TU WIEN Informatics

Efficient and Transparent Model Selection for Serverless Machine Learning Platforms

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Silvio Vasiljevic, BSc
Matrikelnummer 01633650

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Mitwirkung: Dipl.-Ing. Dr. Thomas Rausch
Dipl.-Ing. Philipp Raith

Wien, 22. November 2023

Silvio Vasiljevic	Schahram Dustdar

TU WIEN Informatics

Efficient and Transparent Model Selection for Serverless Machine Learning Platforms

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Silvio Vasiljevic, BSc
Registration Number 01633650

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Assistance: Dipl.-Ing. Dr. Thomas Rausch
Dipl.-Ing. Philipp Raith

Vienna, 22nd November, 2023

Silvio Vasiljevic | Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Silvio Vasiljevic, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 22. November 2023

Silvio Vasiljevic

Danksagung

„Wenn ich weiter gesehen habe, dann indem ich auf den Schultern von Riesen stand."
— Isaac Newton

Ich bin dankbar für die Ermutigung und Unterstützung, die ich auf meinem Weg erhalten habe, sowohl beim Schreiben dieser Arbeit als auch im Leben.

In erster Linie möchte ich Thomas Rausch meinen aufrichtigen Dank aussprechen, der eine ständige Präsenz in meinem akademischen Leben war. Seine nicht endende Geduld und seine großzügige Unterstützung waren die Grundlagen meines akademischen Werdegangs. Er hat mir immer wieder den Weg gezeigt und mein Wachstum als Forscher und Ingenieur unterstützt, von der Konzeption meiner Bachelorarbeit bis zur Fertigstellung dieser Arbeit. Sein fundiertes Fachwissen über verteilte Systeme war für mich sehr hilfreich, und diese Arbeit wäre ohne seine Betreuung nicht möglich gewesen.

Ich möchte Prof. Schahram Dustdar meinen aufrichtigen Dank dafür aussprechen, dass er mir die Möglichkeit gegeben hat, an dieser Diplomarbeit zu arbeiten. Darüber hinaus muss ich Philipp Raith für seine Unterstützung beim Evaluationscluster danken—auch über Zeitzonen hinweg. Er stand mir immer zur Verfügung, wenn ich Probleme mit dem Cluster hatte, und versorgte mich mit den Ressourcen, die ich für die Durchführung der Experimente benötigte.

Mein herzlicher Dank geht an meine Freunde, die mich motiviert und unterstützt haben. Insbesondere möchte ich mich bei Matthias bedanken, mit dem ich mich mehrmals zum Arbeiten an unseren Diplomarbeiten getroffen habe. Auch Rafaels akribisches Korrekturlesen hat dazu beigetragen, die Qualität dieser Arbeit zu verbessern.

Mein tiefster Dank gilt auch Violetta, meiner Partnerin, deren unermüdliche Unterstützung und Ermutigung eine ständige Triebkraft war. Ihr Glaube an meine Fähigkeiten und ihre Ermutigung haben maßgeblich dazu beigetragen, dass ich Herausforderungen und Rückschläge überstehen konnte.

Mein Dank gilt auch den Menschen bei LocalStack, die es mir ermöglicht haben, diese Arbeit zu schreiben, indem sie mir ein flexibles Arbeitsumfeld boten und mir erlaubten, Bildungskarenz zu nehmen. Ich danke der Netidee Internet Stiftung für ihr großzügiges Stipendium.

Zuguterletzt sollte meine Familie besonders erwähnt werden. Seit meiner Kindheit sind meine Eltern eine ständige Quelle der bedingungslosen Unterstützung und Inspiration für mich gewesen. Ihre durchgehende Ermutigung, nach Großem zu streben und niemals aufzugeben, hat mich immer wieder motiviert. Sie haben mir ein Umfeld geboten, in dem ich mich entwickeln und lernen konnte. Maja und Vidica, meine Schwestern, haben mich während meines gesamten Lebens unterstützt, unter anderem auch durch sorgfältige Korrekturlesungen dieser Arbeit. Ich werde meiner Familie für ihre unermüdliche Unterstützung in meinem Leben ewig dankbar sein.

Acknowledgements

"If I have seen further, it is by standing on the shoulders of giants."
— Isaac Newton

I am grateful for the encouragement and support I have received along my journey, both in writing this thesis and in life.

First and foremost, I'd like to convey my heartfelt gratitude to Thomas Rausch, who has been a constant presence in my academic life. His unending patience, generous support, and guidance have been the foundation of my academic journey. He has continuously shown me the route and supported my growth as a researcher and engineer, from the conception of my bachelor's thesis until the completion of this work. His deep expertise of distributed systems has been extremely beneficial to me, and this thesis would not have been possible without his mentorship.

I would like to express sincere thanks to Prof. Schahram Dustdar for providing me with the opportunity to conduct this research. In addition, I must thank Philipp Raith for his assistance with the evaluation cluster—even across time zones. He was always available to help me with any issues I had with the cluster and to supply me with the resources I needed for conducting the experiments.

My heartfelt gratitude goes to my friends, who have been pillars of motivation and focus. I'd want to thank Matthias in particular, with whom I've had several study sessions working on our theses together. Rafael's meticulous proofreading has also helped to improve the overall quality of this work.

My profound thanks also goes to Violetta, my partner, whose unfailing support and encouragement has been a continuous driving force. Her belief in my abilities and her encouragement have been instrumental in my ability to persevere through challenges and setbacks.

I am also grateful to the individuals at LocalStack who enabled me to complete this thesis by providing a flexible work environment and allowing me to take educational leave. I am grateful to the Netidee Foundation for their generous grant.

Last but not least, a particular mention should be made of my family. Since my childhood, my parents have been a constant source of unconditional support and inspiration for me. Their constant encouragement to strive for greatness and never give up has served as a

constant motivator. They have provided me with a nurturing environment in which to develop and learn. Maja and Vidica, my sisters, have also been crucial in their support throughout my life, including careful proofreading contributions to this thesis. I will be eternally grateful to my family for their unwavering support throughout my life.

Kurzfassung

Serverless Computing ist ein neues Paradigma für die Bereitstellung von Anwendungen in der Cloud. Seine Ausweitung auf den Bereich des Machine Learning (ML) ist seit einigen Jahren ein Forschungsthema—insbesondere in der Community des Serverless Edge-Computing. Die nahtlose Integration von Machine Learning-Modellen für den Einsatz in Serverless Computing-Umgebungen ist immer noch eine Herausforderung. Die Auswahl geeigneter Modelle für bestimmte Anwendungsfälle ist entscheidend für die Leistung des gesamten Systems. In anderen Bereichen wurden Dienstauswahl und Load Balancing unter Berücksichtigung der Quality-of-Service (Dienstqualität) eingesetzt, um die Leistung von Anwendungen zu verbessern. Die aktuellen Ansätze zur Dienstauswahl und zum Load Balancing berücksichtigen jedoch weder die Besonderheiten von Machine Learning-Modellen noch die Latenzzeiten von Edge-Knoten. Daher müssen Entwickler:innen von Serverless Anwendungen derzeit das beste Modell für ihren Anwendungsfall manuell auswählen oder sich auf suboptimale Modellauswahlansätze verlassen. Die manuelle Auswahl der Modelle schränkt die Flexibilität der Anwendung ein und setzt voraus, dass Entwickler:innen über Kenntnisse der zugrunde liegenden Infrastruktur verfügen. In dieser Arbeit schlagen wir eine Lösung vor, die automatisch und transparent das beste Modell für einen bestimmten Anwendungsfall aus der zugrunde liegenden Infrastruktur von Serverless Plattformen auswählt und dabei Machine Learning und Edge Computing-spezifische Belange berücksichtigt. Die Anwendungsentwickler:innen müssen dann nur noch die Daten und die gewünschten Leistungsmerkmale (z. B. Latenz, Genauigkeit usw.) des Modells bereitstellen. Wir evaluieren unsere Lösung, indem wir sie mit gängigen Basisansätzen für die Auswahl und dem Load Balancing von Diensten vergleichen. Zunächst führen wir Fallstudien durch, um die Leistung unserer Lösung in verschiedenen Anwendungsfällen zu bewerten. Anschließend vergleichen wir die Leistung unserer Lösung in einer simulierten Umgebung mit den grundlegenden Ansätzen. Die Ergebnisse zeigen, dass unsere Lösung in den meisten Fällen besser abschneidet als die Basislösungen.

Abstract

Serverless computing is a new paradigm for the deployment of applications in the cloud. Its extension to the field of Machine Learning (ML) has been a topic of research for the past few years—especially in the serverless edge computing community. Seamless integration of Machine Learning models for usage in serverless computing environments is still a challenge. Selecting approriate models for particular use cases is crucial for the performance of the whole system. In other domains, service selection and load balancing with quality-of-service consideration have been used to improve the performance of applications. However, current approaches to service selection and load balancing either do not take into account the specifics of Machine Learning models or the latency of edge nodes. So, developers of serverless applications currently have to manually select the best model for their use case or rely on subpar model selection approaches. Manually selecting the models limits the flexibility of the application and requires the developer to have knowledge about the underlying infrastructure. In this thesis, we propose a solution that automatically and transparently selects the best model for a given usecase provided in the underlying infrastructure of serverless platforms while taking into account Machine Learning and edge computing-specific concerns. Application developers then only have to provide the data and the desired performance traits (e.g., latency, accuracy, etc.) of the model. We evaluate our solution by comparing it to common baseline approaches for service selection and balancing. First, we conduct case studies to evaluate the performance of our solution in different use cases. Then, we compare the performance of our solution to the baseline approaches in a simulated environment. The results show that our solution outperforms the baseline approaches in most cases.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Through the rise of cloud computing, the way in which software is developed and deployed has changed significantly. Instead of managing ever-changing hardware and software stacks and dealing with scalability concerns, system administrators can now rely on the infrastructure provided by cloud providers and use virtualized resources. This new paradigm is commonly referred to as Infrastructure-as-a-Service (IaaS) [SGH15]. **Serverless computing** is a paradigm that expands on this idea. It not only eliminates the need of managing hardware resources but also the whole context and state from the developer's end. By offering the deployment of short-lived containerized or virtualized applications—Function-as-a-Service (FaaS)—developers can focus on the business logic of their applications. This way, they do not have to worry about the underlying infrastructure. These commonly stateless functions react to event triggers from services the cloud provider offers. The stateless nature of serverless functions is a key aspect of the paradigm. It allows cloud customers to scale the functions automatically based on demand and only pay for the actual resource consumption. However, similar to the invocation triggers, functions rely on external backend services for data storage and retrieval This represents the second part of the serverless paradigm: Backend-as-a-Service (BaaS) [JSSS+19].

Even though cloud computing, and more specifically serverless computing, managed to solve problems of scalability and availability, there are still limitations to the paradigm. For instance, latency sensitive applications are unsuitable for serverless computing as the data transmission between the cloud and the end user is slow. Computation therefore needs to happen closer to the end user, at the edge of the network. **Serverless edge computing** is a paradigm which tries to close this gap by bringing the ease-of-use of serverless computing to heterogeneous environments at the edge of the network. This enables the development of novel applications which previously were either not possible to develop on plain serverless platforms or were too complex to manage in edge environments.

One class of applications benefitting from the synthesis of these paradigms is **Edge Intelligence (EI)**, a combination of Artificial Intelligence (AI) and edge computing [RND23].

A paradigm which also gained popularity with cloud computing and can gain further significance through the combination with serverless edge computing is **Machine Learning Operations (MLOps)**. MLOps is a set of practices and tools to automate the development, deployment and maintenance of machine learning models and applications. MLOps platforms can thereby oversee machine learning models during their entire lifecycle. This starts from data collection and model training and continues to the deployment, inference and monitoring of the models [KKH23]. An example of such a platform is Amazon SageMaker[1] which includes detailed management of the models and their deployment, and subsequent monitoring with other Amazon Web Services.

1.2 Problem Statement

Serverless edge computing has shown to be useful for dealing with the heterogeneity of edge environments. Serving model inference on serverless edge computing platforms enables the development of Edge Intelligence applications. However, the separation of infrastructure and application logic in serverless computing, while reducing complexity for application developers, leads to a lack of domain-specific Machine Learning knowledge such as the accuracy of the served models. This makes it hard to optimize applications along Quality-of-Service concerns such as latency or accuracy and to balance the trade-off between them and the utilization of the underlying infrastructure. Therefore, serverless edge computing platforms need an extension for intelligent model selection able to take into account domain-specific knowledge about the models and their execution environments.

We propose an *efficient and transparent selection system for machine learning models in heterogeneous execution environments*. For this, we utilize the operational benefits of serverless computing for model serving in edge environments. This model selection system is able to deal with data-intensive workloads by being conscious of the data sizes transferred between clients and distributed execution nodes. Through continuous measurement of Quality-of-Service metrics such as latency or accuracy, the system not only enables the selection of the most suitable model for a given task. It also improves the resource utilization of the underlying infrastructure.

[1] https://aws.amazon.com/sagemaker

Research Questions

The research questions which shape the work of this thesis and their descriptions are as follows.

RQ1 Which benefits can be achieved for latency-sensitive applications by using a weighted model selector compared to a round-robin, random or plain-net-latency model selector?

Edge Intelligence applications are often latency-sensitive as they need to react to events in real time. Selecting fast-running models on a nearby node is crucial for such applications. We compare our approach to simple solutions like round-robin and random selection, as they represent selection without Quality-of-Service concerns. We also deliver a comparison with an approach only considering the network latency between the client and the execution node without further knowledge of the model. All other research questions are against the same baselines.

RQ2 Which benefits can be achieved for accuracy-sensitive applications by using a weighted model selector compared to a round-robin, random or plain-net-latency model selector?

Some Artificial Intelligence applications have no hard latency requirements but instead require high accuracy. For such applications, selecting the most accurate model is crucial and a consideration of the latency is superfluous.

RQ3 What differences in behavior can we expect for applications with varied latency and accuracy preferences between the different use cases in the weighted selection compared to the baselines?

In the average case, applications require a trade-off between latency and accuracy. We first evaluate how our approach and the baselines handle cases with equal preference for latency and accuracy. Then, we evaluate how they handle cases with changing preferences over time.

RQ4 How does the selection scale with increasing message sizes for the different selection algorithms?

Increasing the size of data to be transferred between the client and the execution node can have a significant impact on the latency. By independently increasing the size of the data transferred, we can evaluate the scalability of the selection algorithms for data-intensive workloads.

RQ5 How does the selection scale with an increasing number of concurrent requests served by the platform for the different selection algorithms?

Increasing the overall load on the platform can also have a significant impact on the latency. By only increasing the amount of concurrent messages, we can evaluate the scalability of the selection algorithms for compute-intensive workloads.

1.3 Methodology

The methodological approach we follow in this thesis is as follows.

1. **Literature Review**
   We gather the existing adaptations of the serverless paradigm for machine learning and present it in a comparison matrix. We further analyze related fields such as cloud service selection and data-intensive workloads in serverless computing. The findings of the literature review contribute to the requirements elicitation and the design of our prototype and its evaluation.

2. **Requirements Elicitation**
   Based on different machine learning use cases, we generate design goals for the platform which we need to meet to offer the benefits of the serverless paradigm for machine learning applications effectively. This elicitation includes the findings from the literature review and delivers requirements in terms of performance and feature set. We construct use cases which both aim to show the fulfillment of the design goals as well as helping to answer the research questions.

3. **Prototype Implementation within a Serverless-Capable Runtime**
   We provide a prototypical implementation built atop a platform capable of following the serverless paradigm, in accordance with our design goals. This section of the thesis includes modeling of the meta representation of the functions and models, development of a simple API for automated selection of ML models, and implementation of components that enable this selection.

   These components, which also describe the main contribution of the thesis,primarily include:

   - an object storage solution for the models along with a way to launch those models manually in a containerized environment,
   - monitoring components for measuring the QoS metrics of the models,
   - a metadata storage schema for storing the QoS metrics of the models and their general characteristics,
   - a service for automated selection of models based on the QoS metrics and the characteristics of the models,
   - and client implementations of the use cases utilizing the automated selection service.

4. **Case Study**
   We base our case study on the use cases from the requirements elicitation and their client implementations. The case study is both qualitative and quantitative in nature as we analyze the results in a cumulative manner as well as looking at single experiment runs in more detail. The main performance metric is the latency

experienced during invocation and the accuracy of results as it directly impacts the service quality of end users. The testing environment for the case study offers heterogeneity in latency between execution nodes. This helps in evaluating the performance of the platform in edge environments.

5. **Scalability analysis**
   We use the same mechanisms as in the case study to evaluate the scalability of the platform. However, instead of analyzing QoS metrics based on use case, we analyze how these metrics change for the same use case when increasing different types of load on the platform.

1.4 Structure

The remainder of this thesis is structured similarly to our methodology and is as follows.
Chapter 2 provides an overview of the theoretical background of the thesis. We first introduce the serverless paradigm, its challenges, its extension to the edge of the network and an implementation base commonly used for building serverless platforms. We also discuss the concept of Machine Learning Operations, the tasks it encompasses and an example platform.
Chapter 3 relates our research with works in the field of serverless computing, machine learning and service selection. We first compare our work with approaches for tackling data-intensive workloads in serverless computing. Then, we present existing work combining serverless computing with Machine Learning and how our work relates to those. Finally, we discuss the research field of cloud service selection further contextualizing our work for enabling automated selection of Machine Learning models.
Chapter 4 details our approach and the design of the prototype platform. We first define the platform's design goals, followed by appropriate use cases. Then, we present the architecture of the prototype platform and the implementation details.
Chapter 5 lays out how we evaluate our prototype. We first describe the testing environment and additional components used for the evaluation. Then, we describe the evaluation methodology and how we answer the research questions, both for the case studies and the scalability analysis.
Chapter 6 then presents the results of the evaluation. We go through each of the research questions and discuss the results of the case studies and the scalability analysis. Afterwards, we summarize the findings to give a holistic view on the results.
Finally, Chapter 7 concludes the thesis and gives an outlook on future work.

CHAPTER 2

Background

In this chapter, we present the background concepts on which we base the work in this thesis. We present the concepts of serverless computing and its challenges in Section 2.1. Furthermore, we discuss applications of this paradigm at the edge in Section 2.2 and Kubernetes as a facilitator for serverless computing in Section 2.3. Another concept that has gained popularity through the rise of cloud computing in recent years is Machine Learning Operations (MLOps) which we discuss in Section 2.4.

2.1 Serverless Computing

The emergence of large-scale cloud providers (e.g.,Amazon Web Services (AWS)[1], Microsoft Azure[2] and Google Cloud[3]) has enabled developers to build and deploy applications with a higher degree of abstraction than before. Virtualizing the underlying infrastructure abstracted away the need to manage hardware and led to the models of Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) [SGH15]. Developers can therefore now scale their applications quasi infinitely and only pay for the resources they actually use. However, managing the resulting virtual machines still requires a lot of operational overhead. This is impractical for smaller applications with limited purposes (e.g., thumbnail generation [Ama23c]). Serverless computing is a paradigm further abstracting away even the virtual infrastructure and allows developers to focus on the business logic of their applications.
As the name suggests, serverless computing platforms completely remove the notion of servers from the developer's perspective. Instead, developers implement their application logic as short-running virtualized or containerized packages which get executed in response to events. The events can either be specific triggers like HTTP requests, actions

[1] https://aws.amazon.com
[2] https://azure.microsoft.com
[3] https://cloud.google.com

in databases or specifically designed scheduled events from other services in the offering of the cloud provider. The packages are called functions and the platforms which execute them are called Function-as-a-Service (FaaS) platforms. The functions also do not hold any long-lasting state which means that they need to store and retrieve data from external sources. Cloud providers therefore offer Backend-as-a-Services (BaaSs) that are used in conjunction with FaaS to build complex applications. The combination of FaaS and BaaS is what constitutes serverless computing. Furthermore, the serverless computing platforms also need to handle the scaling of the functions and offer pay-as-you-go pricing models [JSSS+19].

A demonstrative example of a platform offering serverless computing is Amazon Web Services (AWS), offering FaaS, BaaS, auto-scaling and a pay-as-you-go pricing model. It is also, by market share, the most popular cloud platform offering serverless computing [Ric23]. However, other cloud providers like Azure and Google Cloud also offer similar services with different branding but the same underlying components. As for AWS, for Function-as-a-Service it offers AWS Lambda which can act on events from other services such as DynamoDB, S3, API Gateway, AppSync. DynamoDB being a NoSQL database and S3 being a storage service are examples of Amazon's Backend-as-a-Service offering. Instead of having to manage a storage server, a database server and a web server, Amazon's customers use these services to relay requests to their functions and to store and access data.

Shafiei et al. [SKM22] argue that the existence of such providers opens up previously unavailable opportunities for developers As mentioned before, utilizing serverless computing does away with large parts of the operations overhead of distributed applications like managing server hardware, dealing with scaling and handling request contexts. Cloud providers can, e.g., offer more competitive pricing for scaling because they can host multiple customers on the same physical hardware. They can also use older hardware for less demanding workloads and can therefore extend the lifecycle of their hardware. Developers do not need to concern themselves with these operational details and instead can focus on making their own product better. On top of that, they also have more opportunities to deliver their product to those who need it via cloud provider marketplaces. To that end, they can develop generalized solutions to common problems and package them tightly into functions. Several existing domains are active fields of research for serverless computing, as surveyed by Shafiei et al. [SKM22]. Especially fields with parallelism concerns like data analytics, video streaming, scientific computing, IoT and edge computing are promising application domains for serverless computing.

Despite the advantages of serverless computing, there are still challenges to overcome. The following sections will discuss the current challenges of serverless computing in general and in the context of data-intensive applications, such as Machine Learning. Shafiei et al. [SKM22] provide a comprehensive overview of the current challenges developers face when using serverless computing. We divide these challenges into three categories: development, execution infrastructure and cross-cutting concerns. Furthermore, we describe the findings of the survey, discuss related work to those findings and highlight

the connection to our work where applicable.

2.1.1 Developing serverless computing Applications

Regarding the development of serverless computing applications, Shafiei et al. the fundamental **lack of mature development tools and practices**, and the **difficulty of predicting the cost** of serverless computing as core challenges [SKM22].

Due to its novelty, they argue that serverless computing is still lacking ways to properly test and debug applications. We interject that there are novel solutions to this problem, like LocalStack[4] for local testing of serverless applications by containerizing an extensive subset of the service offering of AWS. However, we concede that tools like LocalStack are not yet available for all cloud providers and are still gaining traction among developers. The authors further argue that the serverless model inherently makes it more difficult to verify and reason about the correctness of applications since they are composed of many small functions which are often stateless. This is exacerbated by the fact that serverless computing is often used in combination with other cloud services for storage and trigger events. However, there are also approaches to address this problem [APRS21].

Since cloud providers utilize the heterogeneity of their infrastructure to keep costs low [RDC+17], the cost of serverless computing is highly dependent on the current load of the cloud provider. Developers can work around that by automatically searching for the cheapest cloud provider, since it is easier to switch providers with serverless computing than with traditional data center computing. However, the biggest cloud providers use static pricing to counteract the volatility of their infrastructure cost.

2.1.2 Executing Serverless Computing Applications

After development, the next challenge is the actual execution of the serverless computing applications. The challenges Shafiei et al. identify here are **efficient scheduling**, **data management** and **communication** [SKM22].

In the context of serverless computing, scheduling refers to the allocation of resources to functions—i.e. when and where to execute the function—and the distribution of load [Ste18]. When scheduling functions, cloud providers need to consider multiple constraints, both from the function developer and from the underlying infrastructure. This increases the complexity of scheduling functions and makes it difficult to find an optimal solution. To address this problem, serverless platform providers can follow multiple scheduling strategies either in isolation, or more often in combination with each other. All these strategies base themselves on some kind of awareness of infrastructure or application constraints. Platform engineers can implement these strategies in existing tools, such as Kubernetes[5] [RRD21] or Apache Mesos[6] [SSM20].

[4] https://localstack.cloud
[5] https://kubernetes.io
[6] https://mesos.apache.org

For strategies involving infrastructure constraints, approaches like energy-aware scheduling or resource-aware scheduling are possible. Being energy-aware means that the cloud provider tries to minimize the energy consumption of the infrastructure. Putting function containers into hibernation when they are not needed or delaying their execution when the load is smaller are examples of energy-aware scheduling. However, this can lead to the problem of cold starts, which means that the function container needs to be initialized before it can be executed. Waiting on this initialization can lead to increased latency for the function invocation. Combined with the short actual runtime of functions, Li et al. argue that this can incur unnecessary costs and contributes to the difficulty of predicting the cost of serverless computing [LLW+23] . Mitigating this cold-start problem is an active area of research with approaches involving pre-warming of function containers [SAG23] or utilizing machine learning [VFA23, ARB21].
For resource-aware scheduling, the cloud provider tries to maximize the utilization of the infrastructure. The goal here can either be to maximize the utilization on every single execution node (by, e.g., capping CPU load per function [KHLZ20]), or to optimize the overall distribution of load across all nodes while respecting quality of service constraints. Adding on auto-scaled VMs to the infrastructure to perform some kind of Serverless and serverful hybrid scheduling is also an example of infrastructure-based scheduling.

Beyond the infrastructure, platform engineers can also decide to base their scheduling on application constraints. Here, schedulers can consider serverless application concepts like workflows [XGT+23], packing and caching. Workflows describe how functions interact with each other and which functions are dependent on each other to form a coherent application. Workflow-aware scheduling aims to find these dependencies and tries to schedule them close to each other to minimize latency.
Packing is the concept of packing multiple functions together on the same node to minimize the number of nodes needed to execute the application. The difficulty in packing arises when multiple functions need the same data, or one function needs multiple data sets. Here, the cloud provider needs to decide which functions to pack together and which data to pack with them. A further extension to this is the inclusion of available packages in the packing decision [ZLL+23, CAS23]. For example, if one function needs a specific package (e.g., Tensorflow[7]), the cloud provider can decide to pack it with another function which already has the package installed or schedule it on a node already containing the package. Data placement decisions like these go hand in hand with caching decisions inside the platform. Compared to serverful applications, data stores in serverless computing are more decoupled from the functions which use them. Therefore caching strategies from serverful applications are not directly applicable to serverless computing. Considering workflows and multiple reads across functions can, e.g., lead to more efficient caching strategies [HFC+23].
Migrating these data is also more expensive, since the functions are stateless and can be executed on any node. This can lead to the cloud provider choosing a suboptimal node for a function to avoid data migration costs [LLW+23]. Combining all these considerations

[7] https://tensorflow.org

leads to dataflow-aware scheduling and in the case of packaging also to package-aware scheduling.

