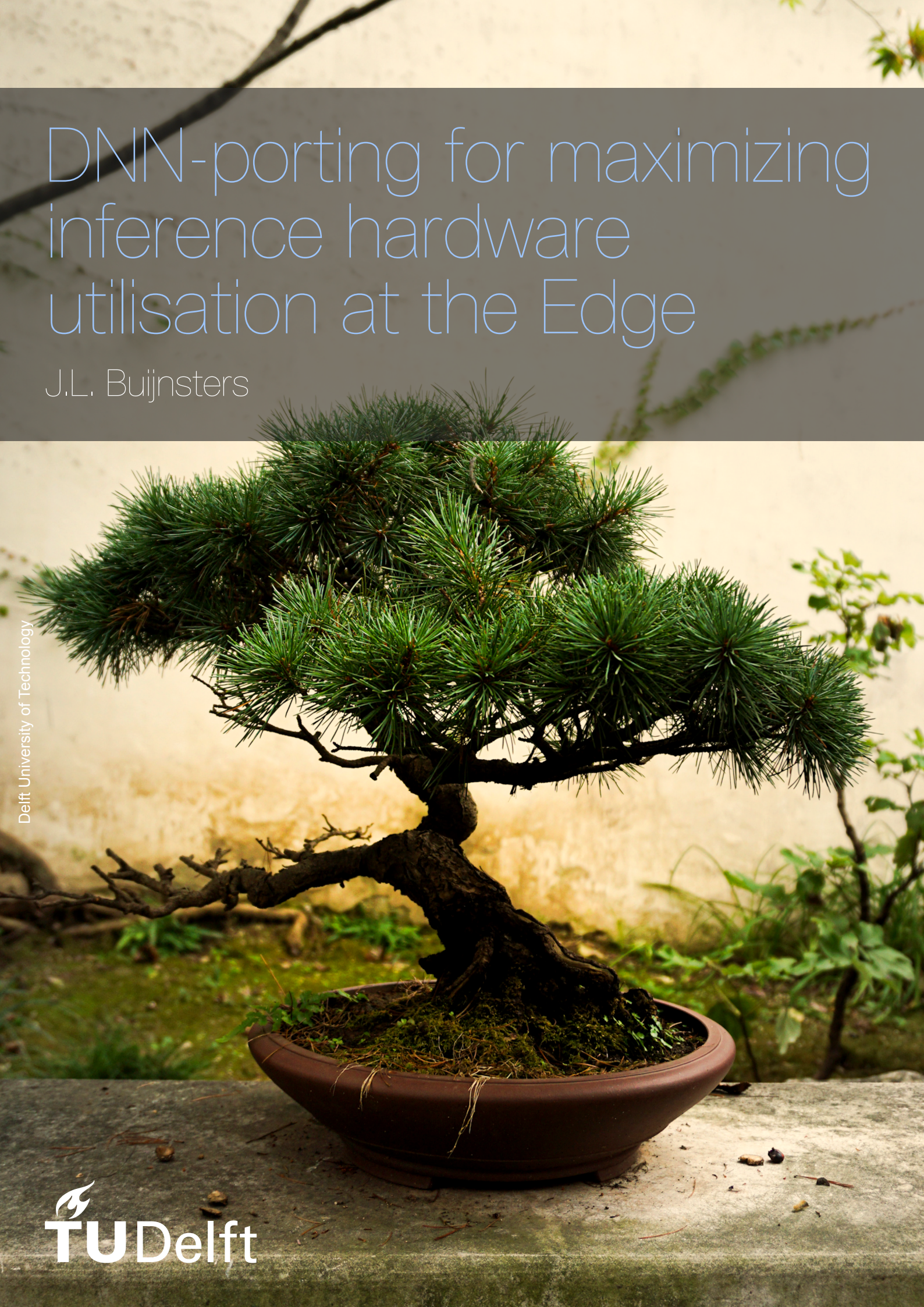
DNN-porting for maximizing inference hardware utilisation at the Edge

J.L. Buijnsters



Delft University of Technology

TU Delft

DNN-porting for maximizing inference hardware utilisation at the Edge

by

J.L. Buijnsters

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday June 26, 2023 at 13:00.

