Theory of Hybrid Systems Informatik 2

RWTH AACHEN UNIVERSITY

Diese Arbeit wurde vorgelegt am
Lehr- und Forschungsgebiet Theorie der hybriden Systeme

Resource Optimized Scheduling of Scientific Simulations on CPU and GPU Computing Platforms

Resourcenoptimiertes Scheduling von wissenschaftlichen Simulationen auf CPU- und GPU-Plattformen

Masterarbeit
Informatik

Dezember 2020

Vorgelegt von Presented by	Serjoscha Bender Hermann-Behn-Weg 1 20146 Hamburg Matrikelnummer: 344493 serjoscha.bender@rwth-aachen.de
Erstprüfer First examiner	Prof. Dr. rer. nat. Erika Ábrahám Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University
Zweitprüfer Second examiner	Prof. Dr. rer. nat. Thomas Noll Lehr- und Forschungsgebiet: Software Modellierung und Verifikation RWTH Aachen University
Externer Betreuer External supervisor	Dr. rer. nat. Pascal Richter Steinbuch Centre for Computing Karlsruhe Institute of Technology

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen meiner Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Dasselbe gilt sinngemäß für Tabellen und Abbildungen. Diese Arbeit hat in dieser oder einer ähnlichen Form noch nicht im Rahmen einer anderen Prüfung vorgelegen.

Hamburg, im Dezember 2020

SERJOSCHA BENDER

Contents

1 Introduction

Since their early days, computers have been used in science. The scientific community develops software to solve a wide variety of problems, perform data analysis or run simulations based on mathematical models to verify assumptions or make completely new findings. Usage of theses scientific software tools can be cumbersome, however.

Operation of the software might be not intuitive and only understandable to experienced users, the execution can require specialized or dedicated hardware and might take a considerable amount of time to finish, in which the computer cannot be used unrestrictedly. Because of such constraints the distribution of scientific software to a broader audience as well as user friendliness and accessibility often remains a difficulty.

A solution could be to adopt a model that has had increasing significance in commercial software distribution over the last years: Software-as-a-Service (SaaS). A central provider takes over the hardware allocation and the execution of programs, which can be triggered and later evaluated by the user via a web interface.

A Case Study Consumption of energy in general and electricity in particular is on the rise all around the world. From 1990 to 2017 the global electricity consumption has more than doubled from 10.9 Pwh to 24.7 Pwh [5]. To meet this growing demand while reducing the emission of greenhouse gases, so called renewable sources such as sunlight or wind are used for energy production. Possible types of renewable energy power plants are offshore wind farms or concentrated solar thermal facilities.

While it seems that there is an unlimited supply of these energy sources, the aim of the plant operators is to maximize energy production while minimizing operating costs and using only a limited and predefined area to build the facilities. This describes an optimization problem based on a large amount of parameters. Scientists are creating models that allow to simulate the power plants and find optimal characteristics.

Two examples are the SunFlower[21] and WindFlower[17] projects. They offer software tools that can perform several computations of varying complexity.

To overcome the aforementioned obstacles of scientific software, a web application for Wind- and SunFlower has been developed during several university labs. It provides a platform independent user interface to control the input, trigger computations and view the corresponding outputs. It also serves as a platform to share projects and results and use the software collaboratively. The computations themselves, using the SunFlower and WindFlower software tools are managed by the web application and run on allocated servers. The user thus does not need to provide and manage any resources, which makes it convenient.

It becomes clear though, that for the providers of such services a new challenge arises. They are dealing with a variety of computations that are requested by different users. Runtime for these computations is distributed over a wide range, as it depends not only on the type, but also on the input parameters. The providers face expectations of their users regarding the time from triggering to completion of a commissioned computational job. This expectation usually is dependent on the adressed runtime

considerations as well as the customer status, i.e. users who pay a fee expect a higher quality of service (= lower completion time) than users who are on a free plan.

The goal of this study is to develop a strategy which allows providers of the Windor SunFlower web application to offer a reasonably good quality of service to their users while minimizing the costs of running the application. Therefore, mainly two techniques are being used: *Scheduling* of computational jobs and *Scaling* of cloud computing resources.

1.1 Outline

In Section 2 the theoretical concepts and technologies that are used in this thesis are described and an overview of works with similar goals is given. Section 3 covers the software tools that perform the actual simulations which are to be scheduled. Then, in Section 4 the web application offering a user interface and computing platform for the simulations is presented. In Section 5 the considerations for developing the scheduling and scaling infrastructure and strategies are explained. Following, in Section 6 different scheduling scenarios are created and the performance of strategies is evaluated. Section 7 contains the conclusion of this work and gives an outlook on possible future research.

2 State of the Art

In this section we will give a brief overview over the scheduling problem and cloud computing technologies. Also, we present scientific works that dealt with similar scheduling scenarios as this thesis.

2.1 Scheduling

Scheduling problems refer to the chronological allocation of tasks to resources (or vice versa), the creation of a schedule. Usually, there exists a set of constraints and the goal is to optimize the schedule w.r.t. a certain set of parameters. This type of problem arises in private life, e.g. the creation of a timetable in school, in industry, where machines and workers have to be assigned time slots to work on certain orders, as well as in computing environments.

One prominent example in a computing environment is the process scheduling in multi-tasking operating systems. The CPU or, to be precise, one CPU core can only handle one process at a time. To achieve the appearance of parallel process execution, a schedule is created that makes the CPU work on slices of the processes in turn. By assigning different priorities to processes, the ratio of CPU time per process is shifted so that more important processes get handled faster in comparison.

On a higher level there is computational job scheduling. Jobs may be program executions, e.g. for scientific calculations, that are to be run on single computers or on a cluster of machines. The job parameters may vary as well as the resources offered by the machines. Scheduling is needed to make efficient use of the provided resources and

offering satisfying execution times to the job creators. [15] This kind of scheduling is an integral part of this thesis.

Comprehensive overviews over scheduling problems, its variations and solutions can be found in the books by Brucker [9] and Pinedo [19]. Below, based on this literature, we provide definitions for further usage.

Scheduling problems in general can be formalized as follows: There is

- A set of m **machines** $M_j (j = 1, ..., m)$
- A set of n **jobs** $J_i, i = 1, ..., n$

A schedule is created that allocates time slots on the machines for certain jobs, as shown in 1. Both, machines and jobs are described by further parameters.

Jobs

- p_{ij} - Processing Time: The processing time that is needed to complete job J_i on machine M_j. There are several units possible for this value, e.g. seconds or CPU cycles.
- w_i - Weight: Acts as a priority for Job J_i.
- r_i - Release Date: At this point in time the job becomes available. Processing on J_i can start at r_i at the earliest.
- d_i - Due Date: The point in time at which the job is expected to be finished. This deadline can be a "hard" or "soft" one. In the first case, the finishing until d_i is mandatory, in the latter a job still can finish after d_i, but this is penalized in the scheduling process.
- *pmtn* - Preemption: If preemption is allowed for job J_i, processing of J_i can be paused and continued later and/or on another machine. If preemption is not allowed, J_i, once started, is processed until completion, accordingly.

Machines The characteristics of machines can take many different forms, of which some are:

- *1* - Single Machine: Only one machine processes jobs, one after another.
- *Pm* - Identical machines in parallel: Any job can be processed on each of the m Machines with the same speed.
- *Qm* - Parallel machines with different speeds: m parallel machines with differing speeds v_j are available. Job J_i takes time $\frac{p_i}{v_j}$ to process on machine M_j.
- *Rm* - Unrelated parallel machines: The speed of one machine is also dependent on the job that is to be processed and is denoted by v_{ij}. Job J_i takes time $\frac{p_i}{v_{ij}}$ to process on machine M_j.

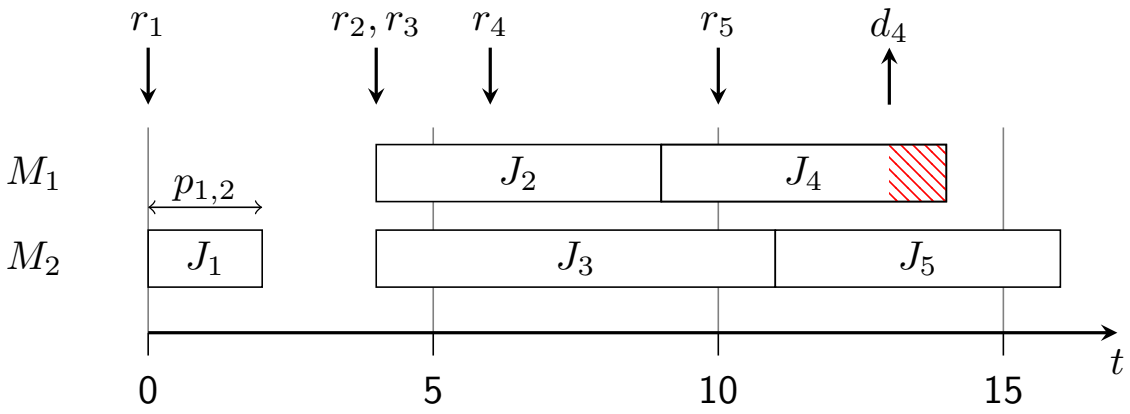


Figure 1: Visualisation of an exemplary schedule. It shows a P_2 machine setup with no preemption and it can be seen that Job J_4 fails to meet its deadline d_4.

Job Cost When assessing the quality of a schedule, a cost function $c_i(t)$ is used which calculates the cost of completing job J_i at time t. It usually depends on d_i and w_i.

The completion time C_i is the point in time at which a scheduled job J_i is completed and leaving the machine. Based on C_i the lateness L_i can be computed, which is a measure of the job's urgency.

$$L_i = C_i - d_i$$

The tardiness T_i works in a similar way, but cannot get negative. That means, it does not differentiate how early a job finishes, as long it does not violate its deadline.

$$T_i = \max\{C_i - d_i, 0\} = \max\{L_i, 0\}$$

Lateness and tardiness both can be used as costs.

Total Cost To calculate the total cost of a schedule there are essentially two possibilities: Using the *max* function, representing the *bottleneck* of the schedule

$$c_{total} = \max\{c_i(C_i), i = 1, ..., n\}$$

or the sum of costs

$$c_{total} = \sum_{i=1}^{n} c_i(C_i)$$

2.1.1 Objective

In general the goal of scheduling can be described as reducing the total cost of the schedule to the least possible value that is reachable under the prevailing constraints. A schedule with cost c_o is optimal, if every possible schedule for the same scenario has a cost c with $c \geq c_o$.

For many scheduling problems optimal solutions can be found by using well-studied techniques, such as linear programming [22] or dynamic programming [8]. It has been show though, that scheduling for multiprocessors in most cases is NP-complete and only specific multiprocessor scheduling problems have a solution in polynomial time [14].

However, often a "good" schedule is sufficient and it might not be necessary to find the optimal schedule or the cost for optimizing simply is too high. There are many "fixed" scheduling strategies that can provide adequate results, depending on the scenarios. Some of which are:

- *First in, First out (FIFO):* Jobs get scheduled by ascending release dates
- *Priority Scheduling:* (Released) jobs get scheduled by descending priorities
- *Round-Robin:* The scheduler cycles through available jobs, of which each gets assigned a certain time slot (which can depend on the priority)

2.1.2 Online Scheduling

While some scheduling problems can be solved before running the schedule (i.e. at compile time), there are many scenarios in which the schedule has to be created at runtime. This is called *online scheduling*. In this case the jobs, or certain properties of jobs, only get known to the scheduler over time. While it is still possible to create optimal schedules for single processor environments based on this limited information, there cannot be an optimal scheduling algorithm for all input instances in multiprocessor environments without full a-priori knowledge of computation times, release and due dates for all jobs [10].

To evaluate the performance of online scheduling algorithms, competitive analysis is used [20]. It compares the worst-case relative error of an algorithm with the optimal the solution. Let $c_{total}(A, I)$ be the total objective cost of using algorithm A on input instance I and OPT the optimal scheduling solution for this problem. Algorithm A is c-competitive if $c_{total}(A, I) \leq c \cdot c_{total}(OPT, I) + b$ for any input instance I.

2.2 Container-Based Virtualization

Traditionally, when talking about virtualization in computing environments, this meant hypervisor-based virtualization. The hypervisor is a software tool that abstracts the hardware of a physical computer (the *host*) and allows running so called *virtual machines* (*guests* of the host) on it. These virtual machines can run an arbitrary operating system and perform like a dedicated computer. The abstraction makes it easy to share physical resources between different users and migrate systems. That way the operation of computation resources (e.g. server farms) can be made more efficient, while maintaining isolation between systems in order to ensure security [13].

In contrast to hypervisor-based virtualization, in container-based virtualization there is no emulation of hardware, but the software is directly run on the operating system.

The operating system therefore has capabilities to isolate processes, preventing access to other containers and the host itself. The result is a reduction in overhead compared to hypervisor environments regarding the size of system images and the time to boot a system up, as there is no operating system included.

Also, containers lend itself to the use of continuous development techniques, such as continuous integration (CI) or continuous deployment (CD). A modular system, consisting of different applications that each are build as a container image can be set up and continuously updated by automatic build systems easily, while avoiding dependency conflicts.

Docker The breakthrough of container technology came with the emergence of Docker [1] from 2013 on. Docker is a software that abstracts the technical configuration and tasks needed for running containers and offers users a simple command-line interface with which they easily can build, run and manage containers.

2.3 Cloud Computing

Cloud computing is a concept to provide computing resources to customers in a rapid and flexible way. The resources, which can be servers, storage, applications, network features and more, are bundled in a 'pool' at the providers premises and customers can request a claim on a certain share of them at any time through a simple interface. To account for this on-demand resource-leasing, billing is based on the actual usage that is metered precisely. E.g. servers are billed per uptime minute/second and network resources per transferred megabyte [11].

2.3.1 Providers

As offering cloud computing services requires running adequately large data centres there are few companies that dominate the market. The biggest cloud service providers (CSPs) are: Amazon with its Amazon Web Services (AWS) holds the largest market share followed by Microsoft Azure and Google Cloud Platform. Theses three make up a 60% share of the cloud computing market as of June 2020 [4].

2.3.2 Scaling

The advantage cloud computing resources hold over long-term-acquired ones is the flexibility to dynamically scale a system to the resource demand at any time. When a customer expects a higher demand of his services during day time in a specific regions, he can request more resources for that timespan which get shut down again for the night while only paying for the actual runtime. Furthermore, CSPs offer functionality to automatically take up the scaling based on a set of metrics, e.g. the CPU usage of a server pool over a certain time period.

2.4 Related Work

The general problem of scheduling has been extensively studied as stated in Section 2.1, but also the more specific task of scheduling scientific computational jobs was topic of several works.

Bittencourt et al. have developed a cost optimized scheduling algorithm for workflows (HCOC) in hybrid cloud environments [7]. Workflows are sets of tasks which are represented by a directed acyclic graph in which the nodes represent the tasks and the relations represent the order in which tasks have to be executed. The algorithm decides when to acquire additional resources from a public cloud. A similar analysis for workflows was done by Malawski et al. [18], highlighting also the case of prioritized tasks.

Azar et al. studied online scheduling of jobs on cloud servers, taking into account the booting time of virtual machines and finding upper bounds for costs and delays for their algorithms [6].

In [16], Jindal et al. described the development of an application that is capable of automatically scaling cloud resources based on a prediction algorithm that generates a forecast of a certain metric, e.g. the CPU load of the running servers.