Apart from infrastructure and application constraints, Raith et al. also present a scheduling approach based on user circumstances: mobility-aware scheduling [RRD+22]. This approach is especially important for edge computing, where users can move around and therefore change their proximity to the edge nodes. The authors define the concept of "pressure", an abstract metric which models the load on the edge nodes and their relation to themselves and the user. With different underlying metrics for pressure, Raith et al. achieve improvements in terms of latency as well as in terms of resource consumption.

Since serverless functions are typically stateless, they need to communicate over the network to exchange data. They also are short-lived and multiple instances of the same function definition can run simultaneously. Li et al. mention that execution time of functions can vary from milliseconds to minutes. This means that finding ways to address each other is difficult. The increased barrier between the functions makes parallelization over multiple functions less efficient than parallelization inside a single function [LLW+23]. We address this point in our work by keeping the communication between client applications and the platform as simple and transparent as possible, utilizing only stateless HTTP requests to the selector and models. This is also a solution model which Shafiei et al. discuss: using an external coordinator and stateless API [SKM22].

Cross-Cutting Concerns

Furthermore, Shafiei et al. also mention cross-cutting concerns like **security, privacy and monitoring** as big challenges for serverless computing as they are for other computation paradigms [SKM22].

For monitoring, external services such as AWS Cloudwatch[8], Datadog[9] or Dynatrace[10] are available. However, these services are restricted in the metrics they can monitor, since they are not aware of the internal state of the application. The survey states that the services often miss metrics such as the data dependencies, invocation rate, cache states, data formats. Often, developers need to go through the vast amount of log data to find the metrics they need [MKW19].

Regarding security and privacy, the survey authors argue that serverless computing is especially susceptible to replay attacks, since functions are stateless and can be invoked multiple times. Strong authentication and authorization mechanisms are therefore necessary—not only for manual invocation, but also for trigger events. A challenge for serverless computing is also the multi-tenancy of the underlying infrastructure and the handling of isolation between functions. To achieve isolation, cloud providers usually employ some kind of virtualization or containerization techniques or isolated language

[8] https://aws.amazon.com/cloudwatch
[9] https://datadoghq.com
[10] https://dynatrace.com

runtimes [LLW+23]. Underlying infrastructure vulnerabilities, like Meltdown [LSG+18] or Spectre [KHF+19], however, could still lead to data leakage between functions. Contextual information, such as the data format, can also leak through the invocation of functions. Lastly, large scale invocation attacks are another problem for serverless computing according to Shafiei et al., since functions can scale to high numbers of instances. Without any protection mechanisms, this can lead to high costs for the function developer [SKM22].

2.2 Serverless Computing at the Edge

Edge computing is a paradigm that stands in stark contrast to the centralized nature of cloud computing. Instead of performing all tasks inside few data centers, edge computing aims to distribute computation and storage to heterogeneous devices the edge of the network [CLMS20]. As we mention in Section 2.1, serverless computing is a paradigm which arose from the cloud computing paradigm. However, there are also approaches to combine serverless computing with edge computing, as Raith et al. discuss in detail [RND23]. They argue that the serverless paradigm can be used to manage the diverse set of devices in the edge-cloud continuum. The combination of these two domains—serverless and edge computing—can be used to create a new generation of applications.
They specifically mention Edge Intelligence (EI) as a promising application domain for serverless edge computing. Edge Intelligence describes the usage of AI on the edge of the network. This can mean to use edge devices to run AI applications to utilize AI for managing the edge of the network [DZF+20]. In the case of serverless edge computing for EI the focus is on the former, i.e. the usage of AI on the edge of the network. The challenges of EI lie mainly in (1) making data available to the vast number of devices, (2) selecting the proper models in regards to accuracy and latency, and (3) coordination of training and inference tasks between heterogeneous devices [DZF+20]. To handle these challenges, Raith et al. argue that specific criteria need to be met for developing and executing EI applications [RND23]. For the development and design of these applications, they argue that serverless edge computing frameworks need:

- Extensive support for the handling of application state and data. This ranges from enabling checkpointing, to implementing data management APIs, to failure handling.
- The awareness of the handled resources and the network state in order to be able to meet service level objectives (SLOs) defined in a service level agreement (SLA).
- General support for the development of AI workloads. This ranges from supporting programming languages commonly used for AI (e.g. Python) to offering their own abstractions for such workloads.
- The ability to compose and orchestrate multiple functions into a single application via some API, configurations files or a UI.

Furthermore, for the execution of EI applications, they argue that serverless edge computing frameworks need:

- Reliable invocations and a way to handle failed invocations. Ideally, invocations happen in transactions able to handle errors gracefully.
- Stateful functions that retain their state between invocations. The implementation of this feature should take into account the heterogeneity of the network conditions and the devices.
- Advanced ways to trigger function executions. These can range from simple HTTP requests to more complex triggers inside an event handling system.
- Distributed gateways for routing requests.
- Intelligent handling of performance goals. The platforms should be able to optimize for multiple goals and should be able to handle the trade-offs between them and their cost.

2.3 Kubernetes and its Use for Serverless Computing

To implement serverless computing, cloud providers need to manage the underlying infrastructure and the execution of functions. As we discuss in Section 2.1, this includes scheduling functions, managing their state and handling their communication. Building a platform from scratch to handle these tasks is a big challenge and requires a lot of resources.
However, container orchestration platforms like Kubernetes (K8S)[11] already offer many of the features needed to implement serverless computing.
K8S supports automatic deployment of pods (virtual machines hosting application containers), scaling and management of containerized applications. For communication, it offers internal services discoverable via DNS and external ingresses for communication with the outside world. Persistent storage is also available via K8S volumes [MPK+22]. Through these features, Kubernetes already offers enough functionality to implement at least the Backend-as-a-Service (BaaS) part of serverless computing. Decker et al. even argue that K8S can be described as "accidentally serverless" [DKK22]. However, to add on the Function-as-a-Service part of serverless computing several extensions emerged for K8S such as KNative[12], Fission[13], Nuclio[14] and the most popular OpenFaaS[15] [DKK22, KF22].

Utilizing serverless platforms based on K8S has shown to be both suitable for high performance applications [DKK22] and for edge computing scenarios [KF22]. However,

[11] https://kubernetes.io
[12] https://knative.dev
[13] https://fission.io
[14] https://nuclio.io
[15] https://openfaas.com

regarding high performance applications, future work can be done to improve the performance of the platforms, like supporting more hardware accelerators [DKK22]. For edge computing scenarios, specific Kubernetes distributions like K3S[16] can be used to reduce the resource footprint of the platform [KF22]. With these performance characteristics, developing a platform based on K8S is a viable option for high-performance and serverless edge computing applications. In Chapter 4, we present our prototypical implementation of a serverless edge computing platform which can be deployed on Kubernetes. Through Kubernetes our implementation can inherit these performance characteristics and FaaS capabilities of extensions to form a full machine learning platform for serverless edge computing.

2.4 Machine Learning Operations (MLOps)

The development of Machine Learning models rarely happens in a vacuum and ends with the training of the model. Models mostly have a specific purpose and are therefore deployed in production environments to be used by other applications. To facilitate this process, Machine Learning Operations (MLOps) is a set of principles and components for managing the lifecycle of ML models. The principles of MLOps include the automation of the ML lifecycle, continuous training, monitoring and evaluation of deployed models. As for the components that enable the principles, MLOps includes tools for continuous integration and deployment, feature storage, model training, model registration, metadata storage and model serving and monitoring [KKH23].
Relevant to this thesis are the components for metadata storage and model registration, serving and monitoring, which we discuss in our approach in Chapter 4.

We present an example of a MLOps pipeline in the SageMaker[17] offering of Amazon Web Services (AWS). Figure 2.1 shows the workflow of SageMaker MLOps[18] which utilizes the components mentioned above. The platform has both graphical interfaces for business analysts, as well as technical interfaces for data scientists to develop and train new models.

We discuss the components of SageMaker MLOps, relevant to our work in the following. These specific implementations inspire our prototypical implementation in Chapter 4 and are therefore useful to highlight. **The model registry** is how SageMaker stores models for further development and deployment. For each ML problem (i.e. application), the registry creates a "model group" with each new training of a model being a versioned "model package" inside that group. The versions start from 1 and increase by 1 per training of the model. The registry exists as an alternative to the AWS marketplace where pre-trained models can be selected instead of custom trained models. The registry features an approval mechanism where models can be tested before they are deployed for inference.

[16] https://k3s.io
[17] https://aws.amazon.com/sagemaker
[18] https://aws.amazon.com/sagemaker/mlops

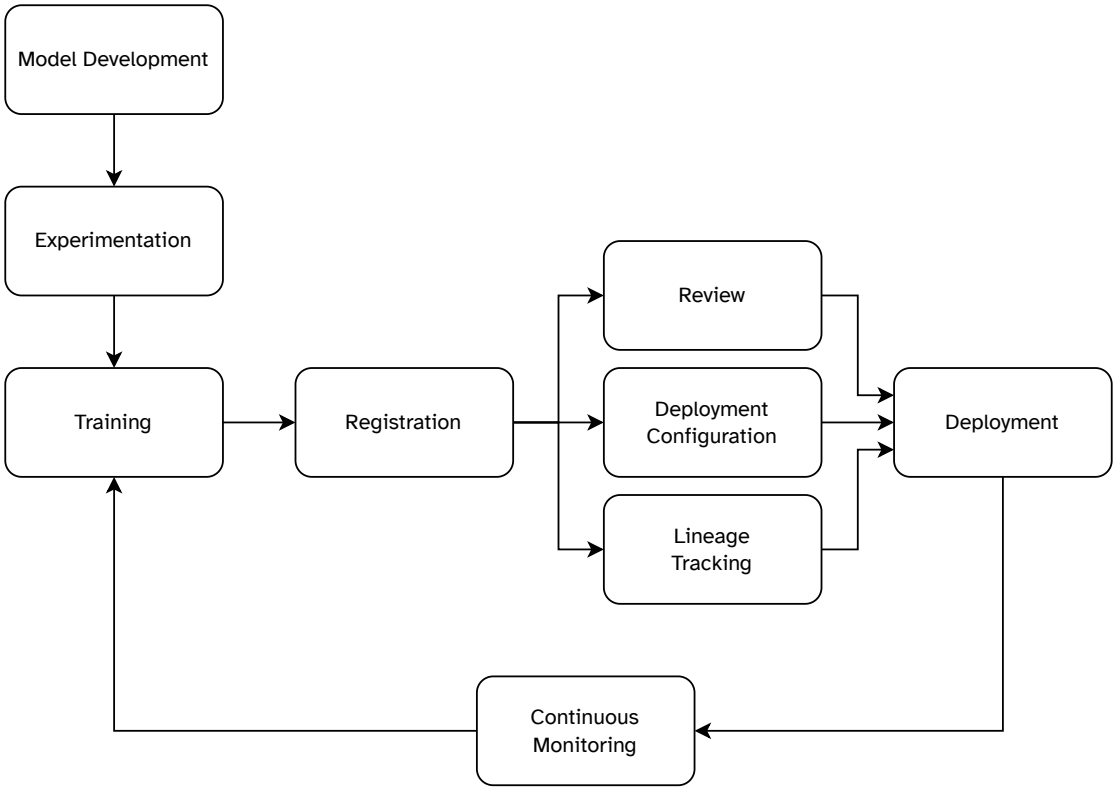

Figure 2.1: Workflow of SageMaker MLOps, adapted from [Ama23a]

For the **deployment of models**, SageMaker offers three options: (1) Real-time inference which hosts models continuously and scales them based on need. (2) Serverless inference which launches the models themselves on demand. (3) Asynchronous inference for requests with large bodies (at Gigabyte scale) with long processing times.
To maintain reliability of the deployed ML services, AWS provides tools for **monitoring** the deployed models. This offering is based on a range of common AWS services like CloudWatch for tracking metrics and creating dashboards, CloudWatch Logs for storing the logs of model runs, Cloudtrail for capturing API calls and subsequent events, and CloudWatch Events for keeping track of status changes in training jobs.

CHAPTER 3

Related Work

In this chapter, we present related work in the areas of serverless computing, Machine Learning Operations, and the combination of both. We discuss how these approaches deal with the challenges we present in Chapter 2 and how they relate to our approach. Section 3.1 deals with existing approaches to using serverless computing with data-intensive workloads, to which Machine Learning also belongs. Section 3.2 presents approaches to combining serverless computing with Machine Learning in particular and especially approaches combining model inference and operations with serverless computing. Section 3.3 is a further discussion of approaches to QoS-based cloud service selection, which is a key component of our approach while not being specific to Machine Learning or serverless computing.

3.1 Approaches Dealing with Data-Intensive Workloads

In Chapter 2 we discuss the challenges of serverless computing, including these in executing serverless applications. We further explain how packing of functions along with caching is used to optimize performance and cost of serverless applications. These are data management aspects of serverless computing which enable dataflow-aware and package-aware scheduling.

Applications with data-intensive workloads can benefit from dataflow-aware scheduling and related techniques in the space of data management. However, the current industrial offerings of serverless computing do not offer enough techniques in this direction to optimize data-intensive workloads. Hellerstein et al. [HFG+19] argue that state-of-the-art serverless platforms, like AWS Lambda[1], suffer from constraints that limit their ability to support data-intensive applications. The authors list constraints such as the lack of support for long-running functions, high-bandwidth data transfers, caching of client data

[1] https://aws.amazon.com/lambda

for subsequent invocations, and the consideration of hardware heterogeneity. They further argue that these constraints are slowing innovation in the fields of distributed computing, hardware-accelerated software and the open source data systems community. Nevertheless, there are also approaches to overcome some of these constraints and challenges, which we will discuss in this section.

First, we present approaches to overcome the lack of support for high-bandwidth data transfers. To our best knowledge, this is the challenge with the most research in literature concerning data-intensive serverless computing. We believe this is the case since it is also most important to data-intensive workloads. One approach to overcome this issue is to reverse the data flow and move the code to the data instead of moving the data to the code [SABG23]. Sethi et al. [SABG23] justify their approach first by showing that the latency of function invocations scales with the size of the data transferred and with the distance of the code from the data. They further validate their approach through mathematical analysis in which they show that after considering basic assumptions about the network, the cost of transferring data to the code is always higher than the cost of transferring code to the data. To the best of our knowledge, the assumptions made in this analysis are not further validated by the authors. After validating their approach, the authors present a prototype implementation of their approach using Kubernetes (K8S) and Fission (which we both present in Section 2.3). They compare this prototype to a baseline implementation using AWS Lambda and show that their approach can reduce the latency of function invocations by up to 50% for large data sizes.
Cheng et al. [CFSS19] also follow the reverse data flow approach but still keep the regular data flow as an option. Their goal is a flexible solution which works for both cloud and edge computing in the context of IoT. They not only offer the reverse data flow approach, but also optimize function orchestration by considering the contextual information such as the locality of the invocation. Through this mechanism, they gain mobility awareness and can optimize the function orchestration for the current context via migrations. Their approach can scale to large numbers of nodes and manages to minimize traffic between nodes and therefore reduce service latency.
Pheromone [YCWC23] is another approach to overcome the lack of support for high-bandwidth data transfers. However, Yu et al. do not reverse the data flow, but rather decouple it from the function flow. They argue that the orchestration of serverless functions should follow the data flow instead of treating functions as independent units. Pheromone achieves this by offering a new programming model for serverless computing, which allows developers to specify the data flow between functions and the data dependencies between functions. With what they call a “data-centric” programming model, the authors achieve improvements in the latency of function invocations and data exchange.

Besides the large body of work on the lack of support for high-bandwidth data transfers, there are also approaches to overcome the other challenges of serverless computing for data-intensive applications. Duo by Huang et al. [HFC+23] presents an approach to overcome the caching challenges of serverless computing (see Section 2.1) additionally to the lack of support for high-bandwidth data transfers. The authors present a caching

mechanism considering the data dependencies between stateful functions invocations by employing two cache lists. The first cache list, which they call "Leader List" only accepts data which is read more than once by never taking in data which is read for the first time. To facilitate this they have a second cache list, called "Wingman List" and which caches the rejected data. They are able to reduce cache misses and improve data sharing between functions with their approach. This way they avoid unnecessary data transfers and lower the latency of function invocations inside stateful workflows.
Rausch et al. [RRD21] present an approach to overcome the lack of support for heterogeneity of devices in serverless computing, especially in the edge computing domain. They achieve this by introducing a novel container scheduling algorithm which considers data and computation flow additionally to hardware constraints like GPU availability. Their approach is flexible since the weights of the scheduling algorithm are adjustable and can change the focus between latency, traffic and execution cost. In their evaluation they show that given a specific set of weights, they are able to trade off between latency and execution cost. They further bring up another limitation of serverless computing, in form of the centralized nature of the control plane that is at odds with the distributed nature of edge computing.
[HTZT21] discusses workload consolidation in the context of data-intensive serverless applications. Workload consolidation is the process of combining multiple workloads on a single server to improve resource utilization. While doing so, the different workloads should not interfere with each others resource demands since this would lead to performance degradation. The authors present a resource controller which takes into account Quality-of-Service (QoS) concerns to guarantee underlying hardware performance to all running workloads. This way they are able to consolidate data-intensive workloads while reducing violations of QoS requirements.

Finally, we discuss how our approach in Chapter 4 relates to the approaches presented in this section. The data-intensive workloads we consider for this thesis are Machine Learning workloads, which experience big data transfers during model launches and inference. Our approach currently does not factor in automated scheduling or orchestration of those workloads, but rather offers an automated selection for accessing already deployed ML models. Nevertheless, by factoring in the size of request data in our selection algorithm our approach considers the data flow from client applications to the ML models. Therefore, it does alleviate the challenge of high-bandwidth data transfers in the context of ML workloads. Furthermore, our approach is also not limited to a specific set of selection criteria, but rather offers a flexible selection algorithm extensible to consider other criteria. This way, we could factor in the heterogeneity of devices and the availability of caches in our selection algorithm. Through these criteria resource utilization also can be balanced with QoS requirements. Overall, while not especially designed for generalized data-intensive workloads, our approach can be extended to also consider other data-intensive workloads besides Machine Learning.

3.2 Combining Serverless Computing with Machine Learning Operations and Inference

After discussing the challenges of serverless computing and how they are addressed in the context of data-intensive workloads, we now delve into the combination of serverless computing with Machine Learning. Machine Learning in itself is a data-intensive workload, making the approaches discussed in Section 3.1 also relevant for Machine Learning. However, there are also approaches which combine specifically Machine Learning with serverless computing in a more specific way, especially in the context of Machine Learning Operations.
Barrak et al. [BPJ22] provide a systematic mapping study of the literature on the combination of Machine Learning and serverless computing considering 53 publications. We will use this work as a basis for our discussion of the approaches to combining Machine Learning and serverless computing. To this end, we will first present the results of their mapping study, then present a matrix of recent approaches relevant to our work. These approaches are partly to be found in the mapping study, but we also include some approaches published after the study or otherwise not considered in the study. Then, we discuss these approaches in more detail, and finally discuss how our approach relates to the approaches discussed in this section.

Barrak et al. show that the combination of Machine Learning and serverless computing is a recent research area with a growing number of publications. They argue that this is connected to the growing popularity of serverless computing and the growing importance of MLOps in the industry. In this growing body of research, they identify multiple distinction criteria for the approaches.
First, they distinguish between the different stages of the ML lifecycle, which they define as data processing, model training, hyperparameter tuning, and model inference. More than half of the approaches they identify focus on model inference—the stage of the ML lifecycle most relevant to our work. After this follow model training and hyperparameter tuning, while data processing is the least considered stage of the ML lifecycle.
Second, they found that AWS Lambda is the most used serverless computing platform in the approaches they identified, followed by other popular frameworks such as Apache OpenWhisk. We do not implement the Function-as-a-Service aspect in our approach, but rather focus on the Backend-as-a-Service aspect of serverless computing. By utilizing Kubernetes for our evaluation however, any extension which adds Function-as-a-Service functionality would be possible (see Section 2.3).
Third, they present the different challenges the approaches address or solve. Here, cost optimization is the most prioritized challenge, followed by resource scalability, inference latency, batching, cold starts and service level objectives (SLOs). Our work is most related to the approaches addressing the challenges of inference latency, and SLOs. Regarding machine learning frameworks, the approaches mostly use industry standard frameworks

like Tensorflow[2], Keras[3], MXNet[4] and PyTorch[5]. Our approach follows this trend and uses Tensorflow as the machine learning framework. However, any other framework would be easy to add, as long as it can run in an application container and receive model data from storage.
Regarding the type of ML models, the approaches mostly use neural networks, followed by supervised learning models like logistic regression and random forests. While our approach is only limited by models available in Tensorflow, our evaluation is limited to simulated models which aim to show a variety of latency and accuracy characteristics.

Table 3.1 presents a matrix of significant approaches discussed by Barrak et al. [BPJ22] and some additional approaches which are relevant to our work. We separate the approaches by their focus on Backend-as-a-Service or Function-as-a-Service, and by their focus on model inference or other aspects of MLOps. The columns of the matrix show the different aspects of the approaches, which we will discuss in the following:

- **Approach:** The name of the approach if available and the reference to the publication.
- **BaaS / FaaS:** Whether the approach focuses on the BaaS or FaaS aspect of serverless computing, or both.
- **ML Aspect:** The stage of the ML lifecycle the approach focuses on, like model inference or model training.
- **Edge:** Whether the approach considers the edge computing domain.
- **Main Contribution:** The contribution of the approach which we deem as the most significant of the work, such as a new scheduling algorithm, model selection algorithm, benchmark results.

Starting from the top, Rausch et al. [RHM+19] present a proposal for combining Edge Intelligence and serverless computing. The authors argue that for enabling EI applications, serverless computing needs new concepts, such as abstractions of the device heterogeneity at the edge, awareness of device context, treatment of models and data as first-class citizens, and detailed policy mechanisms. For this, they present a prototype implementation of a serverless computing platform which is aware of the device context and can schedule functions according to hard and soft constraints. They also present a programming model for the notion of model selectors in the form of function decorators. Next, InFaaS [RLYK21] is a serverless computing platform specializing ML inference serving. The authors argue that current ML inference serving platforms do not handle the variety of ML models well and therefore put the burden of selecting the right

[2] https://tensorflow.org
[3] https://keras.io
[4] https://mxnet.apache.org
[5] https://pytorch.org

Approach	BaaS / FaaS	ML aspect	Edge	Main Contribution
[RHM+19]	both	whole pipeline	✓	scheduling, programming model
InFaaS [RLYK21]	FaaS	inference		selection, scheduling
[JGL+21]	FaaS	training		benchmarking
MLLess [SA22]	FaaS	training		cost analysis
[CPMS21]	FaaS	inference		burst offloading
INFless [YZL+22]	FaaS	inference		scheduling, batching
[BTK22]	FaaS	whole pipeline	✓	edge-cloud scheduling
[WDF+23]	FaaS	training, tuning		cost-efficient scaling
Sagemaker Serverless [Ama23b]	FaaS	inference		industry standard
MuLambda (ours)	BaaS	inference	✓	selection

Table 3.1: Comparison matrix of the approaches to combining Machine Learning and serverless computing

model variant on the user. Their approach can automatically select the right model variant given performance and accuracy requirements. Furthermore, they present a scheduling algorithm able to scale ML inference serving functions based on demand and hardware requirements. Introducing this serverless approach increased throughput in their evaluation by up to 1.3× compared to a baseline approach and reduced SLO violations by 1.6×.

The work of Jiang et al. [JGL+21] compares the performance of ML training on FaaS platforms to the performance of ML training on IaaS platforms. They implement a prototype for ML training on the FaaS platform AWS Lambda and compare it to a baseline implementation on the IaaS platform EC2. Their results show that utilizing regular (i.e., without any optimizations) FaaS platforms for ML training is only feasible for models with low data transfer requirements and low training times.

MLLess [SA22] is another approach to ML training on FaaS platforms. The authors argue that FaaS platforms can be more cost-effective than serverful platforms for ML training and support this argument with a cost analysis on a prototype implementation. They come to a similar conclusion as Jiang et al. [JGL+21] and argue that FaaS platforms are only cost-effective for ML training for fast-converging models and when communication is kept low. They introduce the idea to not update the weights of the model until a significance threshold is reached, thus reducing the communication overhead. Chahal et al. [CPMS21] present an approach to ML inference serving with awareness of service level objectives (SLOs). The motivation for this approach is that ML inference serving is often subject to bursty request patterns, which can lead to SLO violations in IaaS platforms. In order to avoid these violations, the authors suggest to offload requests to a FaaS platform during bursts. However, the main benefit they present is the decreased cost. In regards to SLO violations, they mention that cold starts still can lead to violations.

As many others Yang et al. [YZL+22] argue that current general-purpose FaaS platforms are not suitable for ML inference serving. They mention the problems of high latency, lack of performant batch processing, unsuitable balance between memory and compute availability, and the singular nature of function deployments. They introduce a new platform, INFless, an aim to solve these problems by incorporating a scaling engine capable of batching while considering SLAs. Their evaluation shows that their approach can reduce the latency and cost of ML inference serving compared to industry standard FaaS platforms.
Bac et al. [BTK22] highlight the importance of privacy concerns in the context of MLOps pipelines additionally to the latency aspect. They propose a multi-layered approach consisting of the cloud as a global aggregator, edge nodes as local execution platforms, and IoT devices as data sources. Inside the edge layer, they propose a serverless computing platform which is aware of the heterogeneity of the devices and can schedule functions for training and serving accordingly. Their evaluation shows promising results in regards to latency and cost inside their testbed.
Wu et al. [WDF+23] tackle the problem of cost-efficient scaling of model training and hyperparameter tuning on FaaS platforms. They achieve this by greedily partitioning the available resources between the different tuning jobs. During model training, they use an online prediction to adjust the available resources. Their approach is able to reduce the cost and increase performance of both model training and hyperparameter tuning compared to a baseline approach.
Last, we want to highlight that there is an industry solution for serverless ML inference serving, which is Sagemaker Serverless [Ama23b]. It provides standardized inference containers which can be deployed on-demand and scale automatically. The containers support the most common ML frameworks such as Tensorflow and PyTorch. However, users can also bring their own containers. As this offering is quite new, we were not able to find performance evaluations of this service.