Student number: 4598733
Project duration: November, 2021 – June, 2023
Thesis committee:

Dr. P. Pawelczak,	TU Delft, Chair
Prof. Dr. J.S. Rellermeyer,	Leibniz University Hannover, Supervisor
Dr. Y. (Aaron) Ding,	TU Delft

Cover: Bonsai, Suzhou Jiangsu China by Ala J Graczyk (Modified)

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

Industry 4.0 and the Industrial Internet of Things (IIoT) growth will result in an explosion of data generated by connected devices. Adapting 5G and 6G technology could be the leading enabler of the broad possibilities of connecting IIoT devices in masses. However, the edge solution has some disadvantages, such as the loss of resource elasticity compared to cloud solutions. The research questions of this thesis are whether Deep Neural Networks (DNN)-porting can solve the accuracy-performance trade-off of edge computing solutions and how to implement an edge computing system based on open-source container-orchestrated DNN model inference platforms to enable vertical model autoscaling capabilities.

The thesis shows how porting techniques like structured pruning on DNN enable the accuracy-performance trade-off in hardware-constrained settings. It generates models with reduced complexity and size while minimally degrading the accuracy. By using these ported models in the proposed inference platform, the thesis demonstrates how an edge computing system can achieve vertical model autoscaling capabilities, enabling efficient use of computational resources.

This research focuses on CPU hardware and Real-Time (RT) request scenarios, where the latency Service Level Objective (SLO) combined with current demand are crucial factors. When the resources in an inference system deplete, the latency of individual requests can increase significantly due to queuing. The results show how an orchestrator can make live model version selections based on the model versions and demand. The proposed system increases the maximum possible throughput compared to the state-of-the-art while avoiding creating a queue in the RT scenario and improving system accuracy when CPU resources are available. Additionally, this work proposes a design to implement these benefits in industry-adopted open-source DNN inference platforms.

Preface

In this thesis, I try to solve one of the limitations of running Artificial intelligence (AI) at the Edge: the loss of resource flexibility compared to the cloud. This is done by utilising DNN-porting techniques, which can compress large DNN models to smaller variants. These variants have a smaller computational footprint while taking a small loss in model accuracy. This allows us to choose which variant is used at the Edge at a given time, which can be beneficial if computational resources are scarce. During the design and implementation of the proposed system, industry-adopted open-source technologies have been kept in mind such that a final step to a well-maintained project would be minimal. This has been a personal goal of mine, as I have seen too many research projects isolated from adaptation and further development.

Before my thesis, I was already interested in resource management in hardware clusters by utilising container orchestration with Kubernetes. One of my specific interests was how to cost-effectively move computational load from an on-premise cluster to the cloud when the on-premise resources became scarce. With these interests, I tried to find a thesis project in the Distributed Systems research group. My supervisor Professor Dr. J.S. Rellermeyer, offered me a lot of freedom in choosing the direction for my research and has provided me with many helpful insights along the way, which I am very grateful for. It allowed me to dive deeper into my interests regarding container orchestration and broaden my knowledge with respect to AI and the field of DNN-porting.

It was definitely a hard and lonesome journey, with a good portion of underestimation regarding unforeseen roadblocks. However, I am very glad to finish and leave this university phase of my life with a successful final chapter, even though it took a bit longer than initially anticipated. I would like to give a special thanks to my grandfather Leo Hartman for providing me with valuable feedback to improve my report's readability and understandability, even though it cost him some sleepless nights in the process.

I hope that my research can help anyone in the field. As with many of the research projects I have come across, I still intend to continue and add to my work and provide myself, the industry and the research community with an easy-to-adopt plugin for one of the industry's standard open-source AI inference platforms.

J.L. Buijnsters
Rotterdam, June 2023

Contents

1 Introduction

1.1. Motivation

The industry sector has gone through three revolutions until now [18]. The first started in 1784 with the invention of steam-powered engines. The second began in 1870, with the adaptation of mass production with assembly lines and electrical energy. The third was in 1969, thanks to automation by computers and electronics taking over simple ”human cognitive” tasks. The fourth industrial revolution will create Industry 4.0, including cyber-physical systems, IIoT, networks and AI. Many sensors and devices are connected to this new industry to create an IIoT network. Industries ranging from, e.g. manufacturing, healthcare, retail, agriculture, and smart cities can take advantage. The adaptation of 5G and, soon, 6G technology could be the leading enabler of the broad possibilities of connecting IIoT devices in masses. These new networking technologies have important characteristics like low latency, high reliability, higher bandwidth and broader coverage [30].

