too high. Most likely, the DevOps engineer will have to compromise in the search for a green but viable solution. Having these constraints in mind while experimenting will hopefully lead to knowledge that is usable within the constraint that businesses operate within.

Which exact algorithm to use will be explored in the design phase, but it will need to provide useful data for analyzing the greenhouse gas emissions in simulations, both with different patterns of tests and with and different approaches for allocating the workload in areas with a low carbon footprint.

In the design phase different sources of input data will be looked at, and a procedure for fetching and transforming data to a useful format will be described.

In addition to exploring results in what can be considered normal operations, the model must be tested for robustness under more unusual conditions by looking at edge cases. What happens if a lot of tests are being triggered at the same time? Or what will it do in periods with little or no intermittent green energy production? Things like this do happen so testing for such cases is important to make an algorithm that could perform under real-life conditions.

Different data sources must be examined to find suitable data for the task. We need realistic data on the following in order to achieve meaningful results:

- Surplus of green energy from intermittent green energy sources relative to the locations of different data centers. There are many energy markets in the world where one can find data for production of electricity from different sources, consumption, and exchange. Because of this, it should be possible to use realistic data for this part of the project. For an experiment to give useful results, it is important to look at data that is representative for a real-life scenario in an existing electricity market.
- Greenhouse gas emissions caused by running the tests. This will probably have to be calculated as a function of CPU usage or server minutes, or stipulated as a function of certain factors, like the comprehensiveness of the test.

The model will not be implemented in a tool for automated testing, as this would not contribute any useful information towards the problem statements. However, it is still an ambition to create something which could be used in a CI tool like Jenkins, Travis CI or TestProject for JenkinsX, eventually.

4. Design

This chapter describes the steps that were undertaken in order to form a working model, the model itself and the data collection. Decisions about which data to use and how to generate data where realistic data could not be obtained are also explained here.

4.1 Geographical area for test data

Electricity is provided all over the world with different pricing schemes and market regulations. Some of them are considered primitive, with no dynamic price mechanisms. Others are closed towards their neighbors and have no exchange. For this project, we need to assume that there is a market that employs market-based pricing for each area and where data is publicly available. It also needs to have a significant amount of installed production capacity for IRE (intermittent renewable energy, like wind and solar).

The Nordic power exchange NordPool fits with these criteria and is therefore suited for the project. In real life, a software developer can choose from data centers across the globe, but for this model the assumption is that tests must be run within a limited geographical scope. This is not an unlikely situation, as many will choose to stay within one area due to regulations like the GDPR, trust between countries and areas, or local legislation. Even though the Nordics is chosen for this project, another area with publicly available data and a sufficient amount of IRE could have been used.

NordPool calculates the electricity spot prices for Denmark, Estonia, Finland, Latvia, Lithuania, Norway, and Sweden. Norway, Sweden, and Denmark are divided into multiple grid areas based on transmission capacity constraints (see figure 2.3). This leaves us with the following areas that can be used for the model:

Country	Grid area codes
Finland	FI
Estonia	EE
Latvia	LV
Lithuania	LT
Sweden	SE1, SE2, SE3, SE4
Norway	NO1, NO2, NO3, NO4, NO5
Denmark	DK1, DK2

Table 4-1: the different grid area codes in the NordPool electricity trading system.

There are data centers in most of these areas. Data from Baxtel, a commercial data center information site, show that cloud data centers are placed in all areas except Latvia and Lithuania. It is not realistic that a software developer has access to all these data centers, as most stick to one or perhaps two cloud vendors. Still, for the sake of the model, we will include all areas except Latvia and Lithuania.

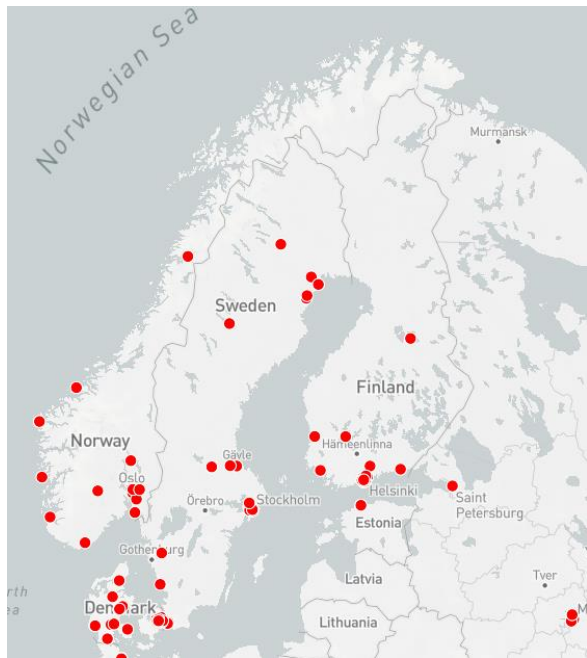


Figure 4.1: map showing cloud data centers in the Nordic countries and the Baltics. Source: baxtel.com/map.

The geographical areas available for placing a server for testing is defined in the model as a list:

$$AREAS = [A_1, A_2, \dots, A_n]$$

4.2 The relationship between exchange cables, IRE surplus and energy prices.

To measure where energy consumption causes the least carbon emissions, one approach is to look at the carbon intensity in the area. The carbon intensity represents the total emission per unit of energy consumed (grams of CO₂ equivalents per kWh). However, in the Nordic countries, this would not be ideal. Because of its mountains, Norway has a large percentage of hydropower in the mix. Where other countries use nuclear and coal to supply the base load, Norway is one of the few countries in the world that uses hydropower for this purpose. If one looked at carbon intensity alone, all workloads could simply be placed in western Norway.

Simply placing the data center near a hydropower-plant would ensure that the supplied energy causes no greenhouse gas emissions, but it would not be helpful towards incentivizing more green energy investments. Neither does it consider the demand side of the equation: maybe the citizens and industry in this area already need the energy that is being produced? If that is the case, placing the load here would simply result in an increase in energy prices, and energy being imported from areas that could have less green production.

What we are looking for is a surplus of intermittent green energy in an area that can be utilized for running tests. Energy production is the obvious factor to look at but looking at production alone will not be representative for where there is a surplus. We know that demand in an area varies greatly, so a production level that causes surplus at night might not be sufficient to cover demand in the daytime. In addition, there are neighboring areas that could also use the produced energy if they have an energy deficiency at the time, as well as exchange cables allowing it to happen. Energy will flow from areas with a surplus to areas with a deficit, until the constraints on energy transportation are met.

If the cables are not fully utilized, the neighboring areas will have the same energy price. Similarly, different prices in two areas mean that the cables are fully utilized, and the area with the lower price has a surplus compared to its neighbors. A surplus of intermittent renewable energy is characterized by the following properties:

- The production of IRE is a significant portion of the total production. Previous studies have calculated this to be around 20% (Otsuki, Komiyama, & Fuji, 2017). In the

simulations, 20% will be used as the base threshold for determining that an area has a significant share of IRE in the production mix, but experiments will also be done with other thresholds. This threshold will have the notation *IRE_THRESHOLD*

- The neighboring areas cannot utilize the surplus because of transmission constraints, which causes higher prices in the neighboring areas.

To express whether an area has a relevant share of IRE, the following is included in the model:

$$IRE_SHARE = \frac{PA_{IRE}}{PA_{Total}}$$

Where PA_{IRE} is the amount of electricity from wind and solar PV produced in an area, and PA_{Total} is the total production of electricity for that same area. The concept of neighbors is described as a nested set of values:

$$NEIGHBOURS = \{A_a [N_{a1}, \dots, N_{an}], \dots, A_n [N_{n1}, \dots, N_{nn}]\}$$

Where A is the production area, and N is an area with exchange to A that is within the same price system.

The price delta for an area is given by:

$$\Delta p = \sum_{n=1}^S pA - pN_n$$

Where S is the number of neighboring areas with exchange, pA is the areas price and pN_n is the price in the neighboring area. A negative delta indicates a surplus of energy in the area.

Based on the above, an IRE surplus is defined as an area where both the following conditions are met:

$$IRE_SHARE \geq IRE_THRESHOLD$$

$$\Delta p < 0$$

4.2.1. Introducing the CAST algorithm

The concepts of IRE_SHARE, NEIGHBOURS, and price delta, together with the conditions for IRE surplus, can be combined and used to locate areas with IRE surplus. In this project, they are used to create the CAST-algorithm.

The CAST-algorithm is used for deciding where to place the workload and it is described by the following pseudo-code:

Listing 4.1: the CAST-algorithm

```
1   For each area with data center
2       If energy IRE_SHARE >= IRE_THRESHOLD
3           Add to array of candidates.
4   Workload area = null
5   For each area in array of candidates
6       Calculate price delta to neighbors ( $\Delta p$ ).
7       if price delta to neighbors < 0 AND
8           price delta to neighbors ( $\Delta p$ ) < workload area price delta
9           workload area = area
10  return workload area.
```

After adding all areas where the condition of a surplus is met to a collection of data, the one with the lowest price delta is loaded to the variable workload area and returned. This variable holds the value of the area with the best conditions for using surplus IRE. If none of the areas have a surplus, the code will return null, which can be used to trigger the fallback option of starting a server in an area without IRE surplus.

4.3 Energy data sources

To calculate surplus according to the previous section, we need the following data:

- The production of IRE per area per hour
- The total energy production per area per hour
- The energy price per area per hour

Many different APIs are available that could be used for obtaining production volume data from wind and solar power. Both Statnett and Energinet provide APIs that can be used to see the volume of production right now. These are the transmission system operators in Norway and Denmark and deliver reliable data. Statnett's data, however, is aggregated per country and Energinet's data is only available for Denmark. There are also vendors who supply real-time data per grid area, but accessing their APIs is quite expensive.

Another approach could be to use the forecasts for production in each geographical area. Wind production forecasts are available for the next day, per hour divided per grid area from NordPool, except for the Norwegian areas. For the Norwegian areas, wind forecast data can be downloaded from Entso-E (European network of transfer system operators). The forecast data together with the calculated spot prices could be used to calculate a profile of the most desirable area per hour for the coming day.

On NordPool, both the producers and retailers must submit orders on how much electricity they plan to buy or sell for each hour in the coming day before 1200 hours CET. The day-ahead price for every hour of the coming day is then calculated for each area. Retailers are the companies the consumer buys electricity from, they operate as a link between Nordpool and the end user.

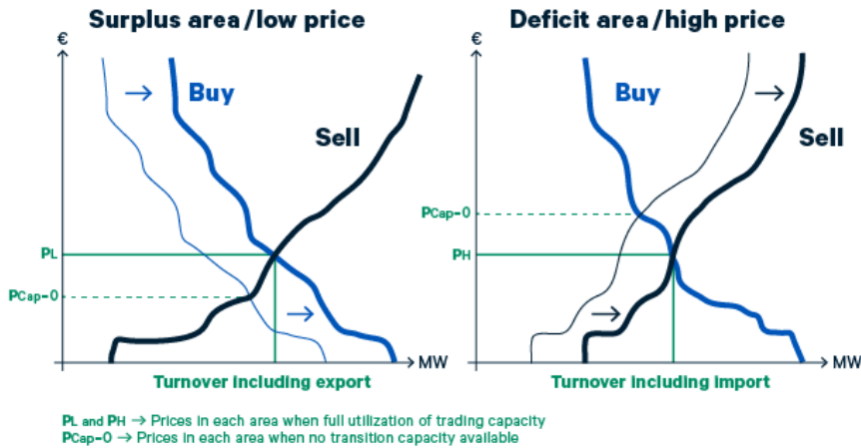


Figure 4.2: price calculation curves per area on NordPool. The arrows shows how the supply shifts when exports or imports are added to the calculation. Source: <https://www.nordpoolgroup.com/trading/Day-ahead-trading/Price-calculation/>

NordPool and Entso-E also supply data for historic hourly wind and solar photovoltaic production (PV) per price area which can be used for simulations. Solar PV is not always an indicator for where it is best to move the workload. High production of solar PV can be correlated with high temperatures, which again calls for more energy spent on cooling. One possibility is to include solar PV only when temperatures are below a given threshold. In this project, production data from solar is included entirely, as it amounts to less than 0.5% of the total production in the area used. Data for solar PV production and for wind production in the Norwegian areas are obtained from Entso E.

4.4 The cost of deploying a test server.

In DevOps, rapid deployments are considered beneficial and there are many systems and technologies developed with this in mind. Container-technology like Docker makes it easy to deploy new and changed software with little effort. Still, a deployment is not without costs. Large and complex systems can have an overhead to each deployment, and deployments can fail, which makes it preferable to limit the number of deployments during a day. Implementing a model that allocates tests to different servers requires the user to have the complex infrastructure in place that enables deploying without manual intervention.

A central question any model in our context must answer, is whether the test infrastructure should be expanded to a new location preemptively based on pre-planning, or if it should be done ad-hoc as part of the test.


		Deploy	
		Ad-hoc	Preemptive
Decide	Ad-hoc	<ul style="list-style-type: none"> • simple algorithm • resources scaled to usage • expensive - many deploys • risk of there not being available capacity 	
	Preemptive	<ul style="list-style-type: none"> • Capacity reserved and possibly cheaper • expensive - many deploys • More complex algorithm • risk paying for unused resources 	<ul style="list-style-type: none"> • Fewer deploys • risk paying for unused resources • less dynamic for allocation optimization

Figure 4.3: matrix showing the pros and cons of choosing ad hoc or preemptive methods for deciding allocation and deploying the test environment.

The table above illustrates four different approaches as to when to decide upon allocation and when to deploy the test environment. As we cannot deploy the test environment before we have decided where to deploy it, the upper right quadrant of the matrix is crossed out. This leaves us with the following options:

Decide allocation on demand and deploy on demand:

This would be the method most in compliance with the DevOps philosophy, and it would require a simple algorithm without prediction. However, depending on the overhead associated with a deployment, this option might be costly. One also risks that there is no available capacity at the optimal location at the time. This approach would be good in cases where the cost of deployments is moderate.

Decide allocation preemptively and deploy on demand:

This approach would call for a more complex algorithm that calculates the best allocation for a coming period and reserves data center capacity at the optimal placement, thus securing capacity at a lower price. However, this requires forecasting the need for deploys, which might not be feasible. Also, one would risk reserving and paying for capacity that was never used, and the number of deploys still is the same as with the previous approach. Doing it this way would be suitable in a situation where the cost of deploying is moderate, there is a predictable pattern of tests, and data center capacity constraints is a problem.

Decide allocation and deploy preemptively:

To do this one would have to reserve capacity in the forecasted optimal allocation for several time periods, for instance the next 6 hours, and deploy there. Doing this comes with the risk of paying for unused capacity. An approach like this could be a solution for projects where deploying is costly.

4.5 The significance of the hour of the day

Energy demand follows a pattern, where the demand is higher in the mornings and afternoons on weekdays, and less during nighttime and on weekends. Energy prices are highly volatile, and even in the Nordics, which are considered to have relatively low volatility it is normal that the most expensive hour of the day is double the price of the cheapest one.

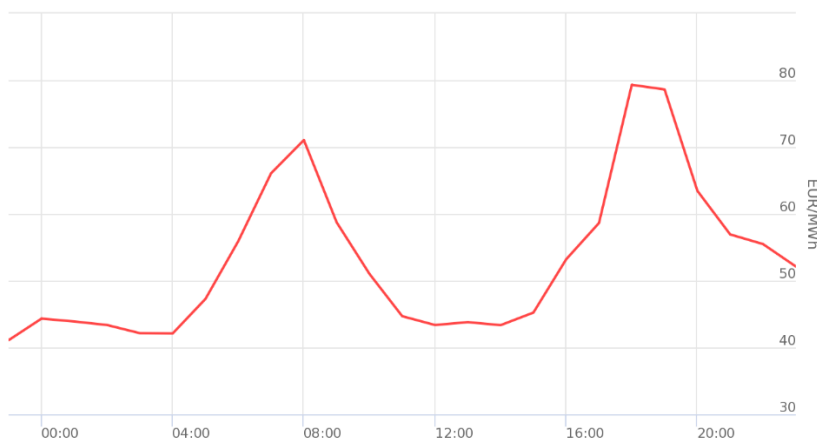


Figure 4.4: Spot prices on Nordpool for the area DK2 on March 2nd 2021. The graph shows a typical "devils head" curve, where increasing demand causes prices to rise in the morning and afternoon. Source: nordpoolspot.com

It is expected that the volatility will increase in the coming years, as the fraction of IRE is increasing (Statnett, 2018). In the scenario where an engineer is committing some code at the end of her workday, seeing the results right away is not important. The same goes for other types of tests like scheduled tests for performance and security. Moving these workloads to a less busy time could make it possible to utilize more IRE.

Traditionally, a DevOps pipeline does not have any concept of waiting. The whole idea is to get fast feedback and ship code rapidly. Looking at the curve presented above it is clear that during peak hours, there can be gains from waiting, especially if one can wait for hours. The NordPool markets spot pricing is per hour. Other markets, like the German, trades in 15-minute intervals. There are ongoing discussions about moving to 15 minutes resolution for the Nordics, but no plans are made at the time of writing. This means that for this model, a short delay of a few minutes will only be useful if the tests are ordered right before the hour changes.

Data centers also bill per hour, so that once a server is running it makes little economic sense to take it down before one hour has passed. We might see a change in the billing system of cloud providers, but this project will use the current scheme in order to make the conditions as realistic as possible. From an operations perspective it therefore makes sense to group tests together and run as much as possible within one server-hour. If we considered green energy alone, we might start and stop servers without consideration for what was already running and what tests are expected to be triggered in the future. This can be an option in the future, when low/no energy operating systems are more widely used. Still for this model the economics of operating in the cloud will also be addressed. The model must consider which servers are already running, and whether we have other tests waiting to be executed.

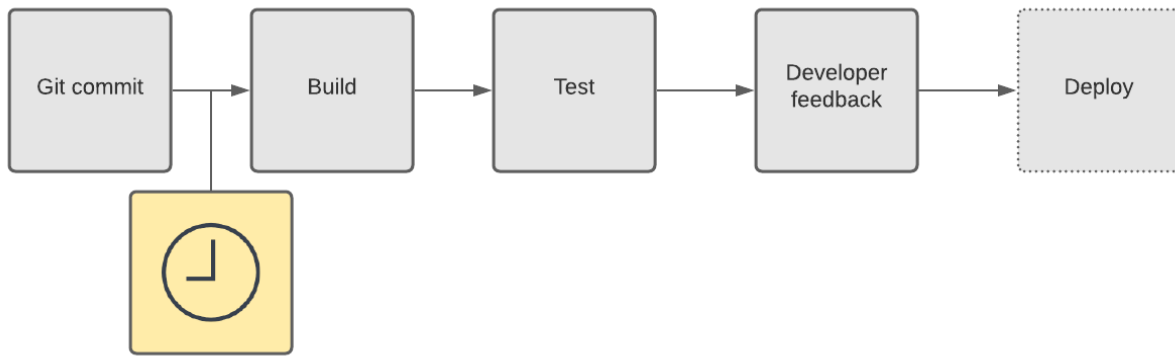


Figure 4.5: CI/CD pipeline with waiting time inserted between code commit and build.

To investigate the possible gains of waiting, the concept of delay tolerance is included in the model. Delay tolerance is defined as the amount of time it is acceptable to wait before running the tests. The delay tolerance will be considered when the algorithm decides where and when to start a new server.

In a real-life scenario there might be some hours of the day that are off limits for running tests. For instance, it is not uncommon that database indexing jobs are scheduled to run during night hours, or other infrastructure maintenance to take place. This will affect performance, and tests running time. For tests that query the database, this can cause false positives due to timeouts.