Finally, we discuss how our approach in Chapter 4 relates to the approaches we present in this section. In our comparison, we first want to highlight that the idea of our approach resulted from the programming model presented by Rausch et al. [RHM+19]. They present the idea of model selectors for edge functions which guided the implementation of our approach as the authors did not present a prototype implementation.

Looking at the matrix in Table 3.1, we see that our approach is the only one solely focusing on the BaaS aspect of serverless computing. This might be because the interpretation of what constitutes serverless computing varies in the literature. The definition we use in this thesis is the one we present in Section 2.1, which is the interpretation of serverless computing as a combination of BaaS and FaaS. In our approach, we do not intend to scale ML models as functions, but rather offer machine learning models for application functions to use. This is analogous to the approach of offering a database as a service, which is also not scaled as a function, but rather offered as a service to other functions. As we mention in Section 2.3 by using Kubernetes as the underlying platform, our approach could be extended to also include the FaaS aspect of serverless computing which would

invoke the models returned by our selection algorithm. Other approaches however, scale ML models as functions, which is also a valid interpretation of the combination of model inference with serverless computing. In other words, our work combines MLOps with the BaaS part of serverless computing, while there are also approaches combining MLOps with the FaaS part of serverless computing.
Apart from that, we can see that there are approaches all along the ML lifecycle, with our approach fitting into the model inference stage. Since our main contribution is the selection algorithm, we do not intend to offer a full MLOps platform, but rather a component which can be integrated into existing serverless MLOps platforms.
Regarding the edge computing domain, we see that there are approaches which consider the edge computing domain, but most approaches do not. Since our approach takes QoS requirements into account and is aware of the latency of nodes, it is suitable for the edge computing domain. Regarding the main contribution of the approaches, we see that there are many approaches presenting new scheduling or scaling algorithms.
However, there is only one other approach for a model selection algorithm—InFaaS [RLYK21]. The main difference between our approach and InFaaS is that InFaaS does not consider heterogenous devices and therefore does not consider the latency of nodes in an edge computing environment.
This presents the research gap which our approach aims to fill: *an efficient and transparent model selection system suitable for serverless machine learning with a consideration of heterogeneous nodes.*

3.3 Cloud Service Selection utilizing Quality-of-Service Metrics

In this thesis we aim to better utilize Machine Learning models in the context of Serverless Computing by offering smart model selection. The research body of smart service selection inside serverless machine learning is still small, which is why we also consider approaches to smart service selection in other domains. We will discuss some of these approaches in this section and examine how they relate to our approach.

To begin with, we discuss the work of Thakur et al. [TSS22], who present a systematic review of the literature on service selection. By service selection, they mean the selection of a service from a set of services offering the same functionality but with different Quality-of-Service (QoS) attributes. This meshes well with our notion of model selection, which is the selection of a model from a set of models offering the same functionality but with different QoS. The authors present two groups of approaches to service selection, multi-criteria decision making and recommender systems. Our work is most related to the approaches in the group of multi-criteria decision making, which we will discuss in the following.

- **Scoring methods** are approaches assigning a score to each service based on the weighted sums of QoS attributes. This score can then be used to rank the services

and select the best one. In our own literature research we found that this is the most common approach to service selection [BMZ+22, NJ21].

- **Divergence methods** are approaches calculating the distance between the QoS attributes of the services and the QoS requirements of the user. The service with the shortest distance to the requirements is then chosen.
- **Outranking methods** are approaches which compare the QoS attributes of the services to each other and create a ranking of the services for each attribute, even with partial data. Depending on the requirements of the user, the services are then ranked and the best one is selected. An approximate example for such an approach is [WL22] since it uses a ranking of the services for each attribute. However, the approach is more advanced and includes a genetic algorithm to find the best service.
- **Pair-wise comparison methods** aim to find the best service by comparing the services to each other in a pair-wise fashion. This comparison, however, depends on the knowledge of the decision maker.
- **Utility methods** are similar to scoring methods, but instead of assigning a score to each service, they assign a utility value to each service. This means, that for each attribute they rate how well the service performs in this attribute. As an example for this we found [ZR11].

Our approach relates most to the scoring and utility methods, since we also assign a score to each model based on the QoS attributes. For each attribute, we normalize the scores of the models, which then represent a form of utility rating, and then calculate the weighted sum of the normalized scores. Based on this score, we select the best model.

Thakur et al. [TSS22] also present current issues with the approaches to service selection. We discuss the issues which fit our domain (MLOps) in the following and discuss how our approach relates to them. These relevant issues are a lack of standardization of QoS attributes, a lack of fully automated attribute evaluation, and a lack of methods to deal with uncertainty. In our work, we also notice the lack of standardization of QoS attributes for ML models. Since the focus of our work is on providing a serverless platform for ML inference and not on the classification of ML models, we do not address this issue in our work. However, we provide simulated models with different QoS attributes to evaluate our approach. To solve the issue of lack of fully automated attribute evaluation, future work could include automated benchmarking of models after they have been trained. The results can then supply the QoS attributes to our approach. In our evaluation, we encounter the issue of uncertainty in the QoS attributes of the models. Before inferencing a model for the first time, we do not know the end-to-end latency of the model as that data has not yet been collected. However, to deal with that uncertainty, we use the latency of the node running the model as a proxy for the total latency of the model. This, however, leads to temporary performance degradations, which we discuss in Chapter 6.

CHAPTER 4

MuLambda: Efficient and Transparent Model Selection for Serverless Machine Learning Inference

This chapter describes the design and implementation of a serverless machine learning inference platform prototype. Specifically, we focus on the automated selection of suitable machine learning models for a given task. First, we explain the platform's goals and the use cases that we will use to assess the platform's suitability to accomplish those goals in Section 4.1. Based on these use cases we evaluate the platform in Chapter 5. Section 4.2 in detail describes the architecture of the platform and the function and implementation of its components. Throughout the chapter we use the terms *platform* and *MuLambda* (i.e. the name of the prototypical implementation publicly available on GitHub[1]) to refer to the serverless machine learning inference platform prototype.

4.1 Design Goals

We aim to design a platform which makes it easy for application developers to use machine learning models in their projects. To this end, we assume that developers do not necessarily have extensive knowledge about how to train, deploy and manage machine learning models themselves. Furthermore, developers using the platform also do not require knowledge which specific models are available for them to use. In general, users of the platform should only have to know the task they are trying to solve with machine learning, e.g., image recognition, and the performance they expect from the underlying

[1] https://github.com/silv-io/mulambda

model that performs the given task. Thus, to satisfy the needs of such users, the platform needs to meet the following design goals (DGs):

DG1 Requesting a model should be reasonably easy to add for a variety of applications and not require extensive knowledge of the architecture of the platform or any specific non-standard technology.

DG2 Hosting of machine learning models should use an industry-standard solution with an existing large set of available models.

DG3 Developers of client applications running on the platform should be able to request a model for their specific task with a minimal latency, while also maximizing the accuracy of the model. As those two aspects describe a trade-off, developers need to define which one is more important to them.

DG4 The platform should serve multiple developers and thus aim to maximize the performance for all clients. Therefore, the load sent to the platform should be balanced between the different available underlying execution nodes.

DG5 The platform should be able to run in heterogeneous environments, such as edge computing, where latency between models and client is variable and can rise and fall based on location. It should therefore not rely on a specific infrastructure or network topology, such as all computation nodes being in the same data center or server rack.

We also explicitly define non goals (NGs) of the prototype. These aspects can be added on in future work but are not part of the implementation and evaluation in this thesis. Each item also includes a reason for its omission in the design goals.

NG1 The platform maintains a repository of a large quantity of various machine learning models. *Reason:* Creating and maintaining a suitable set of machine learning models for proper use should be done by experienced professionals in fields relevant to the users of the platform. The platform itself should only provide the means of facilitating the execution of a large variety of models (see **DG2**).

NG2 The platform schedules model executors automatically on suitable nodes in the execution cluster. *Reason:* The focus of this thesis lies in the experience of the application developer. We will evaluate different manually established schedules for the executors. Future work can explore the optimization of the schedule itself.

4.1.1 Use Cases

We propose use cases in the space of smart public services to evaluate the platform as there we can find a variety of different requirements for machine learning models. In the literature we can find machine learning use cases for Smart-City Pedestrian Safety (SCP) [RHS+21], Medical Diagnosis Assistance (MDA) [LWL+23], Public Sentiment Analysis (PSA) [KAMM23] and Environmental Monitoring (ENV) [HKH+19]. These use cases have different requirements on the latency and accuracy of the models and also different request patterns. Varying all those aspects helps us investigate whether our implementation meets **DG3**, **DG4** and **DG5** from Section 4.1 effectively.

Table 3.1 shows a comparison matrix of the use cases in terms of latency and accuracy importance and request patterns. These aspects take on the following values with the following meanings:

Latency importance:

- **High:** It is important to deliver results as fast as possible.
- **Low:** It is unnecessary to deliver fast results.
- **Balanced:** It is important to deliver fast results but to also consider other requirements (i.e. accuracy).

Accuracy importance:

- **High:** It is important to deliver the best possible (i.e. the most accurate) results.
- **Low:** It is unnecessary to deliver the most accurate results.
- **Balanced:** It is important to deliver accurate results but to also consider other requirements (i.e. latency).

Request pattern:

- **Batched:** Requests arrive as a group with non-varying requirements.
- **Streamed:** Requests change their requirements over time.

For the evaluation of these use cases we do not use real-world models. We instead create approximated simulations of models with a wide variety of performance characteristics (see Chapter 5) and subject them to the loads we can expect from the use cases. In the following we describe the use cases in more detail.

Acronym	Latency importance	Accuracy importance	Requirements
SCP	High	Low	Batched
MDA	Low	High	Batched
PSA	Balanced	Balanced	Batched
ENV	Balanced	Balanced	Streamed

Table 4.1: Comparison matrix of the use cases

Smart-City Pedestrian Safety (SCP)

To keep pedestrians safe, smart-city operators can use ML to identify unsafe situations (e.g., traffic accidents, crowded areas, speeding cars, etc.) and instruct traffic participants to act accordingly by sending information to personal devices or smart cars. Because the pedestrians or smart cars need to get the information quickly in order to reduce reaction time, we need to minimize the latency of the model. The actions which the smart city takes based on these results are precautionary and therefore do not need to be 100% accurate. In real life, models for this use case are for tasks such as image classification or object detection. An example of this is CognitiveXR [RHS+21], a system capable of using ML models to detect unsafe situations and then sending warnings to pedestrians on their virtual reality headsets.

Medical Diagnosis Assistance (MDA)

Actions taken by medical practitioners can have a huge impact on the life of their patients. Therefore, they need the most accurate information achievable with the current state-of-the-art models. Even though it is important to get timely results, practitioners do not need to get results in real time. This domain includes Machine Learning models such as regression models or classification models. An example of this are ML tools to help with the diagnosis of patients via image data [LWL+23].

Public Sentiment Analysis (PSA)

Certain decisions taken by public officials impact many lives and therefore lead to a general sentiment in the public (e.g. in social media or news articles) [KAMM23]. By understanding this sentiment via ML models, officials can evaluate the impact of their decisions. This evaluation can then serve as input for future decisions. Public sentiment needs to both be evaluated as fast as possible and as accurately as possible, therefore requiring an equal trade-off between latency and accuracy. For this use case, models would be specifically trained to evaluate sentiment in text.

Environmental Monitoring (ENV)

Smart cities employ a variety of sensors to monitor the environment (e.g., air quality, noise pollution, etc.) [HKH+19]. Using ML to analyze the sensor data can help decision makers to plan for the future and react to current situations. Here, similarly to the Public Sentiment Analysis, we need to trade-off latency and accuracy. The main difference is the constant stream of data which needs to be analyzed, instead of a batch of data. Therefore, priorities can change over time and the platform needs to be able to adapt to those changes.

4.2 Architecture

To meet the design goals in Section 4.1 for the use-cases in Section 4.1.1 we propose the architecture in Figure 4.1.

In general, the architecture supports the following workflow, including the two main actors *administrator* and *developer* and the auxiliary actor *consumer*:

1. Administrators upload saved model files to the platform.
2. Administrators launch model executors which load the models from the storage and make them available for inference. When launching the model executors, metadata about the model is inserted into the metadata store. While the model executor is running, it periodically monitors the latency between model and clients and updates the metadata in the metadata store.
3. Developers launch clients on the platform, which register themselves in the metadata store and serve consumers. The developer defines the criteria by which the platform selects a model for the client.
4. Consumers send requests to clients that then query the platform for a suitable model and then send the request to the model.
5. Upon receiving the result from the model, the client documents the latency and accuracy performance of the request for further use in the system.

The architecture could be deployed in any sort of execution platform capable of hosting containerized applications and distribute them across multiple nodes. As discussed in Section 2.3, Kubernetes is a suitable platform for the implementation of serverless systems. Therefore, we implement the components as deployments and services inside of a Kubernetes cluster. Especially the concept of pods is useful for the implementation of the model executors, as they can be deployed as pods consisting of two containers: the model serving container and the companion container. Through the deployment of the architecture inside of a Kubernetes cluster, we implement the BaaS portion of a serverless system. Utilizing extensions to add support for FaaS clients would complete

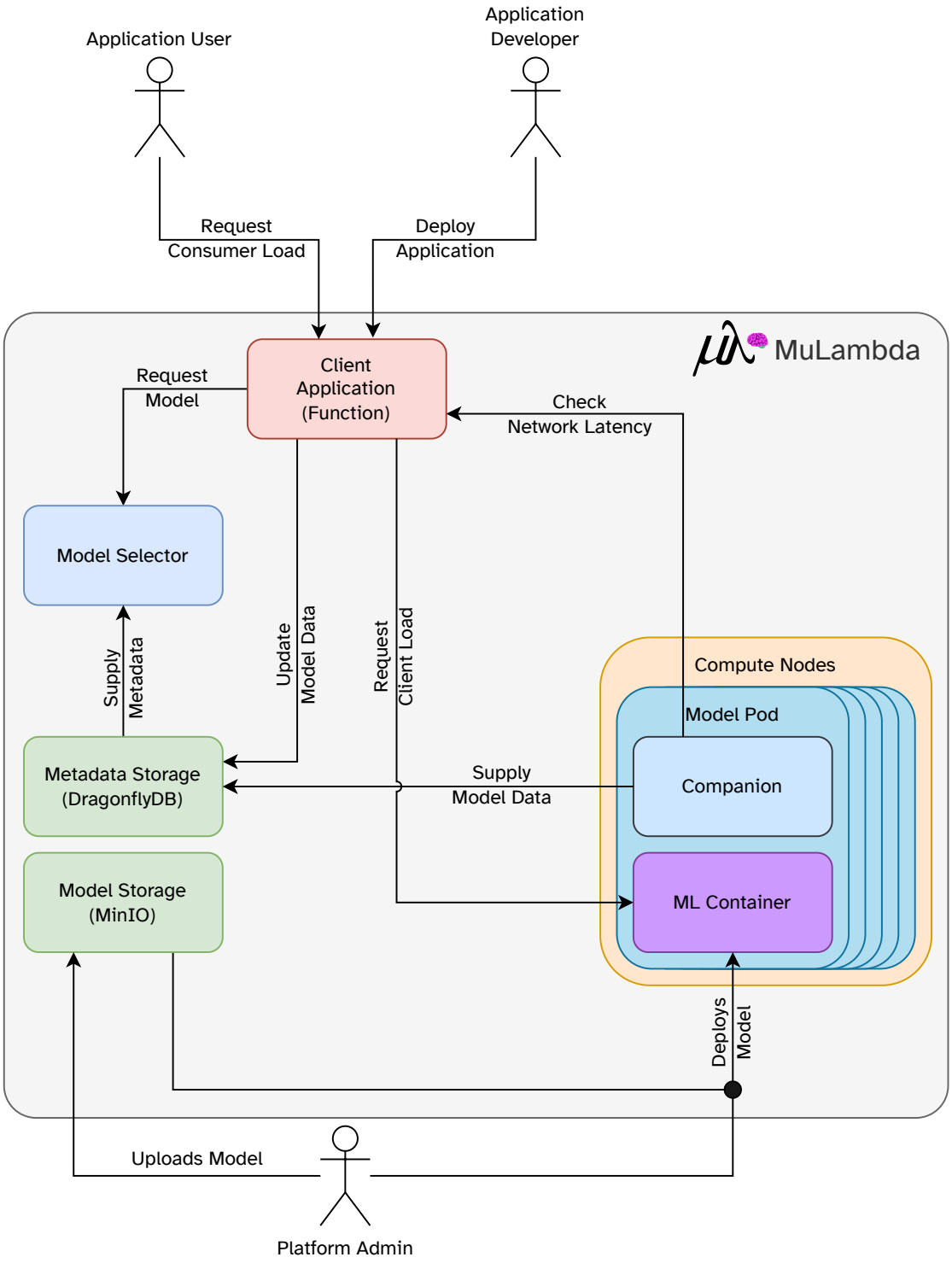

Figure 4.1: Architecture of the Serverless Machine Learning Inference Platform prototype

the system to a full serverless system. However, as our research focus lies on the "Model Selector" component of the architecture, we leave this extension to future work. The cluster runs on multiple nodes with different network latency separated into controller and worker nodes. More details on the concrete cluster setup are described in Chapter 5. The client, selector, metadata store and model storage run on the controller node, while the model executors run on the worker nodes. The following subsections describe the components used in the workflow in more detail.

4.2.1 Model Data Storage

The first step of making models available on the platform is to upload them to the storage. For this, we use MinIO[2], an open-source object storage server compatible with the Amazon S3 API. Amazon S3 is often used as the storage backend for serverless applications (see Section 2.1) which makes an S3 API compatible storage a good fit for the platform. This API is widely used in the industry and therefore provides a large set of tools for interacting with the storage.
The models are stored as Tensorflow SavedModels[3], which is a format for storing machine learning models. This format is widely used in the industry and therefore provides a large set of tools which can be used to interact with the models. Therefore, we meet **DG2** from Section 4.1.

In our evaluation the model data is stored centrally on the controller node because we do not schedule the model executors automatically in our experiments. Therefore, all models are already running on the cluster and do not need to be transferred between nodes during experiments. In future work however, we need to adapt the platform to use MinIO in a distributed configuration [Min23], which would then reduce the transfer time of model data to execution nodes.

The storage also offers the possibility to easily launch the models with Tensorflow Serving[4], a widely used framework for serving machine learning models and easy to integrate with the platform. Other formats like TorchScript[5] can be added in future work. As described in Section 4.1 in **NG1**, the platform does not maintain a repository of models. Instead, administrators can select and upload models to the storage which they deem suitable for the applications developers will deploy on the platform.

4.2.2 Model Executor

After administrators have uploaded model data to the storage, they can launch model executors with it. The model executors are Kubernetes pods and consist of two containers: the model serving container and the companion container. The model serving container is a Tensorflow Serving container which loads the model from the storage and makes it

[2] https://min.io
[3] https://www.tensorflow.org/guide/saved_model
[4] https://www.tensorflow.org/tfx/guide/serving
[5] https://pytorch.org/docs/stable/jit.html

available for inference. The loaded model is kept in memory by the serving container and therefore it does not need to be loaded from the storage for every request. The serving container is accessible via a REST API and a gRPC endpoint which are both exposed by the Kubernetes pod. However, for the purposes of this thesis we will only use the REST API in the evaluation, as it is easier to use and integrate into the processes of the platform. Next to the serving container, the model executor also contains a companion container. The purpose of the companion container is to insert metadata about the model execution into the metadata store. To this end, the companion first inserts general information about the model into the metadata store and then periodically updates the plain network latency between the model and the clients. This metadata includes information such as the model type, the input and output format of the model, the accuracy of the model and the latency between the model and the clients. We describe the metadata in more detail in Section 4.2.3.

For the model selector in Section 4.2.4 we need to create scoring values for the models. The companion container helps us to create such scoring values for the latency of the models. The latency is measured by the companion container by periodically sending requests to all registered clients and measuring the time it takes to receive a response from each. As the network latency often can be assumed to be symmetric, we can use the latency from the model to the client as an approximation for the latency from the client to the model. However, the network latency alone is only part of the latency occurring when sending a request to the model. We notice that the latency depends on three main factors: the network latency, the model itself (different models have different execution durations) and the data which we send to the model. Through continuous measuring we already have the network latency and for each request we also have the data which we send to the model. Additionally we also have the Round-Trip Time (RTT) of each request. Therefore, we can approximate how long the model itself takes to generate a result by subtracting the network latency from the RTT, receiving the *model delay*. Finally, we also have to take into account the size of the data input into account. Larger inputs have an impact on the processing time of the model which is why we need to factor in some value which represents this size. In our evaluation we send strings of different lengths to models which simulate the processing time (see Section 5.2). In real world scenarios, we could derive this data length from the binary size of the input data. Having this representation of input size, we can simply divide the model delay by the data length, receiving the *Model Data Delay (MDD)*.

$$\text{MDD} = \frac{\text{RTT} - \text{network latency}}{\text{data length}}$$

If we then want to estimate the latency for a given request, we can use the the data length, the model data delay, and the network latency like this:

$$\text{estimation} = \text{network latency} + \text{data length} \times \text{MDD}$$

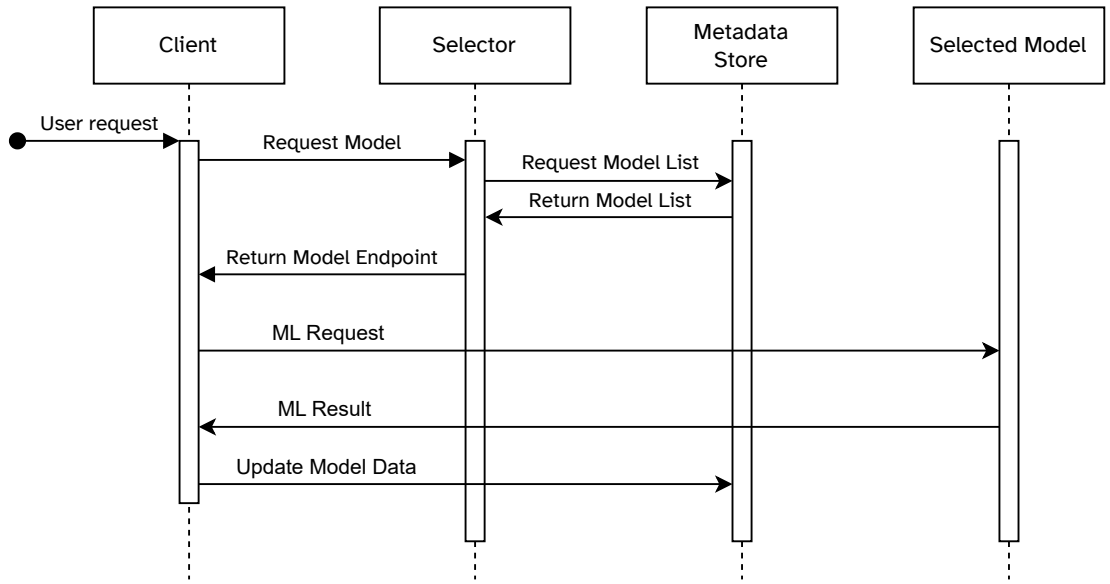

Figure 4.2: Model Request and Update Process

Note that this formula does neither factor the data length with the network latency nor consider how heavily it impacts the model delay. Therefore it does not portray the real world latency that will happen for the request under estimation. However, as it is only a rough scoring estimation, we can use this formula to estimate the latency for a given request. Figure 4.2 illustrates how the platform processes a request. It shows how the model data, like the MDD, gets updated by the client on each request.

Additionally to scores for the latency, the system also measures the accuracy of the model to create a score. As a reminder, we need both the latency and the accuracy to create a score for the model to enable efficient model selection for different use cases. Here, we assume that the model returns a confidence value for each prediction. We can use this confidence value to approximate an accuracy score for the model. Models which are more confident in their results therefore receive a higher accuracy score. As the confidence value depends on what kind of features the input contains, we cannot simply use the latest confidence value as the accuracy score. Instead, we need to calculate the average confidence value for each model.

Since the platform is a continuously running system, we cannot assume that the model accuracy will remain the same over time. Model drifts can occur and change the accuracy of the model over time [MCS23]. Therefore, we need to have a running average of the confidence score. However, this score should not be influenced too heavily by older results and therefore we use a window of the last 10 results. The window size can be adjusted in future work to evaluate the impact of the window size on the performance of the platform.

The process of launching model executors is not automated in the prototype and therefore

not part of the evaluation. Instead, we will launch model executors manually and then evaluate the impact of different selection strategies on the performance of the platform. In future work, we can automate the scaling of model executors by using a scheduler similar to the works described in [RLYK21] and [RRD+22]. This can serve as further input for future work, which can then focus on the scheduling of model executor pods.

4.2.3 Metadata Storage

The metadata storage is a Redis[6] API compatible key-value store which stores metadata about the models and clients. An external high-performance storage like this is necessary because the metadata receives updates very frequently and therefore needs to be able to handle a high throughput of requests. The prototype uses DragonflyDB[7] which implements all the needed Redis API functions and promises better performance than Redis itself. However, the prototype can be easily adapted to use Redis (or any other compatible key-value store) instead of DragonflyDB.

We describe the data model of the metadata store in Table 4.2 and further show which data is stored for each model in Table 4.3.