The amount of data generated in Industry 4.0 will grow significantly [24, 25], the International Data Corporation (IDC) expected exponential growth in the global DataSphere since 2018. In this scenario, the adaptation of AI can play an important role. AI can help extract valuable actions and analyse processes based on data generated by the connected devices. Commonly these computational workloads are performed in the cloud. The increased data could pressure the network of the data centres. A possible solution is Edge or Fog computing [11]. Here AI workloads move closer to the production sites, resulting in lower networking latencies and less networking pressure on the cloud data centres. Another benefit of this approach is the energy savings of the lower amount of transmitted data.

A disadvantage of the Edge solution is the loss of resource flexibility compared to cloud solutions. Adding extra storage, RAM, CPU or GPU is just a mouse-click away when using cloud solutions. This thesis proposes to use DNN model pruning and compression techniques from hardware-constrained resource directions to combat the loss of flexibility. Classically these compression techniques are used to fit DNNs onto hardware-constrained devices, such as mobile phones or embedded devices [13]. This compression can result in a smaller model size or computational footprint for inference. The Edge scenario can utilise these compression techniques to increase the maximal throughput of inferences for a task. A downside of these compression techniques is that there is often a trade-off with accuracy performance.

This work trains multiple DNN models varying in computational resource usage for the same task compromising model performance. A DNN-porting module which can generate these compressed model versions is implemented based on existing DNN-porting libraries. Using such a generative approach has the advantage that neural network engineers do not have to create multiple smaller architectures by hand. An orchestrator can select between the different AI model versions to satisfy the demand given a limited amount of available computational resources. Based on the Dagstuhl perspective on the roadmap for Edge AI, this thesis focuses on the trade-off between model accuracy and resource demand [8].

From the start of the thesis, one of the goals was to make sure that taking the last step from the proposed systems to a production setting would be minimal. As with much related scientific work, the effort to a production setting often requires much refactoring or a build-up from scratch. In an attempt

to make this possible, this thesis evaluates promising and industry-adopted open-source Kubernetes based DNN model inference platforms, like Clipper [5], KServe [20], and Seldon Core [29]. These platforms provide prediction serving systems for DNN model deployments on a Kubernetes cluster. They provide solutions like scalable inference serving by horizontal autoscaling, request routing to the correct models, canary rollout, A/B testing, monitoring and explainability. However, these tools do not include autoscaling capabilities in hardware-constrained settings, like the Edge. Switching between specific model versions for processing inference requests can achieve this. This thesis proposes and analyses this orchestration functionality to enable vertical model autoscaling capabilities to the existing prediction serving systems. Allowing them to make online trade-off decisions between the system's maximal throughput and the deployed models' accuracy performance. They can handle a higher maximal demand by compromising accuracy performance.

There are two main types of application timing when it comes to DNN inference [10], the RT and non-RT requests. The RT requests are time critical and need a proper response given a specific latency SLO, e.g. analysing RT finance activities for malicious actors. The non-RT requests are not delay critical and can get a response in an arbitrary amount of time, e.g. post-processing a photo library for certain entities. This thesis focuses on the RT request demand, and thus an essential factor is the latency SLO of the inference requests. When the resources in an inference system deplete, the latency of individual requests can increase significantly due to queuing. More specifically, this happens whenever the demand exceeds the maximum possible throughput of the system. In this RT scenario, it is beneficial to have the ability to increase the maximum possible throughput of the system to avoid creating a queue.

1.2. Research Questions

This thesis will answer the following research questions. The first research question will establish if DNN-porting can solve the accuracy and resource trade-off problem and which model profiling metric is suitable for quantifying the model resource needs. The second research question gives insight into how a model orchestrator can use the DNN-porting capabilities to fulfil a changing amount of demand from RT clients. The third research question tries to find how the solution can perform in a setting where the resources are distributed over multiple servers serving the DNN models.

RQ1. How can DNN-porting techniques be used to make trade-offs between model accuracy and resource availability?

RQ2. How can DNN-porting be used for coping with the dynamic demand of RT clients in a resource-constrained setting?