4.6 Data center capacity and pricing

To run tests, one does not only need energy which is what has been discussed so far. One also needs free capacity in a data center. The cloud providers all market that they can scale up and down on short notice. They can do this, but they still want their users to spread the load across the hours of the day and to reserve capacity beforehand. This is reflected in their pricing schemes. A common price strategy is to bill the computing units per hour which discourages rapid up and downscaling (Mazrekaj, Shabani, & Sejdiu, 2016). Some cloud providers also offer to bill per minute or second, but this is priced at a significantly higher rate and will only be cost effective in cases where one needs the VM for a very short amount of time. Amazon Web Services also has spot pricing, where prices are dynamically determined based on supply and

demand. The model used for this project will consider the most common strategy, billing per hour.

Most cloud providers also operate with different prices for each data center. These variations are usually small within a limited area, and larger if one compares prices in different continents. The CAST algorithm will in this project operate within a limited geographical area, and the price differences between data centers will therefore not be considered.

4.7 Status and reservations of virtual machines in the cloud

As most cloud providers bill for one hour each time a server has started, it makes economic sense to run it for one hour once it is started. Because of this it can be assumed that it is always better to run tests on a server that is already up if the test can be completed before the end of the current server-hour.

On the other hand, it is preferable to not have idle servers running for the sake of saving energy, which means the model should aim to shut down servers where it is not expected that more tests will be triggered before the hour is done. The model needs to decide upon the following questions each time there is a commit:

- Is there a running server or a scheduled server that can be used for this test run?
- Should the server be shut down after the test run, or left running for the remainder of an hour?

The following initial assumptions are made:

- How long a test will run is known before running it.
- A server has a defined capacity.
- The testing tools will use the full available capacity of the server.
- When a test has very low or no delay tolerance, we expect more test orders to come in soon.
- There are no hours that are unusable due to scheduled jobs running.
- We can reserve capacity in data centers.

The implemented model will check for possible utilization of running or reserved servers before ordering new servers.

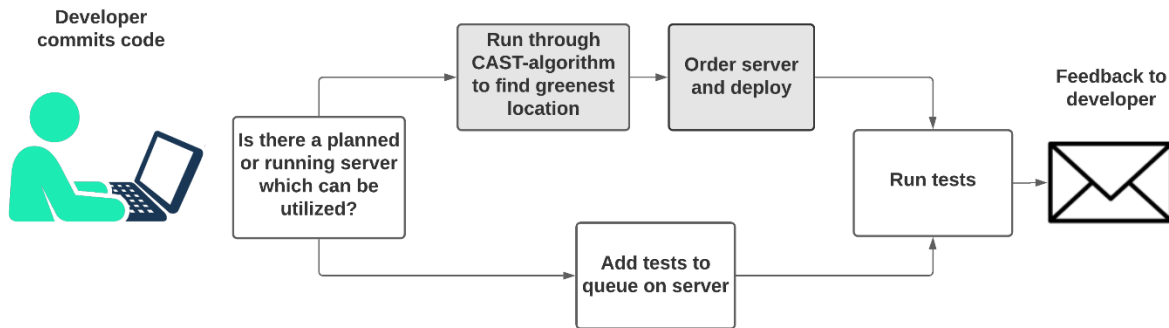


Figure 4.1: the diagram illustrates how the proposed model will check for running or planned servers before either running the decision algorithm described in chapter 4.2 or adding the tests to a server already planned for.

This design relies on a database that records all ordered servers along with a parameter that is set for whether the server should be kept running after the test is done. The algorithm for checking for usable servers is described in the following pseudo-code (all time variables are in epoch-time format):

```
1 Input: commit time, delay tolerance and test duration
2 Query the database for records where:
3     Shutdown variable == false AND
4     end of planned tests on server <= time of commit + delay tolerance AND
5     Server startup + 60 minutes >= time of commit
6 If query returns results:
7     Variable Start_time_for_incoming_test = min. value of planned server
           test end or commit time
8     Variable time_Left_To_Run_Tests = (Server startup + 60 minutes) -
           Start time for incoming test
9     If time time_Left_To_Run_Tests >= test duration
10        Return query result
11 Query the database for records where:
12     Shutdown variable == true AND
13     Server start time <= time of commit + delay tolerance AND
14     end of planned tests on server >= time of commit
15 If query returns results:
16     Variable time_Left_To_Run_Tests = (Server startup + 60 minutes) -
           Start time for incoming test
17     If time time_Left_To_Run_Tests >= test duration
18        Return query result
19 Else return 0
```

4.8 Scenarios for test triggering

In order to analyze the outcome of using the proposed model, it is necessary to find data that represent a realistic series of tests being triggered from developers on a project. This can be accomplished by obtaining the logs from the version control system. For this project, it is desirable to explore the possible outcomes from different types of situations. It is therefore decided to obtain data from three kinds of projects:

- Project 1: A large open-source project with commits coming in from different parts of the world at a high frequency.
- Project 2: A smaller commercial project where all developers work during office hours in one time zone.
- Project 3: A smaller open-source project with little activity.

4.9 Test plan

In order to see how the different parameters affects the server placement and utilization, it is necessary to run several simulations with different parameters and input.

	IRE_Threshold	Delay tolerance	Shutdown parameter limit	Delay tolerance profiles	Test duration	Season
Project 1	Baseline, Low, One	Baseline, Long, Short	Baseline, Zero, long	Static / variable	Baseline, Short, long, High variance	all
Project 2	Baseline, Low, One	Baseline, Long, Short	Baseline, Zero, long	variable	Baseline, Short, long, High variance	all
Project 3	Baseline, Low, One	Baseline, Long, Short	Baseline, Zero, long	static	Baseline, Short, long, High variance	all

Table 4-2: all variations that will be explored in the analysis phase of the project.

The baseline scenario is the one where the most realistic data is used.

4.10 Calculating greenhouse gas emissions from test runs

There are countless ways of estimating the carbon footprint from server usage, but they all depend on information that is unavailable to the consumer of cloud services. To precisely figure out how much energy a computational task uses, one needs to know what kind of server is used, how much energy is used for cooling and other information of the hardware and utilization of the hardware that is installed in the data centers (Mytton, 2020). The organization GoClimate has developed a carbon calculator based on commonly used servers energy usage, server life span of four years, data center energy efficiency and the carbon footprint of the Nordic Energy Mix (GoClimate, 2019). They arrived at the following conclusion:

- A cloud server using 100% green energy will account for 160kg CO₂ per year.
- A cloud server using non-green energy will account for 458kg CO₂ per year.

The numbers from GoClimate will be used to estimate carbon footprint effect of the experiments. This will provide an estimate of the emissions, but it will be quite unprecise, as it uses an average and does not separate active CPU minutes and idle server minutes.

5. Implementation

This chapter describes the process and choices that were made to transform the model described in the previous chapter into code that can produce useful outputs. The aim is to be able to explore how allocation and management of test servers can impact the environmental effects of testing, in order to answer the problem statement.

5.1 Tools

To transform the described model to code that can be used for simulating different scenarios of testing, a Python script was written.

The TinyDB library for Python was used to construct a database with production data, prices, and server reservations. In a real-life scenario, one would use an API to request production and price data for each calculation, but this is not viable if we want to see how the model works at different times of year. Historical data is therefore written to json-files that can be accessed by querying. This also enables us to repeat experiments on the same time period. For transforming data to database-files a combination of excel, notepad++ and python scripting was used.

For data visualizations, the python tools NumPy and Seaborn were used for heatmaps, as well as Excel for tables and line charts. Accessibility was given the high priority when choosing colors for the visualizations.

5.2 Data sources for production volumes and prices

5.2.1. Time period selection

To get test data for the model, four weeks spread throughout the year were selected as sample data. The weeks 31, 43, and 51 of year 2020, as well as week 12 from 2021 were chosen. The selection was done by taking the latest week from the time of writing and iterating backwards per three months in order to get one sample per season of the year. This is important for capturing the different seasonal patterns of IRE production. Because commit history data is used to simulate test orders, it is preferable to avoid holiday weeks. Therefore, week 31 was chosen for summer season data.

5.2.2. Production volume data selection and transformation

To calculate the share of IRE, data on total production and IRE production per area for each hour in the time period selections is needed. Not all necessary data was available from one source, so different sources had to be combined. All data on production totals was collected from Nordpool. Wind production data from areas Estonia, Denmark and Sweden was collected from Nordpool. For Norway and Finland, wind production data was not available from Nordpool, so they were gathered from Entso-E. Production from solar PV is only done in Estonia and Denmark, and the data on this was collected from Entso-E as well.

Downloaded data from Nordpool comes in an one excel-file per country. From Entso-E data is fetched over an API which returns an XML-file. Several tools were used in the process. Firstly, the data from Nordpool was pasted into one Excel-file with three tabs, one for total production, one for solar PV production and one for wind production. Thereafter, data for each time period and area was collected from Entso-E in separate XML-files. The files were stripped of all tags so that the timeseries-data of production volumes was all that remained. This was done by a series of regex-manipulations. After this, all the time series from the Entso-E data were pasted manually into the excel-spreadsheet. The data from wind and solar were added to produce one sheet of data for total production volumes of production from IREs.

date	hour	SE1	SE2	SE3	SE4	FI	DK1
19.10.2020	0	2926	7278	8776	629	6223	1674
19.10.2020	1	2555	7060	8738	606	6245	1566
19.10.2020	2	2423	7047	8744	606	6358	1481
19.10.2020	3	2375	7022	8725	612	6430	1516
19.10.2020	4	2421	7236	8745	632	6663	1517
19.10.2020	5	2742	7546	8779	665	7315	1674
19.10.2020	6	2966	7884	8880	696	7975	1944
19.10.2020	7	3024	7852	9265	770	8487	2065
19.10.2020	8	3383	7661	9305	795	8699	1917
19.10.2020	9	3377	7473	9183	802	8651	1933

Table 5-1: example of production data. The table show total production volumes for the first 10 hours of October 19th 2020 for 6 areas. The production volumes are in given in megawatt-hours (MWh).

To prepare the data for use in a python script, the data for IRE and total production was uploaded to a tool that converted it from excel to JSON format. Lastly, the json-files were run through a script that inserted each record of data into database-files readable by TinyDB querying.

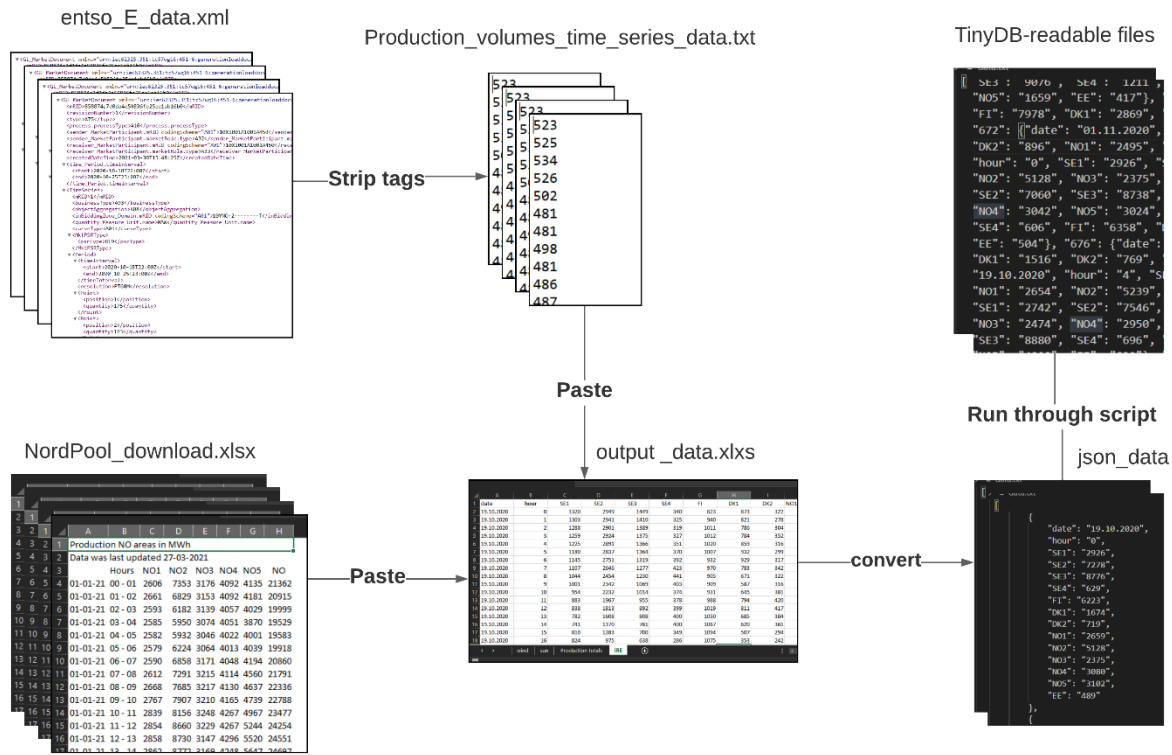


Figure 5.1: illustration of the process of transforming production volume data

After completing this process, two database files were produced and ready for use:

- Volumes_IRE.json
- Volumes_total.json

5.2.3. Price data selection and transformation

Price data was collected from NordPool for all countries included in the model and countries with exchange to these countries, except for Poland and Russia. Prices from the day-ahead market was used, as these carry the largest volumes and best reflects the price for the end user. Data comes in one file per year, and a manual process was done to extract the chosen areas and time ranges and fit them into one file.

For Poland, the day-ahead prices were not available on NordPool, so the local exchange PSE were used. Data from PSE were downloaded for each week. Prices on PSE are given in local currency, so historical exchange rate from XEcurrency was used to convert the prices to euro in Excel. Because the Russian market is partly regulated and will not adhere to the same price mechanisms as the Nordpool system, Russian prices are omitted from the delta calculation.

When all prices had been collected into one spreadsheet, the data was converted on to a json-file readable by TinyDB in the same way as with the production data.

5.3 Test activity data

For simulating testing activity, the commit logs from three different projects were used. These were chosen to represent different types of development.

Visual studio Code (VSC), an integrated development environment project: This is one of the largest open-source projects on GitHub, with over 19.000 contributors. It was chosen as a representative of a large open-source project with many commits coming in from all over the world.

Medium Sized Commercial project (MSC), software for handling memberships, insurances and training for a large Norwegian union: The project was chosen to represent a commercial software project, with developers working in one time zone only. The project has 5 developers working full time based in Oslo.

The Algorithms – Python, a library of algorithms: This is a smaller open-source project with less activity. It has about 1300 contributors and can sometimes go several days without any commits. It was chosen to represent a smaller project with less activity.

The data were recorded to text files containing only the timestamp for the commits, one per new line. One file was made for each project for each of the seasons, a total of 12 different files with commit logs.

5.4 Constructing input data on delay tolerance

As in order to check for planned servers and calculate the most desirable area to run tests in, three data entries are needed: the time the test was triggered, test duration and delay tolerance. The time the test was triggered is available from the commit logs, but the other two have to be constructed.

For projects that follow a steady daily rhythm of commits being done mainly during work hours, the following assumptions are used for constructing delay tolerance data:

- the developers want to have fast feedback during work hours, the delay tolerance is short.
- if a commit is done during nighttime or very early morning it is due to an urgent fix and the developer cannot wait for feedback, the delay tolerance is short.
- if a commit is done between 11 and 1200 hours it can wait a moderate amount of time due to lunch break, the delay tolerance is moderate.
- if a commit is done between 16 and 19, feedback can wait until the next day because the developer is finishing her workday and feedback can wait until the next day.

The files with the commit times are run through a script that appends either short, moderate or long delay tolerance to each line in a csv file, according to the assumptions.

For projects without a particular pattern of commits during the workday the method from the previous chapter cannot be used. These will be given one static delay tolerance value for all commits. For these projects, the files with the commit times are run through a script that appends the same delay tolerance value to each line in a csv file.

5.5 Test duration

For collecting as realistic data as possible on the two open-source projects, they were downloaded and built on a local environment. Both projects contain tests that could be used in a pipeline.

The larger project, VScode, had unit tests, integration tests and an automated UI-test available. The tests were run three times in Electron, and the average times were:

- UI-test: 1 minute, 51 seconds
- Integration test: 8 minutes 50 seconds
- Unit tests: 15 seconds

In addition, the build took 4 minutes and 41 seconds. Each commit to the pipeline needs a new build, so this will be added to the total test duration.

The smaller project was written in python, with a tiny test suite to be executed by the tool pytest. The tests took an average of 41 seconds to run.

With this in mind, it was determined to use 16 minutes for VScode and 1 minute for The Algorithms as the test duration for the baseline test scenario. For MSC, historical test run data was available, and the average run time of 22 minutes will be used.

5.6 Determining surplus areas and their price delta

For the calculation of surplus energy price and production volume, data as described above is used. In addition, the model needs to have a concept of which price areas have energy exchange with each other. A python dictionary listing the neighbors of each area is registered as a global variable, excluding Russia. The data comes from Nordpool, and reflects the exchange illustrated in figure 5.1. Threshold of how large a fraction of the produced energy must come from IRE (IRE_SHARE) to define it a relevant contribution to the surplus, is also added as a global variable.



- DK1 - DK2, SE3, NO2, DE, NL
- DK2 - DK1, SE4, DE
- SE1 - FI, SE2, NO4,
- SE2 - SE1, SE3, NO4, NO3
- SE3 - SE2, FI, SE4, NO1
- SE4 - SE3, LT, DK2, DE, PL
- NO1 - SE3, NO2, NO3, NO5
- NO2 - NL, DE, DK1, NO1, NO5
- NO3 - NO4, SE2, NO1, NO5
- NO4 - SE1, SE2, NO3
- NO5 - NO3, NO1, NO5
- EE - LV, FI
- FI - SE3, EE, SE1

Figure 5.2: map from nordpoolspot.com showing the exchange between the Nordics and surrounding areas and list over neighboring areas.

5.7 Simulation script

The data and scripts described above is utilized by a simulation script that iterates through the commits and runs each of them through the decision algorithms. Each commit that is read into the script goes through the algorithm for checking for planned servers. If no suitable server is found, it will proceed to find the best time and place to run the test by using the CAST-algorithm. Thereafter, it will record the database booking with the appropriate time and place to the server database.