Key	Description
`mulambda:models`	The set of all models which are currently available on the platform
`mulambda:clients`	The set of all clients which currently run in the platform
`mulambda:models:<model_id>`	A hash containing metadata about the model with the id **`model_id`**. ... shows the fields it consists of

Table 4.2: Keys in the metadata store

4.2.4 Model Selector

The selection of models is one of the components which contributes to the automatization aspect of the platform. Because the platform is able to automatically select the best model given specific criteria, developers do not need to explicitly choose a model for their application. This is especially useful for applications which do not have many hard requirements on the model, but instead want a general kind of functionality and performance characteristics. The approach to use machine learning models based on their traits instead of their identity therefore lets developers focus on their product instead of the underlying models, of which there is a vast amount. The usefulness of this is evident

[6] `https://redis.io`
[7] `https://dragonflydb.io`

Field	Description
id	The identifier of the model executor, which is also the name of the Kubernetes service associated with the model executor
name	The name of the model identifying the model data in the model data storage
type	The type of the model, such as classification, regression
input	The format of the input data which the model expects
output	The format of the output data which the model produces
accuracy	A value between 0 and 1, where 1 is the highest possible accuracy. In Section 4.2.2 we explain how we produce this data
latency:<client_id>	The network latency between the model executor and the client with a given **client_id** in milliseconds. We describe how we measure this value in Section 4.2.2
mdd	The Model Data Delay value we introduce in Section 4.2.2
...	Other values which help in addressing the model once it is selected—information such as the path or the port which clients need to access

Table 4.3: Fields in **mulambda:models:<model_id>**

for applications which just want to receive the best possible output for a given input, but do not care which model generated this output. Furthermore, distributed systems benefit by automatically receiving co-located models which can lead to an overall decrease in latency.

To even be able to perform model selection, we first need to devise a classification scheme across the models that the platform serves. Based on this classification scheme, developers can specify criteria by which the platform should select the best model.

Hard criteria	Soft criteria
Model Type	Latency
Input Format	Accuracy
Output Format	

Table 4.4: Criteria for model selection

In Table 4.4 we can see that we have to consider two different kinds of criteria:

Hard Criteria are criteria essential to the client application to function as intended. The platform will never select a model which does not meet these criteria. The available hard criteria are:

- **Model Type:** The main function of the model. Here, we distinguish between models for tasks such as classification and regression. This is a hard criterion because the outputs of different types of models are not compatible with each other.
- **Input Format:** The kind of data client applications infer the model with (i.e. text, image, audio, etc.). Those inputs are in general incompatible with each other and therefore this is a hard criterion.
- **Output Format:** The kind of data the client application expects as output (i.e. text, image, audio, etc.). Client applications have a specific purpose and work on a specific kind of data which is mostly non-negotiable. Thus, this is a hard criterion.

Soft Criteria are criteria which let the client application signal what kind of performance it expects—i.e. if it wants faster or better results. Based on the requested performance profile given by the soft criteria, the platform will select the most suitable model meeting the hard criteria. The available soft criteria are:

- **Latency:** The time it takes for the model to deliver an output to the client for a given input. Even though we always want to be as fast as possible, latency performance can be traded off for better accuracy. This trade-off makes latency a soft criterion. With latency, we also consider the different components which contribute to the overall latency. Those components are the network latency, the model delay and the data length and are described in detail in Section 4.2.2.
- **Accuracy:** The quality of the output of the model. Analogous to latency, higher accuracy can be traded off for worse latency, making accuracy a soft criterion.

To achieve the design goals, we need to make different considerations regarding those soft criteria.

Model Selection Algorithm

To perform model selection on the hard and soft criteria, we use a weighted sum model selection algorithm. We showcase the algorithm by providing simplified listings of the functions used in the algorithm. We leave out implementation details irrelevant for the understanding of the algorithm. These are only relevant to the specific infrastructure we use in the prototype. Listing 1 shows the auxiliary functions we use in the algorithm.

The function `estimate_performance` calculates the performance that we can expect of a given model for a specific request length with a specific request client. It utilizes the inversion of the *Model Data Delay* which we describe in Section 4.2.2 to calculate

```
def estimate_performance(model: Model,
                         data_length: int,
                         request_client_id: str
                         ) -> (float, float):
    latency = (model.mdd * data_length) \
              + model.current_latency_with(request_client_id)
    accuracy = model.latest_accuracy()
    return latency, accuracy

def normalize_latency(latency: int) -> float:
    pass

def normalize_accuracy(accuracy: float,
                       max_accuracy: float) -> float:
    return accuracy / max_accuracy

def score(soft_criteria: Dict[str, float],
          normalized_latency: float,
          normalized_accuracy: float) -> float:
    return - soft_criteria["latency"] * normalized_latency \
           + soft_criteria["accuracy"] * normalized_accuracy
```

Listing 1: Auxiliary functions for model selection

the expected latency for a given request length. It also uses the data the infrastructure provides on the network latency between the model and the client. For estimating the accuracy, we use the moving average of the confidence value that we describe in Section 4.2.2. To be able to compare different performance values to each other we normalize the values between 0 and 1. For the latency, we use predefined buckets which we can adjust to the needs of the application and infrastructure. These buckets are intervals of latency values which we map to a normalized value between 0 and 1. Future work can explore automated adjustment of the buckets.

The centerpiece of the selection is the weighted scoring we defined in the function **score**. It takes into account the soft criteria which we describe above and calculates a weighted sum of the performance values. Note, that we use the negative of the latency as the performance value for latency as we want to decrease the latency, as opposed to the accuracy which we want to increase.

In Listing 2 we can see (a simplified representation of) the actual model selection algorithm and how we utilize the auxiliary functions above. We base our selection on multi-criteria decision making approaches, utilizing the weighted sum of utility values. We discussed these approaches in Section 3.3 where they were presented in [TSS22]. It takes as input the set of models available to the platform, the hard and soft criteria, the length of the request and the id of the client which sent the request. We take the available models from the metadata store before we start the selection. The function then first takes out all the

models which do not meet the hard criteria. Then, all the models receive a performance estimation, by utilizing the **`estimate_performance`** function. We then normalize the performance values to increase comparability. Finally, we calculate the weighted sum of the performance values for each model and select the model with the highest score.

4.2.5 Client implementation

Clients are meant to be implemented by developers of applications which utilize the platform. For the purpose of this thesis, we implement a client which can be used to simulate different request patterns. Those request patterns correspond to the different use cases in Section 4.1.1. For example, the use case for pedestrian safety involves a client which requires low latency responses and does not need high accuracy. We describe the simulation for this in Chapter 5. To test out higher accuracy requirements as in the medical diagnosis use case, we can adjust the weights that are sent to the selector accordingly in the client. For the platform to function properly, clients also need to send updated performance data (i.e. latency and accuracy) to the platform.

The evaluation client is implemented as a Python web application which uses the FastAPI framework to expose a REST API. The web application runs as a Kubernetes pod which makes it easy to deploy it to the platform. On startup the client registers itself in the metadata store so that the companion containers of the models can measure the network latency between the client and the model.

Interfacing with the platform is low-friction and can be easily integrated into existing applications. In Listing 3 we can see a simple example of how a client can request a model from the platform. Clients just need to send simple HTTP requests to the platform, containing the hard and soft criteria which they want to use for model selection, the length of the data they want to send, and the URL of the selector. The result of the request then contains all the information needed to send the request to the model, including the endpoint, the port and the path.

After receiving the result from the model, the client sends the performance data to the platform. This is also as frictionless as possible as we can see in Listing 4. Clients just need to supply the model ID and the performance data to the platform. From the total elapsed time and the most recent network latency measurement, the platform can calculate the model delay and the MDD we defined before in Section 4.2.2. Along with that, the client also sends the current rolling average of the confidence value of the model via a Redis connection to the metadata storage.

Using those simple components, it is easy to create low-effort APIs for applications which utilize the platform. In Python for example, those components can easily be hidden behind a decorator to result in a low-footprint API like in Listing 5. Here, user-defined functions can be decorated with the **`@inject_model`** decorator which then injects a `ModelConnector` in the arguments of the function. This `ModelConnector` then handles the retrieval of a suitable model and the sending of the performance data to the platform.

However, for the evaluation of the platform, we will not use such an API but instead use the lower level API described in Listing 3 and Listing 4.

```python
def select_model(available_models: List[Model],
                 hard_criteria: Dict[str, Any],
                 soft_criteria: Dict[str, float],
                 data_length: int,
                 request_client_id: str) -> Model:
    # Filter out all models which do not fit the hard criteria
    suitable_models = [
        model
        for model in available_models
        if model.fits(hard_criteria)
    ]

    # Estimate the performance of each model
    # for a request of the given length and client id
    model_performance = [
        (model, estimate_performance(model,
                                     data_length,
                                     request_client_id))
        for model in suitable_models
    ]

    # Find the highest possible accuracy
    max_accuracy = max(model_performance,
                       key=lambda x: x[1][1])[1][1]

    # Normalize the latency and accuracy for each model
    normalized_performance = [
        (
            model,
            (normalize_latency(latency),
             normalize_accuracy(accuracy, max_accuracy))
        )
        for (model, (latency, accuracy)) in model_performance
    ]

    # Calculate the score for each model
    # and select the highest score
    selected = max(normalized_performance,
                   key=lambda x: score(soft_criteria,
                                       x[1][0],
                                       x[1][1]))
    return selected
```

Listing 2: Weighted sum model selection algorithm

```python
def get_model(
    hard_criteria: Dict[str, Any],
    soft_criteria: Dict[str, float],
    data_length: int,
    selector_url: str
) -> (str, Dict):
    response = httpx.post(
            selector_url,
            json={
                "required": hard_criteria,
                "desired": soft_criteria,
                "data_length": data_length,
            }
            )
    traits = response.json()["model"]
    return (
        f"http://{response.json()['endpoint']}"
        f":{traits['port']}{traits['path']}",
        traits,
    )
```

Listing 3: Simple HTTP request for receiving a model endpoint

```python
def send_performance(model_id: str,
                     data_length: int,
                     elapsed: float,
                     net_latency: float,
                     avg_accuracy: float):
    mdd = max(elapsed - net_latency, 0) / data_length
    metadata = get_metadata_server()
    metadata.hset(model_id, {"mdd": mdd, "accuracy": avg_accuracy})
```

Listing 4: Sending of performance data to the platform

```python
@inject_model(hard_criteria, soft_criteria)
def user_function(model: ModelConnector):
    result = model(data)
    ...
```

Listing 5: Low footprint API for using the platform

CHAPTER 5

Experimental Setup

This chapter describes the experimental setup for the evaluation of the platform prototype we describe in Chapter 4. First, we showcase the evaluation environment in Section 5.1. Then, we explain the structure of our evaluation. The evaluation is split into two parts: a qualitative evaluation based on the use cases in Chapter 4 and a quantitative evaluation based on benchmarking the platform. The qualitative evaluation is described in Section 5.2 and the quantitative evaluation in Section 5.3.

As a reminder we note that the evaluation investigates the following research questions:

RQ1 Which benefits can be achieved for latency-sensitive applications by using a weighted model selector compared to a round-robin, random or plain-net-latency model selector?

RQ2 Which benefits can be achieved for accuracy-sensitive applications by using a weighted model selector compared to a round-robin, random or plain-net-latency model selector?

RQ3 What differences in behavior can we expect for applications with varied latency and accuracy preferences between the different use cases in the weighted selection compared to the baselines?

RQ4 How does the selection scale with increasing message sizes for the different selection algorithms?

RQ5 How does the selection scale with an increasing number of concurrent requests served by the platform for the different selection algorithms?

5.1 Evaluation Environment

To answer the research questions we evaluate the platform prototype in a partially simulated environment. The platform prototype is deployed on a Kubernetes cluster, with the models being deployed to different nodes. This technology stack resembles deployments already used in existing research [RLYK21, RRD+22]. In a real world deployment though, the models would be deployed to different regions or on the edge, which would result in different latencies. To simulate those different latencies between the models and the selector, we use the Linux Traffic Control (tc) tool [Hub01]. This tool allows us to simulate different network conditions between the nodes. To test different latency scenarios at once, we deploy the platform onto four nodes, with the selector and the client being deployed to the same node. The other three nodes host the models, with the latency between the selector and the models increasing with each node.

The Kubernetes cluster runs in a virtualized environment made available by the Distributed Systems Group at the TU Wien. All nodes are virtual machines running Rocky Linux[1] 9.1 with 8 vCPUs and 16 GB of RAM. For experiment monitoring, we use the extended Galileo framework introduced in [RRP+22]. It is capable of monitoring experiment events for applications running on Kubernetes clusters. Raith et al. also demonstrate the use of the framework in the domain of Edge Intelligence, making the framework a good fit for our evaluation.

Model Simulation

To cover the different use cases, we deploy different models to the nodes. To achieve significant variety between the models, we simulate their characteristics with a dummy model. Using real world models for the evaluation is not feasible, as the variety would be too low and the models would be to difficult to fine-tune for the different scenarios. Therefore, we create a web server which simulates the execution of a machine learning model based on two main values: the *delay* and the *confidence*.

- **Delay of model m for input i (D_{mi})**:
  The delay is the time it takes for the model to reach a result. It consists of (1) an inherent base delay of the model D_m (2) the size of input i (S_i) times the delay data size impact of the model α_m (3) the amount of concurrently served requests (N_{mi}) times the concurrency impact of the model β_m and (4) a random jitter J_m.

$$D_{mi} = D_m + S_i \times \alpha_m + N_{mi} \times \beta_m + J_m$$

- **Confidence of model m for input i (C_{mi})**:
  The confidence is the probability that the result of the model is accurate. To generalize across all different types of models, we consider that models act on inputs i which contain features $f \in F$. The confidence is the probability that the model

[1] https://rockylinux.org

m is correct for a feature f_k in input i. It consists of (1) the maximum confidence (C_{mk}) that m can have for a specific feature f_k (2) the total amount of features m supports F_m (3) the confidence feature set size impact of the model γ_m (i.e. how exact can the model act on different features with an increasing amount of supported features).

The idea behind that is that some features are more actionable than others, leading to a higher maximum confidence for those features. Additionally, the more features a model supports, the less confident it can be about each individual feature. Depending on the quality of the model this confidence descent can be more or less steep. To keep the dimensionality of the experiment low, we assume that all models support the same features, but only differ in their confidence feature set size impact and the maximum confidence score for each feature. To illustrate this, we consider models which support three features $F_m = \{f_1, f_2, f_3\}$, all with different sets C_{mk}. For the purpose of the experiments we choose the maximum confidence for each feature per model in a way that f_1 is the most actionable feature, f_2 is less actionable and f_3 is the least actionable.

$$C_{mi} = \begin{cases} 0 & \text{if } F_m = 0 \\ C_{mk} * \frac{1}{F_m^{\gamma_m}} & \text{otherwise} \end{cases}$$

See Table 5.1 for an overview of the models and their performance traits. We choose these values to simulate the different performance characteristics of real-world models, such as image recognition models or natural language processing models. Models can vary in their execution time and accuracy. In general, the more accurate a model is, the longer it takes to execute.

Name	m	D_m	α_m	β_m	γ_m	max $J_m[ms]$	C_{mk}
FastBad	1	10	0.001	0.1	0.1	5	[0.5, 0.3, 0.1]
FastOK	2	15	0.002	0.35	0.2	5	[0.8, 0.5, 0.3]
FastGood	3	20	0.004	0.7	0.3	5	[0.9, 0.7, 0.5]
MidBad	4	50	0.01	0.1	0.4	30	[0.55, 0.35, 0.15]
MidOK	5	75	0.02	0.35	0.5	30	[0.85, 0.55, 0.35]
MidGood	6	100	0.04	0.7	0.6	30	[0.95, 0.75, 0.55]
SlowBad	7	150	0.06	0.1	0.7	60	[0.6, 0.4, 0.2]
SlowOK	8	175	0.07	0.35	0.8	60	[0.9, 0.6, 0.4]
SlowGood	9	200	0.08	0.7	0.9	60	[1.0, 0.8, 0.6]

Table 5.1: The models used in the evaluation and their performance traits

We explore different schedules for the models to get a better understanding of the behavior of the model selector. The behavior that we observe can then be used to inform the design of an automated scheduler like in [RLYK21] or [RRD+22] in the future. See Table 5.2 for an overview of the different schedules of models $m \in M$. We remind the reader that nodes 1 to 4 have increasing latency to the selector—specifically $[10, 30, 100, 300]ms$.

Schedule	Node 1	Node 2	Node 3	Node 4
Logical	{1, 2, 6, 9}	{4, 5, 6, 9}	{6, 7, 8, 9}	{1, 7, 8, 9}
Arbitrary	{3, 7, 2, 9}	{5, 1, 8, 4}	{6, 9, 2, 1}	{4, 8, 7, 3}
Parity	{1, 5, 6, 9}	{1, 5, 6, 9}	{1, 5, 6, 9}	{1, 5, 6, 9}
Hyperlocal	M	$\emptyset$	$\emptyset$	$\emptyset$

Table 5.2: The model schedules tested in the evaluation

The **Logical** schedule contains logical assumptions about a useful schedule for the use cases in Chapter 4. We put the fastest models on the nearest node and the slower models on the farther nodes, in order to maximize the latency of the results. Furthermore, we put the most accurate, but also slowest model on every node, to maximize the accuracy of the results. This way we assume that the selector has good targets to choose from regardless of the client's preferences.

The **Arbitrary** schedule is a schedule which is not based on any assumptions. We generated it by randomly shuffling the models and assigning them to the nodes. We do not use random schedules every time, as we want to be able to compare multiple experiment runs to each other.

The **Parity** schedule is an edge case where all nodes have the same models deployed. These models offer a range from the fastest possible model to the most accurate model.

The **Hyperlocal** schedule is an edge case where we deploy all models from Table 5.1 to the first node. This way, we remove network latency as a factor for the model selector and can observe the other performance aspects separately.

5.2 Qualitative Evaluation - Case Studies

The results of these case studies answer the research questions **RQ1**, **RQ2** and **RQ3**. Each case study presents a different implementation of the client implementation described in Chapter 4. For each case study, we run experiments sending requests ranging from 500 requests to 1000 requests, with the data length (i.e. the abstract size) of the requests being 10, and the concurrency of the requests being 5 (i.e. 5 requests are sent at the same time). Furthermore, each concurrent stream of request goes through their requests serially, i.e. the next request is only sent after the previous request has finished. Therefore,

choosing models which take long to process slows the arrival rate of requests to the selector.

The research questions are matched to the different use cases as follows:

- **RQ1** ties into the Smart-City Pedestrian Safety use case. To answer it, we implement a client which strongly prefers low latency (weight is -1 to minimize) over high accuracy (weight is 0). We use models with different execution times to simulate the different latencies. We analyze, over the course of the simulation, how the latency of the client changes, depending on the model schedule for each of the different model selectors. We also monitor the side effects to the accuracy of the results and resource utilization of the platform.
- **RQ2** ties into the Medical Diagnosis Assistance use case. Here, we implement a client which strongly prefers high accuracy (weight is 1 to maximize) over low latency (weight is 0). The models that we schedule for this experiment have different static accuracies associated to them. We label each result with the accuracy of the model that produced it. We then analyze how the accuracy of the results changes over time for each of the different model selectors. As with RQ1, we also monitor the side effects to the latency of the results and resource utilization of the platform.
- **RQ3** connects well to the use cases Public Sentiment Analysis (PSA) and Environmental Monitoring (ENV), but the previous two use cases would also apply, as we monitor the side effects to the latency and accuracy of the results. For both the PSA as well as the ENV use case, we implement a client with a balanced preference for latency and accuracy. We differ these use cases by having the PSA use case maintain a perfectly balanced preference (weights -0.5 for latency and 0.5 for accuracy) and having the ENV use case change its preference over time (latency in the interval (-1,0) and accuracy in (0,1) with abs(latency) + abs(accuracy) = 1). We then analyze how the latency and accuracy of the results changes over time for each of the different model selectors in both use case implementations. As with RQ1 and RQ2, we also monitor the resource utilization of the platform.

Table 5.3 summarizes the four different clients which directly correspond to the use cases.

5.3 Quantitative Evaluation - Benchmarking

The results of the benchmarks answer the research questions **RQ4** and **RQ5**. The goal of the benchmarks is to analyze the scalability of the platform. While the case studies in Section 5.2 analyze the behavior of the platform in different scenarios, the benchmarks analyze how performance metrics change with an increasing load on the platform. Since our approach is able to react to changes in the environment, we want to analyze how the platform behaves when model performance degrades due to increased load. Each

client	RQ	request pattern
SCP	**RQ1**	batched
MDA	**RQ2**	batched
PSA	**RQ3**	batched
ENV	**RQ3**	streamed

Table 5.3: Summary of the different clients and their request patterns

benchmark analyzes the three core performance metrics of the platform: latency, accuracy and resource utilization. We perform the benchmarks on the same Kubernetes cluster as the case studies.

The first benchmark analyzes the scalability of the platform with an increasing size of messages and aims to answer **RQ4**. For this benchmark, we change the abstract message size that we send to the models and analyze how the latency and accuracy of the results change for each of the different model selectors. We use the **Logical** schedule from the case studies. The size of the messages increases on a fibonacci-like scale, that being $[1, 3, 5, 8, 13, 21, 34, 55, 89]$. With the way in which we design the model simulation, those sizes result in different execution times for the models. For the other request parameters, we reduce the load to 100 requests and the concurrency to 1, i.e. we send 100 requests one after the other. We want to avoid increasing the load through competing aspects of the platform, such as the concurrency of the requests. Also, increasing the load too much would result in benchmarks that take too long to run.

The second benchmark analyzes the scalability of the platform with an increasing number of concurrent requests and aims to answer **RQ5**. Here, we have a fixed number of models deployed to the platform (we use the **Logical** schedule from the case studies) and increase the number of clients which send requests to the platform. As with the first benchmark, we analyze how the latency and accuracy of the results changes for each of the different model selectors. The amount of concurrent requests follow the same fibonacci-like scale as the message sizes in the first benchmark. Again, the way in which we design the model simulation causes the models to have increasing execution times with more concurrently handled requests. To reduce the interference between different kinds of loads, we use a fixed message size of 10 for all requests. We also only send concurrent requests in one shot. That means, we send all requests at the same time and do not send any further requests to achieve a stable amount of concurrent requests. With this benchmark, we also need to remain mindful of the time it takes to run the benchmark to be able to run it multiple times.

CHAPTER 6

Results and Discussion

This chapter presents and discusses the results of the case studies and benchmarks presented in Chapter 5. We contextualize the results with respect to the research questions from the same chapter. Section 6.1 discusses the results of the case studies and answers research questions 1-3, while Section 6.2 discusses the results of the benchmarks and answers research questions 4-5. We answer each research question individually, and then summarize the results in Section 6.3.

6.1 Case Study Results

RQ1 - Which benefits can be achieved for latency-sensitive applications by using a weighted model selector compared to a round-robin, random or plain-net-latency model selector?

As we state in Chapter 5 we answer this research question by analyzing the results of the latency-focused use case (SCP). To evaluate how well the weighted model selector performs in terms of latency compared to the other selectors, we look at the Empirical Cumulative Distribution Function of request latencies.

Figure 6.1 shows the cumulative distribution of latencies for all clients in the **Logical** and **Arbitrary** schedules. Overall, the weighted model selector has the lowest median latency of all selectors. This holds true for the hand-picked **Logical** schedule as well as for the **Arbitrary** schedule we randomly pre-generated (see Section 5.1). Most of the requests are served in less than 300 ms, with the 95th percentile being at around half a second. We can also notice a clear distinction to the distributions of the other selection algorithms. The algorithm producing the second most requests with a latency of less than half a second is the selector which chooses solely based on host latency. However, we can notice that the distribution of the plain network latency selector is much more spread out. The 95th percentile of the plain network latency selector is the worst of all selectors. For

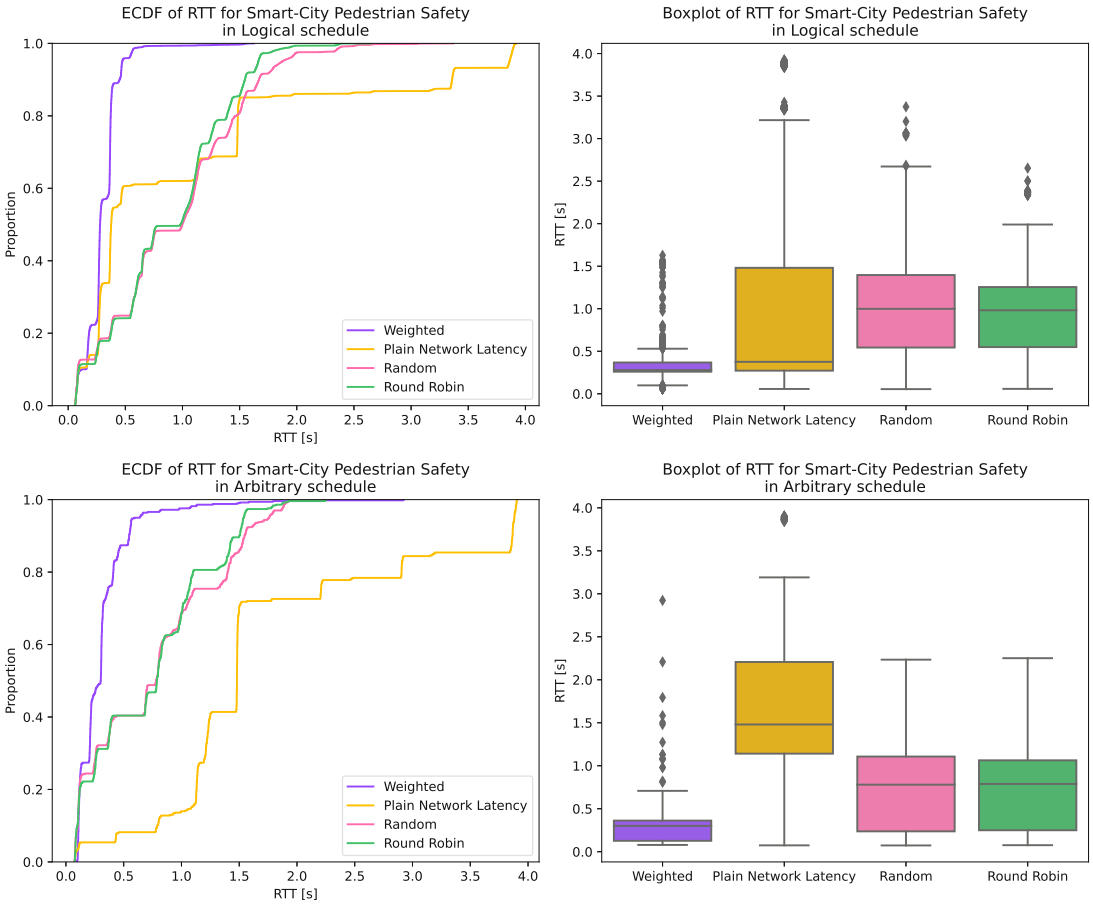

Figure 6.1: Cumulative distribution and box plots of latencies for all clients for **Logical** and **Arbitrary** schedules in the SCP use case

the remaining two selectors we notice a very similar distribution between them, having a median latency of around a second and a 95th percentile of around 1.8 seconds per request.