3 Simulation and Optimization of Power Plants

For several years now the topic of understanding, simulating and optimizing certain renewable-energy power plants has been a research topic at the group of Theory of Hybrid Systems at RWTH Aachen University. Using numerical methods and different optimization techniques a number of models and algorithms have been developed to simulate separate parts of solar power plants or offshore wind farms and optimize parameters in order to maximize the profitability. These developments have been combined in software tools called Windflower and Sunflower, written in C++.

3.1 Sunflower

The subject of all Sunflower calculations are concentrated solar thermal power plants. These are composed of large mirrors that reflect and, through their orientation, concentrate the sunlight onto an absorber, the thermal receiver. In the absorber a certain fluid is being heated up which then can be used to generate electricity, for example by the means of steam turbines.

Sunflower software tools provide a set of calculations of which the following are relevant to this thesis:

- **Simulation:** For given weather data and a certain central receiver solar thermal power plant, the Annual Energy Production (AEP) and a set of economic features is calculated. The simulation is based on a chain of models that cover the whole process: Optical, Thermal, Storage, Electrical and Economic. This is sketched in 2.

- **Optimization:** The goal is to find an arrangement of a given number of heliostats that optimizes a certain economic value, e.g. the total energy that is produced. Therefore it runs a high number of simulations and alters the positions of the heliostats every time, following a certain algorithm. Execution stops when a threshold (time or iterations) is reached or the improvements made are too small.
- **GenerateTopography:** Generates a file with the topographic properties of the site based on satellite data. It is only executed once at the creation of a project and requires only few resources.
- **Generate3DPrinterFile:** Generates a STL-File, containing a 3D model of the plant. This includes the topography, the central receiver and the heliostats.

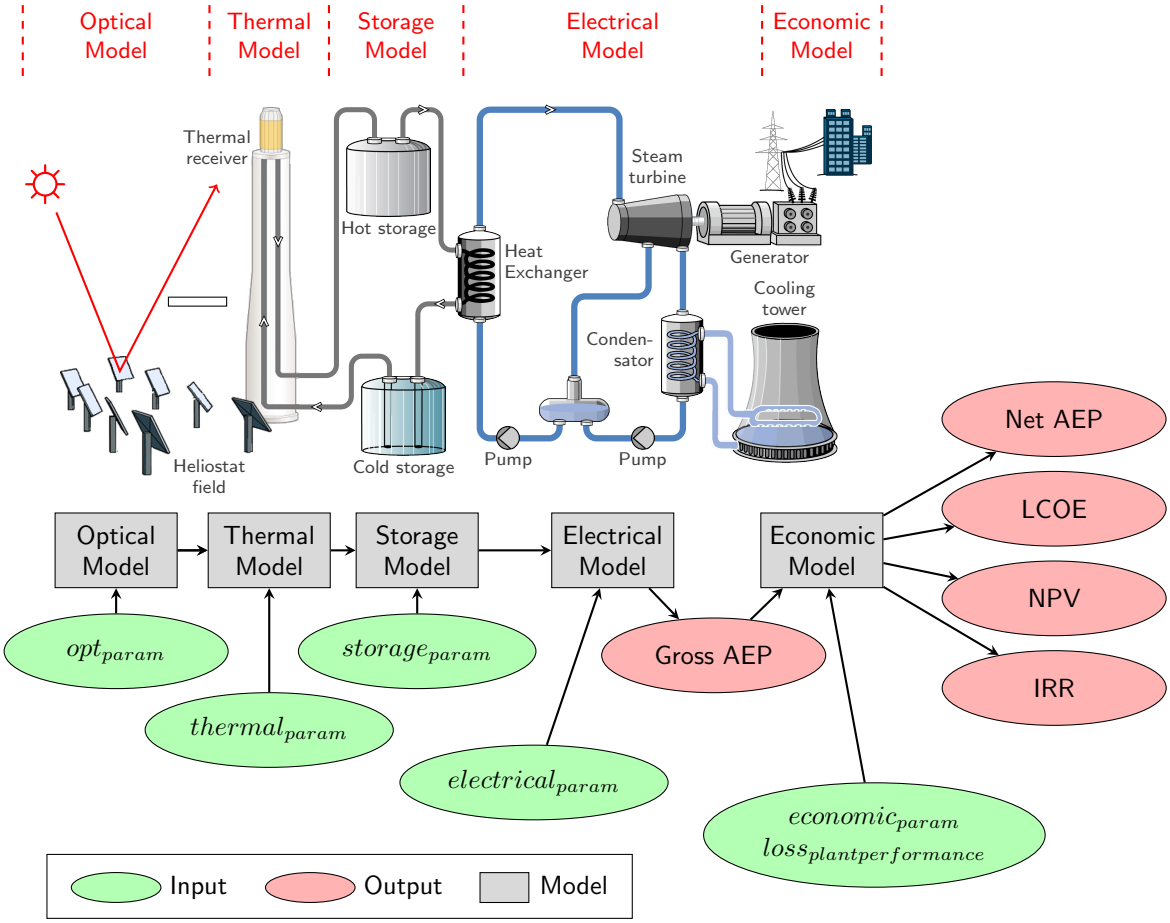


Figure 2: Operation breakdown of a central receiver solar thermal power plant with the corresponding simulation models. The interaction of the different models is shown as well as the ingress points of parameters and outputted values. Partly reprinted and adapted from [21].

3.2 Windflower

Windflower provides similar calculations for offshore wind farms. These are made up from a number of wind turbines arranged in a certain area at sea. At sea, there are no obstructions that slow down the wind and the turbines can be bigger while not needing the height of an equivalent turbine on land. An electrical substation at the wind farm connects the turbines to the electricity grid on land via a subsea cable.

Analogous to Sunflower the software tools of Windflower provide the following calculations:

- **Simulation:** Simulates wind over a full year and subsequently the power generated by the given offshore wind farm. Also a set of economic features is calculated. Similar to Sunflower, a chain of models describes the process: Wind, Wake, Power Generation and Cost. This is depicted in 3.
- **Optimization:** Analogue to the Sunflower optimization, in Windflower the aim is to position a given number of wind turbines inside the site boundaries so that a certain economic parameter is optimized. The optimization process stops when the improvement between iterations falls below a certain threshold or a maximum number of iterations or a time limit is reached.
- **SensitivityAnalysis:** Multiple simulation runs are made with slightly changed input parameters every time.
- **GenerateTopography:** Generates a file with the topographic properties of the site based on subsea terrain data. It is only executed once at the creation of a project and requires only few resources.
- **Generate3DPrinterFile:** Generates a STL-File, containing a 3D model of the wind farm. This can include the ocean ground topography.

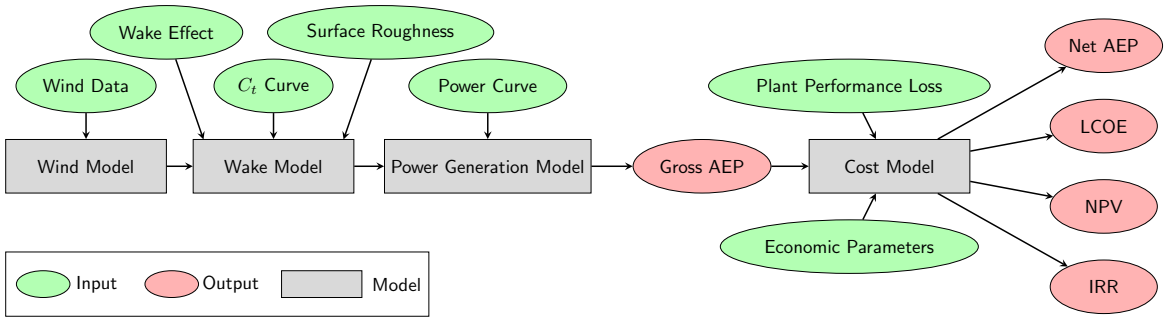


Figure 3: Concept of the individual simulation models and their corresponding inputs and output values for WindFlower offshore wind farm simulations. Reprinted from [12].

3.3 Input Files

Input Data for Sun- and Windflower is provided via several CSV and JSON-Files that contain a large number of parameters. These parameters describe for example the power plant itself, weather, economic assumptions and settings concerning program execution. Depending on the calculation that is carried out, different Input Files may be actually used. Listing 1 shows an example input file, which defines the properties of a single heliostat and is used in the optical model of a Sunflower simulation.

Listing 1: heliostat.json file for PS10 solar thermal power plant

```
{
  "version": "1.0",
  "facet_surface_form": "flat",
  "focal_length": [ 145, 160, 190, 210, 265, 290, 355, 395,
      520, 700 ],
  "heliostat_shape": "rectangular",
  "facet_width": 3.21,
  "facet_height": 1.35,
  "num_horizontal_facets": 4,
  "num_vertical_facets": 7,
  "facet_gap": 40,
  "canting": "on_axis",
  "off_axis_reference_angles": {
    "azimuth": 180,
    "altitude": 77.7
  },
  "pedestal_height": 5.17,
  "cluster_pattern": "single"
}
```

4 Web Application

The web application has been built around Sun- and Windflower and fulfils the following purposes:

- Serving as user interface for the software tools. Before, they were only accessible via the command line, input was provided by editing the corresponding text files and the calculation output also saved as a JSON-file.
- Providing a platform to save projects and calculation outputs
- Offering a platform to collaborate. Projects can be shared with other users of the web application to collaboratively adjust the parameters and share the calculation results

- Providing computation capacity. Calculations are being run on servers, so users do not have to allocate the computing resources.

4.1 Frontend

The frontend is implemented as a cross-platform single-page web application. There exist specific versions for Sun- and Windflower respectively. After logging in, the user can see his projects and the appendant jobs. A project corresponds to a real or fictive power plant. It contains a set of parameters that describe the plant and are valid inputs for the calculations. It can be edited by its creator and all other users that have gotten the necessary permissions by another authorized user, e.g. the creator. For a project the aforementioned computational jobs can be triggered by users in the frontend and the results are displayed there. , where a project is a single set of input data for calculations, corresponding to a real or fictive power plant. Projects are also displayed on a map for easy access.

The pages for setting project parameters are grouped into logical sections and each parameter is manipulated using adequate input methods (e.g. sliders, maps, etc.) as shown in Figure 4a. 3D visualizations help validating and envision the components (wind turbines or heliostats) and the power plants as a whole (see Figure 4b).

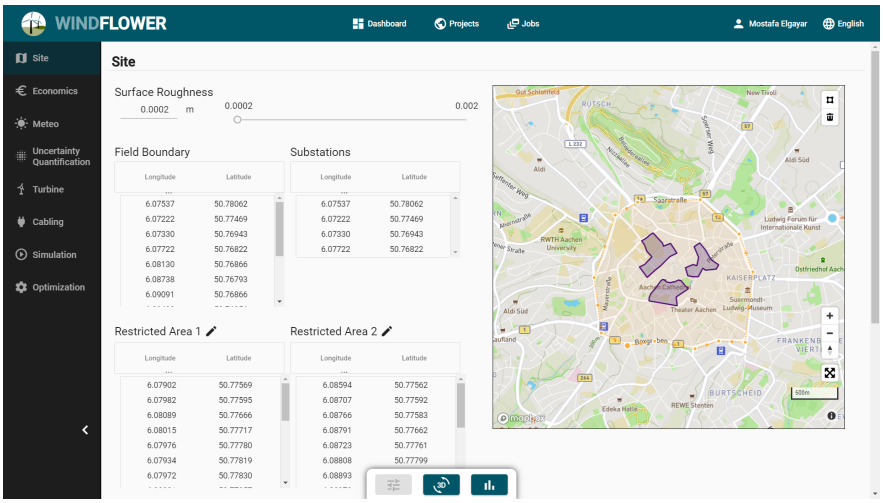
4.2 Backend

The backend is responsible for the data management of the web platform and serves as the connector between the frontend and the execution environment for the Sun-/Windflower calculations. It's logic has been implemented as a Node.js application, MongoDB is used as database and RabbitMQ is used as the message broker towards the workers, which are described in Section 4.3. The Node.js backend application, the MongoDB database and the RabbitMQ message broker are run in separate Docker containers and connected via a virtual private network. The orchestration is done with Docker Compose.

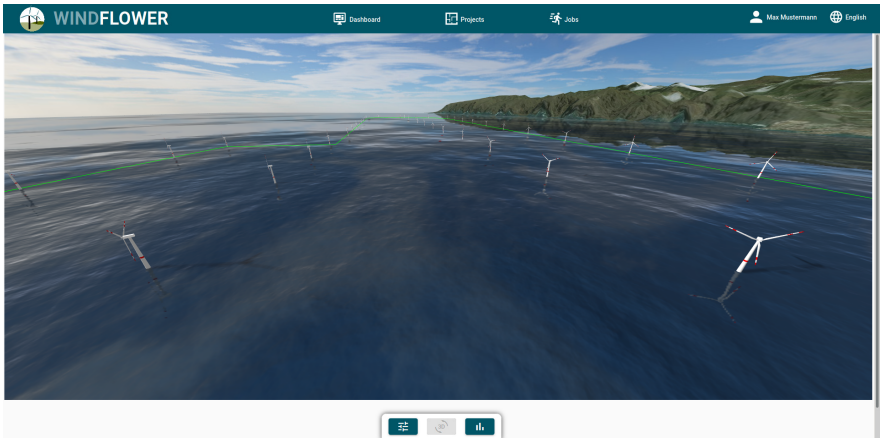
The Node.js application itself is designed to be modular and consists of a number of services that communicate with each other via a bus-like messaging system, called the event bus. Incoming HTTP requests are handled by the API router service, which is built on top of the Express.js framework. Depending on the type of the request, the api router redirects it to one of the other services. Login or other user related requests are handled by the user service, getting a list of existing projects or changing project parameters for example is processed by the project service.

MongoDB is a database software that is widely used with Node.js applications and stores data as JSON-like documents, which makes it suitable for storing project parameters.

RabbitMQ is a middleware software that implements the Advanced Message Queuing Protocol (AMQP). It enables messaging between different programs. Messages get buffered in queues and publishers and consumers get decoupled. It is used to queue and distribute new tasks from the backend to simulation workers and return the results



(a) Setting project parameters



(b) 3D visualisation of an offshore wind farm

Figure 4: Example views of the web application frontend.

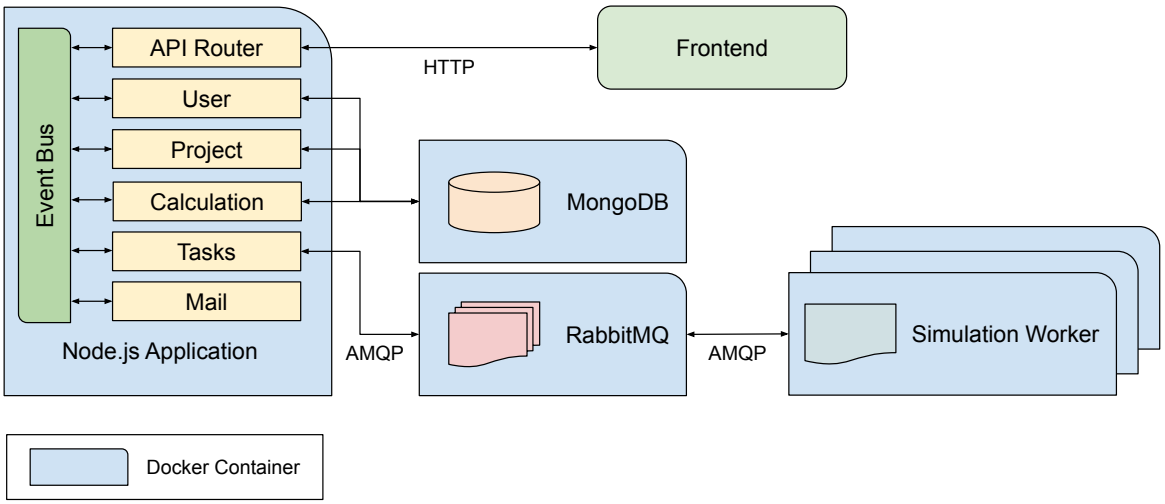


Figure 5: Interaction of the web application's backend components.

after execution. Additionally, execution status updates are sent by the workers. We are using two separate queues: *tasks* for delivering new tasks with the corresponding parameters from the backend application to the simulation workers and *lifecycle* to return status updates and calculation results.

