University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Piotr Blinowski
Student no. 439949

Szymon Mrozicki
Student no. 439978

Szymon Potrzebowski
Student no. 438683

Karol Wąsowski
Student no. 438800

Measuring and improving resource utilization in a Kubernetes cluster serving AI inference tasks

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
Janina Mincer–Daszkiewicz Ph.D.
Institute of Computer Science

Warsaw, June 2024

Abstract

Recent developments in large language models (LLMs) have popularized the usage of AI apps among the general public and have increased the strain on the limited resources available. These large models are often run on distributed systems composed of multiple physical machines and managed by the Kubernetes platform. In this thesis, we evaluate the current strategies available for scheduling AI inference tasks in Kubernetes clusters and try to optimize them based on recent work in the field. To achieve our goal of utilizing the available resources to their full extent, we look for an algorithm well suited for this scenario. To test our solutions, we provide an environment for simulating various scheduling strategies and analyzing how they behave under different circumstances.

Keywords

artificial intelligence, task scheduling, Kubernetes, deep neural networks, resource management

Thesis domain (Socrates-Erasmus subject area codes)

11.4 Artificial intelligence

Subject classification

CCS → Software and its engineering → Software organization and properties → Contextual software domains → Operating systems → Process management → Scheduling

CCS → Computer systems organization → Architectures → Distributed architectures → Cloud computing

Tytuł pracy w języku polskim

Mierzenie i poprawianie wykorzystania zasobów w klastrze Kubernetes obsługującym zadania wnioskowania sztucznej inteligencji

Contents

Introduction

Recent developments in large language models (LLMs) have popularized the usage of AI apps among the general public and have increased the strain on the limited resources available. Services like ChatGPT and DALL–E are seeing billions of users every month, and the growing neural networks behind their success make efficient resource utilization even more important. This is why the Polish start–up RadCode[1], which works on an AI models management platform (Autumn8[2]), tasked us to find an efficient solution to this problem.

Using and developing these models requires powerful machines and, as their neural networks have massive parallelization potential, most of them benefit from the GPUs' architectures. Running these operations at such a large scale and the growth of the number of parameters in the models makes it impossible to run on a single machine.

To remedy this, the industry has identified distributed data centers as a viable solution as they make it easy to scale and accommodate the growth, but pose a challenge as to efficiently managing the available resources. This is especially important as GPU prices are constantly rising.

A popular solution for distributed computing management is Kubernetes. It allows for the distribution of tasks between the available hardware, as well as scaling the infrastructure both horizontally (by introducing more machines) and vertically (by assigning more resources to these machines).

The main focus of this thesis are AI inference tasks processed by a Kubernetes instance. We will provide an environment for simulating various scheduling strategies and analyzing how they behave under different workflows and instance configurations. The simulator we are going to build will allow us to reliably compare numerous approaches and select the one most fitting for real–life workloads. It will also make testing and debugging incremental changes much cheaper than doing it on the actual system. The simulator will both simulate incoming requests and provide mock machines which will pretend to execute the tasks. It will also allow us to generate various statistics and their visualizations for easier comparison.

Then, using the simulator, we will evaluate existing scheduling strategies and look for potential improvements in the field. We want the scheduler to have an understanding of what resources are required by the incoming requests, and therefore be able to more efficiently assign them to physical machines. That way, less computing power will be lost and the cost of running GPU–heavy workloads will be reduced. Our implementation will be based on the current scheduler in use by Autumn8[2].

We will evaluate our solutions using said simulator and, since any publicly–available job traces for deep neural network (DNN) inference that suit our testing scenarios and meet our criteria do not exist, we will also resort to synthesizing our own, based on other publicly available traces, similarly to previous work in the field [3, 9, 10].

[1]`https://www.radcode.co/`

[2]`https://autumn8.ai/`

Best to our knowledge, there is no existing previous work comparing different scheduling strategies for Kubernetes clusters serving AI inference tasks.

This thesis consists of 7 chapters. In Chapter 1 we familiarize the reader with some basic concepts necessary to understand the thesis. In Chapter 2 we briefly describe previous work related to the thesis. In Chapter 3 we describe the architecture of the system we work on as well as request characteristics. In Chapter 4 we present the details of the implementation of the scheduling algorithms. In Chapter 5 we explain how we benchmarked our ideas and how we compared them with each other. In Chapter 6 we present and interpret the results of our testing. In Chapter 7 we summarize our findings and indicate the possible direction of future work.

Chapter 1

Background

The aim of this chapter is to familiarize the reader with concepts related to artificial intelligence and machine learning. We also introduce some technologies that the project is built on and briefly explain the parts that are relevant to our work. These are core concepts that this thesis builds on, and we will often refer to them in the following chapters.

1.1. Artificial intelligence and machine learning

Artificial intelligence (AI) is an approach to problem–solving that focuses on mimicking the human thought process. One of the most popular ways to do this is using machine learning (ML) algorithms, which improve their performance by learning from experience. This is achieved by training them on large amounts of data. The algorithms find patterns and dependencies in them and use this knowledge to predict the result for future inputs.

1.2. Deep neural networks

Deep neural networks (DNNs) are complex machine learning models that simulate the human brain by using multiple layers of connected neurons. The neurons of two consecutive layers are connected by weights, which are adjusted during training. Layers communicate with each other in a simple way: the output of one layer is multiplied by the weights, the result is applied to an activation function (e.g. ReLU, Sigmoid) and passed to the next layer.

1.3. Large language models

Large language models (LLMs) are massive neural networks (one of the most efficient machine learning methods) that can be trained to perform a variety of tasks. Their architecture is based on Transformers, which use a multi–head attention mechanism to process the input data and generate the response. Nowadays, this approach is the state–of–the–art solution in many fields of machine learning, including natural language processing, computer vision and reinforcement learning. The growing popularity of LLMs causes their size to increase, with the number of parameters reaching hundreds of billions. This makes them very expensive to train and perform inferences, which is why usually they are run on powerful GPU machines.

1.4. LLM inference

LLM inference is the process of generating a response by a large language model. Both the input and output of the LLMs consist of text and are represented by sequences of tokens – numbers that represent small parts of words. LLM inference can be divided into two phases: first processing the input, and then generating the output. The length of the output sequence is not known before the inference and its generation is autoregressive, which means that each token depends on the previously generated ones.

1.5. Request batching

Request batching in LLM inference is an optimization technique that combines multiple inference requests so that their output generation can be executed in parallel, leveraging the highly parallelizable nature of Transformers. There are two main types of batching techniques.

1. *Static batching* groups requests together before the inference, and all requests have to wait until all responses are generated before they can be sent back to the users.

2. *Dynamic batching*, first introduced by Orca [10], takes advantage of the autoregressive generation and groups the requests on the *iteration–level*. This enables higher resource utilization and lower latency, as requests with shorter outputs can be sent back to the user before the longer ones finish.

1.6. Cloud computing and GPU clusters

Cloud computing is the on–demand delivery of computing resources, such as virtual machines, databases or storage, over the Internet. External providers manage physical hardware so users can focus solely on designing their systems and applications. Most cloud computing services offer a pay–as–you–go pricing model, which means users pay only for the resources they reserve. Thanks to that, using cloud computing allows for cheaper serving of applications for both large and small companies, as they can easily scale up or down depending on the current demand. Resources available in cloud computing are often divided into clusters, which are groups of connected devices working together as a one system. GPU clusters are a type of clusters that are equipped with graphics processing units, crucial for running most machine learning applications.