For better understanding of these ECDFs we can also look at the time series data of the latencies in a given experiment in Figure 6.2. The experiment ran in the **Logical** schedule with all of the selectors and with a total of 1000 requests sent to each of them (see Section 5.2 for further details on the experiment setup). The first difference we can notice is the length of the different experiments. The weighted model selector experiment took roughly 450 seconds—so around 0.45 seconds per request—while the other experiments took more than double the time. We can also see the reason for that in the time series data, as the weighted model selector has a much lower average round trip time than the other selectors.

The second noticeable difference is in the confidence intervals of the data. The weighted

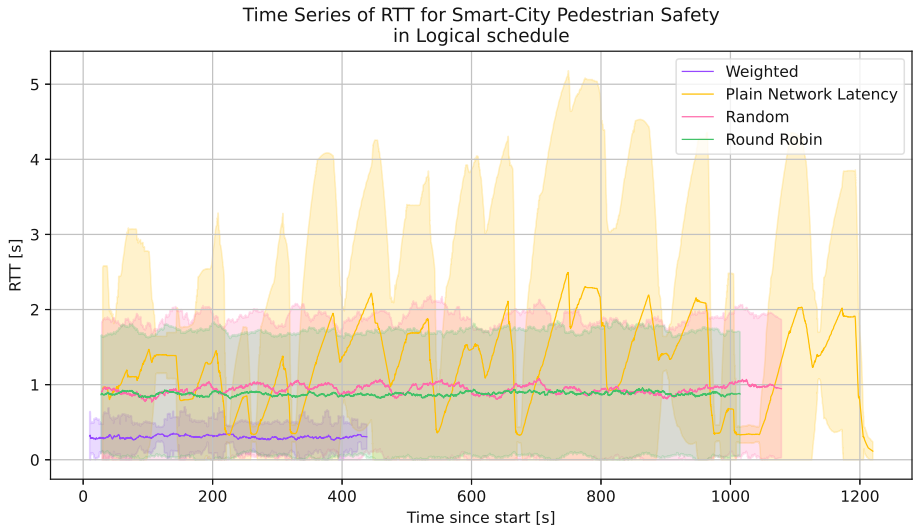

Figure 6.2: Rolling average time series of latencies for all clients for the **Logical** schedule in the SCP use case with a 90% confidence interval

model selector has much more stable latencies than the other selectors. The time series also shows the reason for the increased amount of low-latency response times for the plain network latency selector. The variance of the results of this selector is much higher than the other selectors, which leads to a lower median. This comes at the cost of having a significantly increased amount of high latency results.

To gain deeper understanding of the results we can also look at the accuracy of the different selectors in this use case. In the ECDF of the accuracies in Figure 6.4 the weighted model selector performs worst in terms of accuracy because its curve is above the other selectors. This, however, is not the goal in this scenario, as the optimization should go into the latency of the requests. When we compare the time series results of the accuracy in Figure 6.3 with the time series results of the latency in Figure 6.2 we can see that the points of lowest accuracy for the plain network latency selector are also the points of lowest latency (e.g., around 550 seconds into the experiment). This is trivial because we designed the models in such a way that they have a higher model delay when they also deliver more accurate results. We can therefore deduce that the weighted model selector is able to achieve the lowest latencies by sacrificing accuracy. This is exactly what we want to achieve with the weighted model selector, since we set the weights to just focus on the latency without caring about the accuracy.

We also look at the edge cases in terms of scheduling, i.e. how do the different selectors perform when there is only a single node, or when the nodes contain the same models. In Figure 6.5 the results for the **Hyperlocal** schedule are very similar to the results of the **Logical** schedule.
However, the benefits gained by the weighted model selector are less pronounced in this

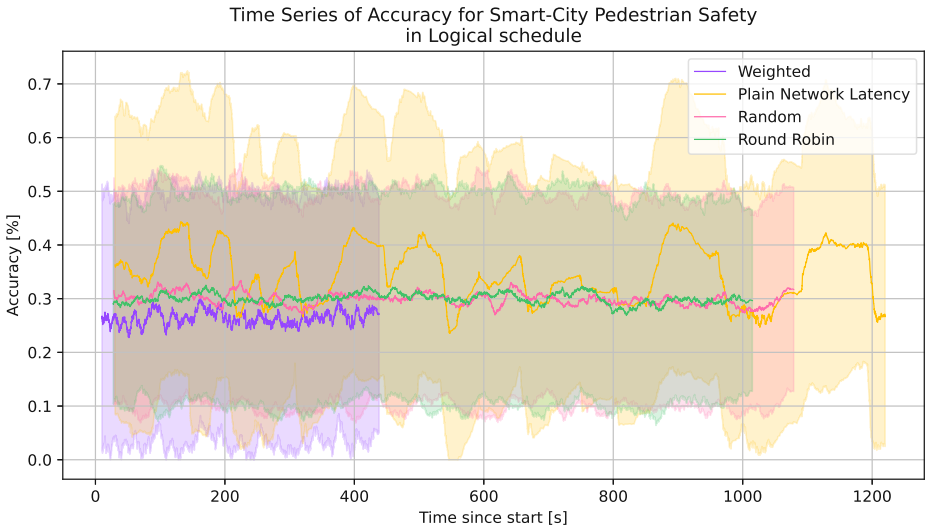

Figure 6.3: Rolling average time series of accuracies for all clients for **Logical** schedule in the SCP use case with a 90% confidence interval

schedule. The reason for this is that there is less variance in the latency of the models because they all run on the same node. Therefore the possible gain of selecting a model with a lower latency is lower.
On the other hand, the plain network latency selector has worsened results since the network latency of all models is around the same—they all run on the same host. This should theoretically deliver similar results to the random selector. However, we can see that the random selector performs better in this case. A possible explanation is that the network latency is updated in 5 second intervals, which means that the latency of the models is not always up to date. This can lead to the plain network latency selector overloading a model and therefore increase its delay while keeping the network latency the same. Also, the distribution of the latencies is possibly not uniform and could be skewed by milliseconds towards models with a higher delay. This leads to the random selector having a higher chance of selecting a model with a lower latency than the plain network latency selector.

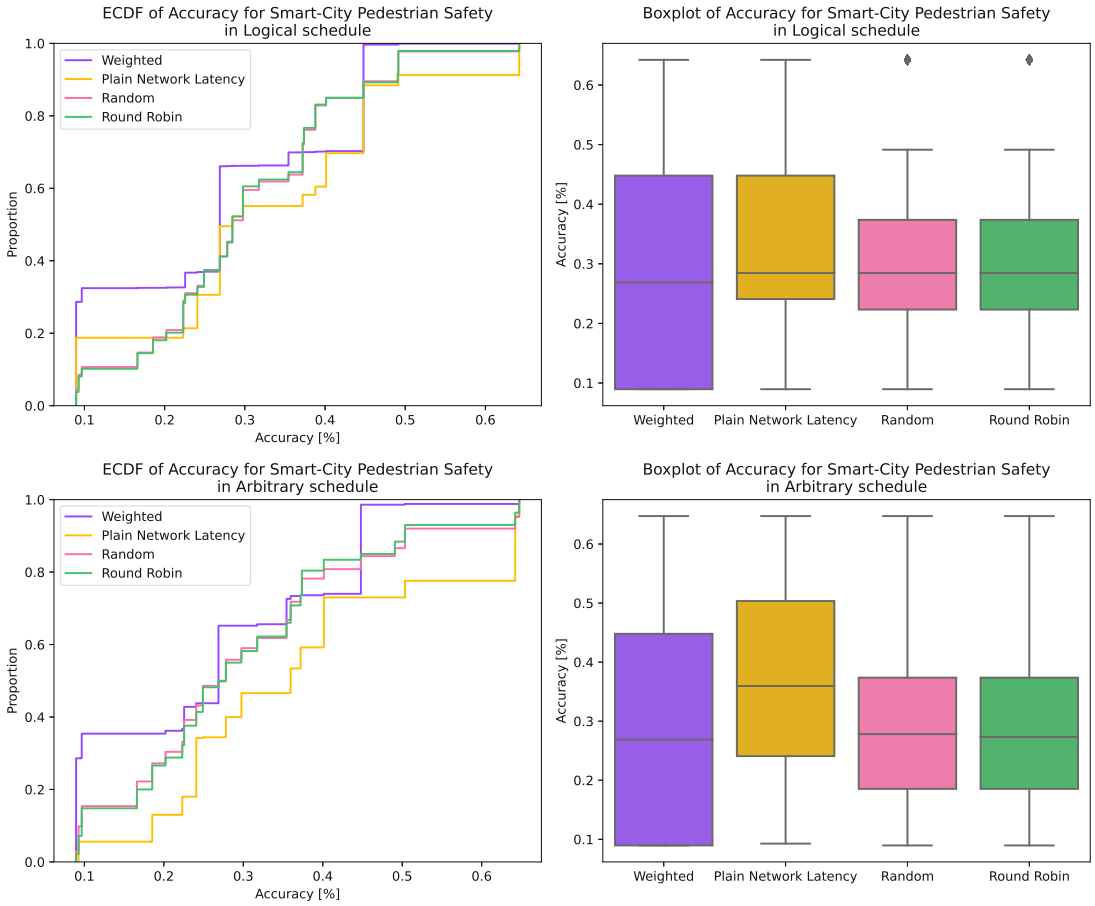

Figure 6.4: Cumulative distribution and boxplots of accuracies for all clients for **Logical** and **Arbitrary** schedules in the SCP use case

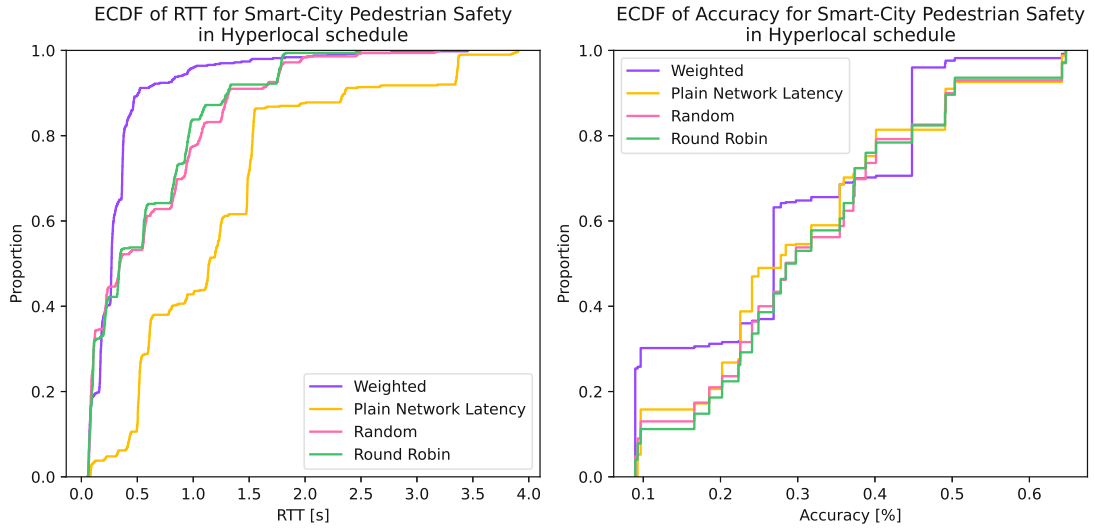

Figure 6.5: Cumulative distribution of latencies and accuracies for all clients for the **Hyperlocal** schedule in the SCP use case

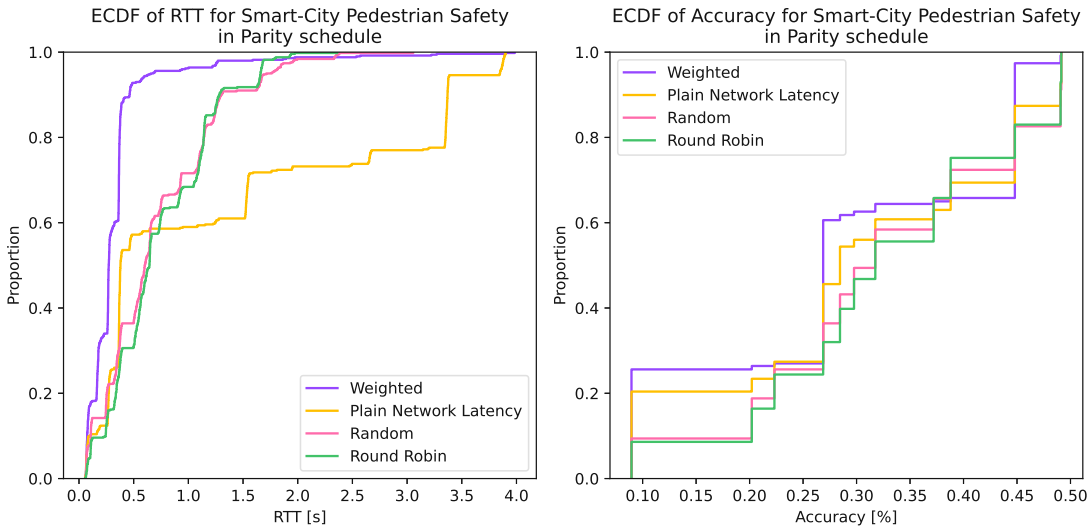

Figure 6.6: Cumulative distribution of latencies and accuracies for all clients for the **Parity** schedule in the SCP use case

For the edge case where all nodes contain the same models (**Parity**) we can see in Figure 6.6 that the results resemble these of the **Logical** schedule. That is, the weighted model selector performs best in terms of latency in all regards, while the plain network latency selector has a high amount of low latency results, but also a high amount of high latency results. The reason for this is that by going strictly with the lowest network latency, this selector will always choose the models on the same node. By over-utilizing one node, the latency of the models will increase, which leads to the high amount of high latency results. This does not happen with the random and round-robin selectors, as they distribute the load more evenly over the nodes. It also does not happen with the weighted model selector, as it does not only consider the network latency, but also the model delay and therefore prefers only the fast models on each node.

We can further compare the utilization of nodes by the different selectors in the **Logical** schedule in Figure 6.7. This figure shows which nodes get utilized the most by the different selectors at given points in time in a given experiment. We use handled requests as a proxy for utilization, as the more requests a node handles, the more utilized it is. The nodes are separated by shades, with the darkest shade being the node with the lowest network latency and the lightest shade being the node with the highest network latency. With both the random and round-robin selector we can see the predictable pattern of the load being distributed evenly over the nodes. There is a slight increase in the load of the node with the highest latency (the lightest shade), as requests arriving at this node take longer to process. Because of this increased processing time there are more requests in the queue of this node, which leads to a higher utilization. The plain network latency selector, however, has a strong focus on the node with the lowest network latency (the

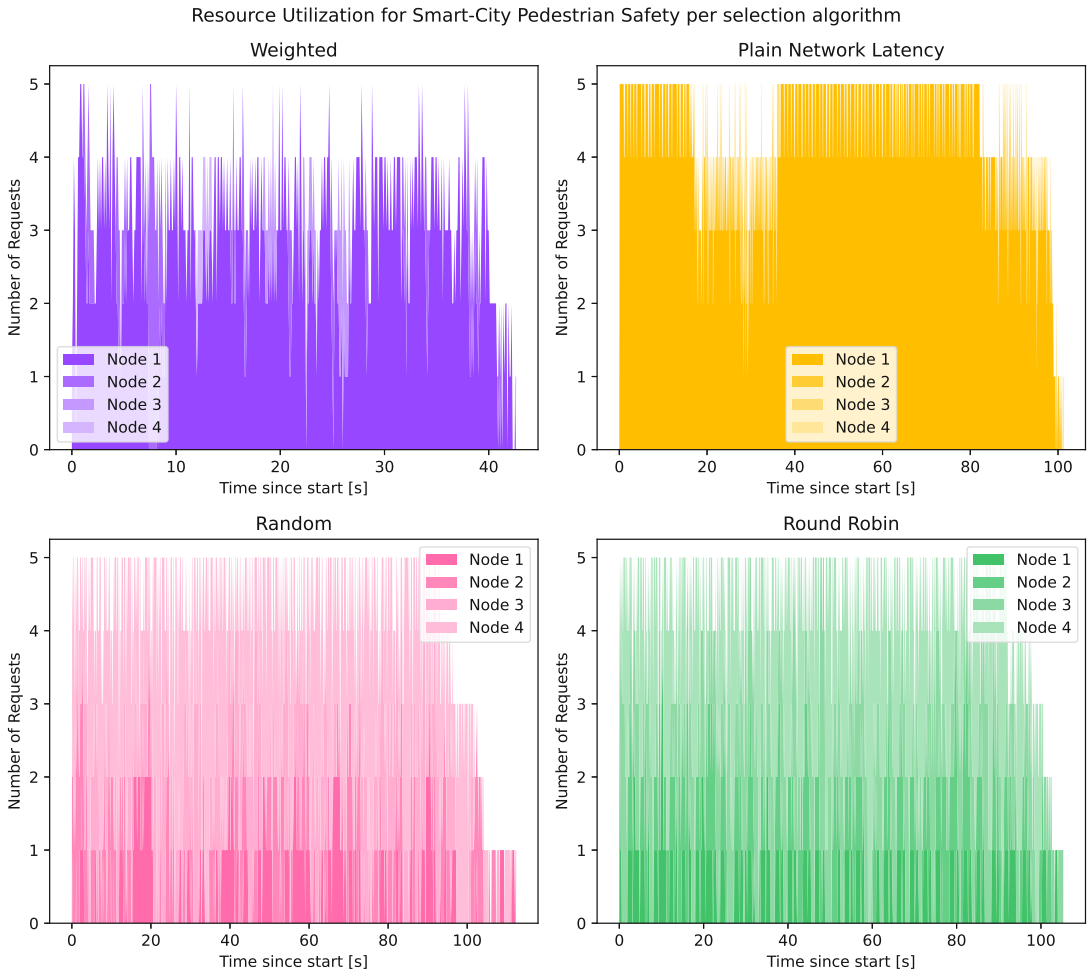

Figure 6.7: Resource Utilization in the SCP usecase

darkest shade). There is only a small amount of requests being handled by other nodes. These requests most likely arrived at the beginning of the experiment and took longer to process. At the beginning there is not sufficient latency data sent from the models to the metadata storage yet. This error tapers out over time, as the latency data gets updated. With the weighted model selector we can see that it also has a focus on the node with the lowest network latency, however, it also distributes the load over the next nodes more.

Answer to RQ1 "Which benefits can be achieved for latency-sensitive applications by using a weighted model selector compared to a round-robin, random or plain-net-latency model selector?"

The main benefit of using the weighted model selector is that putting the focus on latency leads to a noticeable improvement in latency compared to the other selectors. In general, the weighted model selector produces the lowest average latencies, the lowest median latencies and the lowest 95th percentile latency of all selectors under test. This holds true for a variety of schedules, but it is most noticeable for schedules with special considerations for improved latency (e.g, having the fastest models on the nearest node). Using only the host latency as a metric can lead to better average latencies. However, in schedules where low-latency hosts serve models with a high delay this can produce worse results than all other selectors. This can happen for schedules with a single node, where the plain network latency selector has a higher average latency than the random selector.
We also can note that the weighted model selector achieves these low latencies specifically by sacrificing accuracy, which is its implementation goal in the smart-city pedestrian safety use case.

RQ2 - Which benefits can be achieved for accuracy-sensitive applications by using a weighted model selector compared to a round-robin, random or plain-net-latency model selector?

This research question is very similar to the previous one. However, it focuses on the accuracy of the selectors instead of the latency. Again, we answer this research question by analyzing its corresponding use case from Chapter 5—Medical Diagnosis Assistance (MDA). To evaluate how well the weighted model selector performs in terms of accuracy compared to the other selectors, we look at the empirical cumulative distribution of result accuracies.

Figure 6.8 shows that the curve of the weighted model selector ECDF lies significantly below the curves of the other selectors in both displayed schedules. This means that a higher share of the requests receives results with a higher accuracy. Looking at the box plots shows as well that the weighted model selector has the highest median accuracy of all selectors in both schedules. Its median accuracy is above the third quartile of both the random and the round-robin selector in the **Logical** schedule. In the **Arbitrary** schedule the median accuracy of the weighted model selector is above the third quartile of all other selectors. This means that the weighted model selector is able to achieve a higher accuracy than the other selectors in general cases. The plain-network-latency selector achieves slightly better accuracy results than the random and round-robin selectors. This is especially prevalent in the **Arbitrary** schedule where the median accuracy of the plain-network-latency selector is around the third quartile of the random and round-robin

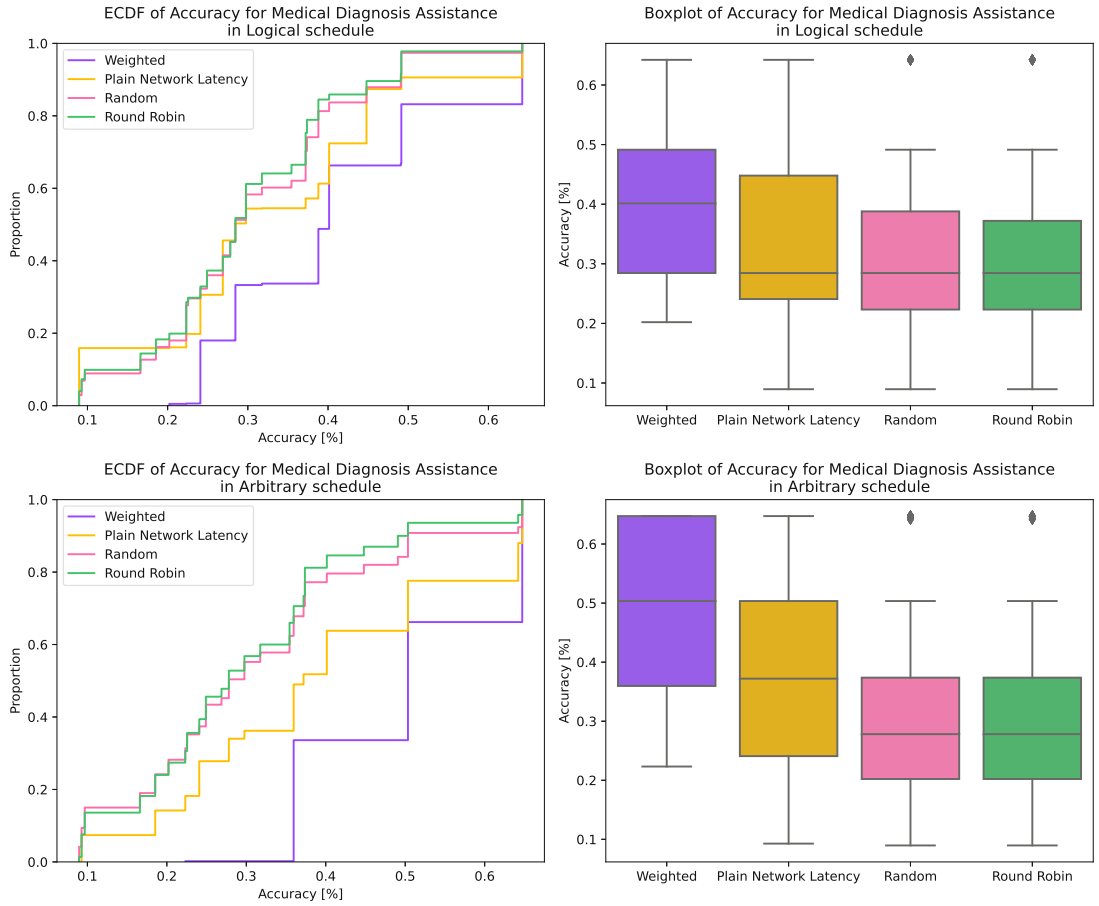

Figure 6.8: Cumulative distribution of accuracies for all clients for **Logical** and **Arbitrary** schedules in the MDA use case

selectors. The **Arbitrary** schedule has a higher amount of high-accuracy models on the nearest nodes which means that choosing only the nearest models leads to better accuracy results than choosing uniformly between all models on all nodes. As with the previous use case, the round-robin and random selectors perform very similarly in terms of accuracy.

To better understand the results we further investigate the time series data of the accuracies in Figure 6.9. This is again an experiment in the **Logical** schedule with all of the selectors. However, here we have a total of 500 requests sent to each client to compensate for the longer request duration when focusing on accuracy in the weighted selector. As with the previous use case, the durations of the experiments are different. However, here the weighted model selector took the longest to finish, while the other selectors took roughly the same amount of time. As we discussed in the previous use case, the reason for this is that results with higher accuracy take longer to compute. Furthermore, we can see that the weighted model selector achieves the highest accuracy

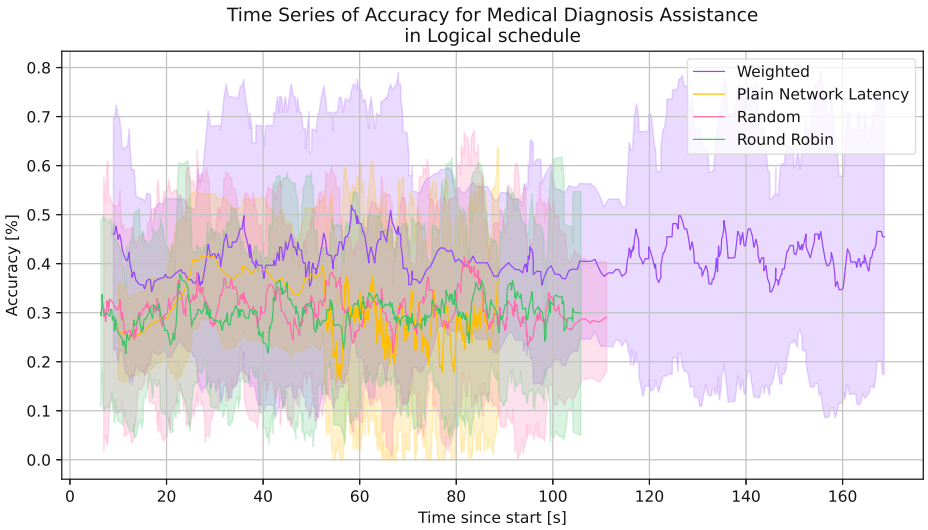

Figure 6.9: Rolling average time series of accuracies for all clients for the **Logical** schedule in the MDA use case with a 90% confidence interval

of all selectors. This is true for the average line in the time series as well as for the confidence intervals.