A task refers to a single execution of a Sun- or Windflower calculation as described in Section 3.1 and 3.2. This thesis aims to provide a scheduling algorithm and implementation along with resource scaling to optimize the execution of these tasks by instances of the simulation worker. When a user triggers a calculation from the frontend, e.g. a simulation run for a wind farm, the following happens in the backend: the calculation service requests the project parameters from the project service, which fetches them from the database. Also a database entry for the task is created. Here the current status of the task and, after successful completion, the simulation results are retained. The calculation service also commits the new task to the tasks service, which enqueues it in the task queue on the RabbitMQ instance.

4.3 Simulation Worker

A simulation worker is a Node.js application which connects to the RabbitMQ message broker and is incorporated in a Docker container together with Sun- and Windflower executables.

As soon as a simulation worker is idle, it can fetch the task from the task queue. The delivered project parameters are saved to temporary files and the requested calculation is started with the corresponding executable. The execution environment inside the Docker container has all required dependencies to run the executables. Via the lifecycle queue the simulation worker gives feedback on the current task's execution status to the tasks service, which relays it to the calculation service to update the status in the database entry. When a user polls the task he now can see that it is running. While

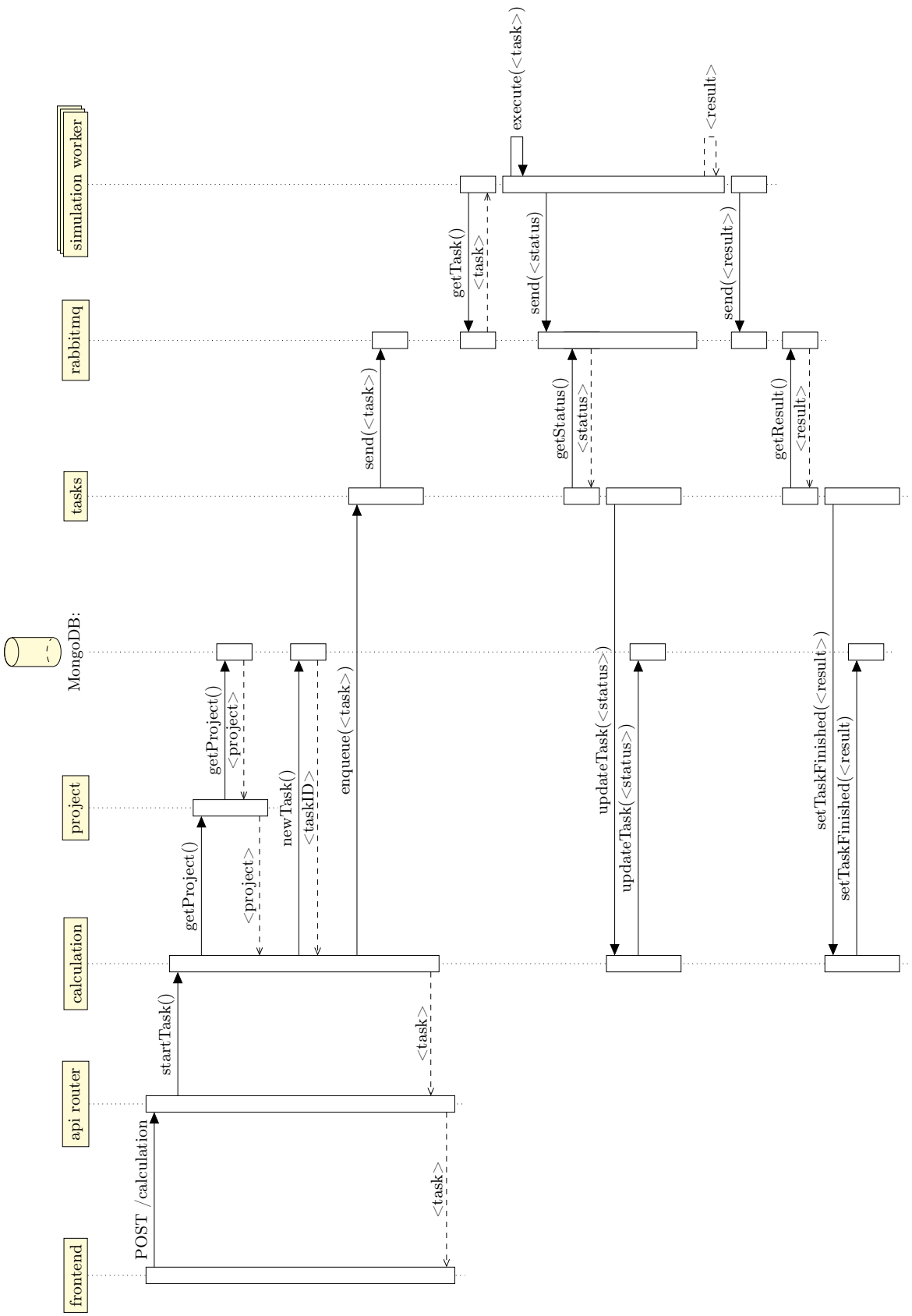


Figure 6: Lifecycle of a calculation task.

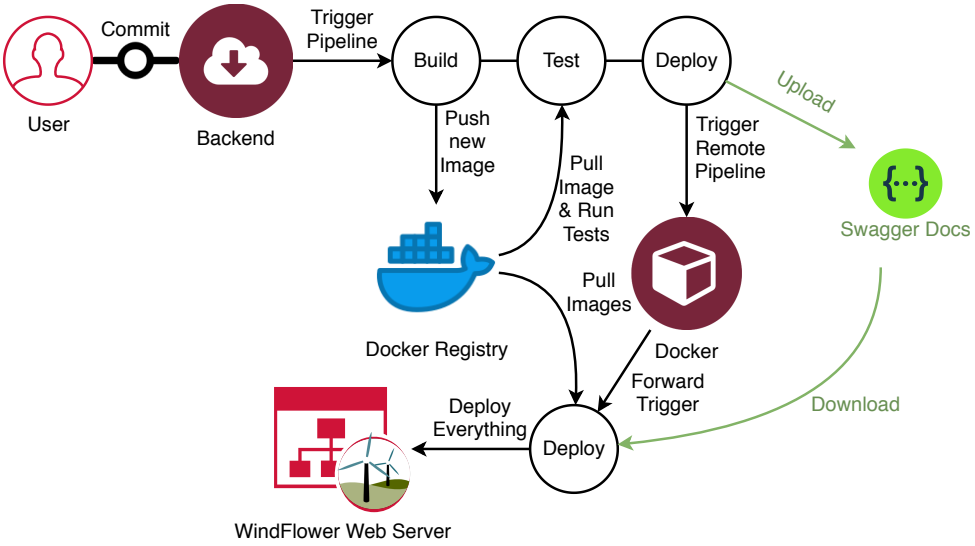


Figure 7: Illustration of the development cycle of the Node backend. The depicted Docker registry is GitLab's internal container registry, thus the entire CI/CD process is facilitated by GitLab.

a calculation is running the worker constantly monitors its progress and sends status updates back to the tasks service. Upon successful completion the result is read from the output file and transferred back to the tasks service via the lifecycle queue.

4.4 Continuous Integration/Continuous Deployment

To automate the build process of the web application's components a setup for continuous integration and deployment (CI/CD) has been created.

CI/CD is done through GitLab pipelines. Docker images are built and published to the local GitLab container registry on every push. Tests are being run with the new containers and upon successful completion the new is being rolled out automatically. Figure 7 depicts the continuous delivery process of the Node backend triggered by a developer's commit and resulting in a new version getting published to the production server.

5 Scheduling and Scaling Strategy

Running calculation tasks for Sun- and Windflower can be computationally expensive, especially for optimizations. When providing a web service where a group of users all can trigger tasks at any time one might be coming to a point where computational resources are not sufficient to process all tasks right away. In this case it is necessary to prioritize tasks and delay less important ones, i.e. to times when emergence of new tasks is low. The priority of a specific task may depend on a multitude of factors.

Another way of handling fluctuating demand for computational resources is resource augmentation. The number of machines that process tasks can be increased or decreased dynamically. This is only possible, if there is sufficient supply of such dynamically available computers and it is only reasonable if it results in an economisation compared to a fixed size computing cluster.

In the following we will concentrate on the Windflower calculations only. But as the concepts and calculation types of Windflower and Sunflower are similar, the results of this thesis should roughly be applicable for Sunflower, too. Furthermore, we will take in consideration only the three calculation types *Simulation*, *SensitivityAnalysis* and *Optimization*. That is, because we are expecting only very small amounts of the two *GenerateTopography* and *Generate3DPrinterFile* and they are very similar to the simulations in terms of their runtime characteristics.

5.1 Simulation Worker Host Platform

For the decision on how to schedule tasks it is important to know what platform, i.e. what hardware, the simulation workers are running on.

An obvious solution would be to use the server on which the rest of the backend is deployed already. This has the advantage of not needing additional hardware. On the downside, the available computation power is very limited, inflexible and the simulation worker competes with the web application for processing power. High workload on either the web application or the worker leads to worse performance of the other and prediction of task runtimes gets difficult.

Another option is to rent a number of dedicated servers or virtual machines. These offer a guaranteed computation capacity and reasonable pricing per time unit. But they take time to provision and their renting period normally is a month, so going with these servers would mean a loss of flexibility.

The third option are cloud servers. These are VMs that can be provisioned quickly and started or stopped at any time. They get billed only for the actual time they run, but the price per time unit is higher than with machines that are rented long-term. The flexibility of cloud servers makes them very suitable for use cases where much scaling of resources is necessary.

In our case we are using a setup that is often referred to as a *hybrid cloud*: a constantly available private server that offers a basic computing capacity as well as on-demand VMs from a public cloud service provider that can be added to the compute pool for absorbing higher loads.

The private server features an AMD Ryzen 7 3700X 8-Core Processor, 32GB of RAM and hardware for GPU computing. Because it is not exclusively used by our project, we restrict the maximum number of parallel executions to four. As the server is running permanently, there are no additional costs occurring for the executions of our simulation tasks. The GPU hardware offers the opportunity to accelerate specific calculations when they are adapted to the hardware.

For the additional public cloud servers we are using services of the Google Cloud Platform (GCP). Many cloud computing providers provide services that allow direct

deployment of containers. GCP for example offers the *Kubernetes Engine*, which is runs containers in a cluster, orchestrated by the *Kubernetes* software. Kubernetes includes an option to automatically scale services, i.e. create more container instances, when CPU load of the existing ones is high or based on other metrices. This can be useful for our purpose, but by reason of wanting to have more fine-grained control of the deployment process, we are not using it here.

Instead, we are using general VMs provided by the GCP Compute Engine. New VMs can be created via a graphical web interface or by a certain call to an API. When creating the new instance, several settings can be controlled, along which are:

- Machine Type
- Region
- Boot Image
- Network

The **machine type** controls on which (virtual) hardware the VM runs. The number of virtual CPU cores and size of memory can be selected. There are options to use more powerful CPUs as well as shared CPUs, where the full computing power may not be available at all times.

A **region** determines where the datacenter is located in which the actual servers stand. Using different regions can decrease latency, depending on the location of customers, as well as increase protection against downtimes because of local effects (e.g. power outages). Depending on the selected region the choice of machine configurations can vary and also prices differ. Two sample regions are schon in Table 1. Because of geographical reasons and the availability of optional additional GPUs we will only use resources from the *europe-west3* region.

Region	**Location**	**Machine Types**	**CPU Families**	**Resources**	**Hourly rate n1-standard-1**
europe-north1	Hamina, Finland	E2, N2, N1, C2, M1	Broadwell, Skylake, Cascade Lake	-	$0.0523
europe-west3	Frankfurt, Germany	E2, N2, N1, M1, M2, C2	Broadwell, Skylake, Cascade Lake	GPUs	$0.0612

Table 1: Example of two GCP regions. Frankfurt offers one more machine type and additional GPU resources, but Finland has lower costs. Data taken from [2] and [3].

The **boot image** is a disk image that includes an operating system to which the VM can boot on its first startup. Because we do not want to configure each machine separately, we have prepared an image that contains Ubuntu Linux as an operating system and has Docker installed. On start-up it will pull the latest Docker image of the simulation worker from the Gitlab docker registry (see Section 4.4) and run it. The IP address of the RabbitMQ instance that shall be used is passed at start-up as an environment variable and then used by the simulation worker to connect to.

Network configurations mostly consider virtual cloud internal networks. In our setups all simulation workers are located in the same virtual network. The workers do not have an associated fixed external IP address, but communicate to the RabbitMQ via a gateway router. This router has a fixed external IP address, which allows for a connection to the RabbitMQ instance via an ssh-tunnel. In any case the network connection to RabbitMQ only has to be set up on the router without changing the workers' configurations.

5.1.1 Machine Type Selection

As already mentioned, cloud computing providers offer machines with different configurations. Amongst other things this affects the number and speed of CPUS as well as memory size. The simulation worker application can run an arbitrary number of calculations in parallel, but each calculation only runs in one thread, meaning that a single calculation will only utilise one CPU at a time. The simulation worker shows only little memory usage that will not be a limiting factor concerning usual memory configurations of servers.

As prices vary for the different server configurations, we want to select the machine type which allows for the fastest and/or most cost efficient execution of simulation tasks. We run the Windflower SensitivityAnalysis with a sample size of 1000 at least 50 times per machine type and capture the needed time for the executions. A sample size of 1000 means that 1000 simulations of the same wind farm are run, each with slightly changed parameters. We use the DanTysk sample data set as basis for the simulations.

A first test is done with the simulation worker only running one calculation at a time. The results are shown in Table 2 and Figure 8. The standard deviation of execution times shows to be very low. The fastest execution time is reached by the c2-standard-4 type, a computing-focused machine with 4 cores, followed by the single-core n1-standard machine. All other multi-core machines produced much worse times, but very similar to each other. This shows that the "highcpu" options indeed just mean a higher CPU-to-memory ratio with otherwise same performance. While it is also not surprising that multi-core machines do not speed up the execution of a single-thread, it is astonishing how much better the single-core machine performed in comparison. In consequence the n1-standard-1 machine by far offers the best cost-performance ratio.