1.7. Docker

Docker is an open–source tool offering OS–level virtualization, delivering software as lightweight containers. Containers run in isolation on a host machine, which is beneficial to the security of such systems. To make it easier to manage large, multi–component projects, each container manages its own set of dependencies and package versions. This also enables developers to quickly set up the necessary development environments, independent of the OS running on the host machine, and improves consistency between different devices.

1.8. Kubernetes

Kubernetes (K8s) is an open–source platform for computer systems that allows for easy scaling. It integrates with various container runtimes, which makes it suitable for managing large cloud computing clusters as well as the workloads running on them. It is widely used in data centers and can be both self–hosted or accessed through a commercial service. A K8s cluster consists of one or more Nodes (physical machines) with each one having the possibility of running one or more Pods (containerized applications). An API enables developers to communicate with a central scheduler that manages all the incoming tasks.

1.9. Service level objective

A service level objective (SLO) is an earlier agreed–upon target that the service should reach over a specified period of time. It is closely related to the service level agreement (SLA) which is the performance that the customer is guaranteed when using the service. They are defined by service level indicators (SLIs), which are quantitative metrics of some aspect of the service.

Summary

Now that we have introduced the basic concepts that this thesis is based on, we will look at and evaluate previous research in cluster scheduling algorithms.

Chapter 2

Previous work

This is a short description of the most important papers in the fields related to our thesis. It's divided into sections based on topics. There are papers strictly related to the method we are using (on which we base our work), as well as ones that are more general and give us a broader view on the problems we faced.

2.1. Cluster management

One of the main challenges when using a cloud computing platform is to manage the resources properly. It's important not to overpay for machines that are not being used. However, it's impossible to set up new machines just–in–time when the demand increases, because loading a model onto a machine takes significantly more time than serving an inference. Swayam [4] faces this problem by assigning one of four states to each Pod:

- `cold` – when the Pod is not active,
- `warm, not in-use, idle` – when it was in use but has been idle for a while,
- `warm, in-use, idle` – when it is available to serve requests,
- `warm, in-use` – when it is currently serving requests.

This solution tries to predict when to scale the cluster up or down based on current Pods' states. The scheduler performs scaling only if some request violates the service level agreement (SLA). The autoscaling algorithm uses a mathematical model to estimate the minimum number of machines required to restore SLA compliance. The model starts with the parameter $n = 1$ and, based on the request history, it tries to predict if n machines are sufficient to satisfy all given SLAs. If they are not, it increases n and verifies the SLAs once again. It repeats the whole procedure until all SLAs are satisfied.

2.2. AI inference systems

As AI models have been popular for a long time and many different algorithms and frameworks exist, there are many papers presenting different AI inference system designs.

One of the most popular works, Clipper [2], presents a prediction serving system that is divided into model selection layer and model abstraction layer. The former is responsible for choosing the right framework for a query, where the model abstraction layer provides a cache for frequent queries and a batching component. The batch size for each framework is set to maximize throughput and satisfy all service level objectives (SLO). The batching is briefly delayed for frameworks that benefit significantly from batched execution. Replicas of each model container are scaled to increase throughput, and each replica performs batching on its own.

Another major work in the field, Clockwork [3], focuses on serving DNNs and undermines the unpredictability of DNN inference executions. It shows that given deterministic performance of DNNs, it is possible to create a system that has predictable execution times. The system has a central controller that manages both loading and unloading of model weights and inference for all workers. The predictability of workers is achieved by limiting the choices that they make to a minimum. Therefore, each time only one of the three actions (load, unload, infer) is being executed by each worker to avoid any impact of internal scheduling/caching policies. The authors argue that many DNNs saturate the GPUs even with relatively small batch sizes, so the cost of no concurrency is overshadowed by the benefits from predictability and a better informed central scheduler with more detailed view of the system state.

The Clockwork scheduler maintains request queues for each model and batch size. To pick which requests to assign to a soon–to–be idle machine, Clockwork searches for a model and batch size such that executing the requests is the closest to the SLO deadlines (so the assignment is as tight as possible, maximizing throughput). Loading models is scheduled based on priorities that reflect the amount of unfulfilled work for each model. For unload actions, LRU (least recently used) eviction policy is used.

2.2.1. LLM inference systems

With the recent popularity of products such as ChatGPT or Copilot, which are based on LLMs, the optimization of LLM inference systems has recently been addressed by many papers. As the characteristics of LLM inference differ significantly from DNN inference in terms of complexity, the papers for both of the types focus on different parts of the design.

One of the most influential papers in the field, Orca [10], first introduced *iteration–level scheduling* and *selective batching*. Compared to request–level scheduling, iteration–level scheduling makes decisions about which requests to process after each single iteration of the token generations. On the level of a single iteration, it is not possible to batch all requests because of the differences in tensor sizes in the corresponding tensor operations. Selective batching is introduced to address this issue by applying batching only to subsets of operations that can be processed together, i.e. those that are in the same phase with the same amount of preceding tokens. Compared to previous systems, Orca's batched requests don't have to wait until all requests from the batch have finished, and therefore the latency of shorter jobs isn't influenced as much by longer jobs.

vLLM [5] finds that LLM serving is memory–bound and builds up on Orca by introducing a novel memory management mechanism. It substantially reduces the memory fragmentation caused by highly dynamic KV cache (a place in the GPU's memory where keys and values for previous tokens are stored during decoding) by adopting virtual memory and paging techniques well known from operating systems. The library, available an as open–source project on GitHub[1], achieves 24× higher throughput than previous state–of–the–art HuggingFace Transformers [8].

Splitwise [6] introduces splitting LLM inference into two phases: *the prompt phase* and *the generation phase. The prompt phase* is responsible for processing the request's prompt and

[1] `https://github.com/vllm-project/vllm`

generating the first output token, whereas *the generation phase* is responsible for generating all the proceeding tokens. Given the autoregressive nature of token generation, the second phase isn't as compute intensive as the first phase. Therefore, the distinction enables improvements in resource utilization because the configurations can be more aligned with the characteristics of the specific phase. Furthermore, the second phase can be executed on less expensive hardware while not impacting the system's latency.

The Splitwise algorithm uses a hierarchical, two–level scheduling system for *Cluster–level* (CLS) and *Machine–level* (MLS) scheduling. On the CLS level, JSQ (join the shortest queue) scheduling is used to assign machines to the requests. MLS uses FCFS combined with *mixed batching* [1] and tensor parallelism. Evaluations of this algorithm were done on production traces from Azure in a simulated setup. The traces consist of arrival time, input size and output size. The request inputs were randomized text of given size, as the work does not impact the quality of outputs. This design proved to increase the system's throughput with the same costs by a factor of 2.35.

2.3. Inference systems workloads

Serverless in the Wild [7] characterizes and publishes[2] Azure Functions production workload. It consists of the number of invocations of each function in 1–minute bins from a two–week–long time window. This workload is considered to be representative for ML inference systems and has already been used in other related papers [3, 11]. The main insights from the paper are:

- 75% of functions have execution time no higher than 10 seconds,
- 81% of functions are invoked at most once per minute, and less than 25% of functions are responsible for more than 99% of all invocations,
- inter–arrival time of most of the applications is hard to predict.

Splitwise [6] conducts a characterization of the LLM inference production trace from Azure. The trace is available on GitHub[2] and contains timestamps, number of input tokens and number of output tokens. The trace represents two use cases – *coding* and *conversation*, which have shown to have different token distributions, i.e. the median number of input tokens for these scenarios is 1500 and 1020 respectively, whereas the median number of output tokens is 13 and 129. The analysis also shows that most of the time is spent in the *token phase* and that the batching of the requests is very limited during that phase.

Summary

In this chapter, we showcased previous work related to our thesis that most of our work will be based on. In the next chapter, we go on to explain how it can be applied in our case and how it benefits our work.