We need to note that the confidence intervals of the weighted model selector are wider than the confidence intervals of the other selectors. This might be because of how we designed the simulated models, since their accuracy is multiplied by a factor based on the input data. If the base accuracy is higher this factor has a bigger impact on the final accuracy, which leads to a higher variance in the results. Furthermore, only the plain network latency selector has short periods where it can match the accuracy of the weighted model selector in the average value. At some periods though, this selector also has a lower accuracy than the random and round-robin selectors. We explain this with the fact that the plain network latency selector only considers the network latency and has no notion of accuracy at all. This makes it sometimes choose high accuracy models on near nodes.

We can also cross-reference these accuracy results with the latency results for the same use case. For this purpose, we investigate the cumulative data of the latencies in Figure 6.11. Here, the weighted model selector has the highest median latency of all selectors in both schedules. The first and third quartile of the weighted model selector are also above the first and third quartile of the other selectors. Dissimilar to the previous use case, low latency is not the goal of the weighted model selector for medical diagnosis assistance. Instead, we want to have the best possible, not the fastest possible results. So, having a higher latency does not pose a problem here. Other than that, the results for latency are similar for the other selectors as in the previous use case. These selectors do not take into account the preferences for use cases which is why their results do not vary between

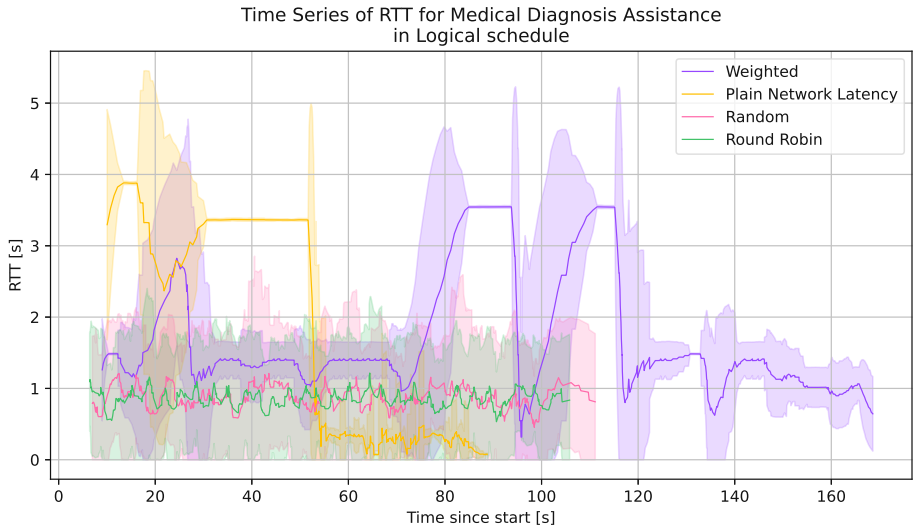

Figure 6.10: Rolling average time series of latencies for all clients for the **Logical** schedule in the MDA use case with a 90% confidence interval

them as with the weighted model selector. We can again compare the time series data of the accuracies in Figure 6.9 to the time series data of the latencies in Figure 6.10 in the same experiment. Additionally, we analyze the cumulative data for the accuracies in Figure 6.11.

The latency of the round-robin and random selectors remains fairly stable throughout the experiment. Compared to that we see high spikes in the latency for both the weighted model selector and the plain network latency selector. This is because we defined the simulation models in such a way that they increase their latency when they receive more concurrent requests. This does not affect the random and round-robin selectors as they distribute the load evenly over all available models. Only slight increases in the top percentiles are noticeable for these selectors for limited periods of time. On the other hand, both the weighted model selector and the plain network latency selector have the possibility of overloading a single model. For the plain network latency selector this happens because it reduces its selection space to only the models on the nearest node. Therefore, if the nearest node is overloaded, the plain network latency selector will also overload the models on that node. For the weighted model selector this happens because of our chosen selection weight. When we advise the selector to only focus on accuracy, then it will choose the most accurate model, even if it is overloaded. This inevitably leads to spikes in latency. However, as we already discussed, we do not consider latency in this use case, but accuracy. An easy fix for this problem would be to increase the selection weight for latency, which would lead the selector to choose a less accurate model, but one which is not overloaded.

Next, we again look at the scheduling edge cases—**Hyperlocal** and **Parity**. First, in

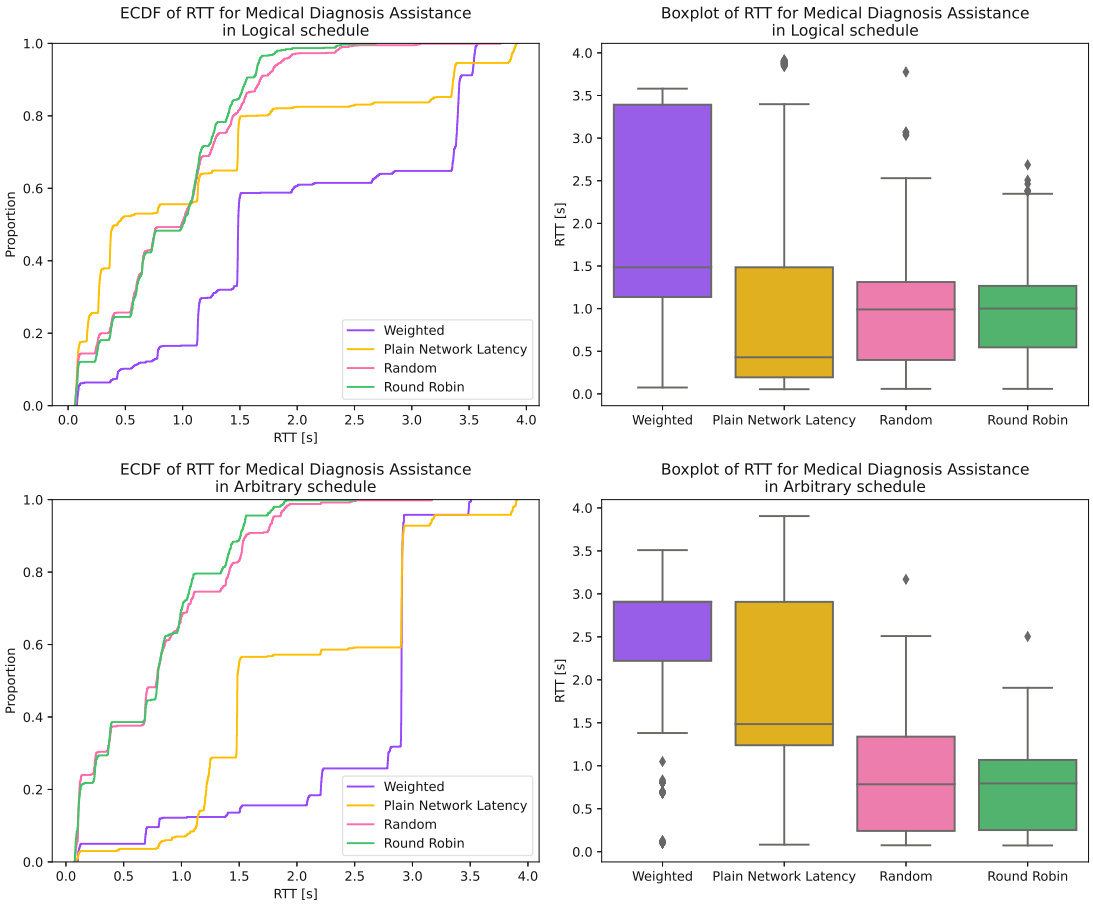

Figure 6.11: Cumulative distribution and box plots of latencies for all clients for **Logical** and **Arbitrary** schedules in the MDA use case

Figure 6.12, which shows the **Hyperlocal** schedule, we notice an even more pronounced difference between the weighted model selector and the other selectors than in the **Logical** schedule. The difference is mainly in the latency of the weighted model selector being even higher than in the **Logical** schedule. This is because instead of having the best possible model on all nodes, like in the **Logical** schedule, we only have the best possible model on one node. This leads to more overloading of the model, which leads to higher latencies. The changes in the other selectors are mainly to be attributed to the fact that all models are only available once instead of the uneven distribution of the **Logical** schedule. This means that the plain network latency selector has similar accuracy to the random and round-robin selectors, as it chooses virtually randomly between the same models. The latency behavior of the baseline selectors is also similar to the **Logical** schedule, as they do not take into account the preferences of the use case.

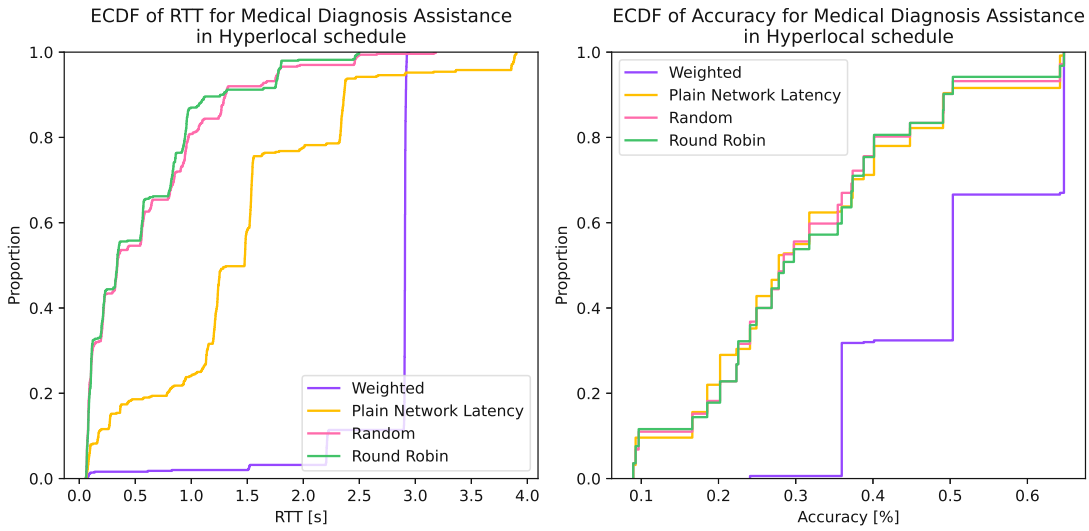

Figure 6.12: Cumulative distribution of latencies and accuracies for all clients for the **Hyperlocal** schedule in the MDA use case

The results for the **Parity** schedule in Figure 6.13 are similar to those of the **Arbitrary** schedule. Here, the weighted model selector outperforms all the other selectors in terms of accuracy, while having a higher latency.

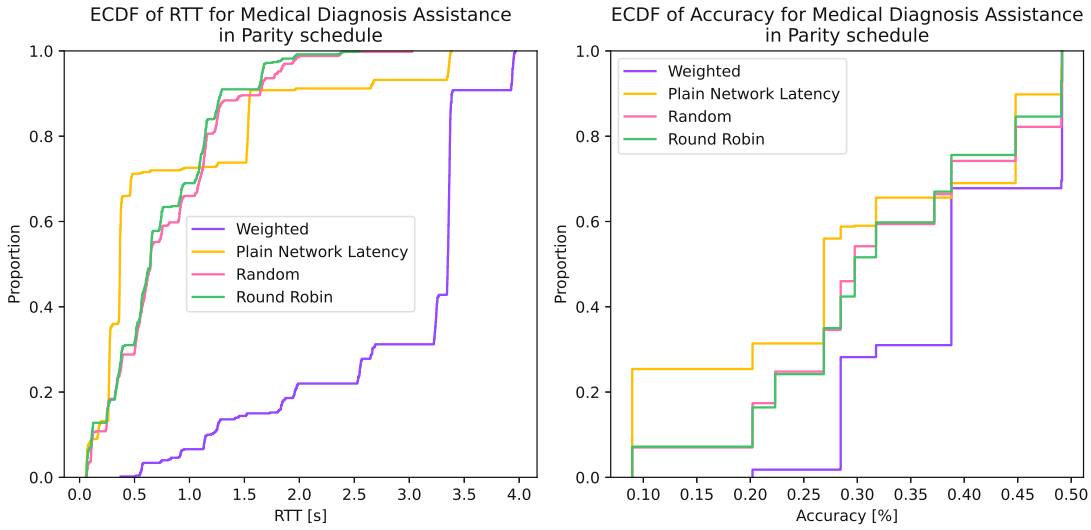

Figure 6.13: Cumulative distribution of latencies and accuracies for all clients for the **Parity** schedule in the MDA use case

Finally, we compare the utilization of nodes by the different selectors in the **Logical**

schedule in Figure 6.14. As a reminder, this graphic shows which nodes get utilized the most by the different selectors at given points in time in a given experiment. There again is a predictable pattern of the random and round-robin providers with the equal distribution of load over all nodes. Notable, again, is that the plain network latency selector has a strong focus on the node with the lowest network latency (the darkest shade). Since the configuration did not change for those selectors between the use cases those are the expected results. However, the weighted model selector shifts the load more evenly over the nodes than in the previous use case. As we do not consider latency in this use case, the weighted model selector does not have a strong focus on the node with the lowest network latency. For the **Logical** schedule we put the most accurate model on all the nodes to increase its availability. This means that in terms of highest possible accuracy, all the nodes are equal. The shifts here mainly happen because of slight changes in the rolling average accuracy of the models on the nodes which we calculate for the selection weight. As soon as the accuracy of the high-accuracy model on a node drops below the accuracy of the high-accuracy model on another node, the weighted model selector shifts the load to the other node.

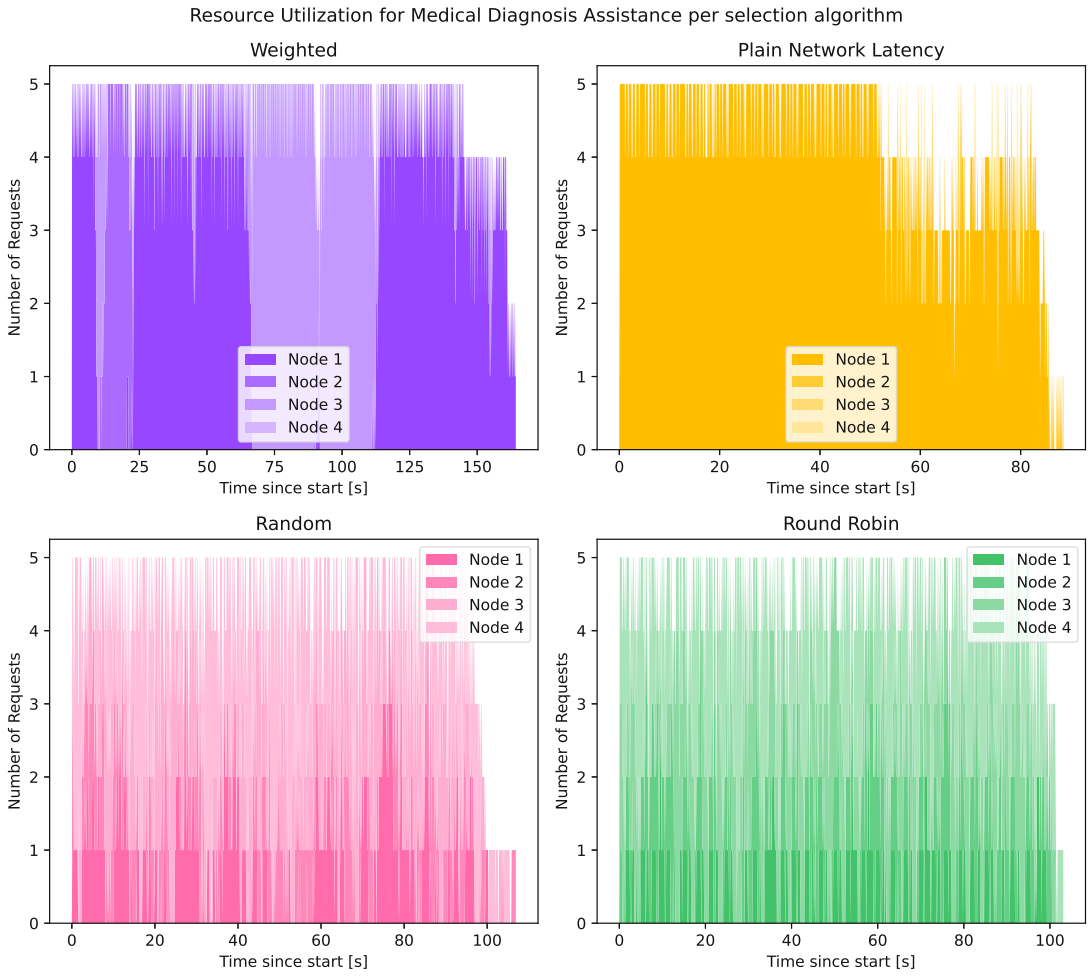

Figure 6.14: Resource Utilization in the MDA use case

Answer to RQ2 "Which benefits can be achieved for accuracy-sensitive applications by using a weighted model selector compared to a round-robin, random or plain-net-latency model selector?"

For accuracy-sensitive applications, the benefit of using the weighted model selector is that it noticeably improves the accuracy of results compared to other selectors. Generally, the weighted model selector produces the highest accuracies of all selectors under test. This holds true for the average, as well as the median accuracies and the lower 90% confidence bound of the accuracies. However, the weighted model selector can—similarly to the plain network latency selector—experience high spikes in latency. This happens because of overloading of a single model as well as selecting a high-accuracy model running on a further away node. This again shows the trade-off between accuracy and latency and how the weighted model selector can be used to tune this trade-off. The weighted model selector exactly achieves this maximized accuracy, by disregarding other factors, like latency or preventing overloads.

RQ3 - What differences in behavior can we expect for applications with varied latency and accuracy preferences between the different use cases in the weighted selection compared to the baselines?

In the previous two research questions we looked at how good the weighted model selector can react to requirements strictly focusing on either latency or accuracy. However, in real-world applications we will most likely have a mix of both requirements. To evaluate the differences between the weighted model selector and the baselines in this case, we have two separate use cases. The first use case is the Public Sentiment Analysis (PSA) use case, which requires equal focus on both latency and accuracy. The second use case is the Environmental Monitoring (ENV) use case, which has periodical shifts in the requirements between latency and accuracy.

First, we analyze the results of the PSA use case, beginning with the effects on latency. Like with the previous use cases, we look at the cumulative distribution of latencies in Figure 6.15.

For the **Logical** schedule, the weighted model selector has a similar median latency to the random and round-robin selectors. However, it has—similarly to the plain network latency selector—a broader range of latencies. In the **Arbitrary** use case the latency performance is even worse, as the median latency is significantly above the upper bounds of the other selectors. The ECDF also displays differences in the curves of the selectors. While the curves of the baseline selectors all have a clear concave shape to them, the curve of the weighted model selector is only slightly convex in the **Arbitrary** schedule and S-shaped (first slightly concave, and then convex) in the **Logical** schedule. This hints at the fact that the weighted model selector has a more uniformly distributed latency

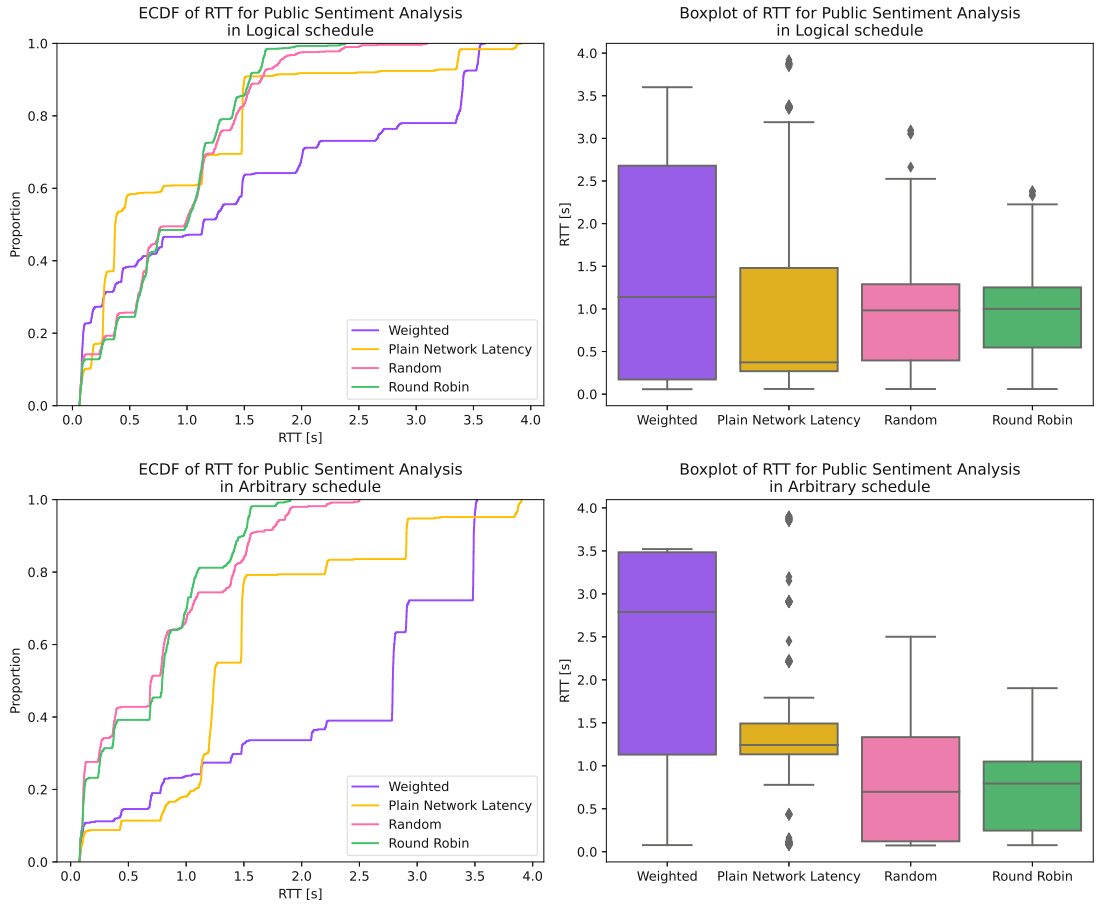

Figure 6.15: Cumulative distribution and box plots of latencies for all clients for **Logical** and **Arbitrary** schedules in the PSA use case

than the other selectors. This would correspond to the idea that the weighted model selector does not focus on a specific latency range, but instead tries to perfectly balance it with the accuracy. To this end, Figure 6.16 further shows the cumulative distribution of accuracies.

The weighted model selector has the highest median accuracy of all selectors in both schedules. Regardless of that, for the **Logical** schedule we can also see that the weighted model selector has the broadest range of accuracies as well as the most uniform distribution of accuracies. We can say this because the distance between each of the quartiles for our selector is the most equal among all selectors. This does not hold true for the **Arbitrary** schedule, where the plain network latency selector has the best distribution and the weighted selector is more top heavy. We attribute this to the fact that the average possible accuracy across all models is higher in the **Arbitrary** schedule than in the **Logical** schedule, where we hand-crafted a more even distribution of accuracies. This

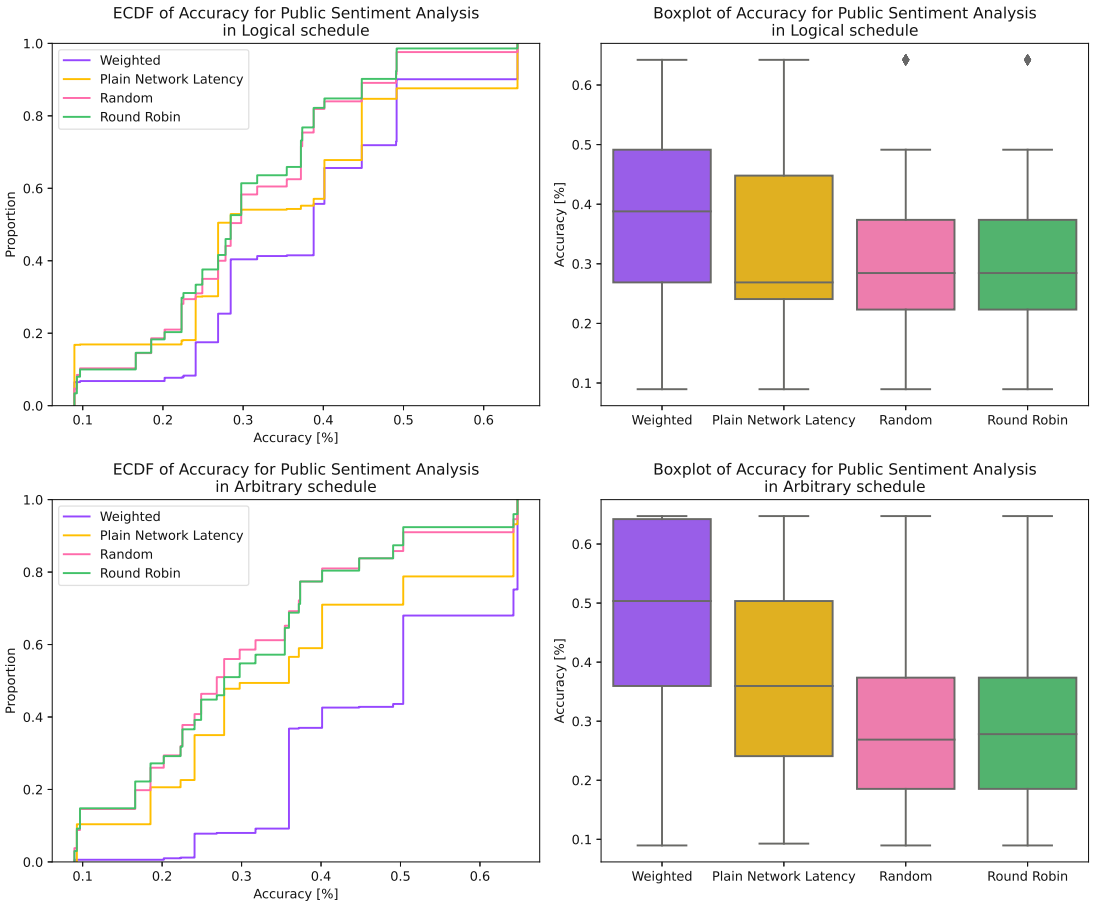

Figure 6.16: Cumulative distribution and box plots of accuracies for all clients for **Logical** and **Arbitrary** schedules in the PSA use case

means that setting the selection weight to 50% focus on accuracy will lead to a higher average accuracy in the **Arbitrary** schedule than in the **Logical** schedule. Nevertheless, the maximum possible accuracy is still the same for both schedules, as they both contain the same models. This makes the weighted model selector in the **Arbitrary** schedule more biased towards the top values.