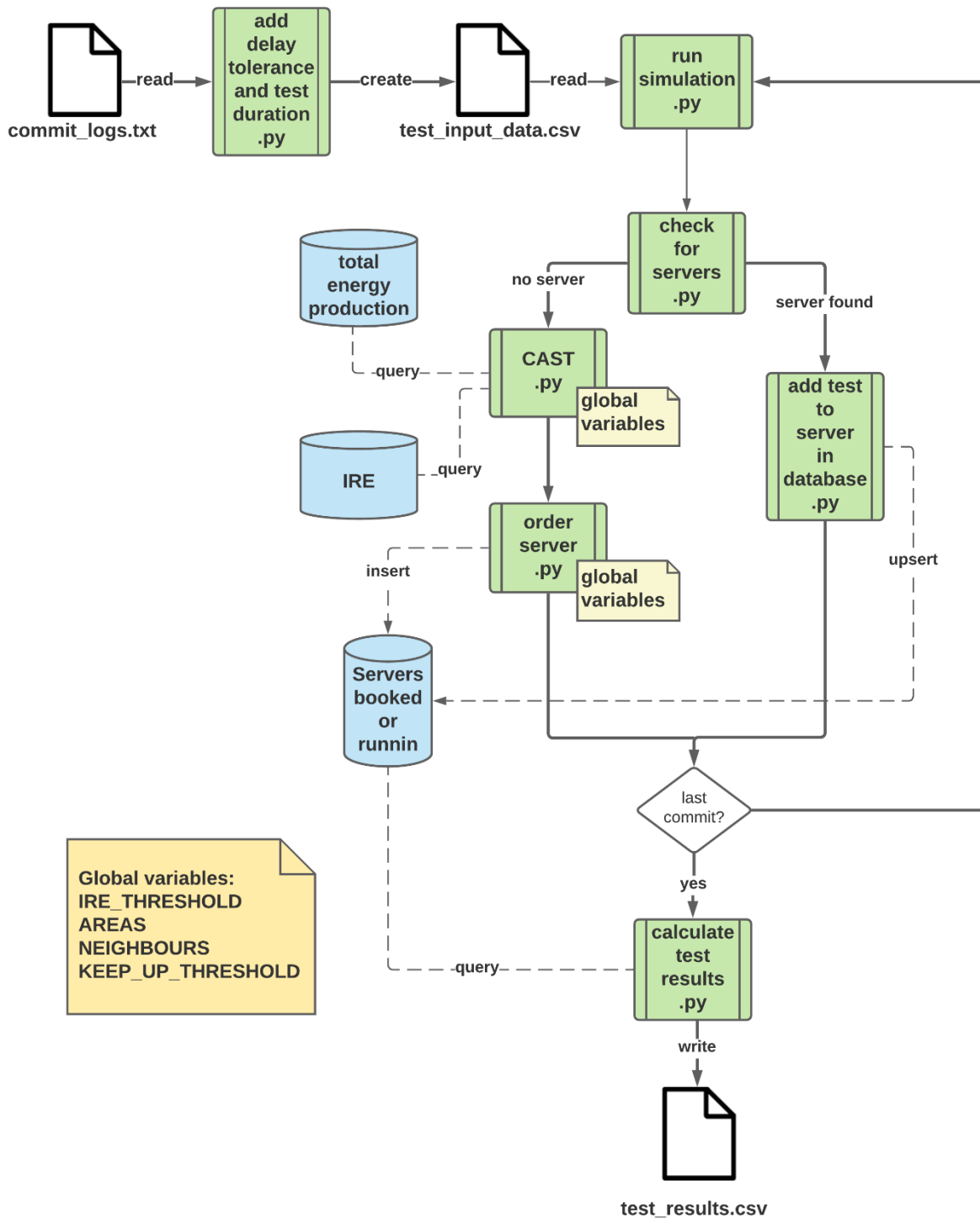


Figure 5.3: flowchart describing the process of simulating test runs and recording the results. The solid lines illustrate the actions of the processes, while dotted lines illustrate interaction with the database files. Green boxes illustrate python-scripts or functions while white icons illustrate files.

After the simulation script has looped through all of the lines in the file with the commit-times, delay tolerance and estimated test duration, it will call on a function that goes through the database with server bookings. The outcome of this process is a list with the following parameters:

- Name of the project
- Season (winter/spring/autumn/summer)
- Scenario (which of the scenarios were tested)
- Average delay tolerance
- Test duration
- Number of commits
- Number of servers started in an area without IRE surplus (gray servers)
- Number of servers started in an area with IRE surplus (green servers)
- Number of re-uses: how many times multiple tests were run on one server.
- Minutes of tests running on a gray server.
- Minutes of tests running on a green server.
- Minutes where a green server was idle.
- Minutes where a gray server was idle.
- The number of servers started in each area.

Project	MSC	MSC	VSC	PA
Season	Summer	Winter	Winter	Summer
Scenario	Baseline	Baseline	Baseline	Baseline
AVG delay tolerance	230	176	183	10
Avg test duration	22	22	16	1
Commits	35	42	538	17
Gray servers	4	0	1	4
Green servers	27	37	370	13
Re-use	4	5	167	0
Gray minutes	110	0	32	4
Green minutes	660	924	8576	13
Green waste	504	1014	7692	0
Gray waste	130	0	28	0
SE1	0	0	0	0
SE2	0	0	0	1
SE3	0	0	0	0

Table 5-2: the first lines and rows of the csv-file with results from the simulation script run.

The file will be used to further analyze how the outcome changes when there are changes to the different parameters used in the script (global variables) or changes to the data input (different seasons, projects, test duration and delay tolerance).

5.8 Constructing test scenarios

5.8.1. The baseline test scenario

In order to observe the effect of altering the different variables, such as delay tolerance, test duration and share of IRE, a baseline scenario was established. This was constructed with the most realistic data in mind, considering all available information. The following was used:

IRE_SHARE: 20% is used as the baseline scenario, as this is what previous research suggests as being the point where IRE production contributes significantly to the power mix.

Test duration: average test run times are used, either from running tests locally several times and calculating the average or by collecting historical data. The process of finding these times were described in chapter 5.5.

Delay tolerance: here, it would be necessary to do some experiments either with surveys or experiments involving humans in order to record real data. This is not in scope of this project, so assumptions are made as to how much delay can be tolerated in different situations. For the profiled distribution of delay tolerance as described in chapter 5.4, the baseline input has been chosen to be 5 minutes delay tolerance during work hours, 30 minutes before lunch and 720 minutes at the end of the workday. For the static input, 10 minutes delay tolerance is chosen as a baseline scenario.

Shutdown parameter limit: the shutdown parameter decides whether the server will be kept running after the test is done or not and is an expression of whether we expect new tests to come in before the server hour is over. Assuming that new tests are most likely to come in when there is active development going on, the baseline limit is set to the same amount of time as the delay tolerance during working hours: 5 minutes.

5.8.2. Other scenarios

Including the base scenario, 9 different scenarios were constructed to look at how variations in the parameters affected the outcome. There are four variants of test scenarios:

IRE_SHARE adjustments: The share of IRE decides how much of the total energy in an area must be IRE in order to consider it to be significant enough to conclude that the energy surplus in an area is a surplus of IRE. The base threshold is set based on previous research, but looking at scenarios with a lower threshold can give an indication as to what will happen when a larger share of the energy produced comes from IRE. The two scenarios constructed for this purpose is one where the threshold is lowered to 15% and one where the threshold is set to 1%.

Delay tolerance variations: Based on analysis of the data sets, it is determined to use a dynamic profile as described in 5.4 for the projects MSC and VSC (explained further in chapter 6.2). For the smallest project, Python Algorithms, no daily rhythm of commits was found, so it is decided to use the same delay tolerance for all commits. In order to analyze how delay tolerance affects the outcome, two scenarios were added. The first had significantly longer delay tolerance than the base scenario, while the other significantly shorter. The shutdown

parameter is set to the same value as the shortest delay tolerance, in order for these scenarios to be similar to the base scenario in which servers are kept running or not.

Shutdown parameter variations: the algorithm uses the shutdown parameter to decide whether the server should be shut down after the end of a test run. The algorithm sets the shutdown-variable to true or false based on whether the delay tolerance is longer or shorter than the shutdown parameter. Adjusting this parameter will give insight on the effects of keeping servers running. Two scenarios are added, one with shutdown parameter of 60 minutes, and one with shutdown parameter zero.

Test duration variations: Altering the test durations will provide information about how much there is to gain from refining the test suite to run faster. Two variations with static test duration were added, one with significantly longer test duration and one with significantly shorter. The data set for Python Algorithms were not given a short test duration scenario, as the tests there already are very fast (one minute) in the base scenario. In addition, one scenario with variable test duration was added, in order to have a look at how this would affect the outcome. Each commit was assigned a random normally distributed test duration value, with a mean of the base value and quite high standard deviations in order to achieve a large spread. The values were converted to absolute values in order to avoid negative test durations.

Scenario	Data-set	IRE share	Delay tolerance work hours	Delay tolerance lunch	Delay-tolerance After work	Delay-tolerance Static (PA)	Test duration VSC	Test duration MSC	Test duration PA	Shutdown Parameter
BASE	base	0.2	5	30	720	10	16	22	1	5
IRE_low	base	0.15								
IRE_one	base	0								
DelayT_long	DT_long		60	120	720	120				60/119
DelayT_short	DT_short		1	15	120	1				1/0
Shutdown_short	base									0
Shutdown_long	base									60

TestDur_ short	TD_ short						6	6		
TestDur_ Long	TD_ long						45	45	45	
TestDur_ noisy-	TD_ noisy						16 / 8	22 / 11	3/1.5	

Table 5-3: the parameters for the different scenarios. Empty cells illustrate that the value is equal to the base scenario. On the last row, the first value is the test duration, while the second is the standard deviation.

5.9 Analyzing input data.

To better understand the outputs of the simulation script, it is useful to have a closer look at the input data.

To see how the supply of IRE varies throughout the day, and if the surplus moves around to different areas like expected, a script is made to produce heatmaps that illustrate the energy situation throughout the day. The script uses the data on IRE production, total production and prices per area/hour, and calculates the price delta for each area with a significant production of IRE. The result of the calculations is stored in a CSV file which is used to draw a heatmap over the situation using the Python libraries NumPy and Seaborn.

To have a look at the chosen projects commit-patterns, the commit logs were imported to excel which was used to do some basic analysis and the excel tools for visualization were used to make diagrams and figures.

6. Results/observations

6.1 Are some areas really better than others? – a closer look at the energy data.

Is there any benefit to moving the workload around in time and space? To determine this a series of heatmaps have been created to illustrate the surplus or deficit of IRE.

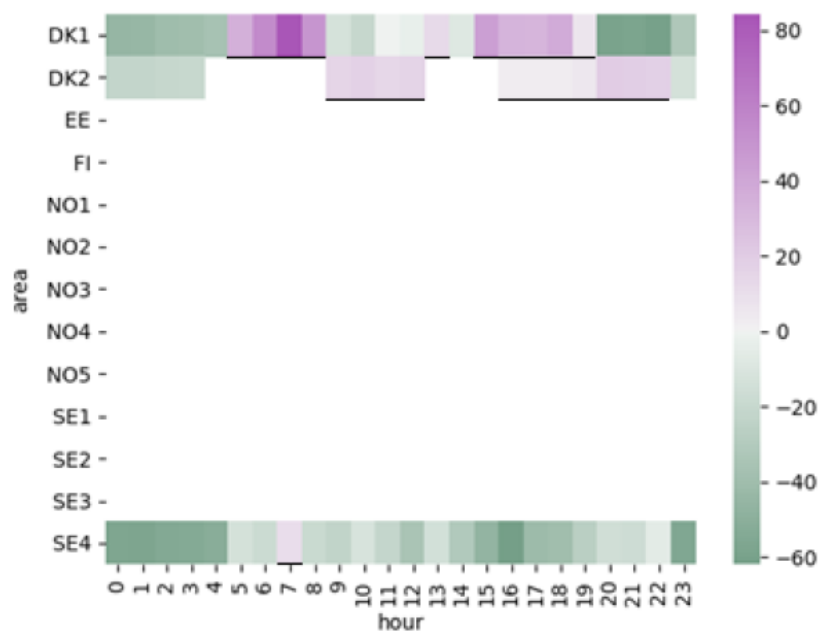


Figure 6.1: heat map illustrating the energy situation in the Nordics, December 16th 2020. Green indicates a surplus while purple indicates a deficit. The scale on the right shows the accumulated price difference between the area and its neighbours in Euros per MWh.

The threshold for significant production of IRE is set to 20%, which means that hours where the share of IRE is less than 20% are without color. Where there is a significant share of IRE, the difference in price between the area and its neighbors are calculated in order to determine whether the area has a surplus of energy.

The hours that have lower prices than the neighbors are colored green. This means that there is a surplus of IRE. For hours with a deficit the color is purple. A purple hour means that there is a significant share of IRE in the area, but the demand is high, and the IRE production is not large

enough to make more energy than the area consumes. The darkest color indicates the largest surplus or deficit.

The figure above illustrates a 24-hours period where only three of the areas have more than 20% IRE. Also, all three areas have a deficit in some hours, and surplus in others. This is not in any way uncommon and it suggests that moving test activity could help both in utilizing more green energy, and relieve areas with energy deficit.

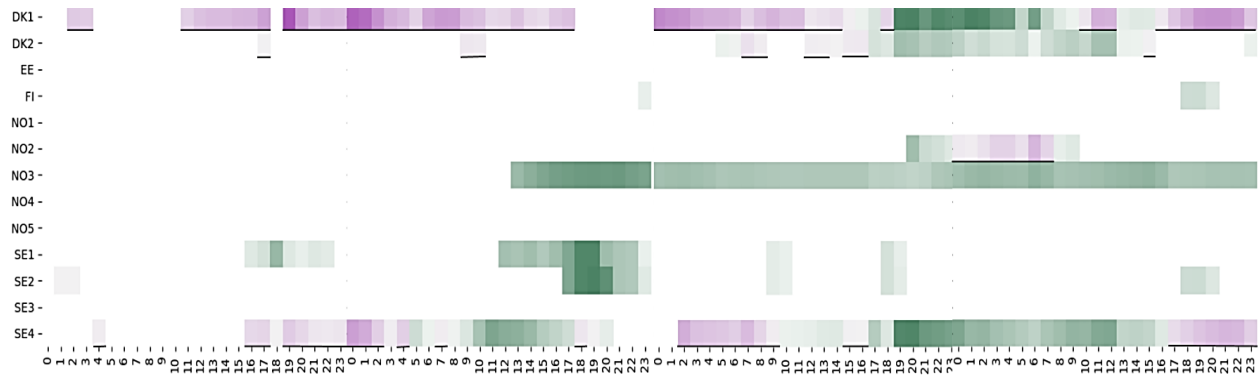


Figure 6.2: heat map showing surplus (green) and deficit (purple) of energy for areas with a significant share of IRE from March 18th to March 21st 2021.

This image shows a combination of the energy production and market situation for each hour of four days, from March 19th to March 21st of 2021.

Looking at the figure proves that it does indeed vary which area has the highest surplus of IRE. While many areas do not have significant shares of IRE at any time, others have it most of the day. The largest surplus is found in 5 different areas over the course of these four days.

This figure also illustrates the effect of new installed capacity. The area NO3 is colored white until midday March 19th 2021. At this time, new wind turbines on Fosen, Trøndelag started delivering energy to the grid. Similar effects are expected in the future, as more wind parks are introduced. One can therefore assume that the occurrence of hours with no surplus will lessen in the coming years.

In the data set we see that the seasons that are lower on wind have quite a few hours where no area has a significant share of IRE. During the summer week, 40 hours were without IRE surplus

in any area. During spring week there were 58 hours, while autumn had four and winter had only one.

6.2 Test distribution.

As expected, the three projects had very different commit logs. VC code had an average of 59.3 commits per day, MSC had 6.7 and Python Algorithms had 1.8. There was no clear pattern as to which season were most busy. VS code had the highest activity in the winter week, and Python Algorithms had the lowest activity that same week with only two commits for the whole week.

One interesting observation was that the commit log for VS code followed a pattern consistent with a normal work week when looking at the amount of commits per day of the week.

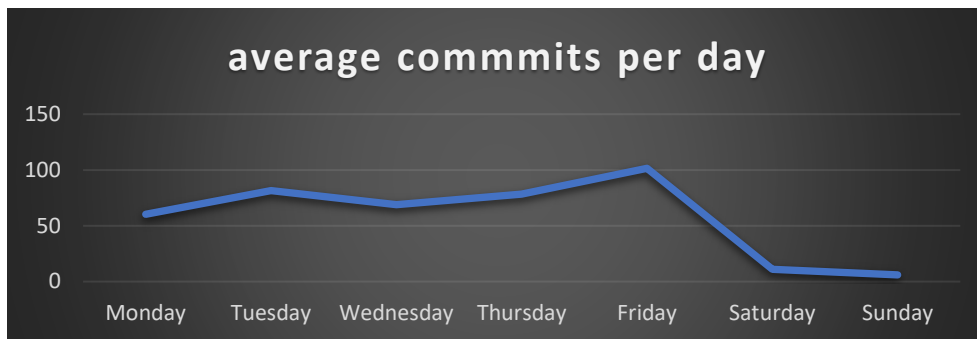


Figure 6.3 line diagram illustrating how many commmits the project VS code has per day of the week, on average.

The weekdays are quite busy, and most commmits are being done on Fridays. Almost no changes are done on Saturdays and Sundays. This indicates that this project is probably being worked on by professionals who do it during their workday. When we look at how the commmits are spread throughout the day, it seems that for VS code there is also a pattern that can look somewhat like a workday, where there is more activity in the hours of 9 and 19. Still, there is activity in all hours of the day. This is not the case for MSC, and the assumption that the commmits mainly happen during the workday holds true. MSC has no commmits during night hours, and most commmits happen before lunch and at the end of the workday.

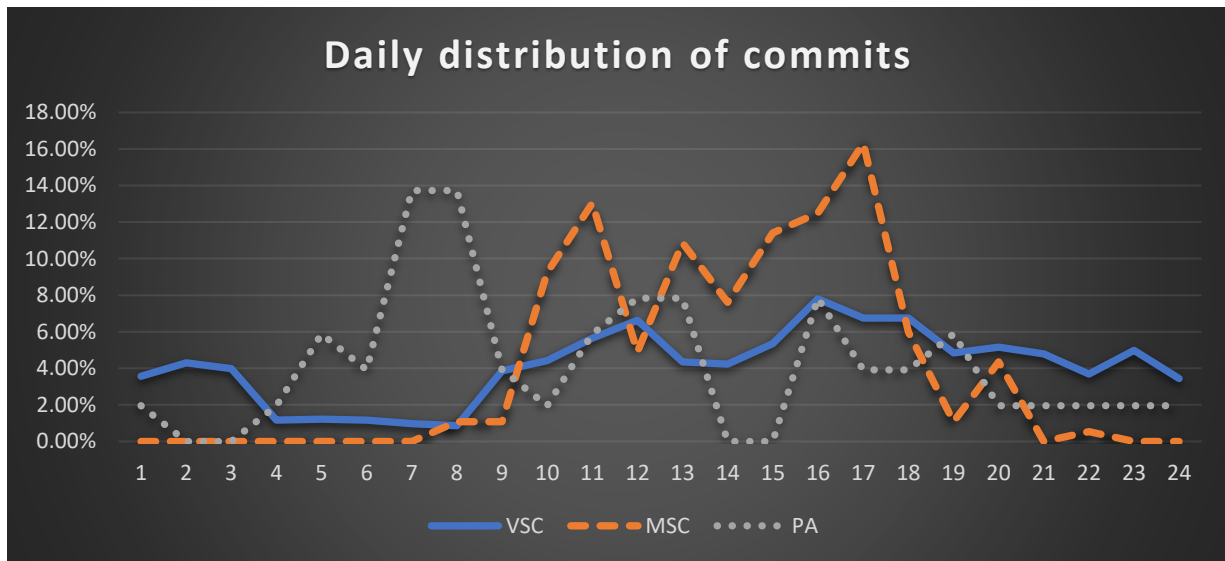


Figure 6.4: diagram showing how many percentages of the commits in each project are done in each hour of the day. The PA-project is illustrated with the gray dotted line, MSC is orange and VSC blue.

6.3 Approach for manual analysis

Looking back at the scenario of the DevOps-engineer set out to optimize the pipeline, it is clear that success must be measured by several different parameters. One goal was to make the pipeline as green as possible. Looking at the number of servers that were deployed in a green area or maybe more so, lack of servers deployed in gray areas will give an indication of the achieved effect on greenhouse gas emissions. Another goal was to avoid unnecessary delays. In the baseline scenario the assumption is that there is some tolerance for waiting, and the effects of waiting longer or shorter will be explored in the scenarios that cover delay tolerance. Cost is also a factor. Running the tests with as few servers as possible is therefore also something to look at when assessing whether a scenario has had successful results or not. Due to startup-costs of a deploy and the per hour pricing model, it is assumed that it is better to run two tests on the same server, even though this results in some waste (idle server time). This means that the overall number of servers started must be considered when assessing the cost.