To better utilise the capacities of the multi-core machines additional tests were run with two and four parallel calculation threads respectively. The results are plotted in Figure 9. They show that the average execution time increases when the number

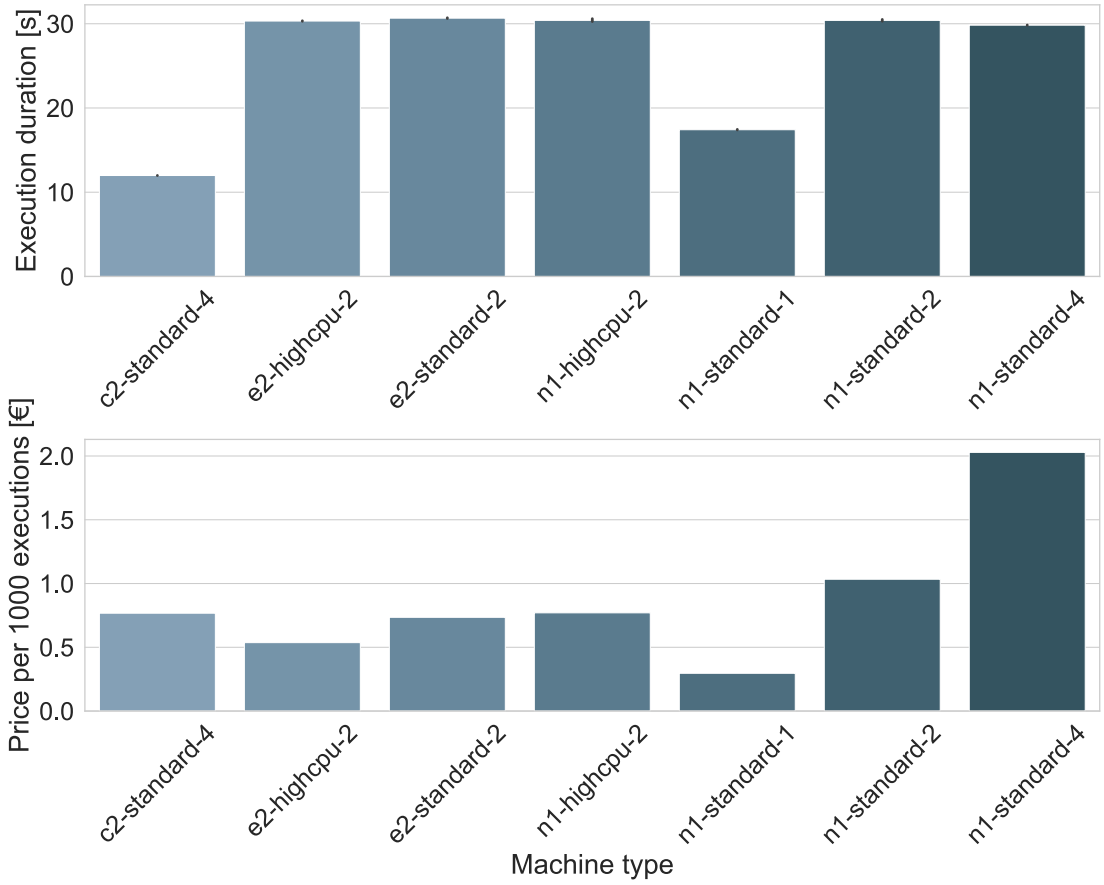


Figure 8: Comparison of different machine types' performances as in Table 2.

Machine Type	vCPUs	Hourly Rate	Avg. Execution Time	Std	Price per 1000 Executions
c2-standard	4	$0.2305	11.984s	0.078s	$0.7673
e2-highcpu	2	$0.0637	30.321s	0.164s	$0.5365
e2-standard	2	$0.0863	30.664s	0.397s	$0.7351
n1-highcpu	2	$0.0912	30.400s	0.1204s	$0.7702
n1-standard	1	$0.0612	17.419s	0.456s	$0.2961
n1-standard	2	$0.1224	30.398s	0.227s	$1.0335
n1-standard	4	$0.2448	29.830s	0.102s	$2.0285

Table 2: Comparison of different machine types' performances with only one calculation running at a time. One Execution refers to one run of the Windflower SensitivityAnalysis with a sample size of 1000 on the DanTysk data set.

of parallel calculations reaches the number of CPUs available. Also, the multi-core machines get more cost-efficient with multiple tasks running in parallel but still cannot match the cost-performance ratio of the n1-standard-1 machine.

In conclusion this means that there are several aspects in favour of using n1-standard-1 machines for our computations:

- They offer the highest flexibility as we do not have to care about the utilisation of the multiple cores. There can be provisioned one machine per running task.
- They offer fast computation in comparison to most of the other availably machine types
- The cost-performance ratio is superior to the alternatives

Therefore, we will use n1-standard-1 machines as cloud servers in the following. It is to be noted however, that c2-standard machines can offer faster computations when not fully utilised, at a higher cost.

5.1.2 Boot Time

While the public cloud machines can be started at any time on demand, they have to boot like every other computer. We have to consider the boot time when scheduling tasks on cloud machines. For the n1-standard-1 machine type we have recorded an average boot time of 21 seconds, from sending the boot command until the simulation worker registers with the RabbitMQ message broker.

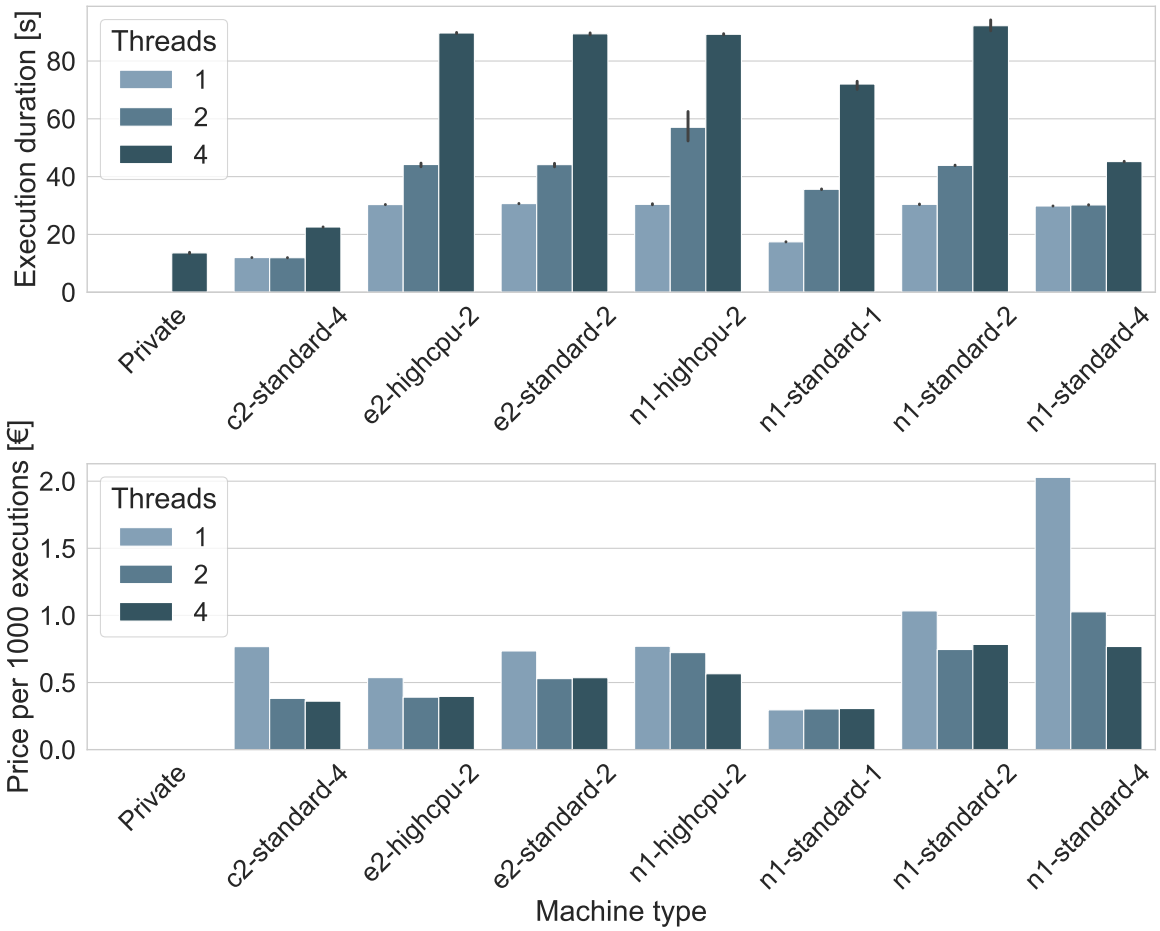


Figure 9: Comparison of different machine types' running the same calculations as in Figure 8, but with multiple calculations in parallel. Also, the execution duration of the same calculations on the private server is shown for comparison.

5.2 Runtime Analysis

For the scheduler to produce the desired result it is crucial that it can generate accurate estimates of the tasks' runtimes. We will use these estimates as a basis for simulating the arrival of calculation tasks in order to evaluate the scheduling performance as well as for the scheduling decision itself.

As we are looking at three different types of tasks that are to be scheduled, the runtime depends on that type in the first place. But depending on certain parameters, execution times may diverge even for the same kind of calculation. We therefore analyse the runtime of the task types in detail.

5.2.1 Simulation

The simulation aims to compute the costs and revenues of a wind farm over the course of one year based on a chain of calculation models, shown in Figure 3. To estimate the runtime, we first identify the set of input parameters that have a non-constant influence on the computational complexity. Looking at the parameter files and the simulation code it becomes clear that most of the parameters represent coefficients, e.g. cost or loss factors, for various calculations that do not affect the runtime.

What has an impact on the runtime, however, is the resolution and type of the wind model as well as the size of the wind farm in terms of the number of installed turbines. Analysis of the code brought up the following candidates to consider for complexity and runtime estimation:

- `num_wind_directions`: The number of wind directions that are possible for the wind model. Wind gets discretised to these directions before simulating wake and power generation. Value range: 4 to 1200.
- `num_wind_speeds`: Wind speed is discretised analogously to wind directions. Value range: 3 to 100.
- `wake_model`: Wake describes the turbulences that form behind a turbine when it is hit by wind. These models are outlined in [23]. Possible values: park, modified_park.
- `num_turbines`: Not a real input parameter, but defined by the number of positions given for the wind turbines. The amount of turbines impacts numerous calculations including wake, cabling and power generation.

To get a picture of how the single parameters influence the runtime, tests were run, each with all parameters fixed except for one of the considered ones. The results of the runs are plotted in Figure 10. For the number of wind directions, respectively speeds a linear dependence clearly shows. Also the choice of the wake model has a small, but noticeable effect.

However, the most interesting curve is produced by the number of turbines involved: Here, an exponential increase of the execution time can be observed dependent on the

increase of turbines. A polynomial regression of order 2 seems to be not able to fit this curve well, while the regression of order 3 provides a very close result. The reason for this exponential behaviour lies in the wake model: The wake of each turbine impacts the wind conditions on potentially every other turbine of the same wind farm.

To find an overall estimator, first, large datasets were generated on the two machine types that are supposed to run the tasks. Simulations were run with random combinations of the input parameters in question and the runtime recorded. On each of the machine-specific datasets a regression model was trained then. The categorical `wake_model` parameter was one-hot encoded and the parameters were expanded to polynomial combinations of up to an order n. Then a linear regression model was fitted to get a polynomial regression overall.

The R^2-score for order $n = 2$ is at $R^2 \approx 96\%$, improves to $R^2 \approx 99.3\%$ for $n = 3$ and even further to $R^2 \approx 99.98\%$ for order $n = 4$. Higher orders of polynomial regression show no enhancement of the R^2-score.

The correspondent residual graphs are plotted in Figure 11. They show the deviation of the runtime estimates compared to the observed values for a separate validation data set which was not used in the fitting process of the model. For order $n = 2$ and $n = 3$ a systematic non-linear residual pattern is clearly visible while the graph for $n = 4$ shows a pattern that is very likely to be only noise.

Accordingly, we will use the polynomial regression of order $n = 4$ for estimating simulation runtimes.

5.2.2 SensitivityAnalysis

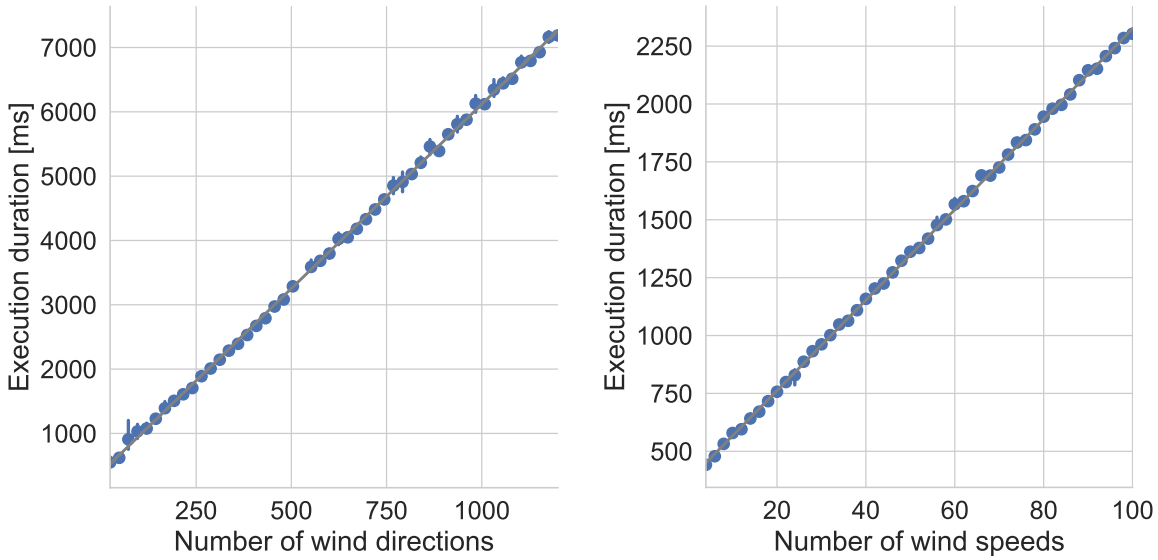
For the sensitivity analysis there is one decisive input parameter: `sample_size`. It defines the number of simulations which are to be carried out. As all relevant input parameters for the simulation runtime are not altered in the course of the analysis the runtime can be calculated as the product of `sample_size` and the simulation runtime according to Section 5.2.1.

5.2.3 Optimization

The runtime of the optimization is harder to estimate. It depends on the type of optimization used, can be parametrically bounded and potentially has a random component affecting it. The multi-step optimization algorithm presented in [23] operates in a number of iterations. This number has a lower bound n_{min_it} and an upper bound n_{max_it}. There are two stop criterions for the optimization:

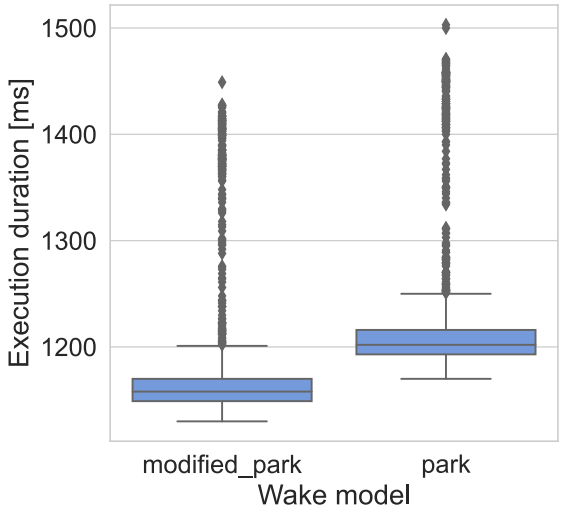
1. n_{max_it} iterations have been performed.
2. At least n_{min_it} iterations have been performed and the improvement between two iterations was worse than a certain threshold ϵ.