Combining the findings from the latency and accuracy results we can say that the weighted model selector is able to achieve a more uniform distribution of latencies and accuracies than the other selectors. We can also say that the schedules have an inherent focus on either latency or accuracy, which means that round-robin and random selection cannot deliver a perfect balance between the two. Instead those selectors will always have a bias towards either latency or accuracy, depending on the schedule. Furthermore, increasing the variety of models in the schedules can also help to achieve a more uniform distribution of latencies and accuracies for the weighted model selector.

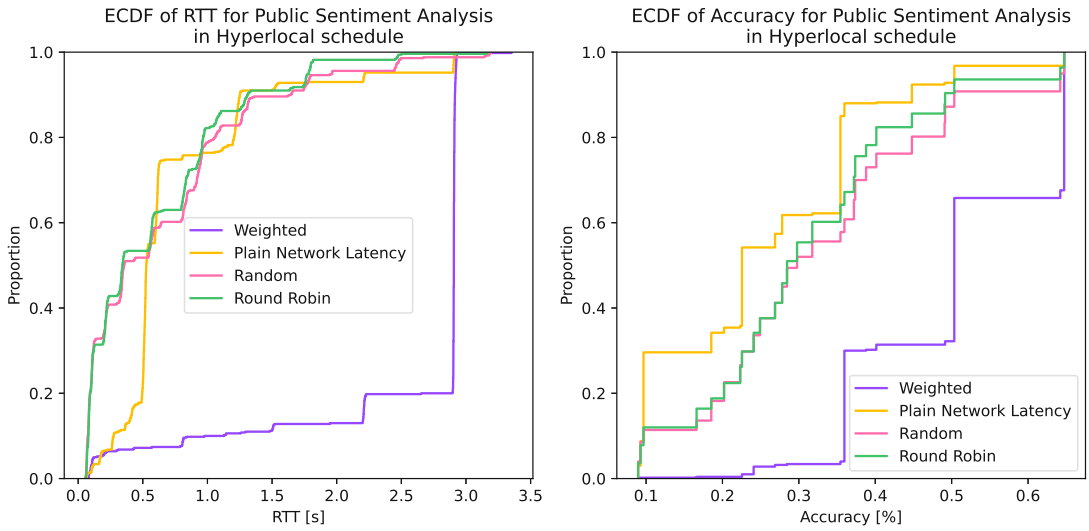

Figure 6.17: Cumulative distribution of latencies and accuracies for all clients for the **Hyperlocal** schedule in the PSA use case

To further understand the impact of the schedules on the uniformity of the results we investigate the edge cases. Figure 6.17 shows the limits of the weighted approach. Having all models on the same node means that the latency impact of choosing the highest accuracy model is higher. This is exacerbated by the fact that the highest accuracy model skews the distribution of accuracies towards the top which makes it more likely to be chosen. We therefore notice the importance of having a variety of models in the schedules. Another fix for that top-heaviness would be to adapt the latency buckets to each schedule.

For the **Parity** schedule in Figure 6.18 the results are the best of all schedules when considering uniformity. With the equal distribution of models over the nodes, the inherent distribution of latency and accuracy in the schedule is also more uniform. This is noticeable in the increased linearity of the curves in the ECDFs for both latency and accuracy. A slight bias towards top latencies is still noticeable, which is because the highest accuracy model which also has the highest model delay is present on all nodes. Therefore, the distribution of model delay per node is less uniform in the top end.

Shifting the focus between latency and accuracy

The power of the weighted model selector not only lies in its ability to balance latency and accuracy, but also in its ability to adapt to changing requirements. Therefore, we look at the results of the Environmental Monitoring use case, which has a periodical shift in requirements between latency and accuracy. First, we present the shift in weights we use for the use case in Figure 6.19 more visually. We can mentally overlay this shape onto the time series data of the latency in Figure 6.20 and the accuracy in Figure 6.21.

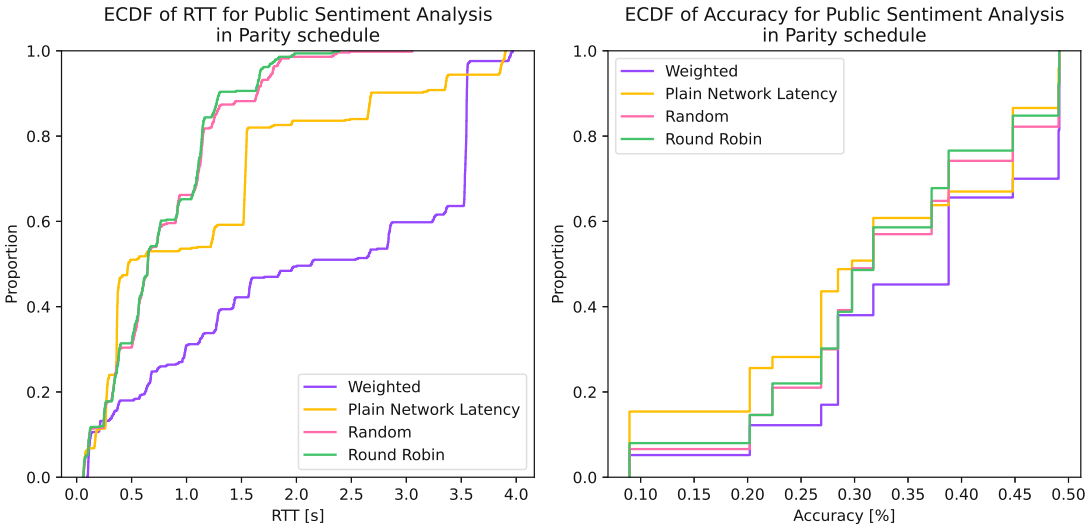

Figure 6.18: Cumulative distribution of latencies and accuracies for all clients for the **Parity** schedule in the PSA use case

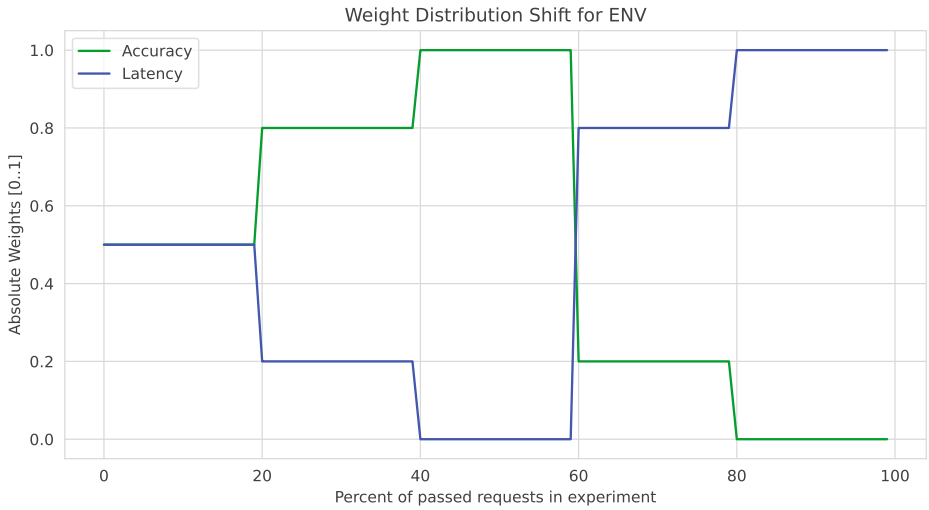

Figure 6.19: Shifts of the absolute values of the selection weights in the ENV use case

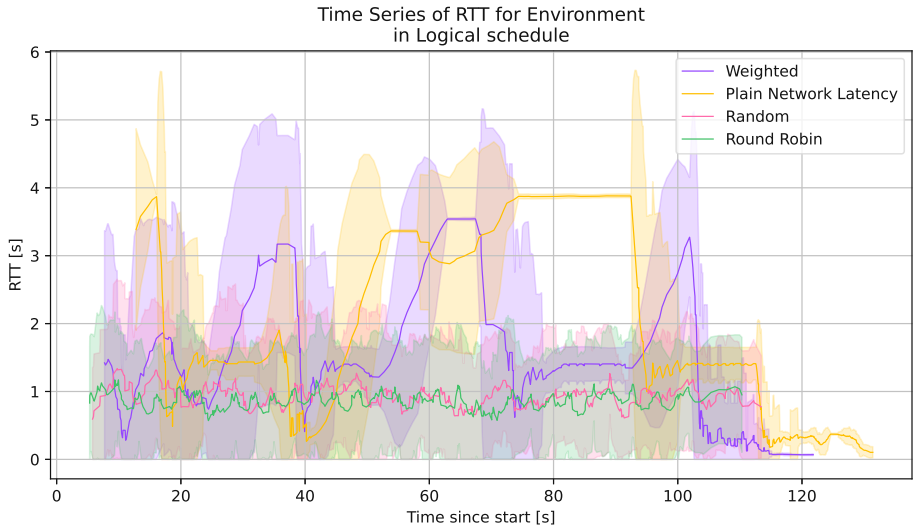

Figure 6.20: Rolling average time series of latencies for all clients for **Logical** schedule in the ENV use case with a 90% confidence interval

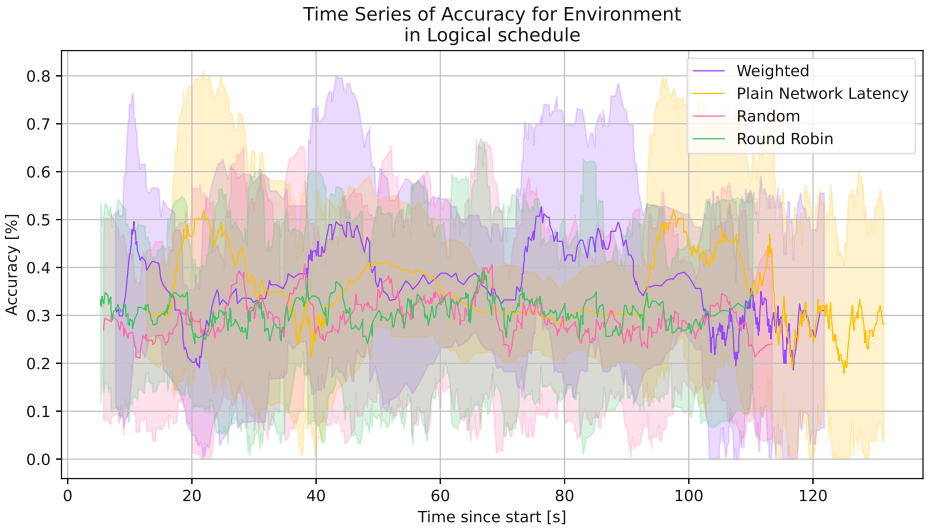

Figure 6.21: Rolling average time series of accuracies for all clients for **Logical** schedule in the ENV use case with a 90% confidence interval

The shifts in weights correspond to events in both time series. For the latency, each decrease in the absolute weight leads to an increase in latency. However, the latency does not hold steady between the shifts. In fact, the latency even drops to the same level as before the shift. We observe this behavior because of insufficient latency data at the beginning of the shift. Before having any knowledge of the Model Data Delays (MDDs) of the models, the weighted model selector will choose the models with the lowest network latency. After calling these models, the selector gains knowledge of the MDDs and after some time chooses a model with enough—but lower—accuracy but also a lower MDD.

For the second shift, however, the highest accuracies and the highest latencies of the experiment occur. Since the weights are set to 100% accuracy, the selector will choose the most accurate model, regardless of its MDD. At the next shift to 20% accuracy and 80% latency, the selector opts for models with similar performance characteristics as in the shift to 80% accuracy and 20% latency. This indicates that in the less extreme shifts, the results tend to be very similar. An ever bigger variety of models might lead to better distinctions between those in-between states. The infrastructure available for our experiments however was limited in terms of the number of nodes and the number of models it can handle. Therefore, exploring the effects of an even higher variety of models is left for future work.

Right around the last shift to 100% latency focus, we can see that the latency peaks again. We can explain this again by models which have yet been called by the selector. After this peak, however, the weighted model selector gains knowledge of their MDDs and refrains from using those models further. The result is a significant drop in latency, which is also the lowest latency of the experiment.

So, the weighted model selector is able to adapt to the changing requirements of the use case. It has difficulties in the beginning of the shifts, but after some time it is able to adapt to the new requirements once it gains enough knowledge of the MDDs and receives updated accuracy information. Over a long period of time, which is the time frame for use cases like Environmental Monitoring, the weighted model selector will have enough data to make informed decisions.

To further understand the effect of weight shifts, we investigate the resource utilization of the different selectors in the **Logical** schedule in Figure 6.22.

As with the previous use cases, this graphic shows which nodes get utilized the most by the different selectors at given points in time in a given experiment. The baseline selectors continue to behave in the same way as in the previous use cases. For the weighted model selector we can see how it reacts to shifts in focus. Right at the start, with an even focus, like for the Public Sentiment Analysis use case, the weighted model selector uses a variety of nodes. As soon as the focus starts shifting towards accuracy, the weighted model selector first starts to select models on node 3. After some time, however, it shifts to node 1, which contains the most accurate model as well. Shifting the focus further towards accuracy however pushes the selection again to the further away node. The model there probably had a better rolling average accuracy at this time. We also note

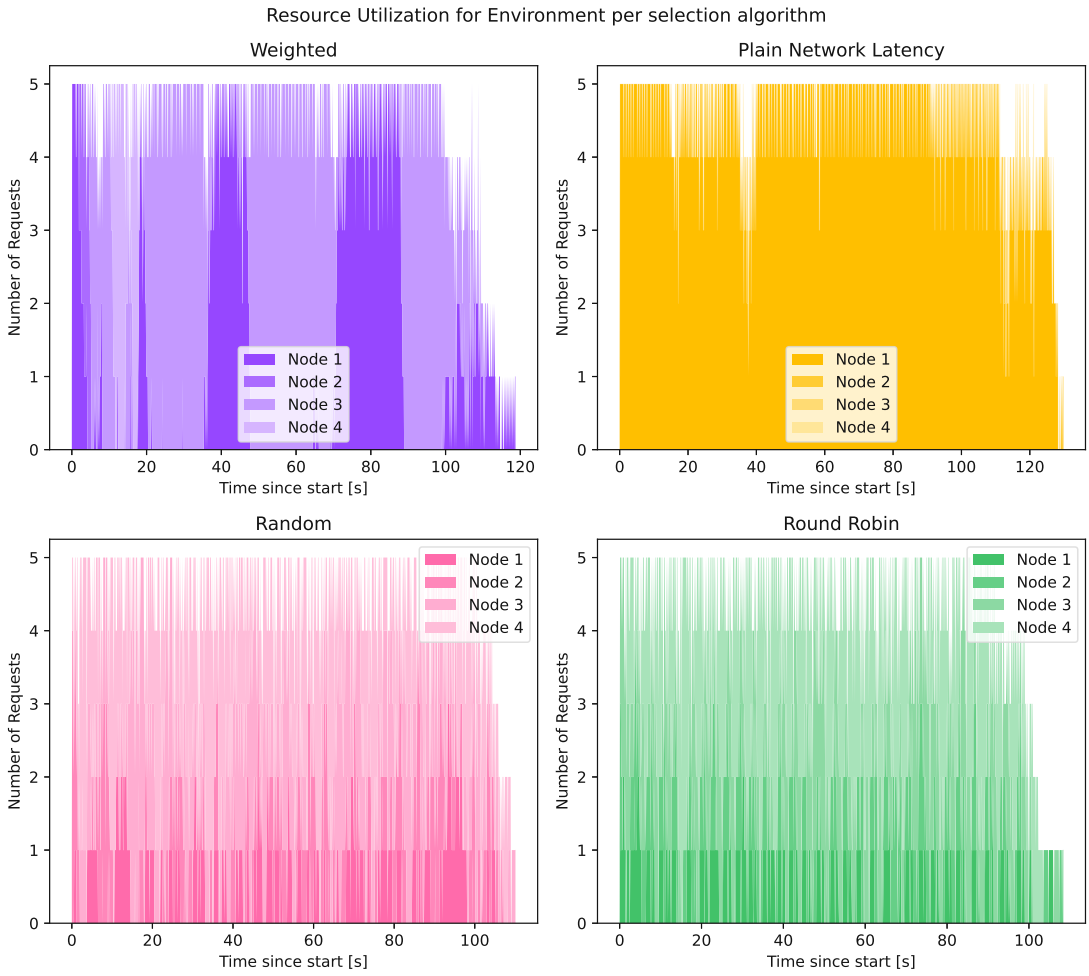

Figure 6.22: Resource Utilization in the ENV use case

that similarly to the Medical Diagnosis Assistance use case, the weighted model selector can cause overloads of a single model when not taking into account latency aspects at all. As soon as the there is some focus on latency again, the weighted model selector shifts back to the nearest node. Then, as we pointed out before, the weighted model selector has difficulties in the beginning of the shift, because it chooses the models with the lowest network latency. After gaining knowledge of all the MDDs, the weighted model selector chooses the fastest models on the nearest node and quickly finishes the remaining 20% of the requests.

Answer to RQ3 "What differences in behavior can we expect for applications with varied latency and accuracy preferences between the different use cases in the weighted selection compared to the baselines?"

The weighted model selector offers multiple benefits for applications with varied latency and accuracy preferences. For applications which try to balance latency and accuracy the weighted model selector is able to achieve a more uniform distribution of latencies and accuracies than the other selectors. We also observe that the schedules themselves have an inherent bias towards either latency or accuracy. This means that approaches which balance the load evenly across all models, like random and round-robin selection, do not also balance the latency and accuracy evenly. Instead, those selectors will always portray the bias inherent to a given schedule. Furthermore, increasing the variety of models in the schedules should also help to achieve a more uniform distribution of latencies and accuracies for the weighted model selector.

For applications which change their requirements over time, the weighted model selector is able to adapt to the changing requirements. This is not possible with the other selectors, as they do not take into account the Quality-of-Service requirements of the use case. We note that it has difficulties in the beginning of the shifts, but after some time it is able to adapt to the new requirements once it gains enough knowledge of underlying infrastructure. It can cause further problems with overloading a single model when not taking into account latency aspects at all—as observed in the previous research question. Over a longer period of time the weighted model selector has enough data to make informed decisions.

6.2 Benchmark Results

To further understand the performance of the different selection algorithms, we discuss the results of the benchmarking experiments. Here, we scale the size of the messages and the number of concurrent requests independently of each other (see Section 5.3). Both these aspects have an impact on the performance of the models and therefore the clients using the different selectors.

RQ4 - How does the selection scale with increasing message sizes for the different selection algorithms?

In order to understand how the selection scales with increasing message sizes, we investigate the results of the message size benchmarks. The message sizes are abstract values representing the size of the input data for the models. The simulated models use these abstract values to calculate the time it takes to process the input data. The sizes increase in a fibonacci-like sequence, which means that the difference between each size increases

exponentially. For better comparability with the baselines we use the Public Sentiment Analysis (PSA) use case on the **Logical** schedule for this research question. The weighted selector has the most similar characteristics to the baselines in this setup. We remind the reader that the PSA use case does not focus on one of the two performance goals, but instead tries to balance them. We also note that for the experiments for increasing size, we do not increase the number of concurrent requests.

First, Figure 6.23 displays the results for the latency. The figure shows the average latency for each message size for all selectors. It also shows the 90% confidence interval for each of those averages.

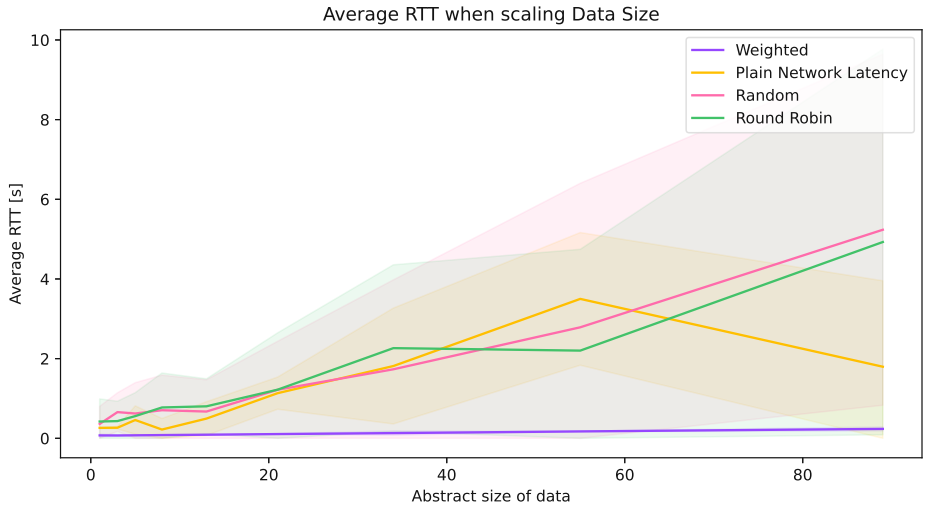

Figure 6.23: Latency for different message sizes for the PSA use case on the **Logical** schedule

For the baselines, increasing the message size leads to a linear increase in latency. For the round-robin and random selectors the average latency growth is almost identical, with the averages matching each other and both having quite wide confidence intervals. The plain network latency selector has a similar growth in latency, but with a narrower confidence interval. This is because the plain network latency selector chooses models from the nearest node, which all have a closer average latency than the entire set of models. With the weighted model selector we can see that the latency growth is almost non-existent. The average latency stays almost the same for all message sizes. The weighted model selector considers message size in its calculations, which means that it chooses models which are faster for larger messages. It stays with the same models for all message sizes, which leads to the almost constant latency.

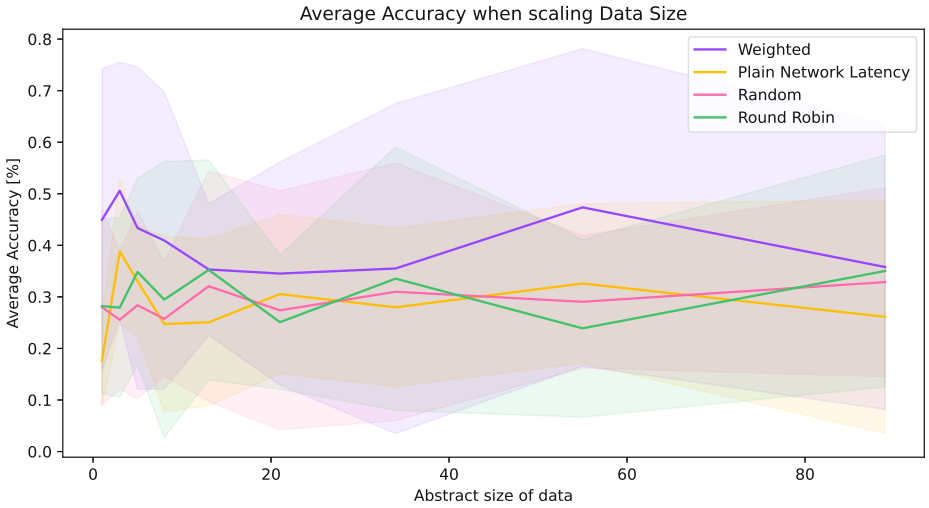

Figure 6.24: Accuracy for different message sizes for the PSA use case on the **Logical** schedule

In Figure 6.24 the accuracy behavior stays the same for all selectors. However, the weighted model selector has a higher average accuracy than the other selectors. We already saw this behavior in the previous research question, where we looked at the results for the PSA use case. This means, increasing the message size does not have an impact on the accuracy of the weighted model selector or any other selector whatsoever. The baseline selectors continue to portray the bias of the schedule, and the weighted model selector continues to balance the latency and accuracy as much as possible.

Answer to RQ4 "How does the selection scale with increasing message sizes for the different selection algorithms?"

The weighted model selector scales better with increasing message sizes than the other selectors. While the latency of the other selectors increases linearly with the message size, the latency of the weighted model selector almost does not increase at all. This is because the weighted model selection takes message size into consideration in its calculations.
Regarding the accuracy, we see that increasing the message size does not have an impact on the accuracy of the weighted model selector or any other selector. Message size is not a factor in the accuracy of the models, which means that the weighted model selector continues to balance the latency and accuracy as much as possible. Also, the baseline models continue to portray the bias of given schedules as we have already seen for **RQ3**.

RQ5 - How does the selection scale with an increasing number of concurrent requests served by the platform for the different selection algorithms?

Similarly to the previous research question, we look at the results of our benchmarks to understand how the selection scales with an increasing number of concurrent requests. The simulated models use the number of concurrently handled requests in the calculations for the time it takes to handle input data. Like the sizes, the number of concurrent requests also increases in a fibonacci-like sequence. We again use the PSA use case on the **Logical** schedule for this research question.

Figure 6.25 shows the average latency for each number of concurrent requests for all selectors along with the corresponding 90% confidence intervals.