RQ3. How can DNN-porting be used to orchestrate model versioning in a distributed hardware-constrained setting?

1.3. Overview

Firstly, in chapter 2, the report discusses the works which give perspective on the future of the Edge. These formed the foundation for the motivation of this thesis. After this, the chapter discusses the state-of-the-art related works implementing similar trade-off capabilities in inference systems.

Secondly, chapter 3 is dedicated to the DNN-porting part of the thesis. This chapter briefly introduces the training of Neural Network (NN) and DNN and the different available compression techniques. Lastly, it explains the implementation of the DNN-porting module and the trade-off complexity arch of the resulting compressed models.

In chapter 4, existing open-source DNN inference systems are discussed and the different identified AI workloads. Later this chapter illustrates the thesis design based on these systems, including a final step to implementing the trade-off capabilities in one of the well-established open-source inference platforms. Finally, the chapter introduces the trivial orchestrator policy used in this thesis. This policy uses the complexity arch to achieve the trade-off capabilities within the proposed system.

To evaluate the system's capabilities, chapter 5 first discusses the experimental setup details and the stress testers used for obtaining the experiment results. The first experimental results perform a baseline evaluation on the state-of-the-art INFaaS inference system [26].

Then the section of the first research question shows how DNN-porting techniques complexity arch results can enable trade-offs in resource availability. Here the results evaluate three identified resource metrics and show that only the maximal throughput profiled on the target hardware is functional. The results also show the correlation between this metric and CPU usage percentage and how this metric behaves in situations with model co-allocation, such that the system serves multiple models simultaneously.

The results of research question 2 focus on using one runtime inference server and show how the orchestrator makes trade-off model versions switching decisions in different settings and the system characteristics during these experiments. Firstly, this section shows the results for a setting with only one DNN task, followed by co-allocation results.

Lastly, research question 3 shows the capability of the system to scale the number of runtime inference servers and utilise the extra hardware resources available by trading them for extra accuracy. Finally, chapter 6 discusses all the results achieved in this thesis and is followed by suggestions for future work.

2

Related Work

Firstly, this chapter discusses the related work regarding the industry's and scientific community's perspective on AI computations at the edge. Followed by background information on the inspiration taken from hardware-constrained model compression frameworks. Lastly, similar Machine Learning as a Service (MLaaS) papers that utilise characteristics of different versions of DNN model variants are discussed.

2.1. Edge Perspective

Edge intelligence is still an emerging topic in research and industry alike. As organisations embrace the potential of AI in Edge computing, different perspectives and approaches have emerged. This section presents two prominent perspectives in this domain: the NVIDIA and Dagstuhl perspectives. While these perspectives converge on the importance of Edge AI in enabling Industry 4.0 and the need for efficient solutions, they also highlight some areas of disagreement. Whereas NVIDIA strongly focuses on their own Edge infrastructure, Dagstuhl identifies challenges that need addressing from the research community. This section explores these perspectives, examining the role of NVIDIA's EGX servers and the cloud-edge collaboration challenges identified by Dagstuhl. Considering these diverse viewpoints, this thesis aims to understand the opportunities and challenges in implementing edge computing solutions for AI.