To find possible optimized settings, the optimization performs numerous simulations, with the exact number depending on the grid settings on which the turbine placement

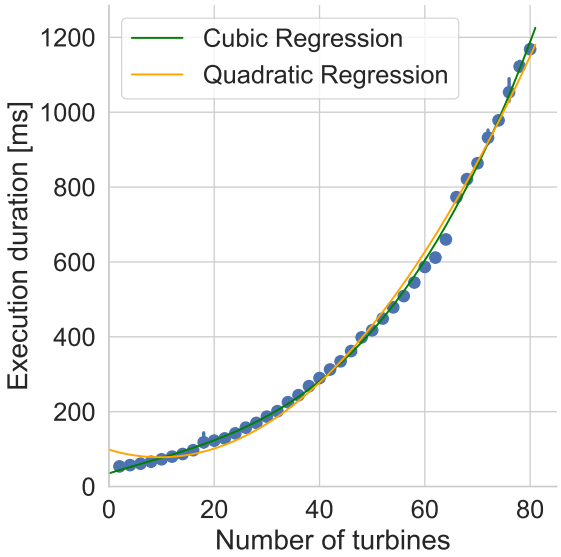


(a)

(b)

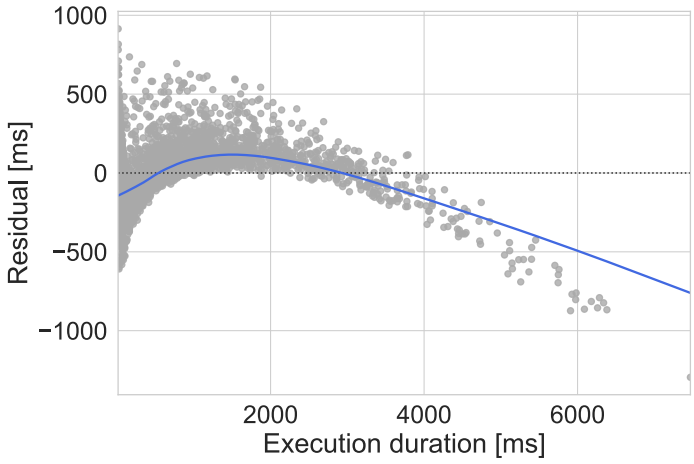


(c)

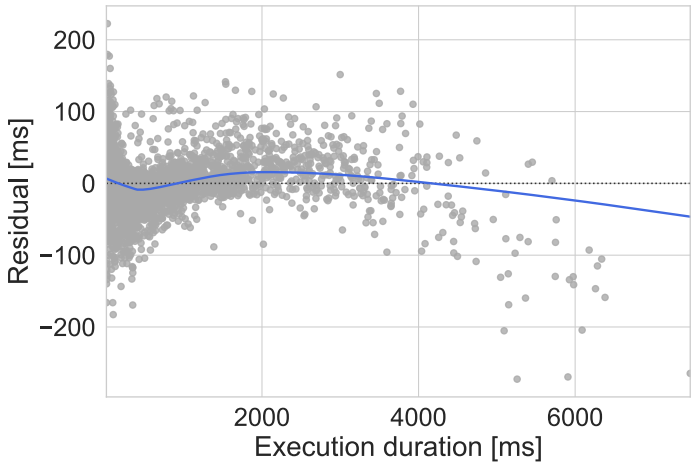


(d)

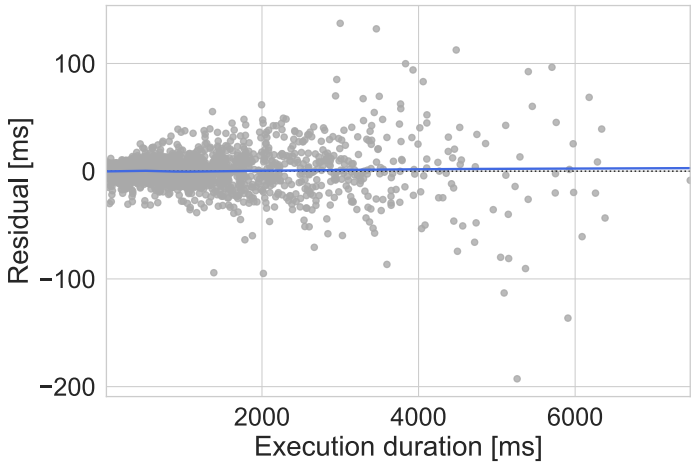
Figure 10: ($n = 5000$ each)



(a) $n = 2$

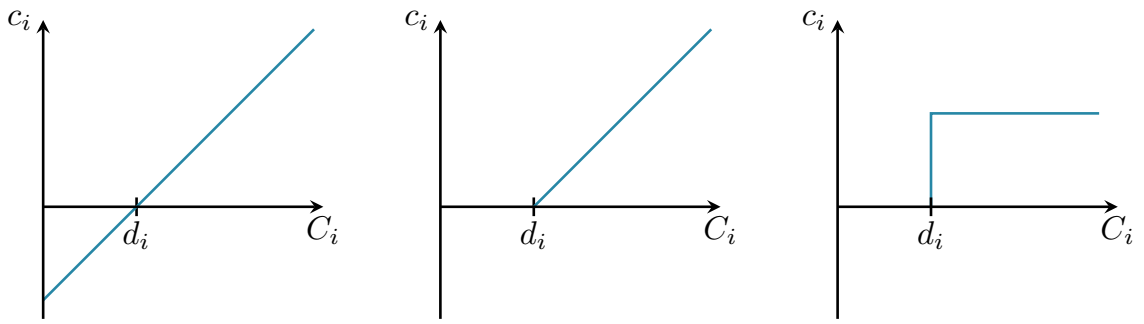


(b) $n = 3$



(c) $n = 4$

Figure 11: Residual graphs for the runtime estimation by polynomial regression of different orders.



(a) Lateness (b) Tardiness (c) Unit Penalty

Figure 12: Simple cost functions for use in scheduling.

takes place. With parameter c representing the number of circular grids, n representing the number of grid positions and t representing the number of turbines, the count of simulations per iterations n_{sim_it} can be computed as $n_{sim\_it} = c \cdot n \cdot t$.

With n_{sim_it}, n_{min_it} and n_{max_it} we can estimate the lowest and highest possible runtimes of the optimization with relatively high precision as per Section 5.2.1. But estimating when the progress of the optimization falls below ϵ, leading to its abortion, is not possible. Thus we cannot estimate the runtime in this case.

5.3 Cost Function

To take a qualified scheduling decision and to further evaluate the quality of a schedule in hindsight we want to define a value that rates the point in time when the execution of a task starts or ends and the computational resources needed. We want to assess this value by the means of a cost function. In Section 2.1 we already gave two examples for such cost functions, each based on the completion time C_i and a fixed deadline d_i for some task (job) J_i:

- Lateness: $L_i = C_i - d_i$
- Tardiness: $T_i = \max\{C_i - d_i, 0\} = \max\{L_i, 0\}$

Another method is to use a kind of unit penalty that does not linearly grow with time, but is a step function that penalizes with a fixed cost if the deadline is violated and imposes no costs otherwise.

- Unit penalty: $U_i = \begin{cases} 1 & , \quad C_i \geq d_i \\ 0 & , \quad \text{otherwise} \end{cases}$

Of course, for these cost functions we need to determine the deadline d_i first. It is usually dependant on the release time r_i and, especially if it is a hard deadline that should be feasible to meet, should not be less than the processing time p_i after r_i: $d_i \geq r_i + p_i$. It is possible to use a constant offset o_c or make it a (linear) function of the runtime p_i.

- Constant: $d_i = r_i + p_i + o_c$
- Linear: $d_i = r_i + (1 + f) \cdot p_i$

In our use case we do not have hard deadlines, because there are no essential measures depending on the timely execution of tasks as there would be in important real time execution environments or also in manufacturing processes for example. Rather it is the user experience that is compromised if the execution of simulation tasks takes too long. It is to assume that a user has a certain tolerance for delayed task runtimes, which may also vary from user to user.

For our cost function this has the following implications:

- It is not desirable to reward overly punctual completion of tasks. If we can have reasonable runtimes for a large share of tasks, we prefer this over fast runtimes for only a few ones. Our cost function will be non-negative and provide a "grace period" in which no costs occur at the beginning.
- A function like the unit penalty which stops growing with time at a certain point tends to make already long waiting tasks "starve" when waiting queues get longer. To minimize costs only very young tasks are selected to run and older ones which already passed the deadline are neglected because they are already "written off" cost-wise. Our cost function must prevent starving and therefore will be (from a certain point on towards infinity) strictly monotonically increasing.
- It is to consider that we are dealing with three categories of tasks which have very different runtimes. We have to make sure that the right balance is given when comparing task delays of different categories. If we take an exemplary simulation with a runtime of $0.5s$ and an optimization with a runtime of $3600s = 1h$ then an additional delay of $10s$ for the execution should lead to a bigger cost increase for the simulation than for the optimization. In contrast we can get over waiting on the simulation for a multitude of its actual runtime (e.g. $50 \cdot 0.5s = 25s$), while we want to prevent the same for the longer optimization task (e.g. $50 \cdot 1h \approx 2d$).

This leads us to use the following cost function based on the delay with which a task has been started and the delay ratio to the tasks' estimated runtime.

$$W_i = \begin{cases} C_i - p_i - r_i & , \quad C_i \geq r_i + p_i \\ 0 & , \quad \text{otherwise} \end{cases}$$

$$c_i(W_i, p_{i,private}) = \sqrt{W_i \cdot \frac{W_i}{p_{i,private}}})$$

When we take the resource scaling by means of cloud resources into account we are dealing with another type of cost: The monetary fees that incur for the time period a cloud machine is running (whether it is actually executing tasks or not).

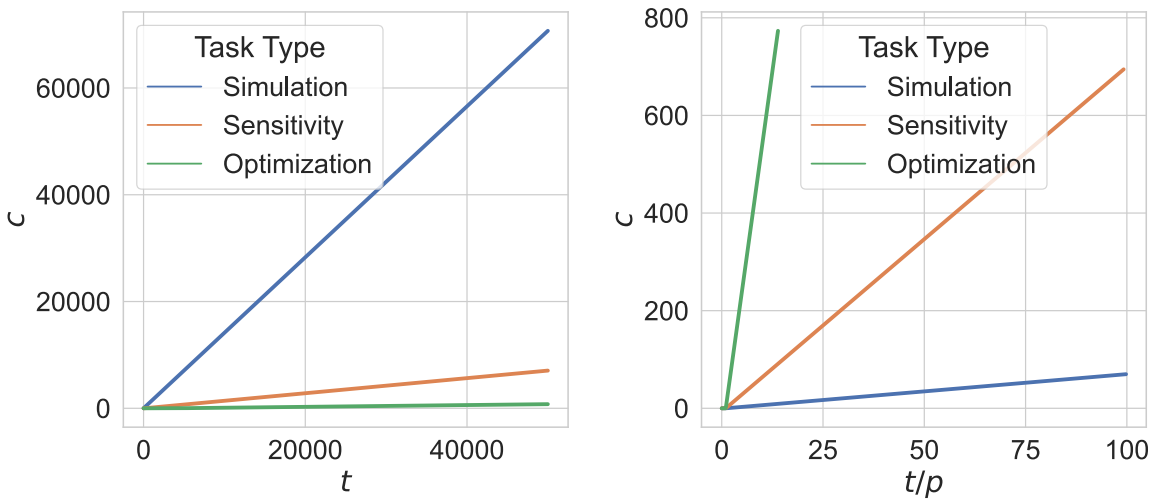


Figure 13: Cost of three exemplary tasks plotted over time. The simulation task has a runtime of $p = 0.5s$, the sensitivity analysis one of $p = 50s$ and for the optimization task it holds $p = 3600s$. All three have the same release time of $r = 0$. Cost is once plotted over the absolute time and once over the factor of time since release over runtime.

Comparing these two types of cost directly is not meaningful and also trying to adjust them with a constant factor does not compensate for the balancing we are doing with our cost function. In a resource scaling scenario we therefore have to minimize the cost characteristics separately from each other.

5.4 Scheduling Simulation

Testing scheduling scenarios in the production environment sketched in Section 4 would take a huge amount of time and in consequence be unfeasible. We therefore have developed a simulation tool to explore different scenarios and validate scheduling algorithms via Monte Carlo simulations.

The simulation tool reproduces all relevant components of the web application. Instead of user input it uses randomly generated events and instead of performing the actual calculations the task runtimes are calculated based on the models developed in Section 5.2 and machine utilization and queues simulated accordingly. The time span that is covered by the simulation can be selected arbitrarily and experiments can be run repeatedly with the same or slightly changed parameters to conduct a sensitivity analysis.

The core classes that make up the simulation tool are depicted in Figure 14 and will be briefly described in the following.

Project The *Project* objects correspond to projects in the web application. They include the set of parameters needed for the runtime estimation. The *activity* variable

defines how many tasks for this project shall be launched per day on average. With the *acitivityType* we can control the pattern of the times at which tasks are created. This way we can not only simulate uniform patterns where the task release rate is constant, but also launch patterns where tasks are only created during working hours or following certain burst characteristics. Following that pattern, launch dates for new tasks are generated step-wise using a Poisson probability distribution. Parameter λ for the Poisson distribution is chosen according to the *activity* value.

Task When created, a *Task* object calculates the estimates for its runtime on different machines. It keeps track of its start and end dates and, when scheduled on a worker, determines a "real" runtime that might differ from the estimate to account for the uncertainty we see in the collected runtimes from Section 5.2. This runtime can scatter around the estimate and also be systematically higher or lower.

Worker The *Worker* objects are responsible for keeping track of currently executing tasks, their end dates and for the handling of finished tasks. They offer statistics on their workload for given time periods.

For a simulation run we define the time period that shall be simulated. At the beginning a set of projects is created randomly or as defined by us. The workers that represent our private computing server are created and booted up. Possible additional cloud resources are created, but not yet booted. The simulation then works off the time period in steps. New tasks are launched and workers clear their finished tasks if necessary. If there are tasks waiting the scheduler takes a decision based on the task queue and the current worker situation. After the simulation run is finished the relevant statistics are collected and plotted.