From all simulations, it is expected to see significant seasonal effects. Previous analysis of the data showed that during the spring week there were 58 hours with no green surplus, while during the winter week there was only one. The results will therefore be analyzed per season and project.

6.4 Results of the baseline simulation

As expected, the spring week is the one which utilizes the lowest share of green servers. On the other hand, we see that for the winter week, there is only one gray server deployed for all projects combined.

In order to measure the effects of the CAST-algorithm, one simulation was run of the base scenario where all servers were placed in SE4, the area with the most hours with IRE surplus. This will provide some insight as to what results would be without the CAST-algorithm. This simulation resulted in 282 gray servers being deployed, for all four weeks and all projects summarized. In the simulation with the CAST-algorithm, 150 gray servers were deployed.

Of the 1848 commits that were made, 793 of them were tested on a previously reserved server. 86% of the servers are placed in a green area (905 out of 1055).

project	season	Avg delay tolerance	avg test duration	commits	gray servers	green servers	re-use	gray minutes	gray waste	green minutes	green waste
PA	winter	10	1	2	0	2	0	0	0	2	0
PA	summer	10	1	17	4	13	0	4	0	13	0
PA	autumn	10	1	18	0	18	0	0	0	18	0
PA	spring	10	1	14	1	13	0	1	0	13	0
MSC	winter	176	22	42	0	35	7	0	0	924	1014
MSC	summer	230	22	35	4	23	8	110	130	660	504
MSC	autumn	210	22	56	1	44	11	22	38	1210	1204
MSC	spring	204	22	51	27	11	13	836	584	286	190
VSC	winter	183	16	538	1	260	277	32	28	8576	6040
VSC	summer	153	16	185	17	93	75	368	652	2592	2360
VSC	autumn	178	16	476	0	253	223	0	0	7616	6776
VSC	spring	165	16	414	95	140	179	2592	2824	4032	3616
SUM					150	905	793				

Table 6-1: the output of the simulation of the base scenario for all projects and seasons. The gray and white rows alternate between the projects.

6.5 Results of the IRE_share scenarios

For these simulations, the data set from the baseline scenario was sent through the allocation-algorithm with IRE_shares of 0.2 (baseline), 0.15 (low) and 0.01 (one). Reducing the threshold will give the allocation algorithm more green hours to choose from. It is expected that a decrease of the threshold will increase the number of servers that are placed in an area with surplus IRE, green servers. It is also expected that it will have a more significant effect in the summer and spring than in the winter and autumn.

project	season	scenario	commits	gray servers	green servers	re-use	gray minutes	gray waste
MSC	spring	base	51	27	11	13	836	584
MSC	spring	IRE_low	51	16	22	13	484	276
MSC	spring	IRE_one	51	0	37	14	0	0
MSC	winter	base	42	0	35	7	0	0
MSC	winter	IRE_low	42	0	35	7	0	0
MSC	winter	IRE_one	42	0	35	7	0	0
VSC	spring	base	414	95	140	179	2592	2824
VSC	spring	IRE_low	414	75	159	180	1968	2260
VSC	spring	IRE_one	414	0	231	183	0	0
VSC	winter	base	538	1	260	277	32	28
VSC	winter	IRE_low	538	1	260	277	32	28
VSC	winter	IRE_one	538	0	264	274	0	0

Table 6-2: selected values from spring and winter simulations of scenarios with different IRE threshold. The gray and white rows alternate between the combination of project and season.

The results are as expected. In the spring we see significant effects of lowering the threshold, the number of gray servers used is significantly reduced. For brevity, only spring and winter for two projects are included in the table above, but the results for the summer and autumn week show the same effect. When accepting as little as one percent IRE as significant, we see that the number of gray servers is reduced to zero. These results can give an indication about what to expect when more production capacity for IRE is installed. A higher share of IRE will increase the selection of green areas, leading to lower greenhouse gas emissions.

6.6 Results of different thresholds for keeping server open.

The shutdown parameter defines whether a server should be kept open after the first test run is finished or not. The rationale behind this parameter is that if there is a short delay tolerance, it is expected that more tests will come in soon. In the base scenario the parameter is set to 5, which means that for all commits with a delay tolerance of 5 minutes or less, the server will be kept open in case of more commits before the server hour is over. In the shutdown_short scenario, the parameter is set to 0, which means all servers will be shut down after the test is done, unless another test comes in while it is still running that can be scheduled to start once the running test is done. In the shutdown_long scenario the parameter is set to 60. For the PA project, this means that all servers will be kept running, and for the other projects all commits during working hours will be kept running.

project	scenario	Avg delay tolerance	avg test duration	gray servers	green servers	re-use	gray min	green min	green waste	gray waste	Server total
PA	base	10	1	4	13	0	4	13	0	0	17
PA	long	10	1	3	11	3	4	13	647	176	14
MSC	base	230	22	4	23	8	110	660	504	130	27
MSC	short	230	22	5	24	6	110	660	0	0	29
MSC	long	230	22	4	23	8	110	660	542	130	27
VSC	base	153	16	17	93	75	368	2592	2360	652	110
VSC	short	153	16	23	106	56	416	2544	0	0	129
VSC	long	153	16	17	90	78	368	2592	2500	652	107

Table 6-3: results of the shutdown scenarios for all projects, summer week. The gray and white rows alternate between the projects.

For the PA project, there is no difference between the base and short scenario, as the delay tolerance is a static value of 10, which means all servers will be shut down after the test runs. For this project, increasing the shutdown parameter to 60 means that all servers are kept open. As expected, this causes a reduction in the number of servers started, but it is small, the total number of serves decreases from 17 to 14. The amount of time where servers have been idle (green and gray waste) has increased significantly, from zero to 823 minutes. So, there is a decrease in cost, but an increase in energy spent.

For the two larger projects, we also see the expected increase in how many servers were used and reduction in waste when we set the shutdown parameter to zero.

6.6.1. The cost of eliminating waste in a large project.

Waste can be eliminated by setting shutdown to zero, but whether it is worth it or not depends on how the prioritizations are done. For VSC, the total amount of servers used increases from 107 in the long scenario to 129 in the short scenario in the summer week.

season	scenario	gray servers	green servers	gray minutes	green minutes	green waste	gray waste	servers total
autumn	Long	0	252	0	7616	6932	0	252
spring	Long	94	142	2592	4032	3832	2864	236
summer	Long	17	90	368	2592	2500	652	107
winter	Long	1	259	32	8576	6288	28	260
autumn	Short	0	332	0	7616	0	0	332
spring	Short	120	169	2592	4032	0	0	289
summer	Short	23	106	416	2544	0	0	129
winter	Short	1	318	32	8576	0	0	319

Table 6-4: selected results for the shutdown long and shutdown short scenarios for the VSC project. The gray and white rows alternate between the scenarios.

For all four weeks, the VSC project started 855 servers in the long-scenario, and 1069 in the short-scenario, a difference of 214 servers over 4 weeks. Extrapolated to a year, this amounts to a difference of 2782 servers. At the time of writing, the average price of an on-demand Linux VM in the Nordic data centers of Microsoft, Google and AWS were 0.039\$ US. This means that the monetary cost of eliminating waste amounts to roughly 108.5\$ per year.

If we look at the amount of waste, the sum is 23096 minutes for the four weeks measured for VSC, approximately 5004 hours per year. 4236 hours is green waste and 768 is gray waste. Using the calculations from GoClimate, the estimated savings of eliminating this waste is 118kg of CO2 per year. This would be the equivalent of driving 677 kilometers with a petrol fueled car (UK Department for Business, Energy & Industrial Strategy, 2020).

These savings are quite expensive, and there can also be a cost of time depending on how fast a new server is provisioned. If there is some overhead energy use to start a VM, this is not accounted for, so the savings can be less than this estimate. Still, the result of shutting the VMs down after a test run is most likely beneficial towards lessening the carbon footprint, and “eliminating waste” does sound good in reports and marketing efforts.

6.7 The effects of different delay tolerances.

A longer tolerance for waiting is expected to have two relevant effects. Firstly, it is expected that more servers will be green because the allocation algorithm gets more hours to choose from. Secondly, it is expected that more servers will be used for more than one test. The delay tolerances have been altered quite dramatically. In the “short” scenario, the daytime delay tolerance has been reduced from 5minutes (base) to 1 minute. For the “long” scenario, the daytime delay tolerance is increased to 60 minutes.

proj	season	scen	Avg delay toler	gray servers	green servers	re-use	gray waste	green waste	total servers	waste factor
MSC	autumn	base	210	1	44	11	38	1204	45	1.01
MSC	autumn	long	251	0	32	24	0	484	32	0.39
MSC	autumn	short	36	1	45	10	38	1204	46	1.01
MSC	spring	base	204	27	11	13	584	190	38	0.69
MSC	spring	long	247	22	10	19	328	146	32	0.42
MSC	spring	short	35	28	11	12	584	190	39	0.69
VSC	autumn	base	178	0	253	223	0	6776	253	0.89
VSC	autumn	long	222	0	178	299	0	2196	178	0.29
VSC	autumn	short	30	0	268	209	0	7480	268	0.98
VSC	spring	base	165	95	140	179	2824	3616	235	0.97
VSC	spring	long	209	61	103	250	992	1388	164	0.36
VSC	spring	short	28	99	155	160	2976	4500	254	1.13

Table 6-5: selected results of the delay tolerance scenario simulations. The green cells mark the most optimal outcomes. The gray and white rows alternate between the combination of project and season.

The data show that the results are as expected, gray servers and total servers decrease with a longer delay tolerance. The movement from gray to green is quite modest for the medium sized project. For the MSC project, increasing delay tolerance by 59 during working hours saves one gray server from running in the autumn week and six in the spring week. For the larger project the effect is more significant in the spring week, with a 38% decrease in gray servers.

A more significant effect is observed on the number of servers deployed and the waste. In all seasons and for all project we see that a longer delay tolerance leads to significantly less resource use. On the left side of the table above, a column of waste factor is added. The waste factor is given as the total minutes of waste over total minutes of active server time. The waste

factor decreases as the delay tolerance increases. The decrease in waste is not caused by servers shutting down more often. The increase in delay tolerance allows the algorithm to re-use servers more often and which is seen from the decrease in number of servers used.

These simulation results support the idea that there are possible savings to both greenhouse gas emissions and cost for those who are willing to wait.

6.8 The results with different test durations

In all the previous scenarios, the test durations have been based on actual processing times for the tests in the different projects. As explained in the background chapter, different techniques can be used to speed up the duration of testing. Running the simulations with different test durations provides some insight on the importance of test duration. The “noisy” scenario with varying test durations was added to see if this made any significant difference to the results.

project	season	scenario	avg test duration	gray servers	green servers	re-use	green waste	gray waste	waste factor	Ops cost
MSC	summer	base	22	4	23	8	504	130	0.76	27
MSC	summer	long	45	5	30	0	270	75	0.20	35
MSC	summer	noisy	22	4	23	8	510	132	0.78	27
MSC	summer	short	6	2	14	19	432	90	2.40	16
MSC	winter	base	22	0	35	7	1014	0	1.10	35
MSC	winter	long	45	0	42	0	465	0	0.25	42
MSC	winter	noisy	23	0	35	7	858	0	0.88	35
MSC	winter	short	6	0	22	20	810	0	3.21	22
VSC	summer	base	16	17	93	75	2360	652	0.91	110
VSC	summer	long	45	27	158	0	1620	405	0.23	185
VSC	summer	noisy	15	16	92	77	2568	604	1.07	108
VSC	summer	short	6	13	65	107	2346	630	2.44	78
VSC	winter	base	16	1	260	277	6040	28	0.70	261
VSC	winter	long	45	1	537	0	5340	15	0.22	538
VSC	winter	noisy	15	1	250	287	5903	29	0.71	251
VSC	winter	short	6	1	137	400	4134	42	1.29	138

Table 6-6: selected results from the simulations with different test durations. The gray and white rows alternate between the combination of project and season.

For the smallest project, the test duration made little difference except from altering the number of active server minutes.

The results for the larger projects were in line with what could be expected. The noisy scenarios result in values close to the base scenario, it does not seem to have any significant impact. The longer test duration of 45 minutes makes quite good use of each server, resulting in low waste factors, but only one test can run per server, so the costs are high. The winter week with long test duration for the VSC project has the highest cost of all simulations done in this project, 538 servers deployed in one week. With the short test duration, the same week results in 138 servers being deployed. The ratio between green and gray servers does not appear to be influenced by the different test durations.

6.9 Summary of cost / benefit analysis

To compare the performance of the different scenarios, a comparison of the annualized values for emissions and cost was made.

Row #	scenario	VSC		MSC		PA	
		emissions	cost	emissions	cost	emissions	cost
1	base	238.2	11167	43.2	1885	0.24	663
2	base-se4	282.1	11180	47.5	1885	0.34	663
3	delay tolerance long	166.2	8112	33.1	1495	0.24	585
4	delay tolerance short	257.9	12194	43.9	1937	0.24	663
5	IRE threshold 15%	227.5	11167	38.3	1885	0.22	663
6	IRE threshold 1%	190.7	11193	30.5	1872	0.20	663
7	shutdown long	241.7	11115	44.3	1872	10.32	468
8	shutdown short	124.5	13897	23.2	2054	0.24	663
9	test duration long	434.2	20982	58.4	2392	10.74	663
10	test duration noisy	226.1	20982	43.6	2392	0.64	663
11	test duration short	128.0	10634	23.6	1781	-	-

Table 6-7: table showing the annual emissions in kilograms of CO2 and cost in number of servers started. For each column, the higher numbers are colored red, while the lower numbers are blue.

Two scenarios stand out as good performers in both cost and emissions: long delay tolerance and short test duration. For all projects, both emissions and cost are on the lower end with these two scenarios compared to the others. The short shutdown scenario shows the expected result of low emissions with increased cost for all VSC and MSC, compared to the baseline (see

row 8). There are no significant differences between the projects, the algorithm has the same relative outcome for a busy project as for a small one using these parameters.

For the smallest project, the emission in most scenarios is negligible, the difference between the base scenario and where all servers are placed in SE4 is only 10 grams of CO2 over one year. The emissions in the base and in the long delay tolerance scenario are the same. Still, the long delay tolerance is favorable due to reduced costs. Because of the very short test duration in this project, keeping servers up after the test is done causes a large amount of waste which leads to a dramatic increase in energy use.

For MSC and VSC, the effects of longer delay tolerance or shorter test durations is more significant. For both projects, reducing the test duration is the most effective way of reducing the carbon footprint, but this is more costly in financial terms than increasing waiting time. However, reducing the test duration also have a nice side effect of giving faster feedback to the developers which might be beneficial for the productivity in the project. The cost of a developer waiting for test results is probably higher than that of extra server time.

6.10 Delay tolerance versus actual waiting time

The results show that there are large potential benefits to having longer delay tolerances. This raises the question of how much longer does the developer have to wait for feedback, in order to gain these benefits? To answer this question, some code was added to record the waiting time between the time of a commit and the actual start of the test run.

When preparing the data, delay tolerance for the two larger projects were divided in three different categories:

- short delay tolerance during the hours when it is assumed that the developers are working actively.
- Long delay tolerances at the end of the workday, when it is assumed that the developers are going home and do not need feedback until the next day.
- Medium delay tolerance during lunch hour.

The most relevant metric with respect to developer feedback is found when looking at the waiting time during the active working hours.

	MSC			VSC		
	Base, delay tolerance 5 minutes	Short, delay tolerance 1 minute	Long, delay tolerance 60 minutes	Base, delay tolerance 5 minutes	Short, delay tolerance 1 minute	Long, delay tolerance 60 minutes
Average waiting time	0.02	0.00	17	0.38	0.00	16
Median waiting time	0	0	11	0	0	11
75-percentile	0	0	31	0	0	25
90-percentile	0	0	49	2	0	48

Table 6-8: statistic values for the waiting time during active working hours for the scenarios base, short delay tolerance and long delay tolerance. The waiting time is recorded in minutes.

The results show that there is little difference between the base scenario, where delay tolerance is 5 minutes, and the scenario with short delay tolerance, one minute. The average waiting time during working hours was reduced to zero from 0.02 (MSC) and 0.38 seconds (VSC). 90 percent of the tests had waiting time of zero already for the MSC project, and less than 2 seconds for the VSC project, so the gain of a small reduction in delay tolerance is negligible.

In the long delay tolerance scenario, the delay tolerance is 60 minutes, but the results show that one rarely has to wait that long. The median waiting time is approximately 11 minutes for both projects, meaning that for half of the commits the waiting time was less than that. 10% of the commits had a waiting time of more than 49 minutes for MSC and 48 minutes for VSC. This is a long time to wait for feedback if one is working and need as fast as possible in order to continue working, but in many cases, it can be perfectly acceptable, for instance if one is attending a meeting or working on something else while the tests are running.

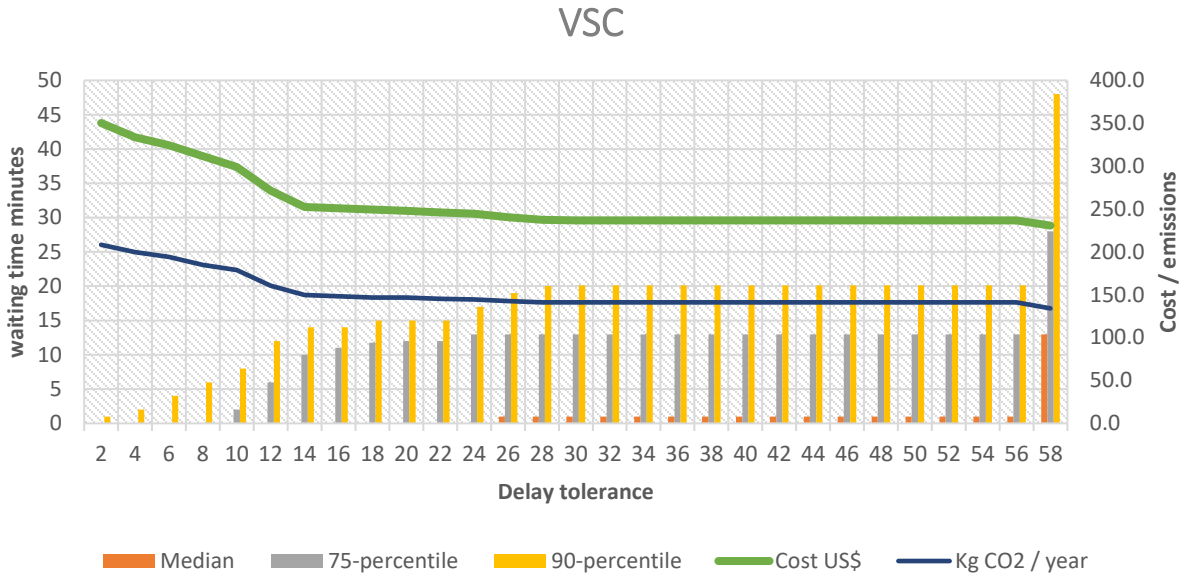


Figure 6.5: diagram illustrating how cost, emissions and actual waiting time develops as delay tolerance increases for the VSC project. Commit logs with commits only during active working hours are used for the simulation.