[2]`https://github.com/Azure/AzurePublicDataset`

Chapter 3

System overview

In this chapter, we give an overview of the Autumn8 inference system's architecture and general request characteristics. We go into the inner workings of each of its subsystems and explain how it communicates with the other parts. We also present the implementation details of the scheduler already used by RadCode, that our work is based on.

3.1. General architecture

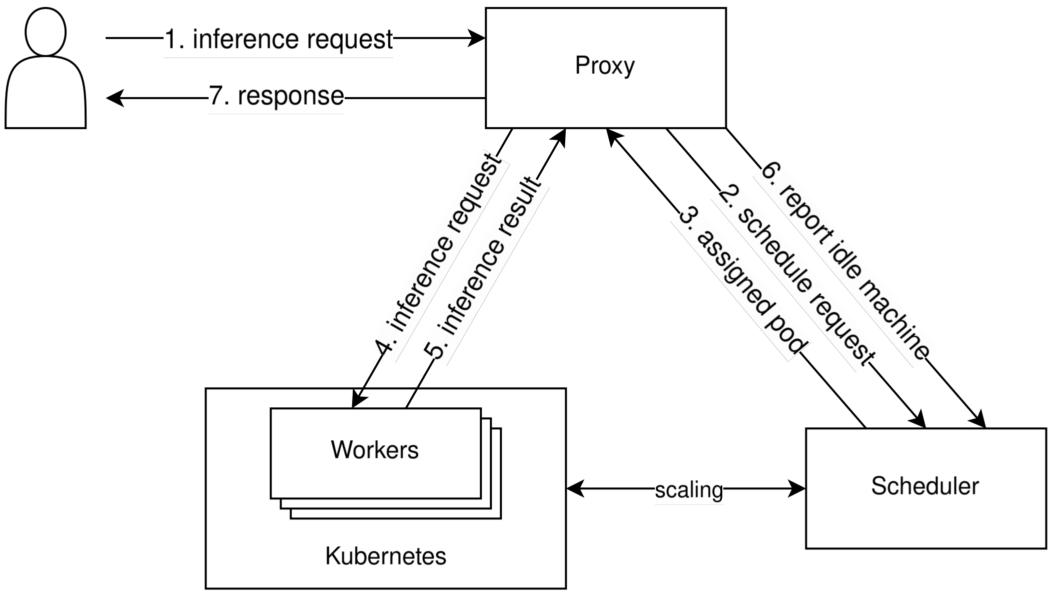

Figure 3.1: Autumn8 System Diagram

In our work, we consider the following system design, based on Autumn8's (Figure 3.1): the client sends a request to the proxy, which forwards it to the scheduler that decides when and where it should be executed. Then the scheduler responds with the worker's IP and the proxy sends the inference request to the machine. When the machine reports to the proxy that the inference is done, the proxy returns the inference results to the client and informs the scheduler, so that it can update its request queues and machine's state.

3.1.1. Kubernetes communication

The scheduler communicates with Kubernetes using its API to manage deployments, get information about the system's workers and their current state. Kubernetes does not provide any endpoints to get information about loaded models or inference processing on the workers. To trigger loading a model on a machine, the scheduler has to use the machine's API. It is also important that the scheduler does not start the inference on a machine directly, as it only responds to the proxy with the machine's IP when the inference is scheduled, and it is the proxy that sends the request to start the inference.

3.1.2. Scheduler

The scheduler provided by RadCode, similarly to Clockwork [3], assigns three types of jobs to the machines in the cluster:

`LOAD` – loading a specific model onto a machine,

`UNLOAD` – removing a specific model from a machine,

`INFER` – executing a model inference on a machine (the model has to already be loaded).

The jobs do not need to be scheduled immediately and can wait until one of the machines becomes idle. What is more, the scheduler can perform two types of actions that change the cluster state:

`SCALE UP` – adding a new machine of a given type to the cluster,

`SCALE DOWN` – removing a machine of a given type from the cluster.

This design gives the scheduler a full view of the system's state, which can greatly impact its decisions. A potential problem in this design is non–scalability of the scheduler, but it has already been described in Clockwork [3]. In that system, the scheduler was shown to become a bottleneck after the number of workers exceeds 110, which is more than RadCode's system is currently using. Therefore, the replication of the central scheduler is left for future work.

3.1.3. Proxy

The proxy is a service, which connects users with the internal part of the system. It receives inference requests from the users and then proceeds to ask the scheduler for the IP address of the Pod that is going to handle the request. When it receives the IP address, it sends the request to the appropriate endpoint of the specified machine (more details in Section 5.2) to handle the inference. It then asynchronously awaits the inference result and returns it to the user.

3.2. User request characterization

The system handles requests consisting of the following, and potentially some additional, parameters:

1. the machine learning model,
2. the model's input,
3. the request's SLA.

Each model has different characteristics. In the system, we distinguish 5 inference types:

1. Non–autoregressive with no batching,
2. Non–autoregressive with batching,
3. Autoregressive with no batching,
4. Autoregressive with static batching,
5. Autoregressive with continuous batching.

Summary

In this chapter, we gave a general overview of the system's architecture and described the roles of each of the components. We also provided the characteristics of requests handled by the system. Now, we are going to dive deeper into the inner workings of the scheduler component.

Chapter 4

Implementation

In Chapter 4 we present a set of data available for the scheduler in the cluster state to base its decisions on. We also discuss the details of different scheduling algorithms we chose for our comparison.

4.1. Cluster state

As Kubernetes does not provide any data regarding models or inference execution on workers, the scheduler needs to store some information about the actual cluster state. We decided to collect details such as:

1. For each machine type separately:
    - a list of active machines,
    - a list of machines that are scaling up,
    - a list of machines that are scaling down,
    - a list of requests on this type of machine.
2. For each machine:
    - its IP address,
    - whether it is idle,
    - whether it is loading a model,
    - whether it had processed any requests after loading a new model,
    - the time of the last inference,
    - the duration of the last inference,
    - a list of loaded models,
    - machine type,
    - current status: `STARTING`, `READY`, `INVALID`, `CRASHED`, `SLEEPING`, `TERMINATING`, `TERMINATED`,
    - last assigned request.

This not only allows us to store more details about the Pods (as mentioned earlier), but also allows the algorithm to react to incoming requests quicker, as it does not require any network communication with Kubernetes. The available workers are only refreshed periodically, which further contributes to decreasing the required network communication.

However, this solution can lead to problems with invalid caches, so it is important to remember about the necessary fail–safes.

4.2. Schedulers

4.2.1. Baseline scheduling algorithm

The Baseline scheduling algorithm, provided by RadCode, has three phases when running INFER actions. In each phase, the scheduler iterates over all requests ordered by arrival time and then selects a machine which is idle and can process the request (based on criteria dependent on the phase) in a Round Robin fashion.

In the first phase, the scheduler tries to schedule only starved requests to idle machines, in the second phase, it schedules requests that can be inferred using continuous batching onto machines which are already processing requests of this type and, in the last phase, schedules all requests onto idle machines that have the same model loaded into memory.

When the request's model is not loaded onto any machine, the scheduler scales it up – it iterates over all machines and looks for an idle one. If it finds a valid candidate, the scheduler removes the currently loaded DNN from the machine and loads the new model onto it by executing a LOAD action.

4.2.2. Modified Baseline algorithm

We implemented some modifications to the Baseline scheduler:

1. We added a condition to the LOAD action that prevents it from continuously loading new models without performing any inferences, which occurs when the load is very high and requests use different models. After this change, the scheduler looks for an idle machine that has already performed an inference after loading a new model or was idle for long enough.
2. We added memory management to the scheduler, so that it keeps track of how much memory is taken up on each machine by the loaded models. This allows us to load multiple models onto a single machine, which is especially useful when performing DNN inferences on smaller models.