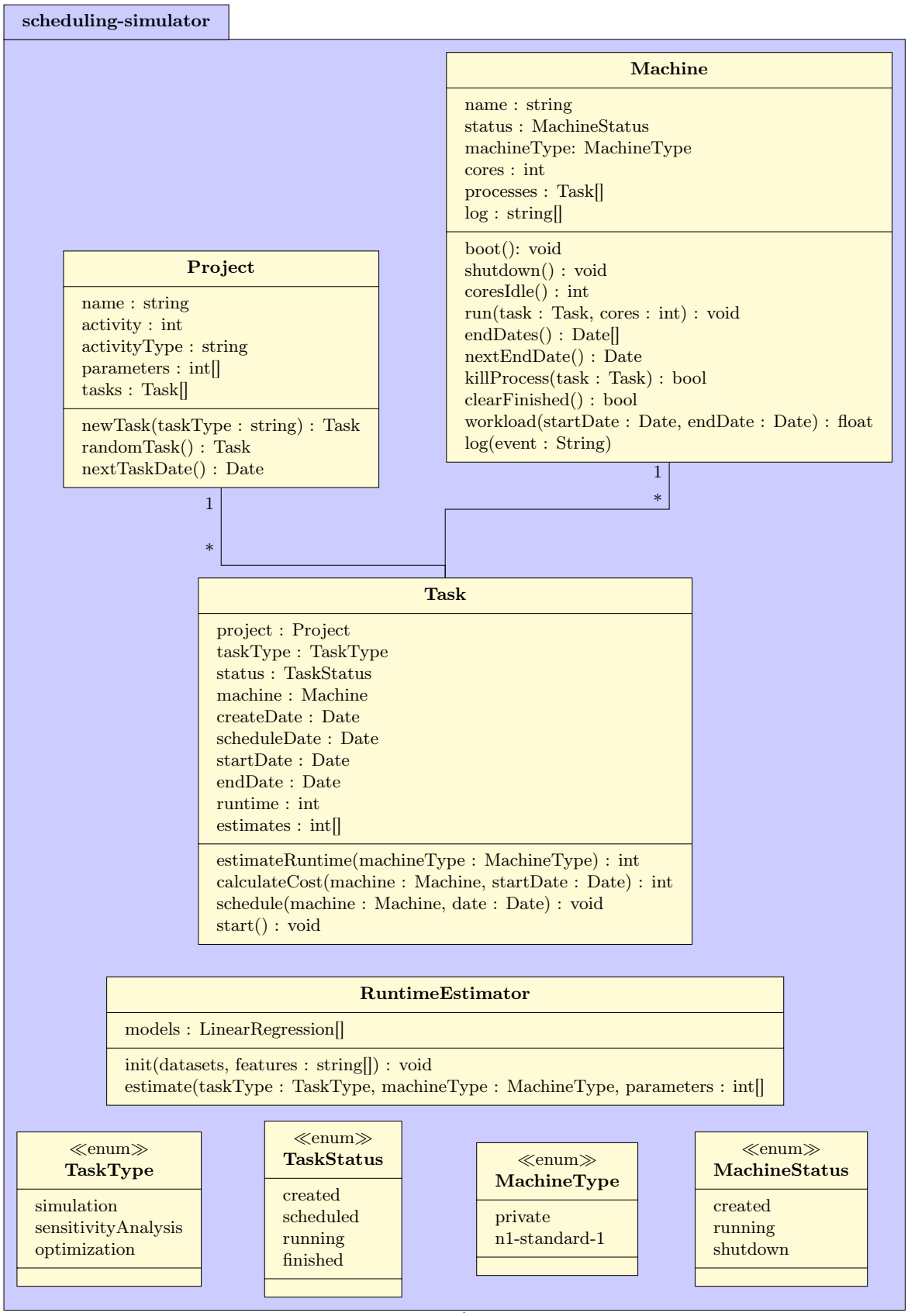


Figure 14: Class diagram showing the core classes of the Monte Carlo simulation tool developed in the course of this thesis.

6 Case Study

To investigate the dynamics of the simulation tasks in our web application environment and to evaluate the performance of different scheduling approaches, a case study is conducted using the simulation tool described in Section 5.4.

6.1 Parameter Settings

During the case study we will vary different parameters of the scheduling simulations that we want to analyse, while others stay consistent. These settings are described in the following.

6.1.1 Task Type Distribution

We expect that the number of times the three relevant task types for WindFlower are each triggered are not equally distributed. Instead, most of the requested tasks will be simulations, followed by optimizations and sensitivity analyses, which together are only expected to make up 15% of the launched tasks. The individual expected shares are shown in Table 3.

Task Type	Expected Share of Tasks
Simulation	80%
Optimization	15%
Sensitivity Analysis	5%

Table 3: Assumed shares of executed task types.

The share of total runtime shows a contrary picture. Due to the huge number of simulations carried out in a single optimization or sensitivity analysis, simulations make up only for a marginal share of the overall execution time as shown in Figure 15.

6.1.2 Project Parameters

When projects get initialized at the beginning of a scheduling simulation run, the relevant project parameters as described in Section 5.2 are set and used for all tasks descending from the project. While some of these will not be varied, others will be chosen randomly to create variance in the runtimes between different projects. The values are picked according to Table 4. For the optimizations we will assume the worst case, which is that they run through n_{max_it} iterations with $c \cdot n \cdot t$ simulations per iteration.

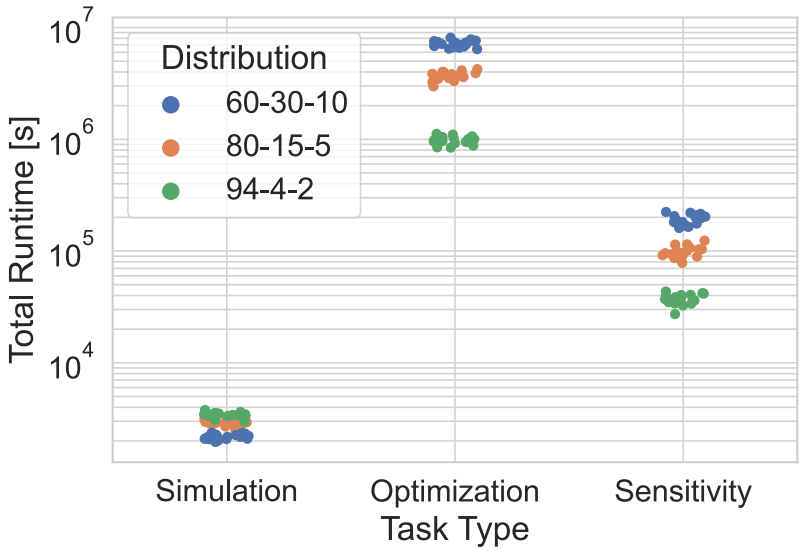


Figure 15: The total run times per task type from 20 runs of the scheduling simulation with different task type distributions for comparison, including the one from Table 3.

6.1.3 Time Period

For each setting we will run 100 simulations, each over a time period of 9 weeks. The first week will be excluded for data analysis to prevent distortion by certain dynamics having to build up at the beginning of a run.

6.1.4 Load Patterns

The requirements for scheduling algorithms depend not only on the characteristics of single task but also on the timings and frequency of the tasks' release dates. We will therefore survey the scheduling behaviour on different patterns of incoming tasks.

- A uniform pattern where the probability of a task's release is equal for every point in time
- A pattern depicting a scenario in which a significantly higher volume of tasks is released during the working hours of a certain geographical region. If the majority of the web application's users is located in a particular part of the world and uses the application for their work, we would expect that more tasks are launched approximately between 7 am and 6 pm of their time zone than during the night.

The uniform and working hours pattern are exemplarily plotted in Figure 16. Besides these two patterns we will analyse the schedulers response to bursts of new tasks that occur irregularly. In this case we will concentrate on the time period around the burst and not a longer span. The impact of such a burst on the incoming task load is depicted in Figure 17.

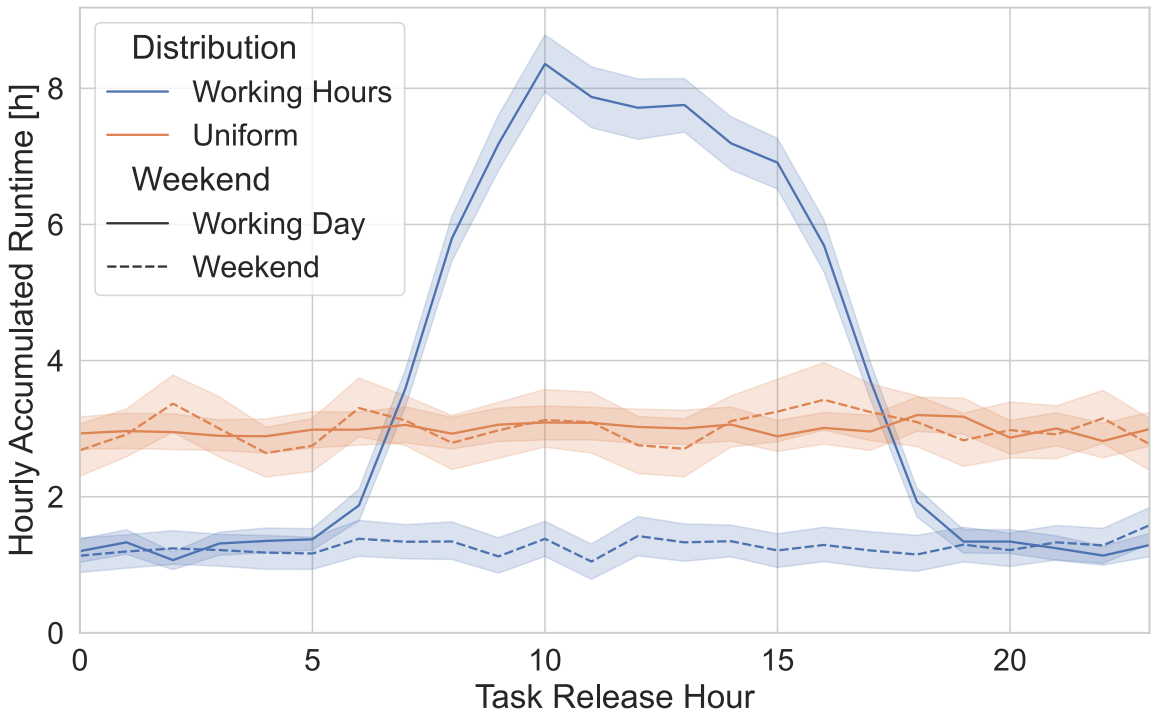


Figure 16: Comparison of the two distributions of task release times. The total accumulated runtime of the tasks is the same for both cases. A value of 1.0 for a certain hour indicates that a single simulation worker thread on the private computing server needs exactly 1 hour to process all tasks that have arrived during that particular hour.

Parameter	Value Range
num_wind_directions	120
num_wind_speeds	56
sample_size	[50, 1000]
wake_model	[park, modified_park]
num_turbines	[30, 120]
$c \cdot n \cdot n_{max\_it}$	72

Table 4: Relevant project parameters (see Section 5.2) used in the case study. Square brackets indicate the value is randomly chosen from this range.

6.2 Scenario 1: Fixed Resources

We start our case study by performing schedules only on the private computing server, which means that we can take hold of exactly 4 simulation workers at any times. This implies that there is a firm upper bound on the task capacity that can be processed in a certain time period.

6.2.1 Sorted Queues

To get a baseline, we will first run scheduling simulations with the two task release distributions and different average incoming loads of tasks. Here, for the scheduling algorithm we will use three basic variants that are based on sorting the queued tasks and starting the first task of the sorted list whenever there is one and a worker is idle at the same time.

- **First in first out (FIFO):** Complete the tasks in the order they arrived.
- **Shortest Runtime:** Select the task with the lowest runtime. This tends to reduce starving of short tasks.
- **Highest Costs:** Select the task that would produce the highest cost if started at this point in time. Due to the form of our cost function this is a middle course between the two aforementioned methods.

The concrete sorted scheduling algorithm with its three variants is shown in Listing 2.

Results for the uniform distribution already show that the FIFO algorithm is not tolerable, because the other two algorithms already produce significantly smaller average costs. This is the case especially for higher loads and shown in Figure 18.

When comparing the results for the two other algorithms in detail, broken down by task type, it emerges from Figure 19 that the costs mostly originate from simulations.

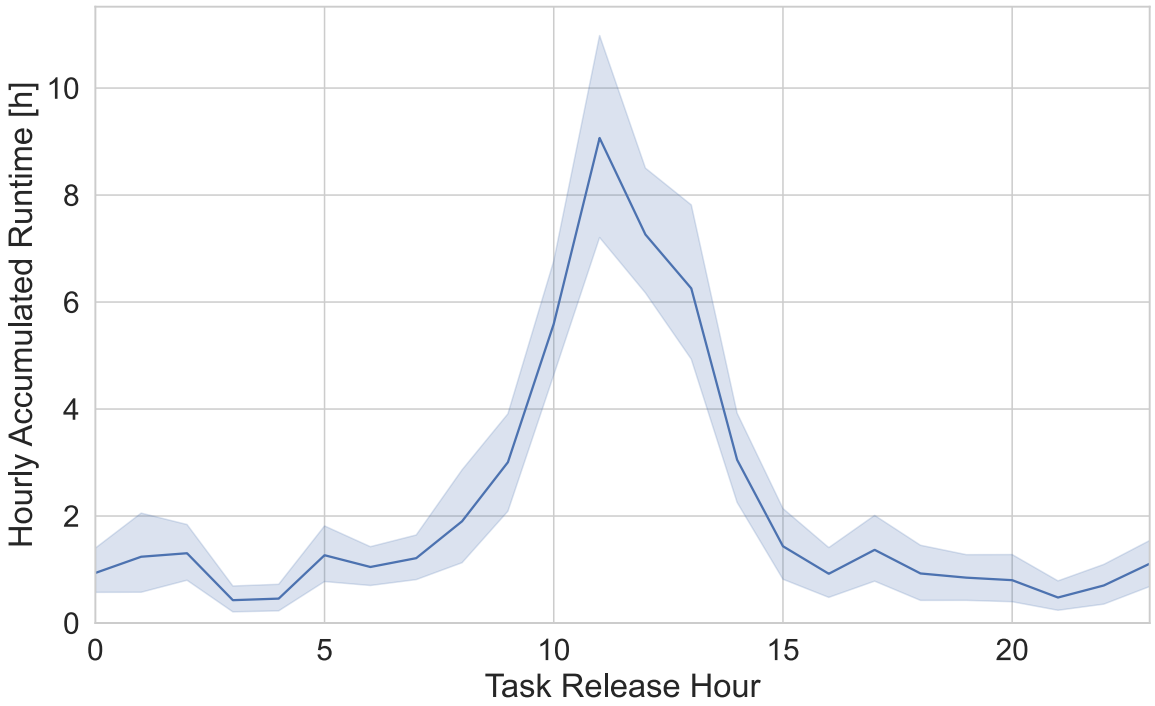


Figure 17: Effect of a burst of task releases around 12 pm.

Listing 2: Basic sorted scheduling algorithm.

```
function schedule_sorted(workers, projects, tasks,
   currentDate, sortBy) :
  for w in workers :
    if tasks['new'] and w.idle() :

      if sortBy == 'fifo':
        tasks['new'].sort(key='createDate', 'ascending')
      elif sortBy == 'shortest':
        tasks['new'].sort(key='runtime', 'ascending')
      elif sortBy == 'cost':
        tasks['new'].sort(key='cost', 'descending')

      t = tasks['new'].pop(0) # Select first element
      tasks['scheduled'].append(t)
      t.schedule(w)
```

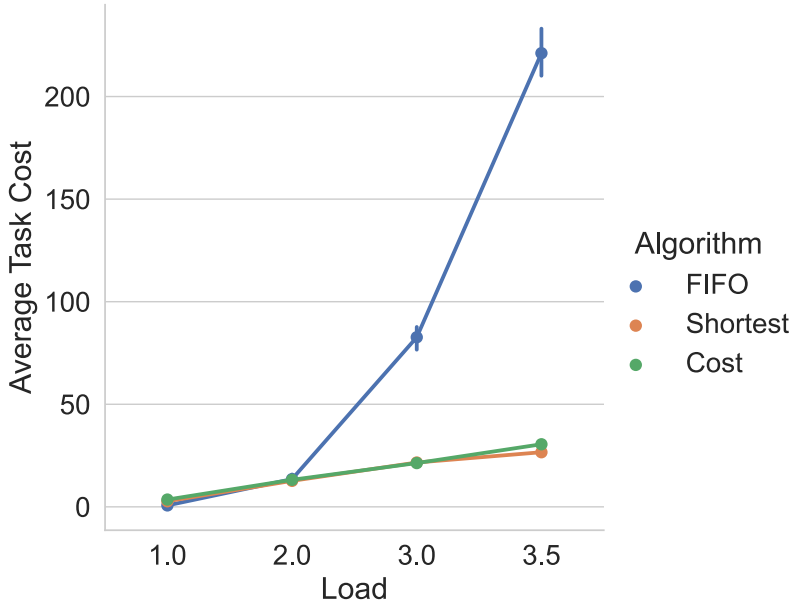


Figure 18: Average task cost for the three basic scheduling algorithms on different uniform load cases. A load of 1.0 means that on average one simulation worker was busy at any point in time. The curves for highest cost and shortest runtime algorithms are almost identical.