The diagram above show that the relationship between delay tolerance and actual waiting time is non-linear. The 75-percentile stays at 13 minutes up until the delay tolerance is 60 minutes. When delay tolerance increases from 2 to 30, there seems to also be an increase in waiting time, but this development flattens, and for delay tolerance between 28 and 58 minutes the waiting time does not increase. When delay tolerance reaches 60 minutes there is a dramatic spike in waiting time.

The data show that the gain is biggest for delay tolerance up to 14 minutes, after this the cost and emission curves flatten and there is no further reduction in emissions or cost until the delay tolerance reaches 60 minutes where there is a small reduction. The graph for MSC show a similar effect, but with a breaking point at 22 minutes. This corresponds with the test duration of 22 minutes.

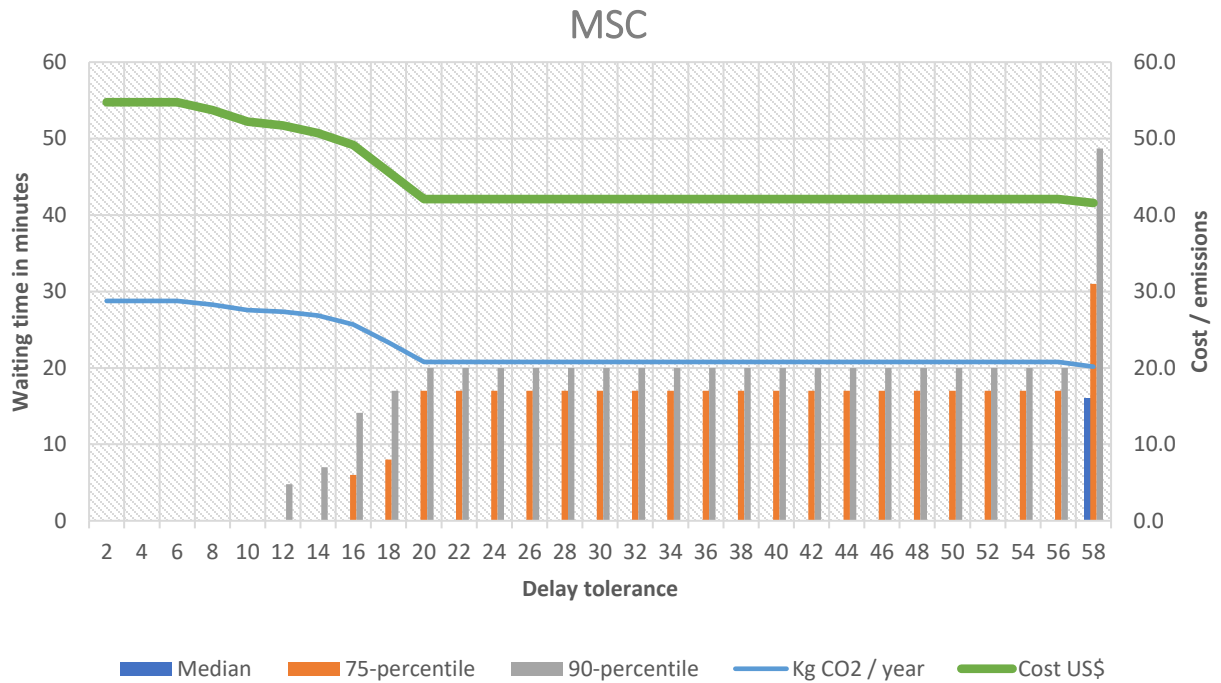


Figure 6.6: diagram illustrating how cost, emissions and actual waiting time develops as delay tolerance increases for the MSC project. Commit logs with commits only during active working hours are used for the simulation.

The 90-percentile for MSC stays at 20 minutes until the test duration reaches 60, which indicates that the developers rarely have to wait for the full extent of the delay tolerance during working hours.

Having a closer look at the commit log gives some insight to why there seems to be high gains of waiting up to the point where delay tolerance equals test duration.

Time between commits, VSC working hours

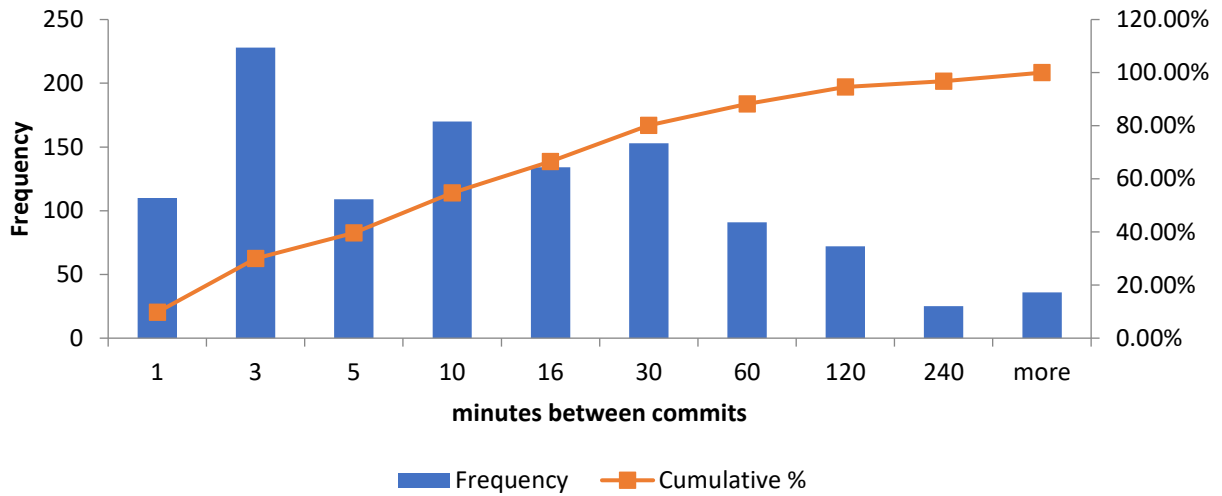


Figure 6.7: frequency diagram illustrating the frequency of different distances in minutes between one commit and the next. The orange line shows cumulative percentage.

66.8% of the tests during working hours were triggered 16 or less minutes after the last test.

With low delay tolerances, these tests will lead to increased costs, because they are not willing to wait until the previous test is done and a new server will be started.

7. Discussion

Looking back at the results, can they be transferred to business value in a real-world use case? Can the CAST-algorithm have a life after this project? This project has sought to find a useful contribution in the fight for a greener future through investigating, building, and testing different variations of a model using the CAST-algorithm.

In this chapter the findings from this investigation and the choices made along the way will be discussed.

7.1 How does these findings answer the problem statements?

The problem statements were written in the very first chapter of this thesis and have been the background for all of the work done. In this sub-chapter they are revisited and the extent to which they were successfully answered is discussed.

7.1.1. Problem statement 1

The first problem statement was: *investigate the development of a model which attempts to optimize the organization of tests in an automated test suite with the objective of least energy greenhouse gas emissions.* Throughout the background and design chapter it was shown that such a model can be made, but it requires some special conditions which are only present in some parts of the world. It will only be useful within geographical areas that have a relevant share of IRE. The area must also have a well-functioning, dynamic, high-resolution and transparent energy market, where data on prices and production is available to the public.

The algorithms used in previous research found and presented in the background chapter used only IRE production or meteorological data for finding the optimal area. This makes the algorithm available in more areas as it removes the requirement of market data. This is sufficient if the only objective is to ensure that the data center uses energy where there is available IRE. However, in this project, the objective is to reduce emissions overall. In this case it is not helpful if one uses green energy where the demand for energy in the area is so large that it causes imports of gray energy from other areas or starts up gas-powered balancing power plants. In areas with large surplus of IRE, there is a risk that wind turbines can be shut

down to prevent overload on the grid if no-one will use the power. In order to locate areas with actual surplus of green energy, one needs dynamic pricing data from a transparent energy market.

This transparency is not so much a technical prerequisite, as a political one. Hopefully, more areas will meet these conditions in the future.

What makes the CAST-algorithm special is that it will consider the entire electricity production and distribution system. This enables it to pursue real greenhouse gas emissions savings, by taking energy from areas with a surplus. But this also limits the commercial potential of this kind of solution. Adding an algorithm like this will increase the complexity in the DevOps-pipeline, an area that is already conceived by many as complex and difficult to manage. At the same time, the immediate benefits are not obvious to the business's stakeholders. To market a business as "green", it is common to make commitments to buy 100% renewable energy, which means buying an equivalent amount of the businesses energy needs from green production sources. This does not guarantee that the energy the business actually use is green, but for marketing purposes it is good enough, and it does not put a strain on the resources of DevOps engineers.

7.1.2. Problem statement 2

The second problem statement was: *evaluate whether the model is successful in making a significant reduction in the usage of non-green energy*. No definition of successful has been made beforehand, but one measure of success is whether the CAST algorithm can be used for other purposes than this research. The algorithm is fairly simple and fast and could be implemented without using a lot of work hours or much computing power. Still, it does add more complexity to a CI/CD pipeline. A large obstacle of using the algorithm in practice, is the limited selection of data centers from each cloud provider.

The extent to which it has made a significant reduction in the usage of non-green energy is more easily observed. A comparison was done where one simulation placed all servers in the area SE4, which is the one with the most installed IRE production capacity in the research area,

and the other with full CAST-algorithm. The number of servers deployed in an area without IRE surplus was reduced by 47% by using the algorithm, which is a significant reduction.

There has been no shortage of data to test on, and three projects were selected as representatives for different types of software development. Even though VS Code is one of the most active projects on GitHub, the automated tests in the repository are small compared to the test suites of some of the larger enterprise solutions. The effects on a large commercial software project with an extensive test suite has not been examined.

The results also indicated that there is a goldilocks-zone of sorts, where delay tolerance is getting near the test duration. At this point data from both MSC and VSC showed that the cost and emissions were significantly reduced. This could be explored further to see whether this is a pattern across all projects, and if it is something that can be detected and used to improve the algorithm.

Also, only four weeks were chosen for representatives of one year. Some manual inspection of the data was done, and holiday weeks were avoided. Still, there extrapolations from these four weeks to annual values have a high degree of uncertainty. Another weakness is the synthetic data used for delay tolerance. In a real-life situation this probably would be a lot more dynamic, and developers would choose delay tolerances based on what they are planning to do in the immediate future after a commit.

7.2 Takeaways from the process

This project followed an exploratory approach, which enabled fast turnarounds and changes along the way as more insight were gained. The approach gives great freedom to the author, something which can be frustrating because it results in unlimited options that one wants to investigate. In the ongoing work the turns and events have been discussed and evaluated frequently along the way. This has made it possible to make full use of the freedom of the exploratory approach and minimizing frustration.

7.2.1. The importance if interdisciplinary backgrounds for this type of projects

In this project, the two disciplines of energy economics and information technology has been combined. The experience and knowledge from my master's degree in Energy Economics has been used actively throughout the process. Having this domain knowledge ready at hand has enabled faster progress than what would otherwise be possible, as the time span of a short thesis is insufficient to consult experts or learn a new domain to the degree where one can be creative and make swift decisions. Future projects with the same type of interdisciplinary form will always have a need to consolidate expertise from several fields. The experience from this project is how important it is to have access to both fields and the advantage of knowing both of them well.

7.3 Deploy overhead.

This model operates under the assumption that there is a cost associated with starting up a new test server, and that it is necessary to shut down the server running the test environment to save energy while no tests are running. In the future it might be possible to keep environments ready in all areas without using energy by utilizing tiny operating system for running only one application that can start up in milliseconds. One such initiative is IncludeOS which was granted research funds from the European Unions Program Horizon (CORDIS EU research results, 2019).

7.4 Data center availability

In this project, the assumption has been made that one can choose to run the tests in any region in the Nordics. This is, however, not practical today, even though there are data centers in all grid areas. A software company will usually stick to one cloud provider, and today none of the providers have data centers in all the Nordic countries. This is likely to change in the future. Today, Google has one data center that is publicly available in the Nordics, in Finland. They are building one new data center in Denmark and they have bought land in Sweden in order to secure an option to build there (Google, 2019) (Moss, 2017). Microsofts Azure has data centers in two areas in Norway today, but they will open in two locations in Sweden in 2021, and plan to build data centers in Denmark as well (Microsoft, 2020) (Microsoft, 2020). Choosing between areas within the same cloud provider will probably be a more accessible option in the future. Also, there will hopefully be tools in place to help "abstract" the cloud interface to some degree, just like configuration management systems mask the actual operating system which it is running on.

7.5 The future of automated software testing

As mentioned in the background chapter, software testing gets little attention in the education of software developers. Still, automated software testing is gaining popularity, and although many companies are yet to introduce it, the expectations are that the market for automated software testing will grow by 18% a year until 2024 (Markets and Markets, 2019). Tools like Katalon TestOps and Github Actions are making it easier to include automated software testing even in smaller project without dedicated test or DevOps engineers to set it up. This indicates that there is future potential for an algorithm like CAST.

7.6 Electricity production in the future

7.6.1. More wind and solar power

At the time of writing, the share of IRE production in the countries Nordic system is approximately 17%. The share of intermittent renewables will increase dramatically in the coming years, and it is forecasted that by 2040, 37% of all electricity delivered to the grid will come from wind and solar powered plants. Parts of the capacity will probably be installed offshore, introducing wind power to areas that have previously not had a large share of IRE.

At the same time, it is planned to reduce production capacity of thermal powered plants (The Norwegian water resources and energy directorate, 2020). This means that in the future, it is likely that most hours of the day have a surplus of IRE in one or more areas.

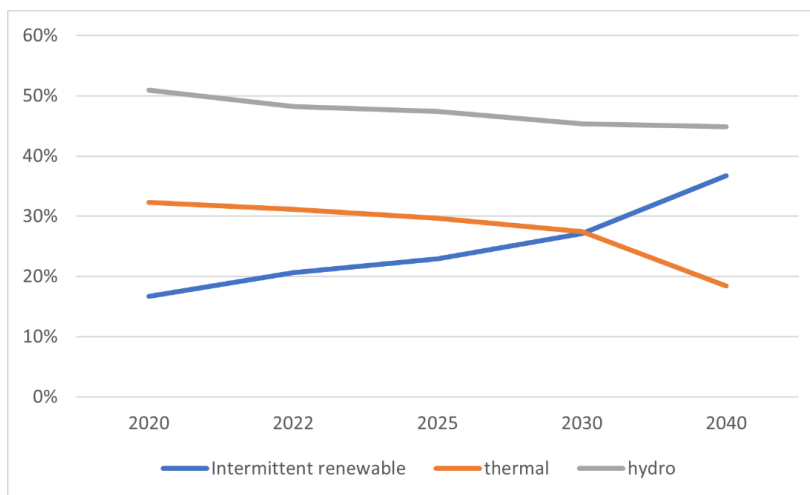


Figure 7.1: Forecast of share of production from different energy sources in the Nordic countries (The Norwegian water resources and energy directorate, 2020).

7.6.2. Thermal production sources - A comeback for nuclear power?

Thermal production plants are delivering large parts of the electricity in the Nordic countries today, and they are important due to their ability to deliver large and steady loads throughout the day. It is a clear political goal to remove all production that causes greenhouse gas emissions, like coal, oil and gas. However, there is no consensus when it comes to nuclear power plants. Germany have decided to completely eradicate nuclear power from their energy mix, due to concern about nuclear disasters and waste disposal accidents. Sweden has been a

bit back and forth on the issue, but the current status is that existing newer reactors shall be kept operational. Nuclear power plants supply over 40% of Sweden's electricity production, so phasing it out would require massive installations of alternative production sources, such as wind and solar (International Atomic Energy Agency, 2020).

Others go in the opposite direction. Nuclear power plants can produce very large quantities of energy without any greenhouse gas emissions, which is why Finland is building new nuclear power plants to support their future energy needs. Either way, the future of nuclear will have a significant impact on the future energy production profile.

7.7 Other applications

Similar models for allocating workloads based on renewable energy supply can be applied to other parts of the software ecosystem. One possible application is to move the process of analyzing the best areas and shift workloads to the cloud providers. In April 2020, Google launched the first version of their carbon-intelligent computing platform, which moves certain non-urgent tasks to the "greenest" hour of the day based on wind and solar forecasts. Analysis of the performance is ongoing, and Google has promised to release research publications on the topic. A second version that moves workloads both in time and location is under development (Radovanovic, 2020). Another possibility is to use the algorithm in testing-as-a-service platforms, where testing from different projects is performed by one party.

Another possibility for the data center customer, would be to apply methods like the one from this project on other parts of the software operations process. Scheduled jobs like indexing and batch data processing could be good candidates. These are inherently easier to schedule than tests in the pipeline, but perhaps more difficult to move between locations.

In this project, the effects were analyzed per project, but the results show that the outcomes are similar in all projects. The smallest project had little to gain from introducing such an algorithm, but there are quite a few smaller projects out there. Commonly used collaboration platforms like GitHub and GitLab offer built-in CI/CD tools with free testing for smaller projects (limited to 2000 minutes a month), larger projects can pay to use the service. If actors who supply these kinds of products were to include an algorithm like CAST, even small projects

would be able to use it. If a small percentage of the projects do, the savings could become significant. GitHub has more than 100 million projects, but how many of these are active is difficult to say.

As previously mentioned, the CAST-algorithm ensures that one uses actual green energy in a way that is not strictly necessary in for branding a company as green after today's standards. This might change in the future, and governments can impose cap-and-trade schemes that limit the amount of gray energy a company can use. The CAST algorithm could be used in a reporting tool, keeping track of how much one has used of the gray quota, in addition to ensuring as much green consumption as possible.

7.8 Possible improvements and further work

7.8.1. Parallelization and sectioning for advanced users

In this project, an underlying assumption has been that a test suite will run on one server only. In order to get faster feedback, it is not unusual to run tests in parallel over several servers. This is faster than running tests in parallel on one server. If one wanted to make a solution that can be used in projects that uses parallelization techniques, it would need to also handle parallel test runs on multiple servers.

For projects with a sophisticated setup for larger test suites, it could also be possible to divide the test suite into smaller subsets and apply the CAST-algorithm for each subset, placing each subset in the most optimal area.

7.8.2. Combine with test ordering algorithms.

Sub test ordering techniques are used mainly to get feedback to the developers faster in projects with large test suites. These work by setting up each test in the test suite in an order designed to provoke test fails as early as possible in the process. Research mentioned in the background chapter has found these methods to be very effective in reducing test duration. From the simulations with different test durations, it was evident that shortening the test duration also can lead to significant benefits related to greenhouse gas emissions and cost

savings. Combining the CAST algorithm with test case prioritization algorithms could be interesting to investigate in a future project.

7.8.3. Further examine the relationship between test frequency and waiting time.

When looking at actual waiting times, the results showed that the distance between commits and the test duration has an impact on the cost and emissions. Queues of tests caused by rapid commits are present in the two largest projects examined, and it is not unlikely that this is a common trait in larger projects. Further work could have a closer look at these patterns and attempt to identify profiles that can be used to suggest optimal delay tolerances.

7.8.4. Apply nudging features to increase delay tolerance.

The experiments have all been done under the assumption that there is a pre-defined number of minutes that a developer can accept to wait before the test starts (delay tolerance). There are different ways that this variable can be set. The initial thought has been that the developer adds how long they can wait as a parameter when committing the code. Other ways can also be explored, for instance one can analyze the coming hours and give some options to the developer, like a pop-up asking “postponing your test start by 15 minutes will make it run in a server that uses pure green energy. Would you like to wait?”. Another possibility could be a menu of possible run times, where the coming hours are displayed with different carbon footprint and the developer can choose at which hour to schedule the tests.