Whenever we refer to the Baseline scheduler in the following chapters, we always mean the version including these modifications.

4.2.3. Round Robin (RR)

The scheduler first arranges models in a circular list and assigns a *quanta* (a time period for which the model is loaded onto a machine) to each of them. Then, it loops through the models from the list and schedules a load on an arbitrary idle machine or waits for the first machine that finishes its quanta, in case all machines are busy. For INFER actions, we use the same strategy as described in Section 4.2.1. What is more, we propose some additional improvements to this algorithm. The improvements fine–tune the algorithm to work better with DNN inferences.

The following improvements are implemented on top of each other, so whenever we refer to one of the latter changes, we assume the former are also included.

Changed quantum logic

The scheduler assigns the quanta to a model not only after it is loaded, but also after it performed any inference. This change lengthens the time popular models stay loaded on the machines and reduces their loading frequency.

Added starvation tracking

When many requests arrive at once, all requiring a model from the end of the queue, it will take a long time for the model to be loaded and, as a result, for the requests to be completed, thus violating their SLAs. To prevent such situations from occurring in our scheduler, we sort the models queue by the number of requests that exceed 60% of their SLA and in case of a draw, we load the least loaded model.

Quanta adapted to different models

As loading times vary between different models and some are more costly to load, we want the bigger ones to stay loaded for a longer time. To achieve this, we assign them longer quanta compared to the smaller ones. Additionally, when unloading a model, we check how many inferences it has performed (how well it made use of its quanta) and adjust the model's quanta depending on its request frequency.

Sorting requests by their deadline

When the Baseline scheduler assigns requests to machines, it begins by serving the ones that have the earliest arrival date (FCFS). To reduce the number of requests that violate their SLAs, we sort them by the earliest deadline before assigning. As a result, the requests that have the closest deadlines are being completed before the other ones. This is safe and will not cause request starvation, because they are required to include a deadline that is a time in the near future.

4.2.4. Clockwork's scheduling algorithm

The Clockwork paper's [3] scheduling algorithm maintains a separate request queue for each model. The algorithm composes requests from the same queues into *strategies* so that each *strategy* is parametrized by a model and batch size. Then, when an INFER action is to be scheduled, the *strategy* which has the corresponding model loaded on a soon–to–be idle machine and which is the closest to its deadline is picked.

In parallel, the scheduler maintains *load* and *demand* statistics for each model, which are used to calculate priorities for scheduling a LOAD action for a given model. The *demand* statistic is the remaining work (in seconds) for each of the models, which can be computed using the profiled data of inference execution times. The *load* statistic is calculated for each machine and corresponds to how much of the work that machine will need to handle (assuming that the workload is split evenly across the machines). Then, the priority of a given model is defined as its demand subtracted by how much of it can be served given the current state of the cluster. Finally, the scheduler periodically picks the model with the highest priority and loads it onto some idle machine. For UNLOAD actions, LRU policy is used.

Our adjustments

As in our setting (Section 6.1) it is not always possible to satisfy all the SLAs, we made an adjustment to the loading policy in order to prevent the scheduler from infinitely reloading the models and therefore stopping the system from performing any work. To achieve this, instead of LRU policy, for UNLOAD actions we pick the model for which the priority after unloading it is the lowest and decide to perform the action only if the resulting priority is lower than the priority of the model for the currently pending LOAD action.

Summary

In this chapter, we presented the scheduling algorithms that our work will be based on. In the next chapter, we will introduce our testing environment – the simulator we built – and present our findings, adjustments and results.

Chapter 5

Evaluation

In this chapter, we explain how we measure the efficiency of the solutions discussed in Chapters 3 and 4. We present the architecture of the K8s simulator we built that allowed us to frequently benchmark changes while keeping operating costs low.

5.1. Simulator architecture

The simulator is built in Python around a microservice architecture. Python was suggested by our client as it has a wide availability of libraries with thorough documentation, which encourages fast development. Figure 5.1 presents a general overview of how the system is designed and is helpful to understand the latter description.

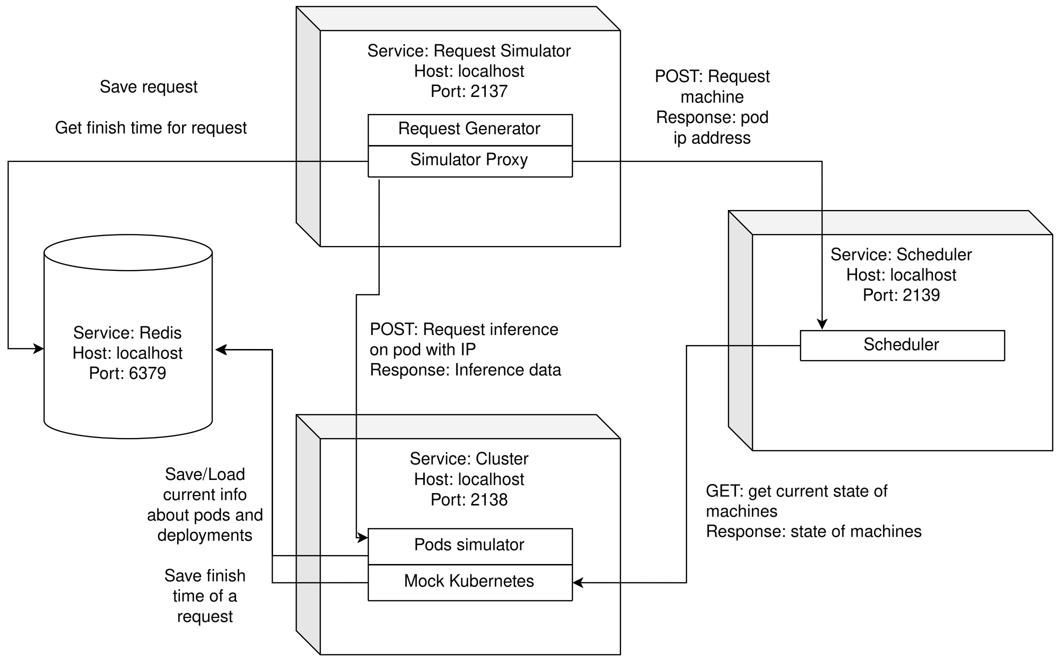

Figure 5.1: Overview of the simulator

We chose the microservice architecture, as it has two advantages for our use–case – it allows us to simulate the real–world network communication, as well as solving Python's multi–threading limitations. Furthermore, each microservice runs a separate codebase, on its own docker instance, which makes it easier to develop multiple components simultaneously, independently of each other.

It is worth noting that the scheduler microservice is not part of the simulator, but rather an external project that can be connected to the system for testing. This makes it possible to use the simulator with any implementation of a Kubernetes scheduler that uses the same protocol to communicate with K8s and the Pods.

All simulations run in real–time. As we are not interested in the inference results themselves, we only simulate loading models and inferences by calculating how long a machine would stay busy, based on our statistics. The inference result is always the same, default value. This makes it possible to run the simulation on a standard personal computer. We have made the decision not to create a separate time instance for the simulation, as it could interfere with the algorithms used to schedule tasks. The scheduling decision may sometimes take a significant enough amount of time, that it would have a noticeable impact on test results. The strategy is optimized for real–world data, and arbitrarily altering run conditions would invalidate the results. Due to this decision, it takes a longer period of time to evaluate changes to the scheduling algorithms, but this is not an issue as running those tests does not consume any expensive resources. The details of how we do it will be explained in the next section.