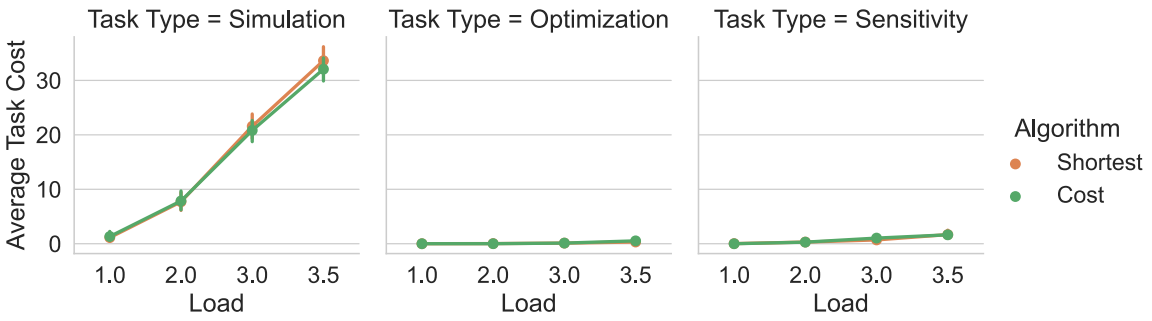
Ideally, the average costs would be similar for each task type, but the simulation costs are particularly crucial because of the task type distribution from Table 3. For a working hours task release distribution we get a similar scenario (see Figure 19b), but scheduling by shortest tasks now clearly performs better than the cost scheduling algorithm.

6.2.2 Dedicated Simulation Worker

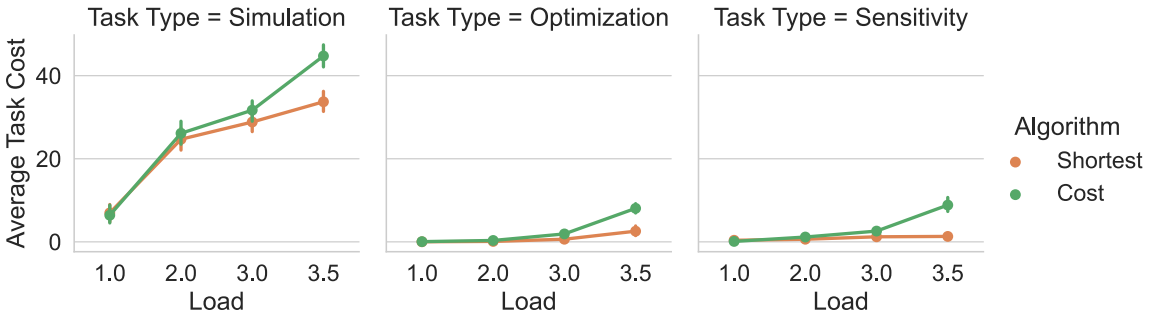
In order to level average costs for different task types we will adjust the shortest runtime scheduling algorithm so that it will dedicate one worker only to simulation tasks and use the other three available workers for optimizations and sensitivity analyses. The corresponding code is to be found in Listing 3.

This has one positive and one negative effect: The benefit is that simulation tasks will mostly be executed (nearly) immediately, because the worker is reserved for them and single simulation tasks take only 0.5 seconds to complete on average. The downside of this, however, is that because of the very short execution times the dedicated worker will idle in large parts and therefore prolong the waiting times of longer tasks especially in time periods with high task ingress.

As it can be seen in Figure 21, the benefits clearly outweigh the drawbacks as the overall average costs per task decrease greatly. While the simulation tasks produce almost no costs at all in this case, the rise of optimization costs shows in Figure 22, but stays in a tolerable range. The idling of the dedicated worker can be observed in



(a) Uniform Distribution.



(b) Working hours distribution.

Figure 19: Average task costs for shortest runtime and highest cost algorithms separated by task type. It shows that most costs are produced by the simulation tasks for these two algorithms. While there is no significant cost difference for the uniform distribution, the cost algorithm performs worse with higher loads in the working hours distribution case.

Listing 3: Scheduling algorithm with a dedicated worker for simulations.

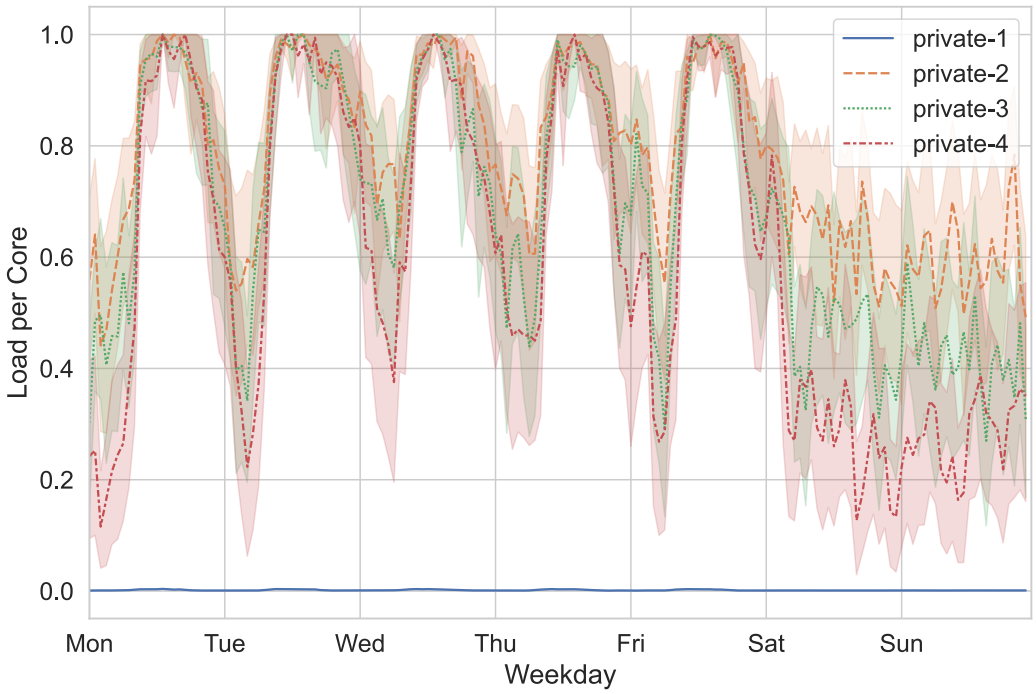
```
function schedule_dedicated(workers, projects, tasks,
    currentDate) :
for (i, w) in workers :
  if tasks['new'] and w.idle() :
    if i == 0 :
      newTasks = tasks['new'].filter('taskType' == '
          simulation')
    else :
      newTasks = tasks['new'].filter('taskType' != '
          simulation')
  if not newTasks:
    continue

  newTasks.sort(key='runtime', 'ascending')

  t = newTasks.pop(0) # Select first element
  tasks['new'].remove(t)
  tasks['scheduled'].append(t)
  t.schedule(w)
```

Figure 20a as there is hardly any load on that one worker.

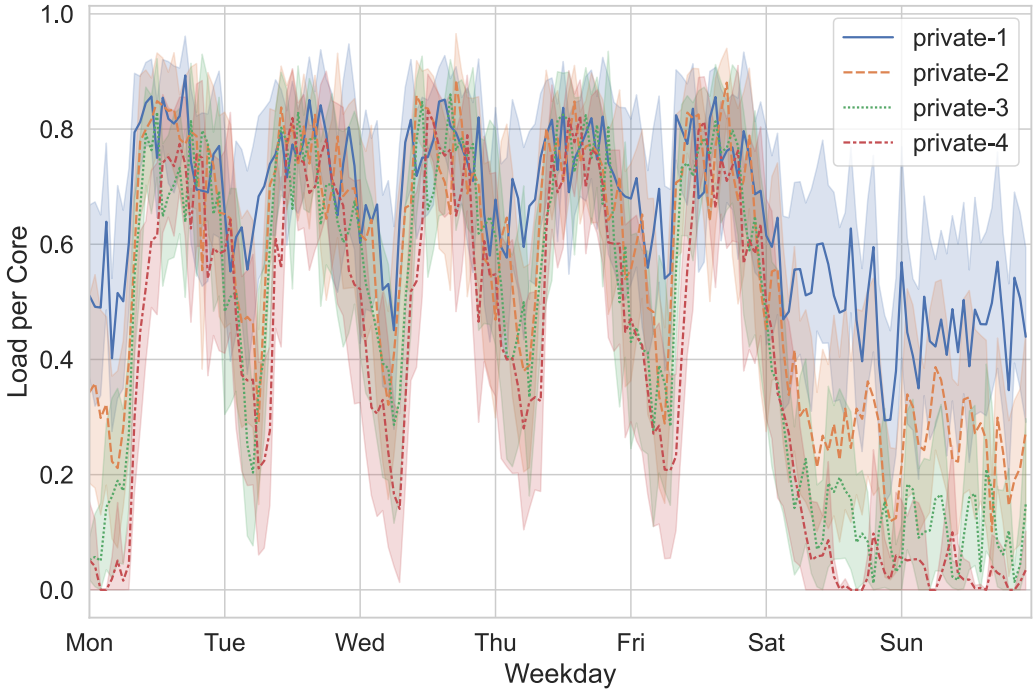
6.2.3 Random Start

From the preceding observations it shows that our biggest source of costs are simulation tasks waiting for long tasks to end so they can be executed. By using a dedicated worker only for simulation tasks we have successfully tackled this problem, but are facing a subpar utilisation of computing resources. In order to increase the utilisation we have developed a partly randomized algorithm. As long as no more than three workers are busy, tasks get executed immediately in order of the shortest runtime. If three of the four workers are already executing tasks, scheduling on the fourth worker works as follows:

- If there are simulations waiting to be scheduled, execute them immediately
- If there are no simulations waiting:
  - If none of the busy workers are expected to finish their execution during the next 15 minutes, do nothing.
  - If one of the busy workers is expected to finish its execution during the next 15 minutes, randomly schedule the shortest available task to a date between now and the expected finish date.



(a) Dedicated simulation worker algorithm.



(b) Random start algorithm.

Figure 20: Loads of the single workers over the course of a week. Tasks are released in a working hours distribution with an average load of 2.0.

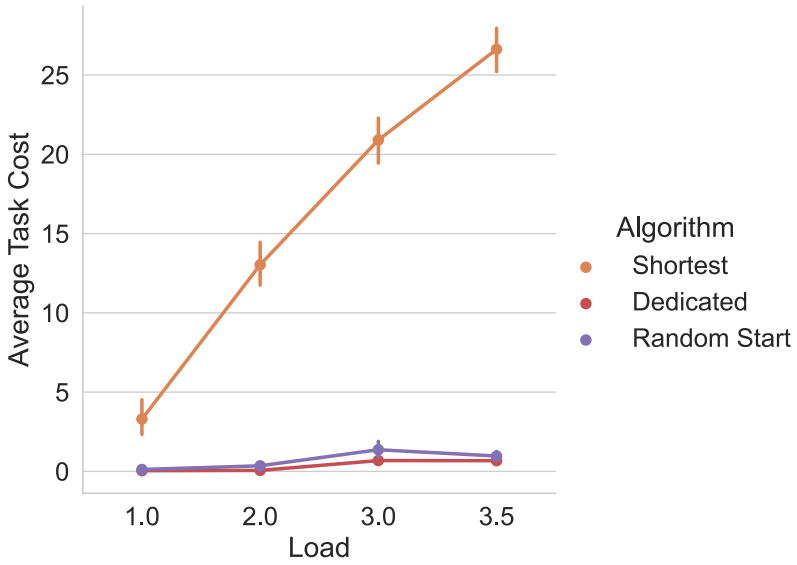
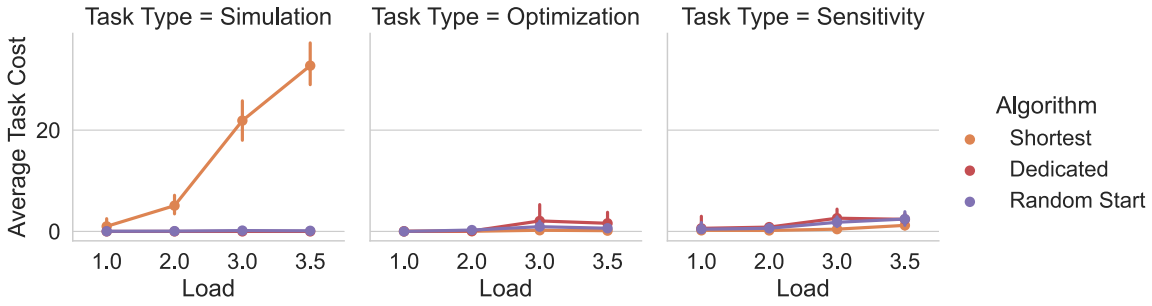
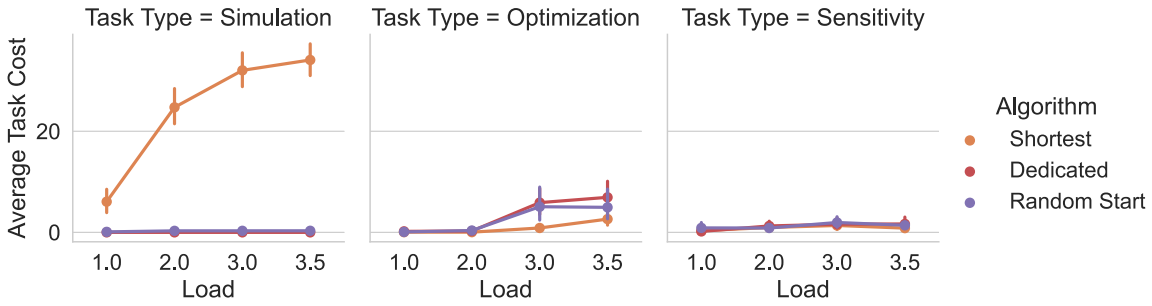


Figure 21: Overall average task costs of the dedicated worker and the random start algorithm compared to the basic shortest runtime algorithm in a working hours scenario.



(a) Uniform Distribution.



(b) Working hours distribution.

Figure 22: Average task costs broken down by task type. No significant difference can be observed between the dedicated simulation worker algorithm and the random start variant.