2.1.1. NVIDIA Perspective

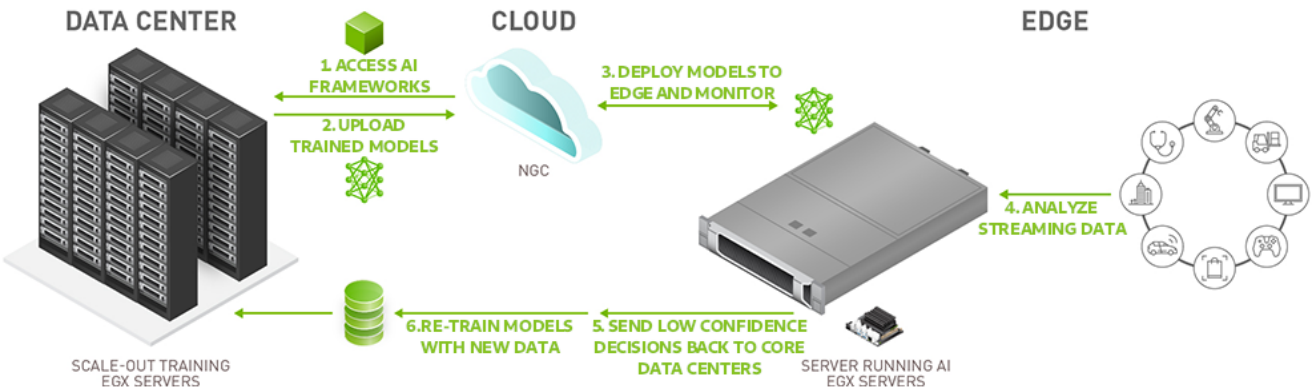


Figure 2.1: NVIDIA Edge infrastructure perspective [1]

[1] https://www.nvidia.com/en-gb/deep-learning-ai/solutions/ai-at-the-Edge/

NVIDIA is a significant part of adapting AI business intelligence in the industry. NVIDIA often plays a role in AI related projects, e.g. autonomous driving, drones, robots, and new AI papers. NVIDIA's Compute Unified Device Architecture (CUDA) toolkit [2] is a fundamental reason for choosing NVIDIA's GPUs for data centres and scientific research. NVIDIA's CUDA toolkit is a software layer that enables developers and scientists to harness the full processing power of NVIDIA GPUs efficiently. CUDA serves as a parallel computing platform and programming model specifically designed for GPUs. It provides a set of programming interfaces, libraries, and tools that allow developers to write high-performance applications that can leverage the parallel processing capabilities of NVIDIA GPUs. By utilising CUDA, developers can offload computationally intensive tasks to the GPU, significantly accelerating the processing speed compared to traditional CPU-based computations. CUDA has been widely adopted in various domains, including data centres and scientific research. NVIDIA's perspective from GTC 2021, a worldwide AI conference hosted by NVIDIA, has been taken into account in the thinking process for this thesis. At the conference, a prominent topic for enabling Industry 4.0 was the future of Edge AI.

In Figure 2.1, the infrastructure of the cloud and Edge envisioned by NVIDIA is depicted. The DNN models are trained in the cloud on the data centre servers. Notice the term "scale-out" for the training servers, alluding to the flexibility of resources in the cloud. After training models, they are uploaded to NGC, the cloud platform for cataloguing, deploying and monitoring the AI models in NVIDIA's ecosystem. From there, the models are deployed to their respective Edge sites. The Edge sites run on EGX servers, one of NVIDIA's products. Those servers process the live data streams and only send back the data of low-confidence inferences. This data can help train a new version of the model.

NVIDIA's perspective strengthens the potential of AI processing at the Edge. NVIDIA logically supports using their expensive resource-heavy EGX servers, which might be necessary for use cases like large automated factories. However, more minor use cases probably fit on modest hardware. The solution this thesis proposes makes more modest hardware approaches a viable choice, as vertical model version scaling provides higher dynamic maximal throughput capabilities. When accuracy performance needs to trade for throughput, one can consider upgrading Edge hardware. However, this is unnecessary if appropriate vertical scaling opportunities exist with smaller model versions, as the systems can free up resources on demand. The scaling can be helpful in cases where the demand load only peaks a small part of the day, as one does not have to invest and set up the extra resources necessary to handle the peak demand at maximal accuracy performance. In this situation, many resources are idle outside the peak demand times.

2.1.2. Dagstuhl Perspective

In the "Roadmap for Edge AI: A Dagstuhl Perspective" paper, a comprehensive discussion regarding Edge AI can be found [8]. This discussion is based on the collective input from the Dagstuhl Seminar 21342[3]. They discuss the technical and societal demands for using AI in Edge computing and share an envisioned roadmap for future research steps within the Edge AI domain.

An interesting argument made is the discussion around the adaptation of Edge AI. They state that to ensure that Edge AI use cases will be adopted, one needs to consider societal needs. Technological advances in the transition to 5G and Edge AI are mainly lower latency and improved reliability. However, societal needs revolve around, e.g. resource-efficient manufacturing, green energy generation and distribution, organic agriculture, and optimisation of retail logistics.

They identify the following four challenges of future perspectives regarding cloud and Edge computing collaboration.