5.2. Implementation details

In this section, we will go over the components of the system discussed in the Section 5.1 and explain their implementation. The implementation of the scheduler component has already been discussed in Chapter 4.

Each component exposes an API built in Python using FastAPI. FastAPI is a popular framework for exposing program functions to a network that makes use of Python's asyncio library, introduced in Python 3.4. It allows implementin multiple endpoints with little overhead.

Redis

Redis is a high–performance, in–memory data store, which we chose to store the state of the K8s cluster. It stores the data as key–value pairs, which allows for easy access. The motivation behind choosing Redis for storage was its fast access to data for very large data sets and thread–safety. Since the system is built on the microservice architecture, it would pose a significant challenge to synchronize the different services. Redis solves this issue in our case.

Request simulator

The request simulator is responsible for generating tasks that the scheduler will have to process. We implemented two operating modes for this component:

- *Retraced* – the user may provide a trace of real–world tasks from a cluster as input. The data must contain at least a timestamp of when the request was received, the time it took to complete, and the machine type it was running on. The request simulator will follow those traces and generate them in the same order and at the same time interval as they appear in the trace.

- *Generated* – the user may provide a distribution that represents the number of requests at different times of the simulation. The request simulator will use that curve to generate arbitrary requests to the scheduler.

These two operating modes allow us to both evaluate every solution on arbitrary data the algorithms have not yet been optimized for and testing against known challenging scenarios and weaknesses from previous versions, to prevent regression.

The request simulator generates a task queue on launch and then only processes them later. Each task is first sent to the scheduler, and it responds with a Pod IP. Then the task is passed on to the Pod corresponding to the selected IP address to process. The request simulator records the time it took to schedule and complete the request and stores those statistics in the Redis data store.

K8s cluster

The simulation of the K8s cluster consists of two parts: simulating K8s API and the Pods themselves. The K8s API is public, which allowed us to easily and accurately mock the necessary endpoints. The endpoints exposed by our K8s simulator are:

GET `/apis/apps/v1/deployments` – returns a list of the available deployments and their details, which include the number of online Pods and the hardware they offer.

GET `/api/v1/pods` – returns a list of the available Pods, together with their IP address, ID, running status and hardware details.

GET `/apis/apps/v1/namespaces/default/deployments/{name}` – returns the details of a single deployment.

PATCH `/apis/apps/v1/namespaces/default/deployments/{name}` – allows the scheduler to alter the parameters of a deployment, for example increase or decrease the number of available pods.

We also mock the endpoints available for each Pod, that are not a part of the K8s API, but rather a part of our internal API:

POST `/preload/{machine_ip}/{tracking_id}/{model}` – begins the process of loading the `model` on the Pod located at `machine_ip`; the request's `tracking_id` is user defined and allows the Pod to send an update, when the loading has finished.

POST `/inference/{ip}/{id}` – runs the specified inference on the pod located at `ip` and identified by `id`.

It is important to note, that in our simulation, the Pod's IP address is always equal to its ID. It doesn't interfere with the functionality in any way, while simplifying the entire process. As mentioned before, request IDs are generated by the scheduler and used for internal tracking of the request's status.

5.3. Metrics

To compare different implementations of the task scheduler, we had to choose the metrics and statistics that should be gathered by the simulator. We settled on these three that were most significant to achieve our goals:

- Total model loading time – to minimize the resources wasted, we want to minimize the time spent loading models and not doing actual work; this metric is measured as the sum of all model loading times. It suits our scenario well, as we have a constant set of active machines,
- Throughput and latency – the system is optimizing costs, but it cannot be at the expense of providing a worse user experience; throughput is measured as the average number of requests processed per minute and latency is the average time a request has spent waiting to be scheduled,
- Deadline compliance – all requests coming into the system are required to be processed by a certain time; deadline compliance is measured as the average number of deadlines missed per hundred requests processed. We also consider the maximum latency achieved by the scheduler during our simulations.

To summarize, the first metric allows us to measure how well we achieve our goal, while the others make sure, that the service stays reliable and provides a good user experience.

5.4. Traces

For our evaluations, we use the following DNN models, which we consider a representative group of models served in the inference server:

- Stable diffusion v2,
- MobileNet–V1,
- SqueezeNet,
- VGG–19,
- DenseNet–169,
- Whisper,
- Resnet18,
- VGG–16,
- Inception–ResNet–v2.

To make the simulations as similar as possible to a real–world scenario, we run them based on data from different sources:

- internal statistics from Radcode's running instances of Autumn8,
- publicly available traces from Azure (see Section 2.3).

For each simulation, we replay 30 minutes of the Azure workload, which is grouped into one minute bins. We assign a model to each Azure Function and then generate the arrival times, similarly to *Serving DNNs in the Wild* [11], using the uniform distribution within each bin. Load and inference times are generated based on the internal statistics provided by RadCode.

Similarly to Clockwork [3], each model is assigned a fixed number of Azure Functions' workloads. As in our work the models vary significantly in terms of model sizes and inference times, we find it important to make these assignments reflect the actual demands for those models. Therefore, we conduct multiple evaluations for different distributions of load across the models.

5.5. Visualizations

To have a better understanding of the simulation setup details and schedulers' performance, our simulator gathers statistics and generates various plots visualizing the data. We decided that to analyze the simulations, we need visualizations of:

- Workload characteristics
  - request arrivals distribution (Figure 5.2),
  - workload distribution across models (Figure 5.3).
- Request processing statistics
  - cumulative distribution function (CDF) of job completion times (Figure 5.4),
  - the number of requests not meeting their deadlines (Figure 5.5) – The data is grouped into one minute bins, and it includes the total number of request arrivals and the number of requests violating their SLAs when using a specified scheduling algorithm,
  - total time spent on loading models onto all machines (Figure 5.6).
- Machine usage (Figure 5.7) – This kind of plot is created for each simulated machine. The x–axis represents time, an intense color indicates that the given model (listed on the y–axis label) was loaded onto the machine, a faint color indicates that the model was loading and the black blocks represent periods when inference was performed on the given model on the machine.

These plots helped us grade the performance of different scheduling algorithms and improve it in our testing setup. The plots below are based on the workload we performed our final tests on.