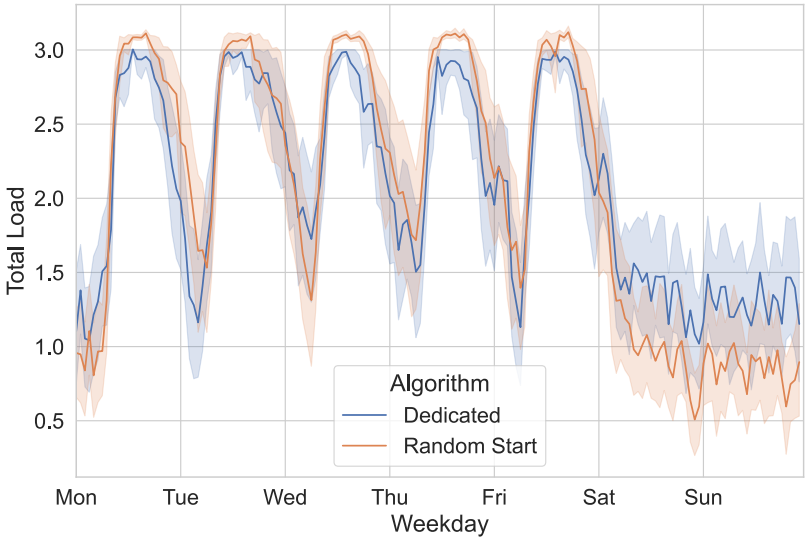


Figure 23: The accumulated loads of all workers compared for the dedicated simulation worker algorithm and the random start variant. Working hours task distribution and average load of 2.0.

  - If a shorter task arrives before the scheduled date or a worker finishes sooner than expected, discard the scheduled execution date

As can be seen in Figure 23, the algorithm indeed manages to increase the worker utilisations a little. But regarding the costs a positive effect cannot be observed. The total costs are higher, but in the same range as for the dedicated worker algorithm as shown in Figure 21. Looking at the individual average costs per task type (Figure 21) there is no significant decrease of costs showing for the optimizations.

6.3 Scenario 2: Public Cloud Resource Scaling

Using additional cloud resources for our computations brings two benefits: We can generally handle a higher number of tasks and we can absorb long waiting times of tasks during temporarily increased loads. The downside obviously is that we have to pay fees for every second a cloud machine is running.

Compared to scheduling only on the private server three additional factors have to be considered:

- To save costs, we would want the public cloud machines not to idle unnecessarily so the boots and shutdowns have to be managed.
- If a cloud machine is shut down it can not be used immediately. There is a delay (21 seconds in our case, see Section 5.1.2) that has to be included in scheduling considerations.
- Tasks take longer to finish their execution on the n1-standard-1 cloud machines compared to the private server.

Because we want to analyse the resource scaling particularly with regard to the flexibility it offers and the corresponding costs, we will restrict the following analyses to task release distributions that follow the working hours scenario.

6.3.1 Cost Threshold Estimation

Our approach is to scale resources on demand if we are suspecting that a task execution on the current resources would exceed a defined threshold c_t. When all workers on the private server are already executing tasks and we want to schedule further ones, we estimate based on our model when the next task on the private server will be finished. Then, we calculate the costs that would occur, if a task would be executed after that worker on the private server has finished its calculation. If these costs are higher then threshold c_t, we boot up a new cloud machine and schedule the task there. The algorithm is shown in Listing 4.

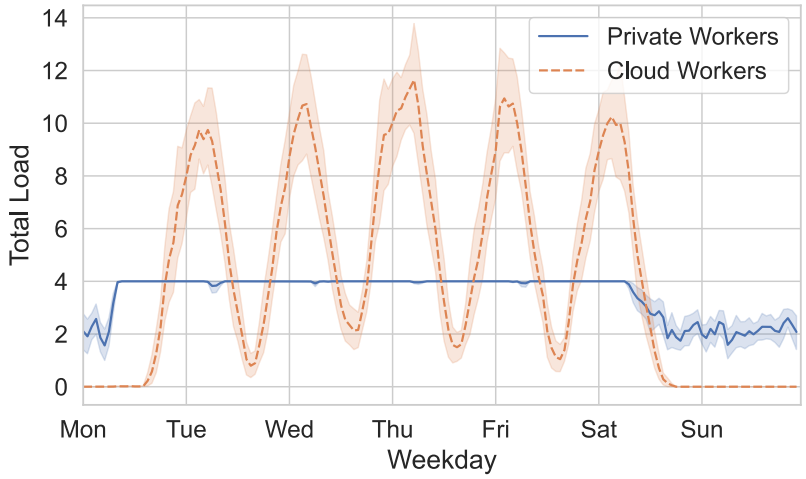
By adjusting the threshold parameter c_t we are able to set the "aggressiveness" of the algorithm which again shapes the balance between scheduling costs and server fees. In Figure 24 the utilisation of private server workers and public cloud ones is shown over the course of a week. It can be observed how a higher cost threshold leads to a delayed set in and an overall reduced utilisation of cloud workers. The drawback is the higher scheduling costs because of longer task waiting times. The comparison of scheduling costs and server fees for different load cases and cost thresholds is presented in Table 5.

A problem that emerges from Figure 24 is that this algorithm with a higher c_t value tends to delay the scaling decision unnecessarily and then produces higher scheduling costs because of the waiting times. In particular this happens if the task load is so high that the private workers are operating at full capacity even at night. In this case it would be appropriate to scale resources right away as there is no chance of executing the task on private resources in reasonable time.

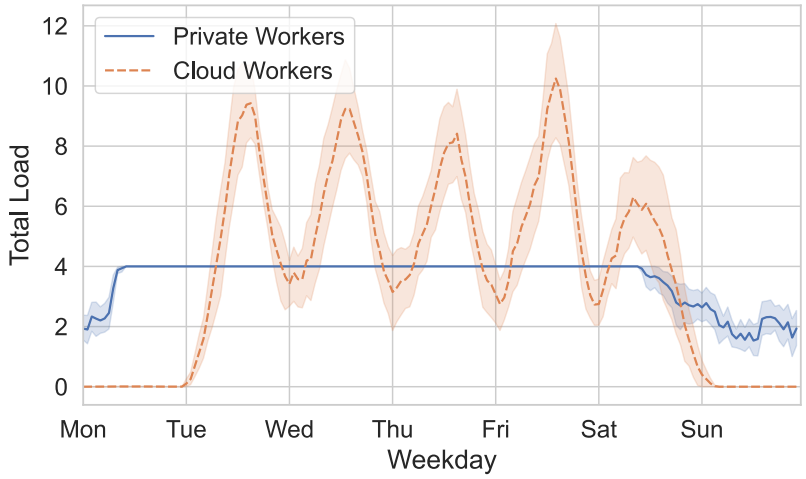
This is backed up by Table 5, where it shows that the loss on the scheduling cost is more explicit than the benefit of lower server fees. In the end however, the choice of threshold c_t is a budget decision.

Listing 4: Cost Threshold Scaling/Scheduling

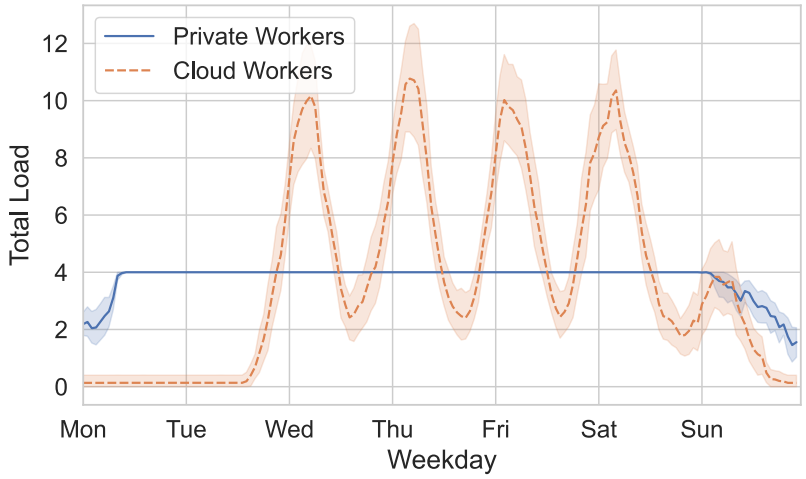
```
function schedule_costthreshold(workers, projects, tasks,
   currentDate, costThreshold):
  for w in workers['private']:
    if tasks['new'] and w.idle():
      t = tasks['new'].sort(key='runtime', 'ascending').pop
         (0)
      tasks['new'].remove(t)
      tasks['running'].append(t)
      t.schedule(w)

  if tasks['new']:
    nextEndDate = tasks['running']
      .filter(t.machineType == 'private')
      .sort(key='endEstimate'), 'ascending')).pop(0)
      .endEstimate
    nextTaskCosts = [(t, t.estimate_cost(nextEndDate)) for t
       in tasks['new']]
      .sort('ascending')
    for t, cost in nextTaskCosts:
      if cost > costThreshold:
        for w in workers['cloud']:
          if w.status == 'running' and w.idle():
            t.schedule(w)
            tasks['new'].remove(t)
            tasks['running'].append(t)
            break
          elif w.status != 'running':
            w.boot()
            t.schedule(w)
            tasks['new'].remove(t)
            tasks['running'].append(t)
            break
```



(a) $c_t = 5$



(b) $c_t = 10$



(c) $c_t = 20$

Figure 24: The utilisation of workers on the private server and on the cloud server compared for an average task load of 5.0.

	$c_t = 5$	$c_t = 10$	$c_t = 20$
Avg. Task Scheduling Cost	1.15	2.35	5.08
Avg. Daily Server Fee	0.52$	0.23$	0.08$

(a) Load: 3.0

	$c_t = 5$	$c_t = 10$	$c_t = 20$
Avg. Task Scheduling Cost	1.46	3.16	6.43
Avg. Daily Server Fee	6.16$	5.42$	4.93$

(b) Load: 5.0

	$c_t = 5$	$c_t = 10$	$c_t = 20$
Avg. Task Scheduling Cost	1.71	3.57	6.89
Avg. Daily Server Fee	13.26$	12.76$	11.75$

(c) Load: 7.0

Table 5: Scheduling cost and server fee comparison for different task loads and cost thresholds.

7 Conclusion

Web applications are a modern way of providing scientific computational services to users. The user's requested calculations are carried out by central servers and results are returned via a web interface. As calculations may require considerable amounts of computing capacity, the order in which requests are processed can be essential for the user experience. In the context of this thesis, we have studied this problem extensively for the particular case of the web applications for the Wind- and Sunflower simulation tools.

A distinctive feature of this application is that there are three types of tasks that are to be handled, which clearly differ in their computational efforts: simulations, optimizations and sensitivity analyses. We therefore have thoroughly studied their runtimes and developed a regression model that is able to estimate the runtime of a task based on its input parameters with high accuracy.

Furthermore, we have explored options to extend the available computing capacity by adding public cloud computing machines to our applications. Here, we found simple single-core VMs to be the best fitting solution due to their recorded performance, cost-performance ratio and the flexibility they offer.

To rate the performance of a scheduling and scaling algorithm we have defined a cost function that evaluates the completion time for each task, adjusted to our scenario. The function takes the absolute completion time into consideration as well as its ratio to the net processing time.

Finally, a case study was carried out to evaluate different algorithms in different scenarios. This was done using a Monte Carlo simulation tool developed during this thesis, which can resemble the computing architecture of the considered web application. First, a general scenario of fixed computing resources was examined, without the possibility of using additional cloud resources. Due to the nature and amount of the short scheduling tasks that have to be processed, it showed to be beneficial to prioritize the execution of these tasks. Even allocating resources for this is to be recommended, even though this results in a lower degree of capacity utilisation during busy time periods.

In case there is the possibility to scale up resources we have analysed the behaviour of a reactive scaling algorithm with different settings. This provides a foundation for estimating incurring server fees and corresponding scheduling costs.

7.1 Future Work

While the findings of this thesis provide a good baseline for task scheduling to use when the Windflower web application gets publicly available, a number of aspects remain to be analysed in the future.

- We have limited and adapted our studies and scheduling strategies to the Windflower simulations. While Sunflower calculations have similar characteristics, the results of this thesis should be validated for Sunflower.
- For our studies we have made assumptions regarding the behaviour of users. During productive operation of the web application it should be monitored how the distribution of incoming tasks is shaped and if it is dependant on other factors, e.g. weekday or time of release. Also, our presented cost function makes assumptions on the user's perception of task completion times. By performing a user study this can be evaluated further and adjustments to the cost function can be made.
- To speed up the computationally intensive calculations, the possibility of performing these on GPU hardware is worth to be evaluated. Due to their structure, GPUs perform significantly better than CPUs on certain calculations, which could be advantageous for Sun- and Windflower.
- The scheduling and scaling strategies presented in this thesis all work in a reactive way. If the scheduler was aware of patterns in the number and types of released tasks, the performance could be improved.
- It is supposable that the web application at some point will have different user categories with different expectations towards the quality of service, i.e. the task completion times. For the scheduling this would introduce additional task priorities that would have to be considered and analysed.

References

[1] Docker company homepage. URL https://www.docker.com/.

[2] Regions and zones, compute engine documentation, google cloud, . URL https://cloud.google.com/compute/docs/regions-zones.

[3] Vm instances pricing, compute engine documentation, google cloud, . URL https://cloud.google.com/compute/vm-instance-pricing.

[4] Chart: Amazon leads 100 billion cloud market. URL https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-provi

[5] International Energy Agency. Electricity consumption, world 1990-2017. URL https://www.iea.org/data-and-statistics?country=WORLD&fuel=Energy%20consumption&indicator=Electricity%20consumption.

[6] Yossi Azar, Naama Ben-Aroya, Nikhil R Devanur, and Navendu Jain. Cloud scheduling with setup cost. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 298–304, 2013.

[7] Luiz Fernando Bittencourt and Edmundo Roberto Mauro Madeira. Hcoc: a cost optimization algorithm for workflow scheduling in hybrid clouds. *Journal of Internet Services and Applications*, 2(3):207–227, 2011.

[8] Earl E Bomberger. A dynamic programming approach to a lot size scheduling problem. *Management science*, 12(11):778–784, 1966.

[9] Peter Brucker. *Scheduling algorithms*, volume 3. Springer, 2007.

[10] Michael L. Dertouzos and Aloysius K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on software engineering*, 15(12):1497–1506, 1989.

[11] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *2010 24th IEEE international conference on advanced information networking and applications*, pages 27–33. Ieee, 2010.

[12] Julian Düstersiek. Uncertainty quantification and multi-objective optimization of wind farms, 2019.

[13] Michael Eder. Hypervisor-vs. container-based virtualization. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, 1, 2016.

[14] Michael R Garey and David S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM journal on Computing*, 4(4):397–411, 1975.

[15] Volker Hamscher, Uwe Schwiegelshohn, Achim Streit, and Ramin Yahyapour. Evaluation of job-scheduling strategies for grid computing. In *International Workshop on Grid Computing*, pages 191–202. Springer, 2000.

[16] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Multilayered cloud applications autoscaling performance estimation. In *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, pages 24–31. IEEE, 2017.

[17] Marwa Maghnie. Simulation and layout optimization of offshore wind farms, 2019.

[18] Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. *Future Generation Computer Systems*, 48:1–18, 2015.

[19] Michael Pinedo. *Scheduling*, volume 5. Springer, 2012.

[20] Kirk Pruhs, Jiří Sgall, and Eric Torng. Online scheduling, 2003.

[21] Pascal Richter. *Simulation and optimization of solar thermal power plants*. PhD thesis, RWTH Aachen University, Aachen, 2017.

[22] Harvey M Wagner. An integer linear-programming model for machine scheduling. *Naval Research Logistics Quarterly*, 6(2):131–140, 1959.

[23] Yin yin Lo. Multi-step layout-optimization of turbines in offshore wind farms, 2020.