- Resource management
- Energy constraints and efficiency
- Security, trust and privacy
- Intermittent connectivity

The resource management challenge is the most relevant one for this thesis. The "security, trust and privacy" challenges are not in the scope of this thesis but are essential for future work.

The work indicates that the locality of Edge servers limits the number of available resources compared to the cloud. This hardware constraint limits the number of Machine Learning (ML) tasks that

[2] https://docs.nvidia.com/cuda/

[3] https://www.dagstuhl.de/en/program/calendar/semhp/?semnr=21342

can run concurrently. A solution they mention would be on-demand loading and executing of the corresponding models. However, this might increase latency, which could be problematic for latency-critical tasks. Keeping these models loaded and reserving resources can limit the total amount of supported ML tasks. It would be essential to determine the risk of failure for these critical tasks with their corresponding failure consequences. In case of back-pressure from to-be-processed requests, priority queuing might be necessary to ensure critical requests are served first.

The second challenge, energy constraints and efficiency, contain two interesting views. Firstly, cloud data centres are more energy efficient than smaller Edge nodes as they profit from economies of scale. It is unclear which of the two can benefit more from renewable energy resources. Cloud centres will have a very high energy demand and create options for large-scale renewable energy. In contrast, Edge nodes will have a significantly lower energy demand and could be able to generate their own renewable energy, possibly taking advantage of their geographical dispersion. The transfer of data to the cloud also consumes power. Processing raw data at the Edge and only transmitting relevant data to the cloud can save this energy. This energy saving depends heavily on the amount of processing performed at the Edge.

The last challenge of ”intermittent connectivity” regards areas with poor connectivity. In these cases, ML tasks must run on the Edge. Combined with energy constraints, these Edge centres can only transfer limited amounts of data to the cloud, or possibly none. An example is water pollution monitoring systems, which must remain functional without connectivity. In these cases, Edge AI has a lot of potential.

Next to these four main challenges for the future of AI computing, they mention three challenges that deserve community attention. Namely, ”availability of accelerators for AI applications”, ”defining a trade-off between accuracy and resource demand”, and ”utilising Federated Learning (FL) of Edge servers”. This thesis focuses on the second challenge. Dagstuhl’s perspective states a pivotal difference for Edge AI regarding the accuracy of ML tasks. Obtaining the highest possible accuracy is no longer the common goal, as metrics like energy consumption and memory also need to be considered in the performance of models. Models must be able to adapt to currently available resources and memory. Changing the size of ML models must be possible. Techniques they mention for miniaturising the models are transprecision, quantisation or approximation. They introduce a trade-off between performance metrics and accuracy. They note that the challenge of adapting models given current and possible future loads is interesting. A possibility for these resource constraints could also be partially offloading the workload to the cloud. For example, in case the accuracy at the Edge is not sufficient.

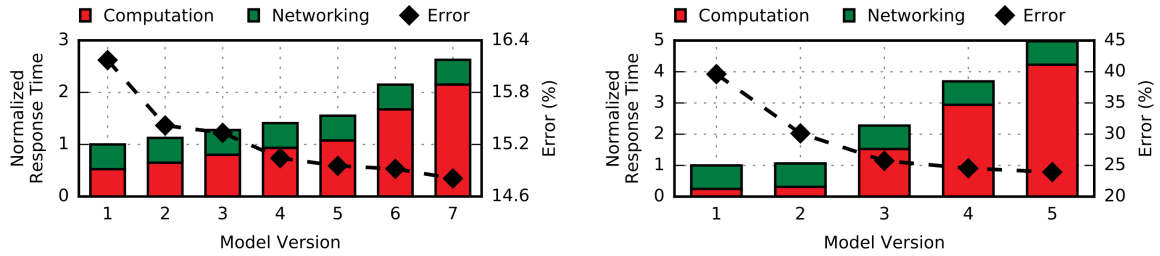
2.2. Hardware-constrained model compression

As Edge AI requires the ability to change the resource usage of ML tasks, it is critical to have the capability of tailoring DNN models to hardware configurations with constrained resources. This section focuses on the motivation behind hardware-constrained model compression techniques and how the Mistify system framework [13] inspired this thesis. The compression techniques are more thoroughly discussed in section 3.3.