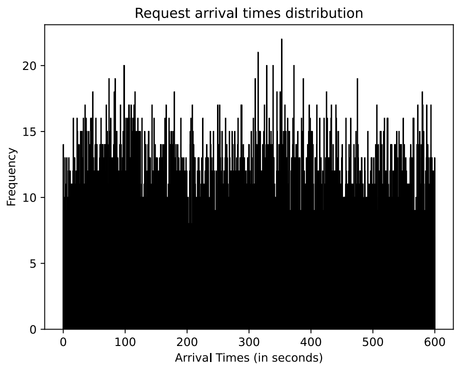

Figure 5.2: Request arrival times distribution

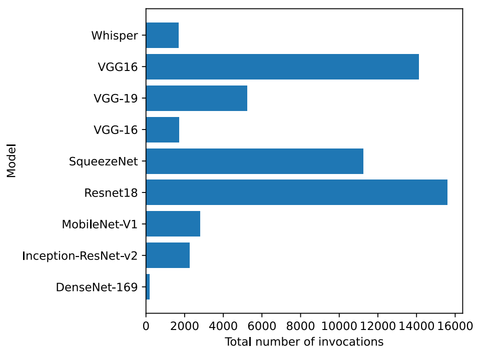

Figure 5.3: Workload distribution across models

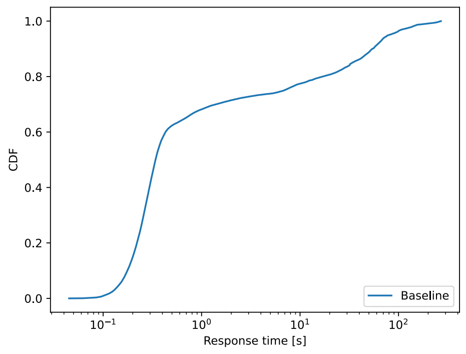

Figure 5.4: Cumulative distribution function of job completion times

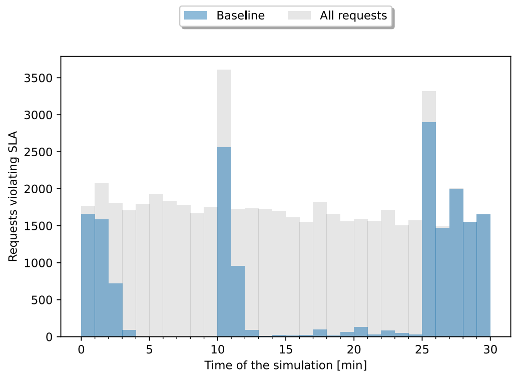

Figure 5.5: SLA violations in time

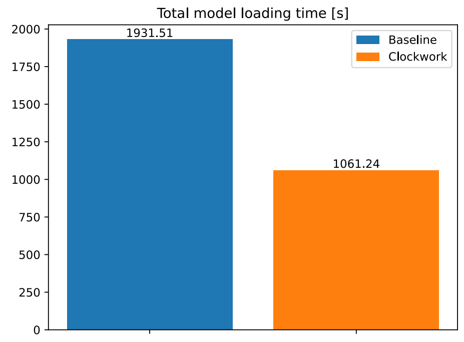

Figure 5.6: Total time spent on loading models onto all machines

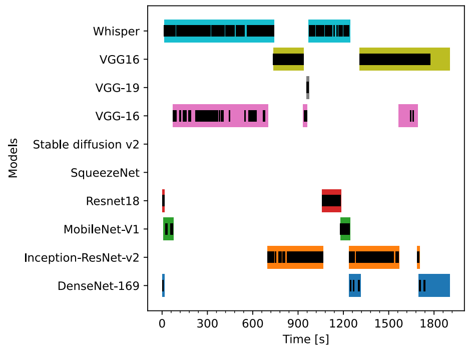

Figure 5.7: Models and inferences on a single machine in time

Summary

In this chapter, we described the architecture of the simulator, and went into the details of each of the simulator's components. We also showcased the statistic types collected by the simulator and presented the plots we use in the following chapters. In the next chapter, we describe our testing setup and present the obtained results.

Chapter 6

Results

In this chapter, we share details about our testing setup and present the testing results. Then, we compare the scheduling algorithms introduced in Chapter 4 based on the metrics and statistics collected by the simulator.

6.1. Testing setup

The results presented in this section were gathered on a setup with a homogenous cluster. We also wanted to limit the capabilities of a single machine to prevent the scheduler from loading all the models onto one worker. This is important, as we also want to compare the scheduling algorithms in terms of their loading and unloading policies. Therefore, we limited to two the number of models that can be loaded onto a single machine at the same time.

The plots and data we provide in the following sections were gathered by running all the scheduling algorithms described in Chapter 4 with the same workload. We provide plots of the requests' arrival times (Figure 5.2) and the distribution (Figure 5.3) of the testing workload.

6.2. Initial testing

Our simulator allowed us to test the Baseline and very quickly identify a problem of infinitely switching models on machines without performing any inference tasks, that surfaced only after prolonged testing. We also found an issue with using invalidated information about the requests in the scheduler, arising from a bug in the handling of asynchronous operations. Fixes for both of these issues have been included in the modified Baseline algorithm.

The bugs we found in the Baseline algorithm would have been very difficult to uncover without the ability to test it under a heavy load. Furthermore, doing this locally enabled us to perform the tests quickly without any additional costs or other complications arising from using an actual deployment.

6.3. Comparing Baseline, Clockwork and Round Robin

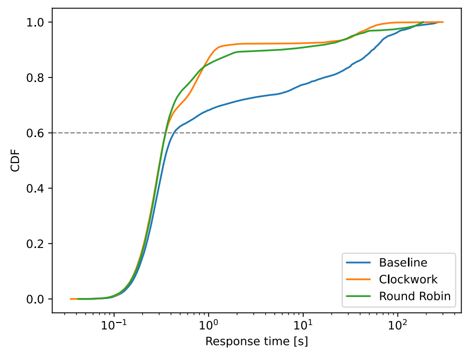

Figure 6.1: CDF comparison of Baseline, Clockwork, and Round Robin

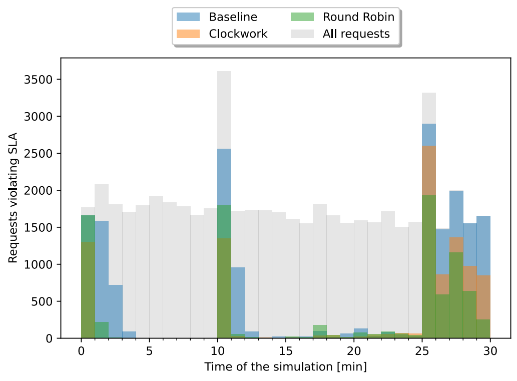

Figure 6.2: SLA violation in time comparison between Baseline, Clockwork and Round Robin

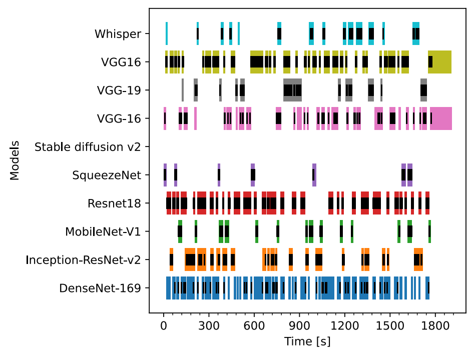

Figure 6.3: Machine usage when using Round Robin

The Baseline algorithm proved to serve well under a stable load. In Figure 6.2 it can be observed that the SLA is met for a large majority of the requests, when the rate of incoming requests is steady at around 1700 per minute. However, when a spike in the number of incoming requests occurs in the system's workload, the SLA violations dramatically increase, which is a result of the reactive nature of the algorithm.

Clockwork's algorithm shows a considerable improvement around the 40–th percentile of the request response times (dashed line in Figure 6.1). The improvement, when compared to Baseline, comes from the fact that the scheduler proactively assigns LOAD actions as it sees the spike of incoming requests and therefore it reacts to dynamic workloads quicker. In Figure 6.2 we can see that Clockwork's SLA violations are noticeably lower than those of Baseline when a spike occurs.

Round Robin performs similarly to Clockwork in terms of request response times, but we do see room for improvement when analyzing a single machine's usage statistics on Figure 6.3. We see that models are frequently loaded and unloaded. This is expected due to the fact, that the assigned quanta are relatively short and cause the scheduler to unload the model, regardless of the number of inferences completed.