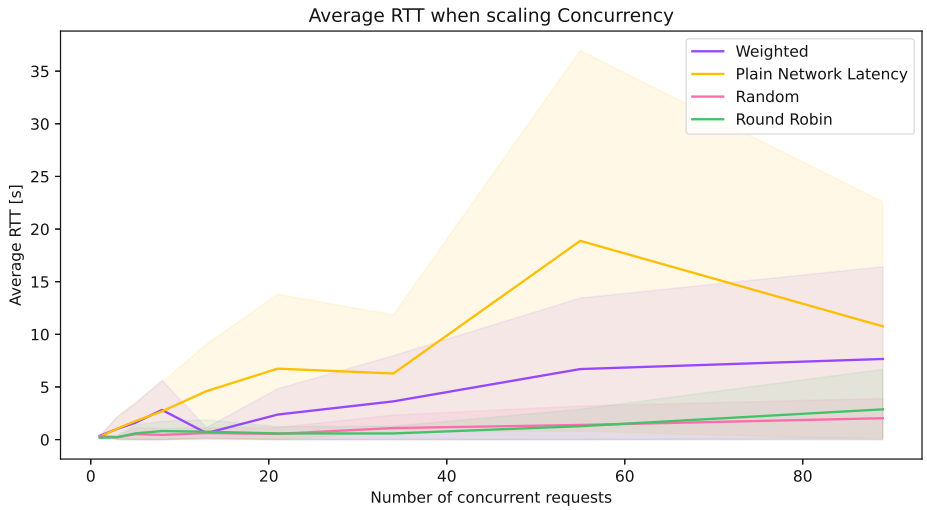

Figure 6.25: Latency for an increasing amount of concurrent messages for the PSA use case on the **Logical** schedule

Here, we observe an increase in latency for all selectors. However, the lowest increase in latency can be observed for the round-robin and random selectors. As they balance the load evenly over all models, they are able to handle the increasing load better than the other selectors. Therefore overloading a single model is the least likely to happen. The plain network latency selector and the weighted model selector on the other hand degrade faster in latency performance. They both limit their selection space, with the plain network latency selector only choosing models on the nearest node and the weighted model selector choosing models with average latencies and average accuracies. This means that the plain network latency selector will overload the models on the nearest node, while the weighted model selector overloads average models until they become too slow.

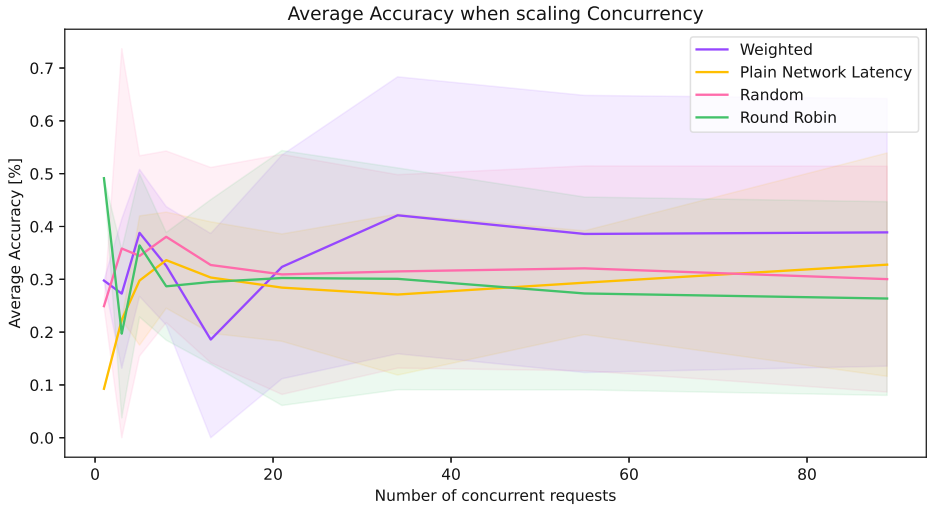

Figure 6.26: Accuracy for an increasing amount of concurrent messages for the PSA use case on the **Logical** schedule

In Figure 6.26 the accuracy of all the selectors remains stable. However, there might be a slight downward trend towards the upper end of the spectrum for the weighted model selector. As it overloads average models, it might have to choose models with lower accuracies than before. Other than that, the baseline selectors again portray the bias of the schedule and the weighted model selector again balances the latency and accuracy as much as possible.

Answer to RQ5 "How does the selection scale with an increasing number of concurrent requests served by the platform for the different selection algorithms?"

While all selectors increase in latency with an increasing number of concurrent requests, the weighted model selector degrades in latency performance faster and scales worse than round-robin and random selection. This is because of the limited selection space, which the weighted model selector and the plain network latency selector both have. This leads to overloads of models which in turn increases the average latencies.
For the accuracy, the weighted model selector continues to balance the latency and accuracy as much as possible. However, overloading of average models might shift the selection window towards lower accuracies in the long run which can decrease the accuracy of the weighted selection results towards the baselines.

6.3 Summary

The overall goal of our approach is to provide a platform which can be used to deploy and run machine learning models in a Serverless environment. This platform should be able to handle the requirements of real-world applications. The research questions we presented in this chapter are designed to evaluate the performance of our approach in terms of latency, accuracy and scalability. In the following we list the main findings of the research questions.

- **Increased Latency Performance:** As we have seen in the experiments we conducted for answering **RQ1** and **RQ3**, setting a focus on latency leads to a noticeable improvement in latency compared to the other selectors. Regardless of the schedule, the weighted model selector improves latency performance on average and also inside the inter-quartile range.
- **Increased Accuracy Performance:** The experiments we conducted for answering **RQ2** and **RQ3** show that focusing on accuracy performance leads to a noticeable improvement thereof. This again, holds true for all schedules and against all other selectors.
- **Balancing Improvements:** Our experiments for answering **RQ3** show that the weighted model selector is able to balance the improvements in latency and accuracy. By setting the selection weight to 50% for both latency and accuracy, the weighted model selector is able to achieve a more uniform distribution of latencies and accuracies than the other selectors. The other selectors either only focus on latency or just relay the inherent bias of the schedule.
- **Change Reactivity:** The experiments in **RQ3** also show that the weighted model selector is able to adapt to changing requirements. When instructing the selector to shift its focus between latency and accuracy, it is able to adapt to the new requirements. Depending on which focus is set, the weighted model selector will either deliver more accurate results or faster results than the other selectors.
- **Data Size Scalability:** The experiments we conducted for answering **RQ4** show that the weighted model selector handles increasing message sizes better than the other selectors. Because the weighted model selector takes the message size into account in the selection decision, it chooses models which are faster for larger messages. This selection does not change for different message sizes, which leads to the almost constant latency.

CHAPTER 7

Conclusions

Serverless computing is a growing research field that has been gaining a lot of attention in the last few years. Especially its combination with edge computing and Machine Learning has been the focus of many research papers, as we have seen in Chapters 2 and 3. Such a combination is very promising and can open up new applications and use cases. However, there are still challenges which need to be addressed regarding the utilization of serverless computing for data-intensive workloads, for Machine Learning Operations tasks and for the edge computing domain.

In our work, we have focused on the combination of serverless edge computing with MLOps and have proposed a novel architecture that combines them. In this architecture we implemented an efficient and transparent model selection system suitable for serverless machine learning with a consideration of heterogeneous nodes. We evaluated our system with use cases lent from real world scenarios and have answered our research questions.

In our evaluation we were able to show that utilizing a weighted model selector can lead to latency and accuracy improvements in the platform, depending on the set focus. The baseline approaches could not achieve those improvements as they are not able to take the different selection criteria of the clients into account. Balancing accuracy and latency is also improved by the weighted model selector, while baseline approaches only show the bias inherent to the schedule of models on the nodes. Shifts in focus are also handled better by the weighted model selector, as it is able to react to those shifts and adapt to them. Again, without a notion of the different selection criteria of the clients, the baseline approaches are not able to react to those shifts. Regarding scalability we were able to show increased consideration of data-intensive workloads by the improved handling of data size through incorporation in the selection process.

7.1 Limitations

Regardless of the positive results we have achieved, there are still some limitations to our approach. Those limitations and how they impact the performance of the weighted model selector are as follows.

- **Schedule Dependence:** As we have seen in all experiments the schedules impact the performance of all selectors. The weighted model selector is no exception to this. If there is not enough variety in the models on the nodes, the weighted model selector will, e.g., not be able to achieve a uniform distribution of latencies and accuracies when instructed to do so. Also, when the platform does not offer enough alternatives which fit the hard and soft criteria of running serverless applications, the weighted model selector will have difficulties in keeping up with the requirements over time. This leads to the next limitation.
- **Tendency to Over-Utilize Single Models:** When there is not enough variety in the models on the nodes, the weighted model selector will tend to over-utilize a single model. This mainly happens when the selection weight is set to 100% accuracy. When the weighted selector does not take into account the latency at all, then it will not be able to react to a model degrading in latency performance. The selector will continue to use that single model, which leads to high spikes in latency.
- **Concurrency Scalability:** This over-utilization of single models turns into a bigger problem when the number of concurrent requests increases. When the load on the platform increases, the weighted model selector will tend to overload a single model. This will in turn lead to higher latencies for the requests. It can also impact the accuracies of the models down the line, as the weighted model selector will choose models with lower accuracies to prevent overloading if instructed to do so.
- **Impact of Missing Data:** When not having enough dynamic data about the models, the weighted model selector will choose models based on static data. For the accuracy this means it will choose the maximum base accuracy of the models. When latency data is missing it will choose the minimum network latency of the models. Those data points are not necessarily representative of the actual dynamic performance of the models. This can lead to sub-optimal results when first using those models in the platform. First usages of specific models in the platform happen at the beginning of using the platform and also after shifts in focus. This leads to spikes in latency or troughs in accuracy at those points in time. Even though this balances out over time, it still is a limitation of our current implementation.

7.2 Future Work

Finally, we discuss possible future work to improve the weighted model selector and the platform and to overcome the limitations we discuss in Section 7.1.

Incorporation of a Model-Aware Scheduler

In our approach we only evaluated four pre-defined schedules. We chose two average cases, one of which we considered to be a logical default schedule, and two edge cases. In our work we were able to show that schedules have a big impact on all selection approaches, as they define the models which are available for selection.

The insights we have gained from our evaluation of the different schedules can be used to create a model-aware scheduler. Such a scheduler would be able to create schedules that emphasize models with high demand from the clients. Through analysis of the weights sent to the weighted model selector and the resulting selection decisions, the scheduler would be able to scale up models which get selected often and scale down models which get selected rarely. This would help mitigate the problem of over-utilization of single models, as the scheduler would be able to react to the increased load on the platform.

Real-World Evaluation

Our evaluation of the weighted model selector was done in a simulated environment. This offered us a lot of control over the experiments and the ability to repeat them. For example, we were able to tightly control the network latency between nodes and the performance traits of the models.

However, this also means that the results we have achieved are not necessarily representative of a real-world scenario. To achieve this, we would need to deploy our system in a real-world environment and evaluate it there. This means, we need to use real edge nodes and use real world machine learning models. The machine learning models would need to trained with enough variety so that the weighted model selector can choose between them. The edge nodes have to be connected to a real distributed network, which would introduce network latency and bandwidth constraints. Furthermore, the edge nodes should have different hardware configurations, so that the weighted model selector can choose between them.

Utilization of FaaS clients

While our implementation of the weighted model selector and the backend models fits the Backend-as-a-Service aspect of serverless computing, the clients in our evaluation are not deployed as Function-as-a-Service functions. As the focus of our evaluation was on the efficiency of the weighted model selector, we did not want to introduce additional overhead by using FaaS clients.

However, to have a holistic view of the model selection process inside a serverless edge computing platform, we would need to use FaaS clients. This way, we would be able to evaluate the implications of weighting decisions on short-lived functions.

List of Figures

List of Tables

List of Listings

Acronyms

AI Artificial Intelligence. 2, 3, 12

API Application Programming Interface. 12, 33, 34

AWS Amazon Web Services. 2, 7, 8, 11, 14, 15, 17, 18, 20, 22

BaaS Backend-as-a-Service. 1, 8, 13, 20–24, 31, 83

DG design goal. 28, 29, 33

DNS Domain Name System. 13

EC2 Elastic Compute Cloud. 22

ECDF Empirical Cumulative Distribution Function. 51–53, 58, 66, 69

EI Edge Intelligence. 2, 3, 12, 13, 21, 46

ENV Environmental Monitoring. 29–31, 49, 50, 66, 69–73, 86

FaaS Function-as-a-Service. 1, 8, 13, 14, 20–24, 31, 83, 84

gRPC gRPC Remote Procedure Calls. 34, 91

IaaS Infrastructure-as-a-Service. 1, 7, 22

IoT Internet of Things. 18, 23

K8S Kubernetes. 13, 14, 18, 20, 23

MDA Medical Diagnosis Assistance. 29, 30, 49, 50, 58–63, 65, 73, 85

MDD Model Data Delay. 34, 35, 37, 38, 40, 72, 73

ML Machine Learning. xi, xiii, 2, 4, 5, 8, 14, 15, 17, 19–25, 30, 31, 81, 87

MLOps Machine Learning Operations. xv, 2, 5, 7, 14, 15, 17, 20, 21, 23–25, 81, 85

NG non goal. 28

PaaS Platform-as-a-Service. 7

PSA Public Sentiment Analysis. 29–31, 49, 50, 66–70, 72, 75–78, 85, 86

QoS Quality-of-Service. xv, 2–5, 17, 19, 24, 25, 74

REST Representational State Transfer. 34

RTT Round-Trip Time. 34

S3 Simple Storage Service. 8, 33

SCP Smart-City Pedestrian Safety. 29, 30, 49–57, 85

SLA service level agreement. 12, 23

SLO service level objective. 12, 20, 22

UI User Interface. 12

VM Virtual Machine. 10

Bibliography

[Ama23a] Amazon Web Services, Inc. How it works: Amazon SageMaker MLOps. https://aws.amazon.com/sagemaker/mlops/?sagemaker-data-wrangler-whats-new.sort-by=item.additionalFields.postDateTime&sagemaker-data-wrangler-whats-new.sort-order=desc, 2023.

[Ama23b] Amazon Web Services, Inc. Serverless Inference. https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html, 2023.

[Ama23c] Amazon Web Services, Inc. Tutorial: Using an Amazon S3 trigger to create thumbnail images. https://docs.aws.amazon.com/lambda/latest/dg/with-s3-tutorial.html, 2023.

[APRS21] Kalev Alpernas, Aurojit Panda, Leonid Ryzhyk, and Mooly Sagiv. Cloud-scale runtime verification of serverless applications. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, pages 92–107. ACM, 2021.

[ARB21] Siddharth Agarwal, Maria Alejandra Rodriguez, and Rajkumar Buyya. A reinforcement learning approach to reduce serverless function cold start frequency. In *21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021, Melbourne, Australia, May 10-13, 2021*, pages 797–803. IEEE, 2021.

[BMZ+22] Wenhao Bi, Junwen Ma, Xudong Zhu, Weixiang Wang, and An Zhang. Cloud service selection based on weighted KD tree nearest neighbor search. *Appl. Soft Comput.*, 131:109780, 2022.

[BPJ22] Amine Barrak, Fábio Petrillo, and Fehmi Jaafar. Serverless on machine learning: A systematic mapping study. *IEEE Access*, 10:99337–99352, 2022.

[BTK22] Ta Phuong Bac, Minh-Ngoc Tran, and Young-Han Kim. Serverless computing approach for deploying machine learning applications in edge layer. In *International Conference on Information Networking, ICOIN 2022, Jeju-si, Republic of Korea, January 12-15, 2022*, pages 396–401. IEEE, 2022.

[CAS23] Faeze Azimi Chetabi, Mehrdad Ashtiani, and Ehsan Saeedizade. A package-aware approach for function scheduling in serverless computing environments. *J. Grid Comput.*, 21(2):23, 2023.

[CFSS19] Bin Cheng, Jonathan Fürst, Gürkan Solmaz, and Takuya Sanada. Fog function: Serverless fog computing for data intensive iot services. In *SCC*, pages 28–35. IEEE, 2019.

[CLMS20] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE Access*, 8:85714–85728, 2020.

[CPMS21] Dheeraj Chahal, Surya Palepu, Mayank Mishra, and Rekha Singhal. Sla-aware workload scheduling using hybrid cloud services. In *HiPS@HPDC 2021: Proceedings of the 1st Workshop on High Performance Serverless Computing, Virtual Event, Sweden, 25 June, 2021*, pages 1–4. ACM, 2021.

[DKK22] Jonathan Decker, Piotr Kasprzak, and Julian Martin Kunkel. Performance evaluation of open-source serverless platforms for kubernetes. *Algorithms*, 15(7):234, 2022.

[DZF+20] Shuiguang Deng, Hailiang Zhao, Weijia Fang, Jianwei Yin, Schahram Dustdar, and Albert Y. Zomaya. Edge intelligence: The confluence of edge computing and artificial intelligence. *IEEE Internet Things J.*, 7(8):7457–7469, 2020.

[HFC+23] Zhuo Huang, Hao Fan, Chaoyi Cheng, Song Wu, and Hai Jin. Duo: Improving data sharing of stateful serverless applications by efficiently caching multi-read data. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023, St. Petersburg, FL, USA, May 15-19, 2023*, pages 875–885. IEEE, 2023.

[HFG+19] Joseph M Hellerstein, Jose M Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. www.cidrdb.org, 2019.

[HKH+19] Dan Häberlein, Lars Kafurke, Sebastian Höfer, Bogdan Franczyk, Bernhard Jung, and Erik Berger. Resilient environmental monitoring utilizing a machine learning approach. In *Artificial Intelligence and Soft Computing - 18th International Conference, ICAISC 2019, Zakopane, Poland, June 16-20, 2019, Proceedings, Part I*, volume 11508 of *Lecture Notes in Computer Science*, pages 85–93. Springer, 2019.

[HTZT21] M. Reza HoseinyFarahabady, Javid Taheri, Albert Y. Zomaya, and Zahir Tari. Data-intensive workload consolidation in serverless (lambda/faas) platforms. In *20th IEEE International Symposium on Network Computing and Applications, NCA 2021, Boston, MA, USA, November 23-26, 2021*, pages 1–8. IEEE, 2021.

[Hub01] Bert Hubert. tc - show / manipulate traffic control settings. https://man7.org/linux/man-pages/man8/tc.8.html, 2001.

[JGL+21] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 857–871. ACM, 2021.

[JSSS+19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[KAMM23] Andreas Kanavos, Nikos Antonopoulos, Alaa Mohasseb, and Phivos Mylonas. Analyzing public sentiment towards the covid-19 pandemic: A twitter-based sentiment analysis and machine learning approach. In *18th International Workshop on Semantic and Social Media Adaptation and Personalization, SMAP 2023, Limassol, Cyprus, September 25-26, 2023*, pages 1–6. IEEE, 2023.

[KF22] Vojdan Kjorveziroski and Sonja Filiposka. Kubernetes distributions for the edge: serverless performance evaluation. *J. Supercomput.*, 78(11):13728–13755, 2022.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[KHLZ20] Young Ki Kim, M. Reza HoseinyFarahabady, Young Choon Lee, and Albert Y. Zomaya. Automated fine-grained CPU cap control in serverless computing platform. *IEEE Trans. Parallel Distributed Syst.*, 31(10):2289–2301, 2020.

[KKH23] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine learning operations (mlops): Overview, definition, and architecture. *IEEE Access*, 11:31866–31879, 2023.

[LLW+23] Yongkang Li, Yanying Lin, Yang Wang, Kejiang Ye, and Cheng-Zhong Xu. Serverless computing: State-of-the-art, challenges and opportunities. *IEEE Trans. Serv. Comput.*, 16(2):1522–1539, 2023.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval

Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[LWL+23] Yongjia Lei, Zujian Wu, Zhiying Li, Yuer Yang, and Zhongming Liang. Bp-capsnet: An image-based deep learning method for medical diagnosis. *Appl. Soft Comput.*, 146:110683, 2023.

[MCS23] Dimitrios Michael Manias, Ali Chouman, and Abdallah Shami. Model drift in dynamic networks. *IEEE Commun. Mag.*, 61(10):78–84, 2023.

[Min23] MinIO, Inc. Deploy MinIO: Multi-Node Multi-Drive. `https://min.io/docs/minio/linux/operations/install-deploy-manage/deploy-minio-multi-node-multi-drive.html`, 2023.

[MKW19] Johannes Manner, Stefan Kolb, and Guido Wirtz. Troubleshooting serverless functions: a combined monitoring and debugging approach. *SICS Softw.-Intensive Cyber Phys. Syst.*, 34(2-3):99–104, 2019.

[MPK+22] Subrota Kumar Mondal, Rui Pan, Hussain Mohammed Dipu Kabir, Tan Tian, and Hong-Ning Dai. Kubernetes in IT administration and serverless computing: An empirical study and research challenges. *J. Supercomput.*, 78(2):2937–2987, 2022.

[NJ21] Falak Nawaz and Naeem Khalid Janjua. Dynamic qos-aware cloud service selection using best-worst method and timeslot weighted satisfaction scores. *Comput. J.*, 64(9):1326–1342, 2021.

[RDC+17] Eduardo Roloff, Matthias Diener, Emmanuell Diaz Carreño, Luciano Paschoal Gaspary, and Philippe O. A. Navaux. Leveraging cloud heterogeneity for cost-efficient execution of parallel applications. In *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*, volume 10417 of *Lecture Notes in Computer Science*, pages 399–411. Springer, 2017.

[RHM+19] Thomas Rausch, Waldemar Hummer, Vinod Muthusamy, Alexander Rashed, and Schahram Dustdar. Towards a serverless platform for edge ai. USENIX Association, 2019.

[RHS+21] Thomas Rausch, Waldemar Hummer, Christian Stippel, Silvio Vasiljevic, Carmine Elvezio, Schahram Dustdar, and Katharina Krösl. Towards a platform for smart city-scale cognitive assistance applications. In *IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops, VR Workshops 2021, Lisbon, Portugal, March 27 - April 1, 2021*, pages 330–335. IEEE, 2021.

[Ric23] Felix Richter. Amazon maintains lead in the cloud market, 2023. https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/.

[RLYK21] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. Infaas: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 397–411. USENIX Association, 2021.

[RND23] Philipp Raith, Stefan Nastic, and Schahram Dustdar. Serverless edge computing - where we are and what lies ahead. *IEEE Internet Comput.*, 27(3):50–64, 2023.

[RRD21] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Gener. Comput. Syst.*, 114:259–271, 2021.

[RRD+22] Philipp Raith, Thomas Rausch, Schahram Dustdar, Fabiana Rossi, Valeria Cardellini, and Rajiv Ranjan. Mobility-aware serverless function adaptations across the edge-cloud continuum. In *15th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2022, Vancouver, WA, USA, December 6-9, 2022*, pages 123–132. IEEE, 2022.

[RRP+22] Philipp Raith, Thomas Rausch, Paul Prüller, Alireza Furutanpey, and Schahram Dustdar. An end-to-end framework for benchmarking edge-cloud cluster management techniques. In *IEEE International Conference on Cloud Engineering, IC2E 2022, Pacific Grove, CA, USA, September 26-30, 2022*, pages 22–28. IEEE, 2022.

[SA22] Pablo Gimeno Sarroca and Marc Sánchez Artigas. Mlless: Achieving cost efficiency in serverless machine learning training. *CoRR*, abs/2206.05786, 2022.

[SABG23] Biswajeet Sethi, Sourav Kanti Addya, Jay Bhutada, and Soumya K. Ghosh. Shipping code towards data in an inter-region serverless environment to leverage latency. *J. Supercomput.*, 79(10):11585–11610, 2023.

[SAG23] Biswajeet Sethi, Sourav Kanti Addya, and Soumya K. Ghosh. LCS : Alleviating total cold start latency in serverless applications with LRU warm container approach. In *24th International Conference on Distributed Computing and Networking, ICDCN 2023, Kharagpur, India, January 4-7, 2023*, pages 197–206. ACM, 2023.

[SGH15] Nicolás Serrano, Gorka Gallardo, and Josune Hernantes. Infrastructure as a service and cloud technologies. *IEEE Softw.*, 32(2):30–36, 2015.

[SKM22] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: A survey of opportunities, challenges, and applications. *ACM Comput. Surv.*, 54(11s):239:1–239:32, 2022.

[SSM20] Alessandro Di Stefano, Antonella Di Stefano, and Giovanni Morana. Scheduling communication-intensive applications on mesos. *Int. J. Grid Util. Comput.*, 11(1):103–114, 2020.

[Ste18] Manuel Stein. The serverless scheduling problem and NOAH. *CoRR*, abs/1809.06100, 2018.

[TSS22] Neha Thakur, Avtar Singh, and Amrit Lal Sangal. Cloud services selection: A systematic review and future research directions. *Comput. Sci. Rev.*, 46:100514, 2022.

[VFA23] Parichehr Vahidinia, Bahareh J. Farahani, and Fereidoon Shams Aliee. Mitigating cold start problem in serverless computing: A reinforcement learning approach. *IEEE Internet Things J.*, 10(5):3917–3927, 2023.

[WDF+23] Hao Wu, Junxiao Deng, Hao Fan, Shadi Ibrahim, Song Wu, and Hai Jin. Qos-aware and cost-efficient dynamic resource allocation for serverless ML workflows. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023, St. Petersburg, FL, USA, May 15-19, 2023*, pages 886–896. IEEE, 2023.

[WL22] Ronghan Wang and Junwei Lu. Qos-aware service discovery and selection management for cloud-edge computing using a hybrid meta-heuristic algorithm in iot. *Wirel. Pers. Commun.*, 126(3):2269–2282, 2022.

[XGT+23] Renchao Xie, Dier Gu, Qinqin Tang, Tao Huang, and Fei Richard Yu. Workflow scheduling in serverless edge computing for the industrial internet of things: A learning approach. *IEEE Trans. Ind. Informatics*, 19(7):8242–8252, 2023.

[YCWC23] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, pages 1489–1504. USENIX Association, 2023.

[YZL+22] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 768–781. ACM, 2022.

[ZLL+23] Senjiong Zheng, Bo Liu, Weiwei Lin, Xiaoying Ye, and Keqin Li. A package-aware scheduling strategy for edge serverless functions based on multi-stage optimization. *Future Gener. Comput. Syst.*, 144:105–116, 2023.

[ZR11] Yanran Zhang and Ming-Lun Ren. Web service selection based on utility of weighted qos attributes. In *Emerging Research in Web Information Systems and Mining - International Conference, WISM 2011, Taiyuan, China, September 23-25, 2011. Proceedings*, volume 238 of *Communications in Computer and Information Science*, pages 417–425. Springer, 2011.