7.8.5. Include data center pricing as a decision parameter.

The CAST algorithm does not consider data center prices in its current form. When examining the pricing scheme, it was found that the price difference of data centers from the same vendor was very small. Prices across vendors had more variation. Some vendors offer spot pricing for consumers that can plan ahead, which is something that can be used in an improved version of the CAST-algorithm to drive cost reduction.

7.8.6. Build a prototype.

So far, nothing has been discovered to indicate that one could not implement the CAST-algorithm as part of a DevOps pipeline. Still, there are several ways one could go forward with

such work. The allocation and server management could be added as a script in a tool like Jenkins or Azure DevOps, or one could build it as a service that is called upon from the pipeline. This, or other options could be explored and tested in a prototyping experiment to learn more about how to best implement an algorithm like CAST.

8. Conclusion

Automated software testing is one of the many energy consuming activities in software development. This thesis has sought to explore the possibility of reducing the carbon footprint from this activity by dynamically placing the test activity in areas where there is a surplus of green energy generated by wind and solar power plants.

The result of the background and design phase is the Carbon-free Automated Software Testing – CAST algorithm. The algorithm considers the production of energy from different sources and the demand situation to determine where there is a surplus of green energy that can be utilized.

Simulations using a combination of real and synthetic data showed that the CAST algorithm is successful in reducing the carbon footprint of automated testing for both small and large projects. An essential condition for the success of the algorithm is the availability of data from a well-functioning electricity market with high resolution dynamic pricing. At the moment, this is not available across the globe, however, the availability and sophistication of that data is increasing.

The simulations showed that reducing the test duration had a large positive impact on the carbon footprint of the test. It would therefore be interesting to look closer at combining the CAST-algorithm with techniques to shorten the test duration, like parallelization and test ordering in future work.

9. Litterature

Amazon. (2020, 12 10). *amazon*. Hentet fra news:

<https://www.aboutamazon.com/news/sustainability/amazon-becomes-the-worlds-largest-corporate-purchaser-of-renewable-energy>

Belkhir, L., & Elmeligi, A. (2018, 03 10). Assessing ICT global emissions footprint: Trends to 2040 & recommendations. *Journa of Cleaner Production*, ss. 448 - 463.

Butgereit, L. (2019). "Using Fiction to Inspire Agility in Information Technology and Manufacturing: A Look at The Phoenix Project and The Goal. *2019 IEEE 10th International Conference on Mechanical and Intelligent Manufacturing Technologies (ICMIMT)* (ss. 138-141). Cape Town: IEEE.

Catal, C., & Mishra, D. (2012, 07 26). Test case prioritization: a systematic mapping study. *Software quality journal*, ss. 445-478.

CORDIS EU research results. (2019, 11 11). *Cordis Horizon 2020*. Hentet fra cordis.europa.eu: <https://cordis.europa.eu/project/id/829668>

European commission. (2021, 05 13). *Climate strategies and targets*. Hentet fra European comission, energy, climate change, environment: https://ec.europa.eu/clima/policies/strategies/2030_en

Forsgren, N., Humble, J., & Kim, G. (2018). *Accellerate*. Portland: IT revolution.

Gao, Y., Zeng, Z., Liu, X., & Kumar, P. (2013). The answer is blowing in the wind: Analysis of powering Internet data centers with wind energy. *Proceedings IEEE INFOCOM* (ss. 520 - 524). Turin, Italy: IEEE.

GoClimate. (2019, 5 23). *GoClimate*. Hentet fra Blog: <https://www.goclimat.com/blog/the-carbon-footprint-of-servers/>

Google. (2019). Hentet fra about:

<https://www.google.com/about/datacenters/locations/fredericia/>

Haghighatkah, A., Mäntylä, M., Oivo, M., & Kuvaja, P. (2018, 12). Test prioritization in continuous integration environments. *Journal of Systems and Software*, ss. 80-98.

Hematti, H., Fang, Z., Mäntylä, M. V., & Adams, B. (2016, 07 13). Prioritizing manual test cases in rapid release environments. *Journal of software: testing Verification and reliability*.

International Atomic Energy Agency. (2020). *Country Nuclear Power Profile (CNPP) Sweden*. IAEA.

International Energy Agency. (2020). *Data Centres and Data Transmission Networks - Tracking report*. Paris: IEA.

Liu, Z., Wierman, A., Ling, M., & Low, S. (2011, 12). Geographical Load Balancing with Renewables. *Sigmetrics Performance Evaluation Reviw SIGMETRICS*, ss. 62 - 66.

Markets and Markets. (2019). *Automation Testing Market by Component (Testing Types (Static, Dynamic (Functional, Non-functional)), Services), Endpoint Interface (Mobile, Web, Desktop, Embedded Software), Organization Size, Vertical, and Region - Global Forecast to 2024*. Northbrook: Markets and Markets.

Mazrekaj, A., Shabani, I., & Sejdiu, B. (2016, 02 01). Pricing Schemes in Cloud Computing: An Overview. *International Journal of Advanced Computer Science and Applications*.

Microsoft. (2020, dec 7). *microsoft.com*. Hentet fra news:

<https://news.microsoft.com/europe/features/microsoft-announces-plans-to-establish-a-new-datacenter-region-in-denmark-to-accelerate-the-countrys-green-digital-transformation/>

Microsoft. (2020, 11 24). *microsoft.com*. Hentet fra news:

<https://news.microsoft.com/europe/2020/11/24/microsoft-announces-investments-to-accelerate-swedens-digital-transformation-and-plans-to-open-its-sustainable-datacenter-region-in-2021/>

- Moss, S. (2017, 10 16). *Google acquires 109 hectares of land in rural sweden*. Hentet fra Data center dynamics: <https://www.datacenterdynamics.com/en/news/google-acquires-109-hectares-of-land-in-rural-sweden>
- Murugan Tanggiah, S. B. (2016). A preliminary analysis of various testing techniques in agile development - a systematic literature review. *International conference on computer and information sciences* (ss. 600-605). Kuala Lumpur: 3rd International Conference on Computer and Information Sciences (ICCOINS).
- Mytton, D. (2020, 08 08). Accessing the suitability of the Greenhouse Gas Protocol for calculation of emissions from public cloud computing workloads. *Journal of cloud computing*, s. Article number 45.
- Otsuki, T., Komiyama, R., & Fuji, Y. (2017, 7 3). Study on Surplus Electricity under Massive Integration of Intermittent Renewable Energy Sources. *Electrical Engineering in Japan*, ss. 17-31.
- Pham, R., Kiesling, S., Singer, L., & Schneider, K. (2017, 11 07). Onboarding inexperienced developers: struggles and perceptions regarding automated testing. *Software Quality Journal*, ss. 1239 - 1268.
- Radovanovic, A. (2020, 04 22). *company news: data centers and infrastructure*. Hentet fra Google : <https://blog.google/inside-google/infrastructure/data-centers-work-harder-sun-shines-wind-blows>
- Saff, D., & Ernst, M. (2003). Reducing wasted development time via continuous testing. *14th International Symposium on Software Reliability Engineering* (ss. 281 - 292). Denver, Colorado: ISSRE.
- Schmidt, S. (2010, 4 30). *The Guardian*. Hentet fra <https://www.theguardian.com>: <https://www.theguardian.com/environment/2010/apr/30/cloud-computing-carbon-emissions#:~:text=According%20to%20a%20recent%20Greenpeace,emissions%20would%20reach%201%2C034%20megatonnes>

- Statnett. (2018). *Fleksibilitet i det nordiske kraftmarkedet 2018 - 2040*. Oslo: Statnett.
- T. Sakamoto, Yamada, H., Horie, H., & Kono, K. (2012). Energy-Price-Driven Request Dispatching for Cloud Data Centers. *2012 IEEE Fifth International Conference on Cloud Computing* (ss. 974-976). Honolulu: IEEE.
- The Norwegian water resources and energy directorate. (2020). *Langsiktig kraftmarkedsanalyse 2020*. Oslo: NVE.
- Toosi, A. N., Qu, C., Assunuco, M., & Buyya, R. (2017, February 6). Renewable-aware Geographical Load Balancing of Web Applications for Sustainable Data Centers. *Journal of network and computer applications*.
- U. S. Energy Information Administration. (2021, 04 17). *electricity-generation-from-wind: webarea for EIA*. Hentet fra EIA: <https://www.eia.gov/energyexplained/wind/electricity-generation-from-wind.php>
- UK Department for Business, Energy & Industrial Strategy. (2020, 06 17). *gov.uk/government/publications/*. Hentet fra gov.uk/government: <https://www.gov.uk/government/publications/greenhouse-gas-reporting-conversion-factors-2020>
- World Economic Forum. (2019, 12 19). *weforum.org*. Hentet fra world economic forum: <https://www.weforum.org/agenda/2019/12/with-thank-you-emails-polite-britons-burn-thousands-of-tonnes-of-carbon-a-year/#:~:text=Britons%20send%20more%20than%2064,list%20of%20most%20common%20offenders.&text=Sending%20one%20less%20%27thank%20you,tonnes%20of%20>

10. Appendices

Appendix A Python files

A1 globalVariables.py

```
AREAS = [ 'SE1', 'SE2', 'SE3', 'SE4', 'FI', 'DK1', 'DK2', 'N01', 'N02', 'N03', 'N04', 'N05', 'EE' ]
IRE_THRESHOLD = 0.2
NEIGHBORS = { 'DK1': ['DK2', 'SE3', 'N02', 'DE-LU', 'NL'], 'DK2': ['DK1', 'SE4', 'DE-LU'], 'SE1': ['FI', 'SE2', 'N04'], 'SE2': ['SE1', 'SE3', 'N04', 'N03'], 'SE3': ['SE2', 'FI', 'SE4', 'N01'], 'SE4': ['SE3', 'LT', 'DK2', 'DE-LU', 'PL'], 'N01': ['SE3', 'N02', 'N03', 'N05'], 'N02': ['NL', 'DE-LU', 'DK1', 'N01', 'N05'], 'N03': ['N04', 'SE2', 'N01', 'N05'], 'N04': ['SE1', 'SE2', 'N03'], 'N05': ['N03', 'N01', 'N05'], 'EE': ['LV', 'FI'], 'FI': ['SE3', 'EE', 'SE1'] }
KEEP_UP_THRESHOLD = 5
```

A2 simulate.py

```
import datetime
from tinydb import TinyDB, Query
from decideServer import checkIfWeHaveServer, orderServer, addTestCollectionToServer
from decideArea import findOptimalArea
from globalVariables import AREAS
import csv
import os
from shutil import copyfile

# script for feeding commit-logs into the servercheck and allocation algorithms
# in order to simulate traffic to the "green testing module"
# the results are recorded in two files, server_order.json (server log) and results.csv

serverDB = TinyDB('storage/server_orders.json')
Ask = Query()
project = 'UDF'
indata = 'base'
scenario = 'WT60'
season = 'all'
filename = 'storage/commitlogs/' + indata + "/" + project + '_' + season + '.csv'
fileOutput = [project, season, scenario]
dbStored = 'storage/db_files/' + scenario + "_" + project + '_' + season + '.json'
```

```

waitingTime = 0
waitingTimeFileName = 'storage/' + scenario + "_" + project + '_' + season + '.txt
'

def countServersPerArea():
    areaCountList = []
    for area in AREAS:
        count = serverDB.count(Ask.area == area)
        areaCountList.append(count)
    return areaCountList

def CalculateServerminutes(testingMinutes):
    greenServerMinutes = 0
    grayServerMinutes = 0
    greenWaste = 0
    grayWaste = 0
    outputList = []
    for n in range (1, (len(serverDB)+1)):
        serverOrder = dict(serverDB.get(doc_id=n))
        if serverOrder['green'] == 1:
            greenServerMinutes += serverOrder['tests'] * testingMinutes
            if serverOrder['shutdown'] == 0:
                greenWaste += 60 - (serverOrder['tests'] * testingMinutes)
        else:
            grayServerMinutes += serverOrder['tests'] * testingMinutes
            if serverOrder['shutdown'] == 0:
                grayWaste += 60 - (serverOrder['tests'] * testingMinutes)
    outputList.extend([grayServerMinutes, greenServerMinutes, greenWaste, grayWas
te])
    return outputList

# loops through a file with commitlog timestamps and prints the output of each co
mmit
with open(filename) as f:
    csvReader = csv.reader(f)
    commits = 0
    delayTolerance = 0
    testDuration = 0
    greenServers = 0
    grayServers = 0
    utilizedServers = 0
    for line in csvReader:
        commits +=1
        delayTolerance += int(line[1])

```

```

        testDuration += int(float(line[2]))
        timeOfCommit = datetime.datetime.fromtimestamp(int(line[0]))
        haveServer = checkIfWeHaveServer(int(line[0]), int(line[1]), int(float(line[2])))
        if haveServer != 0:
            waitingTime = addTestCollectionToServer(haveServer.doc_id, timeOfCommit, int(float(line[2])), line[1])
            utilizedServers += 1
        else:
            where = findOptimalArea(int(line[0]), int(line[1]))
            if where != "no surplus area":
                x = orderServer(where["date"], where["hour"], where["minute"], where["area"], int(line[1]), int(float(line[2])), 1)
                startTimeString = where['date'] + '-' + str(where['hour']) + '-' + str(where['minute']) # added for waiting time count
                startTime = datetime.datetime.strptime(startTimeString, '%d.%m.%Y-%H-%M') # added for waiting time count
                waitingTime = int((startTime - timeOfCommit).total_seconds() / 60.0) # added for waiting time count
                greenServers += 1
            else:
                x = orderServer(timeOfCommit.strftime('%d.%m.%Y'), timeOfCommit.hour, timeOfCommit.minute, "N01", int(line[1]), int(float(line[2])), 0)
                grayServers += 1
                waitingTimeFile = open(waitingTimeFileName, "a")
                waitingTimeFile.write(str(waitingTime) + "\n")
                waitingTimeFile.close()
                print(waitingTime)
            delayTolerance = delayTolerance/commits
            testDuration = testDuration / commits
            fileOutput.extend([delayTolerance, testDuration, commits, grayServers, greenServers, utilizedServers])

f.close()

minutes = CalculateServerminutes(fileOutput[4])
fileOutput.extend(minutes)
fileOutput.extend(countServersPerArea())

outfile = open('storage/results.csv', "a")
csv_writer = csv.writer(outfile)
csv_writer.writerow(fileOutput)

outfile.close()

```

```
copyfile('storage/server_orders.json', dbStored)
os.remove('storage/server_orders.json')
```

```
print(fileOutput)
```

A3 decideServer.py

```
import json
from tinydb import TinyDB, Query
import datetime
import json
import ast
from globalVariables import KEEP_UP_THRESHOLD

serverDB = TinyDB('storage/server_orders.json')
Ask = Query()

# calculate remaining time on server-hour and record reservation in database
# if delay tolerance is less than 15 minutes, expect more tests, don't shut down
server after execution.
def orderServer(date, hour, minute, area, delayTolerance, testDuration, green):
    timeString= date + ' ' + str(hour) + ':' + str(minute)
    startTime = datetime.datetime.strptime(timeString, '%d.%m.%Y %H:%M')
    testEndTime = startTime + datetime.timedelta(minutes=testDuration)
    timeOut = startTime + datetime.timedelta(minutes=60)
    timeLeft = (timeOut-testEndTime).total_seconds() /60
    shutdown = 1
    if delayTolerance <= KEEP_UP_THRESHOLD:
        shutdown = 0
    x = serverDB.insert({"area": area, "start": int(startTime.timestamp()), "end"
: int(testEndTime.timestamp()), "timeout": int(timeOut.timestamp()), "shutdown":
shutdown, "timeleft":int(timeLeft), "green": green, "tests": 1})
    return x

# See if server is available that can complete the test within acceptable delay,
and before the one hour mark
# returns server reservation from database or zero
def checkIfWeHaveServer(timestamp, delayTolerance, testDuration):
    commitTime = datetime.datetime.fromtimestamp(int(timestamp))
    maxTestStart = commitTime + datetime.timedelta(minutes=delayTolerance)

optionsNoShutdown = serverDB.get(
    (Ask.shutdown == 0) &
```



```

        (Ask.end <= maxTestStart.timestamp()) &
        (Ask.timeout >= int(timestamp)) &
        (Ask.timeleft >= testDuration)
    )

    if optionsNoShutdown:
        startTimeForIncomingTest = max(optionsNoShutdown['end'], timestamp)
        timeLeftToRunTests = (datetime.datetime.fromtimestamp(optionsNoShutdown['
timeout'])-
datetime.datetime.fromtimestamp(startTimeForIncomingTest)).total_seconds() /60
        if timeLeftToRunTests >= testDuration:
            return optionsNoShutdown

optionsWithShutdown = serverDB.get(
    (Ask.shutdown == 1) &
    (int(timestamp) <= Ask.end) &
    (maxTestStart.timestamp() >= Ask.end) &
    (Ask.timeleft >= testDuration)
)

if optionsWithShutdown:
    return optionsWithShutdown

return 0

# Add a test run to a server that is alerady ordered
def addTestCollectionToServer(orderID, timeOfCommit, testDuration, delayTolerance
):
    serverOrder = serverDB.get(doc_id=orderID)
    endTime = datetime.datetime.fromtimestamp(serverOrder['end'])
    startTimeForIncomingTest = max(endTime, timeOfCommit)
    waitingTime = int((startTimeForIncomingTest - timeOfCommit).total_seconds() /
60.0) # added for waiting time count
    newEndTime = startTimeForIncomingTest + datetime.timedelta(minutes=testDurati
on)
    newTimeLeft = (datetime.datetime.fromtimestamp(serverOrder['timeout'])-
newEndTime).total_seconds() /60
    newNumberOfTests = serverOrder['tests'] + 1
    serverDB.update({'end': int(newEndTime.timestamp()), "timeleft": newTimeLeft,
"tests": newNumberOfTests }, doc_ids=[orderID])
    return(waitingTime) # added for waiting time count

```

A4 CAST.py

```
import json
from tinydb import TinyDB, Query
import datetime
from collections import defaultdict
from globalVariables import AREAS, IRE_THRESHOLD, NEIGHBORS

IREDB = TinyDB('storage/volumes_IRE.json')
priceDB = TinyDB('storage/prices.json')
totalProductionDB = TinyDB('storage/volumes_total.json')

Ask = Query()

def calculateDelta(date, hour, area):
    neighbors = NEIGHBORS[area]
    prices = dict(priceDB.get((Ask.date == date) & (Ask.hour == hour)))
    delta = 0
    for n in neighbors:
        difference = float(prices[area]) - float(prices[n])
        delta += difference
    return delta

def findOptimalArea(timestamp, delayTolerance):
    firstTime = datetime.datetime.fromtimestamp(int(timestamp))
    timeEnd = firstTime + datetime.timedelta(minutes=delayTolerance)
    surplusDays = dict()
    counter = 0
    time = firstTime
    minutes = 0
    while time <= timeEnd:
        nowDate = time.strftime("%d.%m.%Y")
        productionIRE = dict(IREDB.get((Ask.date == nowDate) & (Ask.hour == str(t
ime.hour))))
        productionTotal = dict(totalProductionDB.get((Ask.date == nowDate) & (Ask
.hour == str(time.hour))))
        areas = AREAS
        for item in areas:
            fractionOfIRE = int(productionIRE[item]) / int(productionTotal[item])
            if fractionOfIRE >= IRE_THRESHOLD:
                delta = calculateDelta(nowDate, str(time.hour), item)
                if delta < 0:
```

```

        counter += 1
        instance = {"date": time.strftime("%d.%m.%Y"), "hour": time.h
our, "minute": minutes, "area": item, "delta": delta}
        surplusDays[counter]=instance
        time = time + datetime.timedelta(minutes=60)
    if surplusDays:
        lowestDelta = min(surplusDays, key=lambda v: surplusDays[v]['delta'])
        bestAllocation = surplusDays[lowestDelta]
        if bestAllocation["hour"] == firstTime.hour: #make sure we start right a
way if the first hour is the best
            bestAllocation["minute"] = firstTime.minute
    else:
        bestAllocation = "no surplus area"
    return bestAllocation