6.4. Round Robin with improved quantum logic

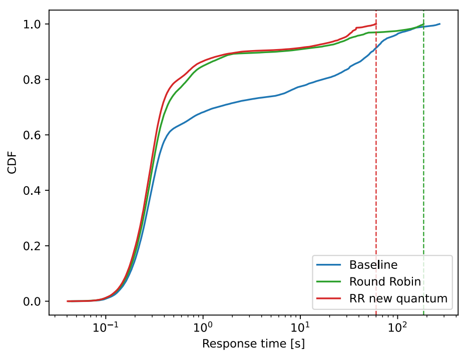

Figure 6.4: CDF comparison of Baseline, Round Robin and Round Robin with improved quantum logic

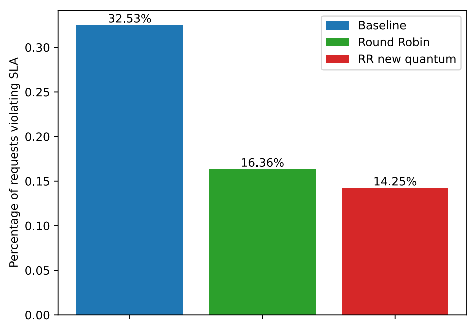

Figure 6.5: SLA violation comparison of Baseline, Round Robin and Round Robin with improved quantum logic

After modifying the quanta assignment logic (described in Section 4.2.3) we can see (Figure 6.4) that although there is not much improvement in most of the requests' latency, the processing time of the longest requests has dropped by about 70% (from around 200s to 60s). Additionally, there is a slight drop in the number of requests violating their SLAs (Figure 6.5).

6.5. Round Robin with starvation tracking

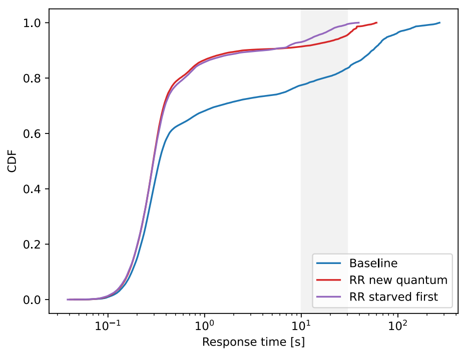

Figure 6.6: CDF comparison of Baseline, Round Robin with improved quantum logic and Round Robin with starvation tracking

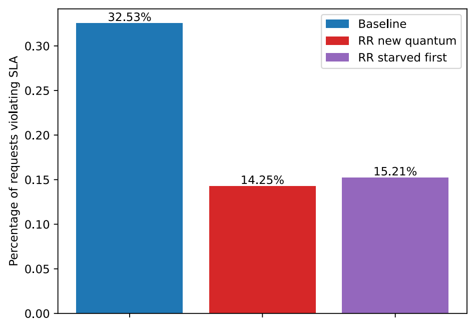

Figure 6.7: SLA violation comparison of Baseline, Round Robin with improved quantum logic and Round Robin with starvation tracking

In Figure 6.6, we can see a visible increase in the number of requests finishing under 30 seconds (the area where the improvement is visible is marked in gray). After introducing this change (described in detail in Section 4.2.3) we also observe the overall number of requests violating their SLAs increasing slightly. This is due to the fact that even though more requests violate theirs SLAs, the average violation length is shortened, which is the reason we observe further improvements in lowering the maximum latency in Figure 6.6.

6.6. Round Robin with quantum per model

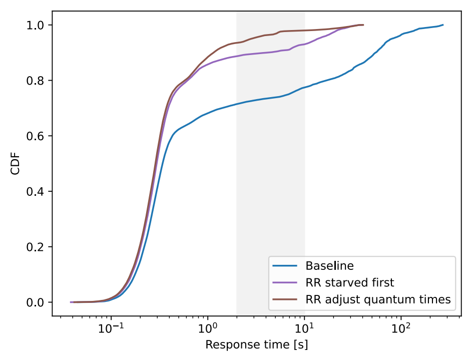

Figure 6.8: CDF comparison of Baseline, Round Robin with starvation tracking and Round Robin with quantum per model

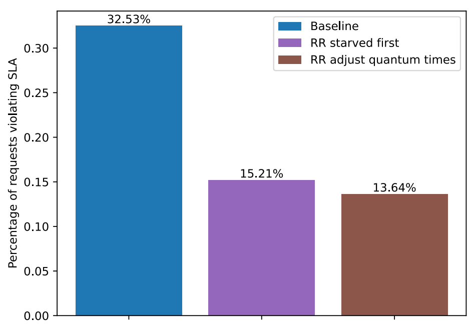

Figure 6.9: SLA violation comparison of Baseline, Round Robin with starvation tracking and Round Robin with quantum per model

After introducing per–model quanta length calculation, based on the model's size and the number of inferences it performed, to the scheduler's logic, we notice (Figure 6.8) that the number of requests with a latency between 2 and 10 seconds has increased (marked in gray). Applying this change also increases the number of requests meeting their SLAs (Figure 6.9) by 1.6 percentage points.

6.7. Round Robin with deadline compliance

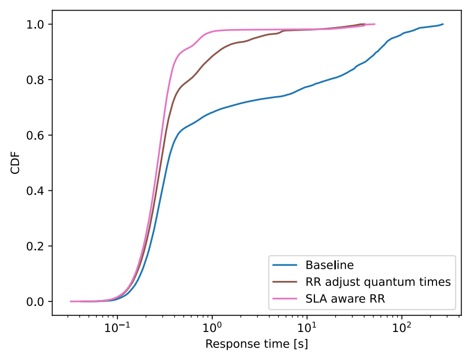

Figure 6.10: CDF comparison of Baseline, Round Robin with quantum per model and Round Robin with deadline compliance

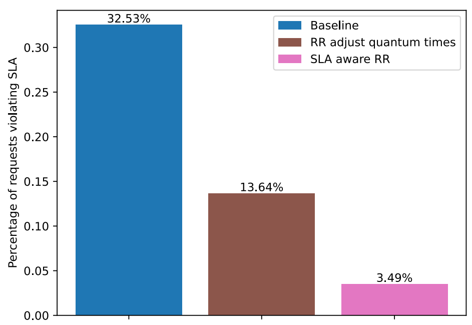

Figure 6.11: SLA violation comparison of Baseline, Round Robin with quantum per model and Round Robin with deadline compliance

Our final modifications to the Round Robin algorithm have significant impact on the system's overall latency. Judging by the CDF alone (Figure 6.10), we could assume that this algorithm outperforms both Baseline and Clockwork algorithms. Furthermore, the SLA violation plot

(Figure 6.11) displays another advantage of applying the changes to the algorithm, as we see the biggest improvement yet in the number of requests meeting their SLAs.

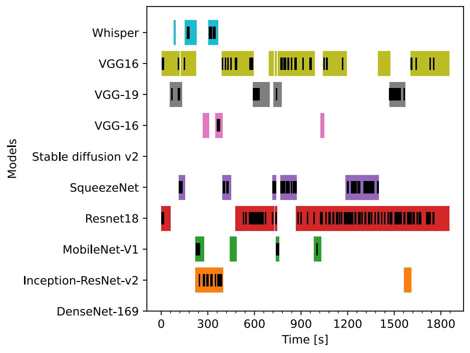

Figure 6.12: Machine usage when using our improved Round Robin

As mentioned in Section 6.3 and presented in Figure 6.3 we had an issue with models being frequently loaded and unloaded. After applying all of the above changes to the initial Round Robin algorithm, we see that the issue is no longer present in Figure 6.12. We can also observe that frequently used models stay loaded for a longer time after the last inference, thanks to the dynamic quanta calculation logic.