```

A5 delayTolerances.py

```
import datetime
import csv
from numpy import random

# loops through txt file with commitlog timestamps
# adds delay tolerance based on the hour of the day
# writes to csv-file for use in simulations

def addDelayTolerance(filename, mean, standardDeviation):
    splitFileName = filename.split("/")
    newFileName = splitFileName[2][:-3]
    outFileName = "storage/commitlogs/withDelayTolerance/" + newFileName + ".csv"
    outfile = open(outFileName, "a")
    delayTolerance = 0
    csvWriter = csv.writer(outfile)
    with open(filename) as f:
        for line in f:
            testDuration = abs(random.normal(loc=mean, scale=standardDeviation))
            timeOfCommit = datetime.datetime.fromtimestamp(int(line))
            if timeOfCommit.hour == 11:
                record = [int(line), "30", testDuration]
                csvWriter.writerow(record)
            elif timeOfCommit.hour >= 16 and timeOfCommit.hour < 20:
                record = [int(line), "720", testDuration]
                csvWriter.writerow(record)
            else:
                record = [int(line), "5", testDuration]
                csvWriter.writerow(record)
    outfile.close()

# Test Duration is the second argument
# Standard deviation third argument, set to 0 for non-noisy test durations

seasons = ["summer", "winter", "spring", "autumn"]
projectname = "UDF"

for n in seasons:
    filename = "storage/commitlogs/" + projectname + "_" + n + ".txt"
    addDelayTolerance(filename, 22, 11)
```

A6 heatmap.py

```
import json
from tinydb import TinyDB, Query
import datetime
from collections import defaultdict
from globalVariables import AREAS, IRE_THRESHOLD, NEIGHBORS
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib import colors
import numpy as np

IREDB = TinyDB('storage/volumes_IRE.json')
priceDB = TinyDB('storage/prices.json')
totalProductionDB = TinyDB('storage/volumes_total.json')

Ask = Query()

# returns a nested list with price deltas and time for all areas
def calculateDelta(date, hour, area):
    neighbors = NEIGHBORS[area]
    prices = dict(priceDB.get((Ask.date == date) & (Ask.hour == hour)))
    delta = 0
    for n in neighbors:
        difference = float(prices[area]) - float(prices[n])
        delta += difference
    return delta

def findOptimalArea(year, month, day, hour):
    time = datetime.datetime(year, month, day, hour, 0, 0, 0)
    surplusDays = ""
    counter = 0
    nowDate = time.strftime("%d.%m.%Y")
    productionIRE = dict(IREDB.get((Ask.date == nowDate) & (Ask.hour == str(time.
hour))))
    productionTotal = dict(totalProductionDB.get((Ask.date == nowDate) & (Ask.hou
r == str(time.hour))))
    areas = AREAS
    for item in areas:
        fractionOfIRE = int(productionIRE[item]) / int(productionTotal[item])
        if fractionOfIRE >= IRE_THRESHOLD:
            delta = calculateDelta(nowDate, str(time.hour), item)
            if delta < 0:
```

```

        counter += 1
        instance = str(time.hour) + ", " + str(item) + ", " + str(delta)
+ "\n"
        surplusDays+=instance
    else:
        counter += 1
        instance = str(time.hour) + ", " + str(item) + ", " + str(delta)
+ "\n"
        surplusDays+=instance
    else:
        counter += 1
        instance = str(time.hour) + ", " + str(item) + ", " + "0" + "\n"
        surplusDays+=instance
time = time + datetime.timedelta(minutes=60)
return surplusDays

def background_gradient(s, m, M, cmap='PuBu', low=0, high=0):
    rng = M - m
    norm = colors.Normalize(m - (rng * low),
                             M + (rng * high))
    normed = norm(s.values)
    c = [colors.rgb2hex(x) for x in plt.cm.get_cmap(cmap)(normed)]
    return ['background-color: %s' % color for color in c]

def generateHeatmapOneDay(date):
    filenameCSV="heatmaps/" + date.replace(".", "") + ".csv"

    f = open(filenameCSV, "a")

    for n in range(0, 24):
        gold = str(findOptimalArea(int(date[6:10]), int(date[3:5]), int(date[:2])
, n))
        f.write(gold)

    f.close()

    imageFileName = "heatmaps/" + date.replace(".", "") + ".png"
    dfData = pd.read_csv(filenameCSV, names=['hour', 'area', 'delta'])
    sbdata= dfData.pivot("area", "hour", "delta")
    colormap = sns.diverging_palette(145, 300, s=60, as_cmap=True)

    ax = sns.heatmap(sbdata, cmap=colormap, center=0, mask=(sbdata==0))
    plt.savefig(imageFileName)

def generateHeatmapsOneWeek(weekstart):

```

```
startDay = datetime.datetime.strptime(weekstart, '%d.%m.%Y')
for n in range(0, 7):
    generateHeatmapOneDay(startDay.strftime('%d.%m.%Y'))
    startDay += datetime.timedelta(days=1)
```

A7 convert.py

```
import json
import tinydb

# convert from json-files to files readable by tinyDB

db = tinydb.TinyDB("storage/volumes_total.json") # create a new storage for the
database

with open("data.txt", "r") as f: # open the unmodified file
    json_data = json.load(f) # parse its JSON

for entry in json_data:
    print(entry)
    db.insert(entry)
```

Appendix B – results

B1 Simulation result data Python Algorithms

project	season	scenario	Avg delay tolerance	avg test duration	commits	gray servers	green servers	re-use	gray minutes	green minutes
PA	autumn	base	10	1	18	0	18	0	0	18
PA	summer	base	10	1	17	4	13	0	4	13
PA	spring	base	10	1	14	1	13	0	1	13
PA	winter	base	10	1	2	0	2	0	0	2
PA	autumn	base-se4	10	1	18	1	17	0	1	17
PA	summer	base-se4	10	1	17	7	10	0	7	10
PA	spring	base-se4	10	1	14	11	3	0	11	3
PA	winter	base-se4	10	1	2	0	2	0	0	2
PA	winter	DT_long	120	1	2	0	2	0	0	2
PA	summer	DT_long	120	1	17	4	12	1	4	13
PA	autumn	DT_long	120	1	18	0	17	1	0	18
PA	spring	DT_long	120	1	14	1	9	4	1	13
PA	summer	DT_short	1	1	17	4	13	0	4	13
PA	autumn	DT_short	1	1	18	0	18	0	0	18
PA	spring	DT_short	1	1	14	1	13	0	1	13
PA	winter	DT_short	1	1	2	0	2	0	0	2
PA	autumn	IRE_low	10	1	18	0	18	0	0	18
PA	summer	IRE_low	10	1	17	2	15	0	2	15
PA	spring	IRE_low	10	1	14	0	14	0	0	14
PA	winter	IRE_low	10	1	2	0	2	0	0	2
PA	autumn	IRE_one	10	1	18	0	18	0	0	18
PA	summer	IRE_one	10	1	17	0	17	0	0	17
PA	spring	IRE_one	10	1	14	0	14	0	0	14
PA	winter	IRE_one	10	1	2	0	2	0	0	2
PA	summer	sdwn_long	10	1	17	3	11	3	4	13
PA	autumn	sdwn_long	10	1	18	0	13	5	0	18
PA	spring	sdwn_long	10	1	14	1	6	7	1	13
PA	winter	sdwn_long	10	1	2	0	2	0	0	2
PA	autumn	sdwn_short	10	1	18	0	18	0	0	18
PA	summer	sdwn_short	10	1	17	4	13	0	4	13
PA	spring	sdwn_short	10	1	14	1	13	0	1	13
PA	winter	sdwn_short	10	1	2	0	2	0	0	2
PA	autumn	TD_long	10	45	18	0	18	0	0	810
PA	summer	TD_long	10	45	17	4	13	0	180	585
PA	spring	TD_long	10	45	14	1	13	0	45	585
PA	winter	TD_long	10	45	2	0	2	0	0	90
PA	summer	TD_noisy	10	2	17	4	13	0	10	32
PA	autumn	TD_noisy	10	3	18	0	17	1	0	45

project	season	scenario	Avg delay tolerance	avg test duration	commits	gray servers	green servers	re-use	gray minutes	green minutes
PA	spring	TD_noisy	10	3	14	1	10	3	3	40
PA	winter	TD_noisy	10	4	2	0	2	0	0	7

Continued

project	season	scenario	green waste	gray waste	se1	se2	se3	se4	fi	dk1	dk2	no2	no3
PA	autumn	base	0	0	4	0	0	12	0	2	0	0	0
PA	summer	base	0	0	0	1	0	10	0	0	0	2	0
PA	spring	base	0	0	0	0	0	3	0	2	0	0	8
PA	winter	base	0	0	0	0	0	1	0	1	0	0	0
PA	autumn	base-se4	0	0	0	0	0	17	0	0	0	0	0
PA	summer	base-se4	0	0	0	0	0	10	0	0	0	0	0
PA	spring	base-se4	0	0	0	0	0	3	0	0	0	0	0
PA	winter	base-se4	0	0	0	0	0	2	0	0	0	0	0
PA	winter	DT_long	0	0	0	0	0	1	0	1	0	0	0
PA	summer	DT_long	0	0	0	0	0	11	0	0	0	1	0
PA	autumn	DT_long	0	0	3	0	0	11	1	2	0	0	0
PA	spring	DT_long	0	0	0	0	0	3	0	2	0	0	4
PA	summer	DT_short	0	0	0	1	0	10	0	0	0	2	0
PA	autumn	DT_short	0	0	4	0	0	12	0	2	0	0	0
PA	spring	DT_short	0	0	0	0	0	3	0	2	0	0	8
PA	winter	DT_short	0	0	0	0	0	1	0	1	0	0	0
PA	autumn	IRE_low	0	0	3	0	1	12	0	2	0	0	0
PA	summer	IRE_low	0	0	0	0	1	7	0	0	0	7	0
PA	spring	IRE_low	0	0	1	0	0	3	0	2	0	0	8
PA	winter	IRE_low	0	0	0	0	0	1	0	1	0	0	0
PA	autumn	IRE_one	0	0	0	0	0	10	0	2	0	5	0
PA	summer	IRE_one	0	0	0	0	1	3	0	0	0	13	0
PA	spring	IRE_one	0	0	0	0	8	4	0	0	0	0	1
PA	winter	IRE_one	0	0	0	0	0	1	0	1	0	0	0
PA	summer	sdwn_long	647	176	0	1	0	9	0	0	0	1	0
PA	autumn	sdwn_long	762	0	2	0	0	9	0	2	0	0	0
PA	spring	sdwn_long	347	59	0	0	0	3	0	1	0	0	2
PA	winter	sdwn_long	118	0	0	0	0	1	0	1	0	0	0
PA	autumn	sdwn_short	0	0	4	0	0	12	0	2	0	0	0
PA	summer	sdwn_short	0	0	0	1	0	10	0	0	0	2	0
PA	spring	sdwn_short	0	0	0	0	0	3	0	2	0	0	8
PA	winter	sdwn_short	0	0	0	0	0	1	0	1	0	0	0
PA	autumn	TD_long	0	0	4	0	0	12	0	2	0	0	0

project	season	scenario	green waste	gray waste	se1	se2	se3	se4	fi	dk1	dk2	no2	no3
PA	summer	TD_long	0	0	0	1	0	10	0	0	0	2	0
PA	spring	TD_long	0	0	0	0	0	3	0	2	0	0	8
PA	winter	TD_long	0	0	0	0	0	1	0	1	0	0	0
PA	summer	TD_noisy	0	0	0	1	0	10	0	0	0	2	0
PA	autumn	TD_noisy	0	0	3	0	0	12	0	2	0	0	0
PA	spring	TD_noisy	0	0	0	0	0	3	0	1	0	0	6
PA	winter	TD_noisy	0	0	0	0	0	1	0	1	0	0	0

Continued

project	season	scenario	no4	no5	ee	waste factor	Ops cost	Gray emissions	Green emissions	Emissions total
PA	autumn	base	0	0	0	0.00	18	0.00	0.01	0.01
PA	summer	base	0	0	0	0.00	17	0.00	0.00	0.01
PA	spring	base	0	0	0	0.00	14	0.00	0.00	0.00
PA	winter	base	0	0	0	0.00	2	0.00	0.00	0.00
PA	autumn	base-se4	0	0	0	0.00	18	0.00	0.01	0.01
PA	summer	base-se4	0	0	0	0.00	17	0.01	0.00	0.01
PA	spring	base-se4	0	0	0	0.00	14	0.01	0.00	0.01
PA	winter	base-se4	0	0	0	0.00	2	0.00	0.00	0.00
PA	winter	DT_long	0	0	0	0.00	2	0.00	0.00	0.00
PA	summer	DT_long	0	0	0	0.00	16	0.00	0.00	0.01
PA	autumn	DT_long	0	0	0	0.00	17	0.00	0.01	0.01
PA	spring	DT_long	0	0	0	0.00	10	0.00	0.00	0.00
PA	summer	DT_short	0	0	0	0.00	17	0.00	0.00	0.01
PA	autumn	DT_short	0	0	0	0.00	18	0.00	0.01	0.01
PA	spring	DT_short	0	0	0	0.00	14	0.00	0.00	0.00
PA	winter	DT_short	0	0	0	0.00	2	0.00	0.00	0.00
PA	autumn	IRE_low	0	0	0	0.00	18	0.00	0.01	0.01
PA	summer	IRE_low	0	0	0	0.00	17	0.00	0.00	0.01
PA	spring	IRE_low	0	0	0	0.00	14	0.00	0.00	0.00
PA	winter	IRE_low	0	0	0	0.00	2	0.00	0.00	0.00
PA	autumn	IRE_one	1	0	0	0.00	18	0.00	0.01	0.01
PA	summer	IRE_one	0	0	0	0.00	17	0.00	0.01	0.01
PA	spring	IRE_one	1	0	0	0.00	14	0.00	0.00	0.00
PA	winter	IRE_one	0	0	0	0.00	2	0.00	0.00	0.00
PA	summer	sdwn_long	0	0	0	49.77	14	0.16	0.20	0.36
PA	autumn	sdwn_long	0	0	0	42.33	13	0.00	0.24	0.24
PA	spring	sdwn_long	0	0	0	26.69	7	0.05	0.11	0.16
PA	winter	sdwn_long	0	0	0	59.00	2	0.00	0.04	0.04

project	season	scenario	no4	no5	ee	waste factor	Ops cost	Gray emissions	Green emissions	Emissions total
PA	autumn	sdwn_short	0	0	0	0.00	18	0.00	0.01	0.01
PA	summer	sdwn_short	0	0	0	0.00	17	0.00	0.00	0.01
PA	spring	sdwn_short	0	0	0	0.00	14	0.00	0.00	0.00
PA	winter	sdwn_short	0	0	0	0.00	2	0.00	0.00	0.00
PA	autumn	TD_long	0	0	0	0.00	18	0.00	0.25	0.25
PA	summer	TD_long	0	0	0	0.00	17	0.16	0.18	0.33
PA	spring	TD_long	0	0	0	0.00	14	0.04	0.18	0.22
PA	winter	TD_long	0	0	0	0.00	2	0.00	0.03	0.03
PA	summer	TD_noisy	0	0	0	0.00	17	0.01	0.01	0.02
PA	autumn	TD_noisy	0	0	0	0.00	17	0.00	0.01	0.01
PA	spring	TD_noisy	0	0	0	0.00	11	0.00	0.01	0.01
PA	winter	TD_noisy	0	0	0	0.00	2	0.00	0.00	0.00

B2 Simulation results MSC

project	season	scenario	Avg delay tolerance	avg test duration	commits	gray servers	green servers	re-use	gray minutes	green minutes
MSC	autumn	base	210	22	56	1	44	11	22	1210
MSC	spring	base	204	22	51	27	11	13	836	286
MSC	winter	base	176	22	42	0	35	7	0	924
MSC	summer	base	230	22	35	4	23	8	110	660
MSC	autumn	base-se4	210	22	56	1	44	11	22	1210
MSC	spring	base-se4	204	22	51	29	9	13	880	242
MSC	winter	base-se4	176	22	42	8	27	7	198	726
MSC	summer	base-se4	230	22	35	5	22	8	132	638
MSC	autumn	DT_long	251	22	56	0	32	24	0	1232
MSC	spring	DT_long	247	22	51	22	10	19	814	308
MSC	winter	DT_long	219	22	42	0	28	14	0	924
MSC	summer	DT_long	269	22	35	2	21	12	88	682
MSC	autumn	DT_short	36	22	56	1	45	10	22	1210
MSC	spring	DT_short	35	22	51	28	11	12	836	286
MSC	winter	DT_short	30	22	42	0	35	7	0	924
MSC	summer	DT_short	39	22	35	6	23	6	176	594
MSC	autumn	IRE_low	210	22	56	1	44	11	22	1210
MSC	spring	IRE_low	204	22	51	16	22	13	484	638
MSC	winter	IRE_low	176	22	42	0	35	7	0	924
MSC	summer	IRE_low	230	22	35	4	23	8	110	660
MSC	autumn	IRE_one	210	22	56	0	45	11	0	1232
MSC	spring	IRE_one	204	22	51	0	37	14	0	1122
MSC	winter	IRE_one	176	22	42	0	35	7	0	924

project	season	scenario	Avg delay tolerance	avg test duration	commits	gray servers	green servers	re-use	gray minutes	green minutes
MSC	summer	IRE_one	230	22	35	0	27	8	0	770
MSC	autumn	sdwn_long	210	22	56	1	44	11	22	1210
MSC	spring	sdwn_long	204	22	51	27	10	14	836	286
MSC	winter	sdwn_long	176	22	42	0	35	7	0	924
MSC	summer	sdwn_long	230	22	35	4	23	8	110	660
MSC	autumn	sdwn_short	210	22	56	1	46	9	22	1210
MSC	spring	sdwn_short	204	22	51	33	11	7	836	286
MSC	winter	sdwn_short	176	22	42	0	38	4	0	924
MSC	summer	sdwn_short	230	22	35	5	24	6	110	660
MSC	autumn	TD_long	210	45	56	1	55	0	45	2475
MSC	spring	TD_long	204	45	51	38	13	0	1710	585
MSC	winter	TD_long	176	45	42	0	42	0	0	1890
MSC	summer	TD_long	230	45	35	5	30	0	225	1350
MSC	autumn	TD_noisy	210	24	56	1	37	18	49	1310
MSC	spring	TD_noisy	204	23	51	28	9	14	883	302
MSC	winter	TD_noisy	176	23	42	0	35	7	0	970
MSC	summer	TD_noisy	230	22	35	4	23	8	108	651
MSC	autumn	TD_short	210	6	56	1	25	30	18	318
MSC	spring	TD_short	204	6	51	17	8	26	228	78
MSC	winter	TD_short	176	6	42	0	22	20	0	252
MSC	summer	TD_short	230	6	35	2	14	19	30	180