6.8. Results summary

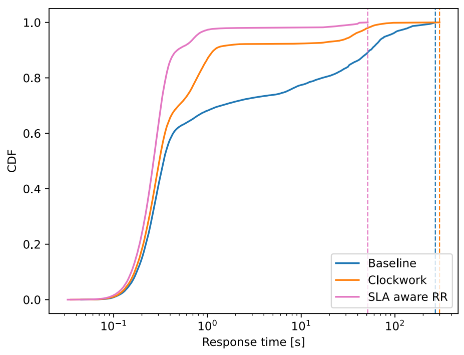

Figure 6.13: CDF comparison of Baseline, Clockwork, and Round Robin with deadline compliance

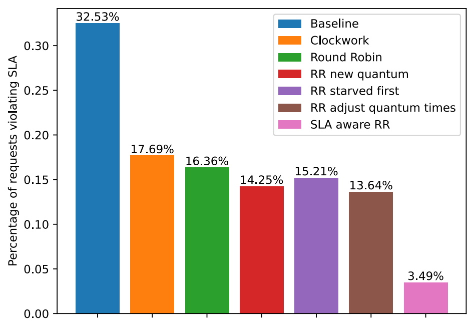

Figure 6.14: SLA violation in time comparison of all presented algorithms

From our experiments, we concluded that the scheduling algorithm has a significant impact on the system's responsiveness and SLA compliance. Implementing the Round Robin scheduler and introducing the previously described changes showed significant improvements in the number of requests finishing before their deadline (Figure 6.14), as well as both the average and maximum response latency (Figure 6.13).

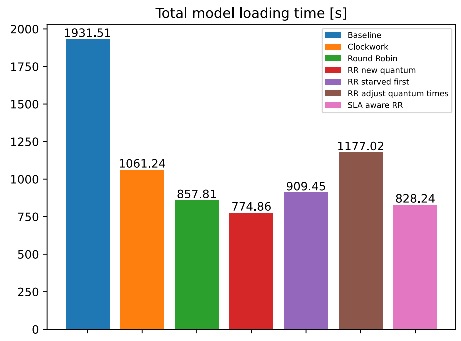

Figure 6.15: Total model loading times for the tested algorithms

Figure 6.15 presents the total mode loading time metric for the tested scheduling algorithms. We want to remind the reader that this metric shouldn't be interpreted apart from the other ones. The objective of this research was to find a balance between the time spent managing and loading the models and complying with the SLA. From the above figures, we conclude that the modified Round Robin algorithm satisfies this objective, by having high SLA compliance during our spiky workload, fast response times and relatively low total model loading times.

Additionally, we can see that Clockwork's ideas also pose significant improvements over the Baseline scheduler in terms of total model loading times and SLA compliance.

Summary

In this chapter, we presented the details of our testing setup and showcased the results of our tests. We then showed how the changes we made resulted in improvements to scheduling. In the next chapter, we will summarize this thesis and discuss potential directions of future work.

Chapter 7

Conclusions

In this chapter, we summarize the work we have done and discuss potential directions of future work. We believe that our research could be beneficial to future developments in scheduling algorithms and should be considered in future work.

7.1. Thesis summary

The goal of the project was to improve scheduling algorithms used in K8s clusters serving AI inference tasks. We began by researching other papers in the field and collecting ideas for our improvements. What we found, was that no research had been previously dedicated to comparing different scheduling strategies.

Our next step, was building a simulator of a DNN and LLM inference system to test the ideas we came upon during our research. It simulates both the K8s cluster itself and provides a proxy microservice that generates the requests. Additionally, it gathers statistical data and creates useful plots about the simulations. The simulator allowed us to easily test different scheduler implementations, managing the K8s cluster.

Using the simulator, we were able to evaluate and debug the Baseline scheduler, as well as to benchmark the algorithm proposed by Clockwork [3]. Later, we introduced a new Round Robin algorithm and implemented crucial changes, that fine–tuned it to give better results when scheduling DNN inference tasks. These changes included adjustments to quanta assignment logic and special starvation prevention measures.

We found that our Round Robin implementation outperformed both the modified Baseline and Clockwork, managing a cluster serving DNN inferences under a heavy and spiky workload. We also tested our Round Robin implementation, to make it production ready. RadCode is satisfied with the result of our research and looks forward to implementing our findings and using the simulator in future developments.

7.2. Direction of future work

Due to time constraints, we weren't able to conduct extensive tests using large language models and scalable clusters. The simulator contains implementations of both dynamic batching and cluster scaling, so we encourage future work to build on top of our findings and widen the scope of the research. We tested the scheduler algorithms on LLM workloads, but didn't feel confident that we did our due diligence in evaluating all the possible scenarios to present the results.

The simulator is missing support for heterogeneous clusters, which would be required for more extensive tests. We advise future researchers to add this functionality when basing on our work.

7.3. Our work availability

The simulator we built is available under the MIT license on GitHub[1]. Feel free to use it in future research as well as private projects. The scheduling algorithms are free to use, but the implementations are owned by RadCode and therefore not included in the GitHub repository. Please credit our work when citing this thesis or using our simulator.

7.4. Credits

The simulator was designed by the entire team and implemented mostly by Szymon Potrzebowski and Karol Wąsowski. The Baseline scheduling algorithm was designed by RadCode. Improvements to Baseline, the implementation of Clockwork's ideas and Round Robin was adjusted and implemented by Piotr Blinowski and Szymon Mrozicki.

The thesis was written equally by the entire team under the supervision of Janina Mincer–Daszkiewicz Ph.D., who we want to thank for the involvement and constant professional support during the course of the project.

We want to credit the Azure Public Dataset[2] for providing the data we used to generate traces in the simulator.

[1] `https://github.com/technical-tigers/tiger-simulator`
[2] `https://github.com/Azure/AzurePublicDataset`

Bibliography

[1] Amey Agrawal et al. *SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills.* 2023. arXiv: `2308.16369 [cs.LG]`.

[2] Daniel Crankshaw et al. "Clipper: A Low-Latency Online Prediction Serving System". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17).* Boston, MA: USENIX Association, Mar. 2017, pp. 613–627. ISBN: 978-1-931971-37-9. URL: `https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw`.

[3] Arpan Gujarati et al. "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* USENIX Association, Nov. 2020, pp. 443–462. ISBN: 978-1-939133-19-9. URL: `https://www.usenix.org/conference/osdi20/presentation/gujarati`.

[4] Arpan Gujarati et al. "Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency". In: *Middleware '17.* Dec. 2017. URL: `https://www.microsoft.com/en-us/research/publication/swayam-distributed-autoscaling-meet-slas-machine-learning-inference-services-resource-efficiency/`.

[5] Woosuk Kwon et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention.* 2023. arXiv: `2309.06180 [cs.LG]`.

[6] Pratyush Patel et al. *Splitwise: Efficient generative LLM inference using phase splitting.* 2023. arXiv: `2311.18677 [cs.AR]`.

[7] Mohammad Shahrad et al. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20).* USENIX Association, July 2020, pp. 205–218. ISBN: 978-1-939133-14-4. URL: `https://www.usenix.org/conference/atc20/presentation/shahrad`.

[8] Thomas Wolf et al. *HuggingFace's Transformers: State-of-the-art Natural Language Processing.* 2020. arXiv: `1910.03771 [cs.CL]`.

[9] Bingyang Wu et al. "Fast Distributed Inference Serving for Large Language Models". In: *arXiv preprint arXiv:2305.05920* (2023).

[10] Gyeong-In Yu et al. "Orca: A Distributed Serving System for Transformer-Based Generative Models". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22).* Carlsbad, CA: USENIX Association, July 2022, pp. 521–538. ISBN: 978-1-939133-28-1. URL: `https://www.usenix.org/conference/osdi22/presentation/yu`.

[11] Hong Zhang et al. “SHEPHERD: Serving DNNs in the Wild”. In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 787–808. ISBN: 978-1-939133-33-5. URL: `https://www.usenix.org/conference/nsdi23/presentation/zhang-hong`.

All webpages have been visited on May 31, 2024.