Continued

project	season	scenario	green waste	gray waste	se1	se2	se3	se4	fi	dk1	dk2
MSC	autumn	base	1204	38	2	0	0	42	0	0	0
MSC	spring	base	190	584	2	2	0	5	0	2	0
MSC	winter	base	1014	0	0	0	0	18	0	17	0
MSC	summer	base	504	130	0	0	0	20	0	3	0
MSC	autumn	base-se4	1204	38	0	0	0	44	0	0	0
MSC	spring	base-se4	152	622	0	0	0	9	0	0	0
MSC	winter	base-se4	770	244	0	0	0	27	0	0	0
MSC	summer	base-se4	466	168	0	0	0	22	0	0	0
MSC	autumn	DT_long	484	0	1	0	0	31	0	0	0
MSC	spring	DT_long	146	328	2	2	0	4	0	2	0
MSC	winter	DT_long	594	0	0	0	0	14	0	14	0
MSC	summer	DT_long	384	32	0	0	0	20	0	1	0
MSC	autumn	DT_short	1204	38	6	0	0	39	0	0	0

project	season	scenario	green waste	gray waste	se1	se2	se3	se4	fi	dk1	dk2
MSC	spring	DT_short	190	584	3	1	0	5	0	2	0
MSC	winter	DT_short	1014	0	0	0	0	18	0	17	0
MSC	summer	DT_short	564	130	0	0	0	21	0	2	0
MSC	autumn	IRE_low	1204	38	2	0	0	42	0	0	0
MSC	spring	IRE_low	498	276	13	2	0	5	0	1	0
MSC	winter	IRE_low	1014	0	0	0	0	18	0	17	0
MSC	summer	IRE_low	504	130	0	0	1	18	0	2	0
MSC	autumn	IRE_one	1242	0	0	0	0	18	0	0	0
MSC	spring	IRE_one	774	0	6	0	0	5	0	1	0
MSC	winter	IRE_one	1014	0	0	0	1	3	0	14	0
MSC	summer	IRE_one	634	0	0	0	1	10	0	2	0
MSC	autumn	sdwn_long	1242	38	2	0	0	42	0	0	0
MSC	spring	sdwn_long	206	638	2	2	0	4	0	2	0
MSC	winter	sdwn_long	1052	0	0	0	0	18	0	17	0
MSC	summer	sdwn_long	542	130	0	0	0	20	0	3	0
MSC	autumn	sdwn_short	0	0	2	0	0	44	0	0	0
MSC	spring	sdwn_short	0	0	2	2	0	5	0	2	0
MSC	winter	sdwn_short	0	0	0	0	0	20	0	18	0
MSC	summer	sdwn_short	0	0	0	0	0	21	0	3	0
MSC	autumn	TD_long	555	15	2	0	0	53	0	0	0
MSC	spring	TD_long	75	405	3	3	0	5	0	2	0
MSC	winter	TD_long	465	0	0	0	0	23	0	19	0
MSC	summer	TD_long	270	75	0	0	0	27	0	3	0
MSC	autumn	TD_noisy	818	11	2	0	0	35	0	0	0
MSC	spring	TD_noisy	184	706	2	1	0	4	0	2	0
MSC	winter	TD_noisy	858	0	0	0	0	17	0	18	0
MSC	summer	TD_noisy	510	132	0	0	0	20	0	3	0
MSC	autumn	TD_short	978	42	1	0	0	24	0	0	0
MSC	spring	TD_short	210	540	1	1	0	4	0	2	0
MSC	winter	TD_short	810	0	0	0	0	10	0	12	0
MSC	summer	TD_short	432	90	0	0	0	12	0	2	0

Continued

project	season	scenario	no2	no3	no4	no5	ee	waste factor	Ops cost	Gray emissions	Green emissions	Emissions total
MSC	autumn	base	0	0	0	0	0	1.00	45	0.05	0.73	0.79
MSC	spring	base	0	0	0	0	0	0.66	38	1.24	0.14	1.38
MSC	winter	base	0	0	0	0	0	1.10	35	0.00	0.59	0.59
MSC	summer	Base	0	0	0	0	0	0.76	27	0.21	0.35	0.56

project	season	scenario	no2	no3	no4	no5	ee	waste factor	Ops cost	Gray emissions	Green emissions	Emissions total
MSC	autumn	base-se4	0	0	0	0	0	1.00	45	0.05	0.73	0.79
MSC	spring	base-se4	0	0	0	0	0	0.63	38	1.31	0.12	1.43
MSC	winter	base-se4	0	0	0	0	0	1.06	35	0.39	0.46	0.84
MSC	summer	base-se4	0	0	0	0	0	0.73	27	0.26	0.34	0.60
MSC	autumn	DT_long	0	0	0	0	0	0.39	32	0.00	0.52	0.52
MSC	spring	DT_long	0	0	0	0	0	0.47	32	1.00	0.14	1.13
MSC	winter	DT_long	0	0	0	0	0	0.64	28	0.00	0.46	0.46
MSC	summer	DT_long	0	0	0	0	0	0.56	23	0.10	0.32	0.43
MSC	autumn	DT_short	0	0	0	0	0	1.00	46	0.05	0.73	0.79
MSC	spring	DT_short	0	0	0	0	0	0.66	39	1.24	0.14	1.38
MSC	winter	DT_short	0	0	0	0	0	1.10	35	0.00	0.59	0.59
MSC	summer	DT_short	0	0	0	0	0	0.95	29	0.27	0.35	0.62
MSC	autumn	IRE_low	0	0	0	0	0	1.00	45	0.05	0.73	0.79
MSC	spring	IRE_low	0	1	0	0	0	0.78	38	0.66	0.35	1.01
MSC	winter	IRE_low	0	0	0	0	0	1.10	35	0.00	0.59	0.59
MSC	summer	IRE_low	2	0	0	0	0	0.76	27	0.21	0.35	0.56
MSC	autumn	IRE_one	19	0	8	0	0	1.01	45	0.00	0.75	0.75
MSC	spring	IRE_one	11	14	0	0	0	0.69	37	0.00	0.58	0.58
MSC	winter	IRE_one	17	0	0	0	0	1.10	35	0.00	0.59	0.59
MSC	summer	IRE_one	14	0	0	0	0	0.82	27	0.00	0.43	0.43
MSC	autumn	sdwn_long	0	0	0	0	0	1.03	45	0.05	0.75	0.80
MSC	spring	sdwn_long	0	0	0	0	0	0.72	37	1.28	0.15	1.43
MSC	winter	sdwn_long	0	0	0	0	0	1.14	35	0.00	0.60	0.60
MSC	summer	sdwn_long	0	0	0	0	0	0.82	27	0.21	0.37	0.58
MSC	autumn	sdwn_short	0	0	0	0	0	0.00	47	0.02	0.37	0.39
MSC	spring	sdwn_short	0	0	0	0	0	0.00	44	0.73	0.09	0.82
MSC	winter	sdwn_short	0	0	0	0	0	0.00	38	0.00	0.28	0.28
MSC	summer	sdwn_short	0	0	0	0	0	0.00	29	0.10	0.20	0.30
MSC	autumn	TD_long	0	0	0	0	0	0.22	56	0.05	0.92	0.97
MSC	spring	TD_long	0	0	0	0	0	0.13	51	1.84	0.20	2.04
MSC	winter	TD_long	0	0	0	0	0	0.25	42	0.00	0.72	0.72
MSC	summer	TD_long	0	0	0	0	0	0.20	35	0.26	0.49	0.75
MSC	autumn	TD_noisy	0	0	0	0	0	0.62	38	0.05	0.65	0.70
MSC	spring	TD_noisy	0	0	0	0	0	0.61	37	1.38	0.15	1.53
MSC	winter	TD_noisy	0	0	0	0	0	0.88	35	0.00	0.56	0.56
MSC	summer	TD_noisy	0	0	0	0	0	0.78	27	0.21	0.35	0.56
MSC	autumn	TD_short	0	0	0	0	0	3.08	26	0.05	0.39	0.45
MSC	spring	TD_short	0	0	0	0	0	2.69	25	0.67	0.09	0.76
MSC	winter	TD_short	0	0	0	0	0	3.21	22	0.00	0.32	0.32
MSC	summer	TD_short	0	0	0	0	0	2.40	16	0.10	0.19	0.29

B3 Simulation results VSC

project	season	scenario	Avg delay tolerance	avg test duration	commits	gray servers	green servers	re-use	gray minutes	green minutes
VSC	winter	base	183	16	538	1	260	277	32	8576
VSC	autumn	base	178	16	476	0	253	223	0	7616
VSC	spring	base	165	16	414	95	140	179	2592	4032
VSC	summer	base	153	16	185	17	93	75	368	2592
VSC	winter	base-se4	183	16	538	34	227	277	1184	7424
VSC	autumn	base-se4	178	16	476	5	248	223	112	7504
VSC	spring	base-se4	165	16	414	157	77	180	4368	2256
VSC	summer	base-se4	153	16	185	24	88	73	512	2448
VSC	winter	DT_long	227	16	538	0	196	342	0	8608
VSC	autumn	DT_long	222	16	477	0	178	299	0	7632
VSC	spring	DT_long	209	16	414	61	103	250	2416	4208
VSC	summer	DT_long	199	16	185	11	75	99	336	2624
VSC	winter	DT_short	31	16	538	1	290	247	16	8592
VSC	autumn	DT_short	30	16	477	0	268	209	0	7632
VSC	spring	DT_short	28	16	414	99	155	160	2592	4032
VSC	summer	DT_short	26	16	185	25	100	60	528	2432
VSC	winter	IRE_low	183	16	538	1	260	277	32	8576
VSC	autumn	IRE_low	178	16	476	0	253	223	0	7616
VSC	spring	IRE_low	165	16	414	75	159	180	1968	4656
VSC	summer	IRE_low	153	16	185	12	99	74	288	2672
VSC	winter	IRE_one	183	16	538	0	264	274	0	8608
VSC	autumn	IRE_one	178	16	476	0	255	221	0	7616
VSC	spring	IRE_one	165	16	414	0	231	183	0	6624
VSC	summer	IRE_one	153	16	185	0	111	74	0	2960
VSC	winter	sdwn_long	183	16	538	1	259	278	32	8576
VSC	autumn	sdwn_long	178	16	476	0	252	224	0	7616
VSC	spring	sdwn_long	165	16	414	94	142	178	2592	4032
VSC	summer	sdwn_long	153	16	185	17	90	78	368	2592
VSC	autumn	sdwn_short	178	16	476	0	332	144	0	7616
VSC	winter	sdwn_short	183	16	538	1	318	219	32	8576
VSC	spring	sdwn_short	165	16	414	120	169	125	2592	4032
VSC	summer	sdwn_short	153	16	185	23	106	56	416	2544
VSC	winter	TD_long	183	45	538	1	537	0	45	24165
VSC	autumn	TD_long	178	45	477	0	477	0	0	21465
VSC	spring	TD_long	165	45	414	160	254	0	7200	11430
VSC	summer	TD_long	153	45	185	27	158	0	1215	7110
VSC	winter	TD_noisy	183	15	538	1	250	287	31	8305
VSC	autumn	TD_noisy	178	16	477	0	236	241	0	7634

project	season	scenario	Avg delay tolerance	avg test duration	commits	gray servers	green servers	re-use	gray minutes	green minutes
VSC	spring	TD_noisy	165	15	414	86	137	191	2489	3871
VSC	summer	TD_noisy	153	15	185	16	92	77	356	2391
VSC	winter	TD_short	183	6	538	1	137	400	18	3210
VSC	spring	TD_short	165	6	414	59	75	280	990	1494
VSC	autumn	TD_short	178	6	477	0	130	347	0	2862
VSC	summer	TD_short	153	6	185	13	65	107	150	960

Continued

project	season	scenario	green waste	gray waste	se1	se2	se3	se4	fi	dk1	dk2	no2	no3
VSC	winter	base	6040	28	0	0	0	177	0	77	0	0	6
VSC	autumn	base	6776	0	14	0	0	226	0	9	4	0	0
VSC	spring	base	3616	2824	22	23	0	35	0	9	0	0	51
VSC	summer	base	2360	652	0	2	0	81	0	6	0	3	0
VSC	winter	base-se4	5292	776	0	0	0	227	0	0	0	0	0
VSC	autumn	base-se4	6588	188	0	0	0	248	0	0	0	0	0
VSC	spring	base-se4	1832	4564	0	0	0	77	0	0	0	0	0
VSC	summer	base-se4	2316	740	0	0	0	88	0	0	0	0	0
VSC	winter	DT_long	2264	0	0	0	0	133	0	61	0	0	2
VSC	autumn	DT_long	2196	0	12	0	0	158	0	6	2	0	0
VSC	spring	DT_long	1388	992	16	21	0	20	0	6	0	0	40
VSC	summer	DT_long	1428	324	0	1	0	67	0	4	0	3	0
VSC	winter	DT_short	7708	44	0	0	0	190	0	91	0	0	9
VSC	autumn	DT_short	7480	0	22	0	0	231	0	11	4	0	0
VSC	spring	DT_short	4500	2976	24	18	0	40	0	12	0	0	61
VSC	summer	DT_short	3020	756	0	2	0	84	0	11	0	2	0
VSC	winter	IRE_low	6040	28	0	0	0	177	0	77	0	0	6
VSC	autumn	IRE_low	6776	0	33	4	0	207	0	9	0	0	0
VSC	spring	IRE_low	4180	2260	56	18	0	18	0	5	0	0	62
VSC	summer	IRE_low	2624	432	0	0	7	74	0	2	0	15	0
VSC	winter	IRE_one	6112	0	1	0	18	104	0	56	0	84	0
VSC	autumn	IRE_one	6776	0	2	0	1	150	0	9	0	57	2
VSC	spring	IRE_one	6440	0	12	0	28	26	0	1	0	95	61
VSC	summer	IRE_one	3056	0	0	0	6	62	0	2	0	41	0
VSC	winter	sdwn_long	6288	28	0	0	0	176	0	77	0	0	6
VSC	autumn	sdwn_long	6932	0	14	0	0	225	0	9	4	0	0
VSC	spring	sdwn_long	3832	2864	22	23	0	37	0	9	0	0	51
VSC	summer	sdwn_long	2500	652	0	1	0	79	0	6	0	3	0

project	season	scenario	green waste	gray waste	se1	se2	se3	se4	fi	dk1	dk2	no2	no3
VSC	autumn	sdwn_short	0	0	21	0	0	296	0	11	4	0	0
VSC	winter	sdwn_short	0	0	0	0	0	209	0	101	0	0	8
VSC	spring	sdwn_short	0	0	26	25	0	41	0	13	0	0	64
VSC	summer	sdwn_short	0	0	0	2	0	94	0	6	0	3	0
VSC	winter	TD_long	5340	15	0	0	0	359	0	169	0	0	9
VSC	autumn	TD_long	4950	0	24	0	0	433	0	16	4	0	0
VSC	spring	TD_long	2685	1935	36	54	0	58	0	19	0	0	87
VSC	summer	TD_long	1620	405	0	2	0	139	0	9	0	7	0
VSC	winter	TD_noisy	5903	29	0	0	0	163	0	80	0	0	7
VSC	autumn	TD_noisy	5891	0	15	0	0	209	0	9	3	0	0
VSC	spring	TD_noisy	3761	2368	21	21	0	34	0	12	0	0	49
VSC	summer	TD_noisy	2568	604	0	1	0	80	0	5	0	5	0
VSC	winter	TD_short	4134	42	0	0	0	94	0	38	0	0	5
VSC	spring	TD_short	2232	1842	11	11	0	21	0	6	0	0	26
VSC	autumn	TD_short	4596	0	8	0	0	115	0	5	2	0	0
VSC	summer	TD_short	2346	630	0	1	0	55	0	5	0	3	0

Continued

project	season	scenario	no4	no5	ee	waste factor	Ops cost	Gray emissions	Green emissions	Emissions total
VSC	winter	base	0	0	0	0.70	261	0.05	4.45	4.50
VSC	autumn	base	0	0	0	0.89	253	0.00	4.38	4.38
VSC	spring	base	0	0	0	0.90	235	4.72	2.33	7.05
VSC	summer	base	0	0	1	0.91	110	0.89	1.51	2.40
VSC	winter	base-se4	0	0	0	0.71	261	1.71	3.87	5.58
VSC	autumn	base-se4	0	0	0	0.88	253	0.26	4.29	4.55
VSC	spring	base-se4	0	0	0	0.81	234	7.78	1.24	9.03
VSC	summer	base-se4	0	0	0	0.95	112	1.09	1.45	2.54
VSC	winter	DT_long	0	0	0	0.26	196	0.00	3.31	3.31
VSC	autumn	DT_long	0	0	0	0.29	178	0.00	2.99	2.99
VSC	spring	DT_long	0	0	0	0.33	164	2.97	1.70	4.67
VSC	summer	DT_long	0	0	0	0.54	86	0.58	1.23	1.81
VSC	winter	DT_short	0	0	0	0.90	291	0.05	4.96	5.01
VSC	autumn	DT_short	0	0	0	0.98	268	0.00	4.60	4.60
VSC	spring	DT_short	0	0	0	1.12	254	4.85	2.60	7.45
VSC	summer	DT_short	0	0	1	1.24	125	1.12	1.66	2.78
VSC	winter	IRE_low	0	0	0	0.70	261	0.05	4.45	4.50
VSC	autumn	IRE_low	0	0	0	0.89	253	0.00	4.38	4.38
VSC	spring	IRE_low	0	0	0	0.90	234	3.68	2.69	6.37

project	season	scenario	no4	no5	ee	waste factor	Ops cost	Gray emissions	Green emissions	Emissions total
VSC	summer	IRE_low	0	0	1	0.98	111	0.63	1.61	2.24
VSC	winter	IRE_one	1	0	0	0.71	264	0.00	4.48	4.48
VSC	autumn	IRE_one	34	0	0	0.89	255	0.00	4.38	4.38
VSC	spring	IRE_one	8	0	0	0.97	231	0.00	3.98	3.98
VSC	summer	IRE_one	0	0	0	1.03	111	0.00	1.83	1.83
VSC	winter	sdwn_long	0	0	0	0.73	260	0.05	4.52	4.58
VSC	autumn	sdwn_long	0	0	0	0.91	252	0.00	4.43	4.43
VSC	spring	sdwn_long	0	0	0	0.95	236	4.75	2.39	7.15
VSC	summer	sdwn_long	0	0	1	0.96	107	0.89	1.55	2.44
VSC	autumn	sdwn_short	0	0	0	0.00	332	0.00	2.32	2.32
VSC	winter	sdwn_short	0	0	0	0.00	319	0.03	2.61	2.64
VSC	spring	sdwn_short	0	0	0	0.00	289	2.26	1.23	3.49
VSC	summer	sdwn_short	0	0	1	0.00	129	0.36	0.77	1.14
VSC	winter	TD_long	0	0	0	0.22	538	0.05	8.98	9.03
VSC	autumn	TD_long	0	0	0	0.23	477	0.00	8.04	8.04
VSC	spring	TD_long	0	0	0	0.23	414	7.96	4.30	12.26
VSC	summer	TD_long	0	0	1	0.23	185	1.41	2.66	4.07
VSC	winter	TD_noisy	0	0	0	0.71	251	0.05	4.33	4.38
VSC	autumn	TD_noisy	0	0	0	0.77	236	0.00	4.12	4.12
VSC	spring	TD_noisy	0	0	0	0.97	223	4.23	2.32	6.55
VSC	summer	TD_noisy	0	0	1	1.07	108	0.84	1.51	2.35
VSC	winter	TD_short	0	0	0	1.29	138	0.05	2.24	2.29
VSC	spring	TD_short	0	0	0	1.49	134	2.47	1.13	3.60
VSC	autumn	TD_short	0	0	0	1.61	130	0.00	2.27	2.27
VSC	summer	TD_short	0	0	1	2.44	78	0.68	1.01	1.69