Tailoring DNN models to a large range of target hardware, from low-power devices to GPU-accelerated EGX servers, is complex. One needs to keep track of all previously generated models and their characteristics, and when one adds new hardware configurations, tailored models may need to be trained. The Mistify system framework [13] tries to solve the challenges of manually generating many different model versions, specifically for tailoring models for a large variety of mobile devices. They use Google’s Morphnet under the hood [4] for the compression of the DNN models. Mistify takes care of the automation of tailoring the models. Whenever the system receives a request for a new hardware configuration, it checks if it has model versions adhering to the hardware constraints. If the hardware constraints of the returned model are smaller than the requested hardware configuration, then the system can queue up a new tailoring job for the specific hardware constraints. Morphnet iteratively prunes the model to the correct constraints. Whenever the model finishes, the new better-fitting model replaces the smaller model on the device.

The Mistify framework intrigued me and seemed like a very nice starting point for this thesis. Unfortunately, the published code of Mistify did not include the promised demo or helper modules. After unsuccessfully getting a response from the author and encountering many bugs during attempts to set up the framework, I moved on with inspiration for the next steps.

2.3. Inference systems



(a) Automatic Speech Recognition Service trade-off

(b) Image Classification Service trade-off

Figure 2.2: Accuracy/latency trade-off found in the paper ”One size does not fit all” [14]

Related works propose different inference systems using the characteristics of model versions for solving different problems. The first work uses varying characteristics to allow the consumer to make trade-offs between accuracy and latency. The second one uses model-switching for returning requests within a latency SLO during load spikes, like the solution proposed in this paper. However, they pre-load all models and use the load in terms of requests per second as the switching decision metric. At last, there is INFaaS, the state-of-the-art system, which proposes a system which automatically generates the different model versions optimised for three different hardware targets, which gives costs as an additional trade-off characteristic by choosing between the hardware used.

The first work using tolerance tiers is ”One size does not fit all” [14]. Each tolerance tier exposes specific accuracy/latency/cost characteristics. Tolerance tiers can combine multiple model versions to provide more fine-grained characteristics than only one version could achieve. Consumers can select the appropriate inference tier when making a request. This can be useful as different applications have different needs, e.g. time-critical applications might want to trade off some accuracy for reduced latency. They realise model versions for image classification using different published models for the 2012 ImageNet Large Scale Visual Recognition Competition [6]. They realise automatic Speech recognition by changing the hyper-parameters of the hidden Markov model used. Figure 2.2 shows their model version trade-off differences. The trade-off between accuracy and normalised response time is clearly shown and shows a similar pattern in both use cases. Note that the black line error range is much larger for image classification, meaning more model accuracy loss between versions. It also shows that networking latency is relatively more prominent than computation time for the less accurate models. From a certain point, reducing the accuracy of these versions further will not result in a beneficial trade-off.

They observe that not all inputs benefit from more accurate models. The performance varies for some inputs; for some, it even degrades. Combining the versions and using output confidence create different tolerance tiers. They use a statistical approach to ensure accuracy degradation for a given tolerance tier is not more than specified, with 99.9% confidence. Unfortunately, they do not give extensive insight into model versions used within the proposed tolerance tiers. Neither how the evaluation results compare to the naive approach, which does not adhere to their strict accuracy degradation metric.

In the paper Model-Switching [33], the authors have observed that end-users of MLaaS mainly care about the accuracy of responses returned within a specific deadline. They call this effective accuracy. They propose to switch from more accurate and complex models to simpler ones in the presence of load spikes to adhere to a set latency deadline of 750ms. They use different depth ResNet models pre-trained on ImageNet [6], namely ResNet-18, ResNet-34, ResNet-50, ResNet-101 and ResNet-152. In Figure 2.3, their results of the effective accuracy for an increasing load are shown. The smaller models can handle a much larger load, given the latency deadline compared to the larger models.

They use an event-based windowing approach for the decision-making of switching between model versions, monitoring the load in terms of requests per second at run-time and using the measured load in the current window for the following model version selection. To improve responsiveness, one can tune the window size offline. Unfortunately, their implementation presumes that all models are pre-loaded. As they mention, this pre-loading causes significant memory overhead. As a result, the paper makes an unreasonably small assumption about the computational resource usage of model switching since they do not have to load the new model into memory. Unfortunately, they did not publish the code

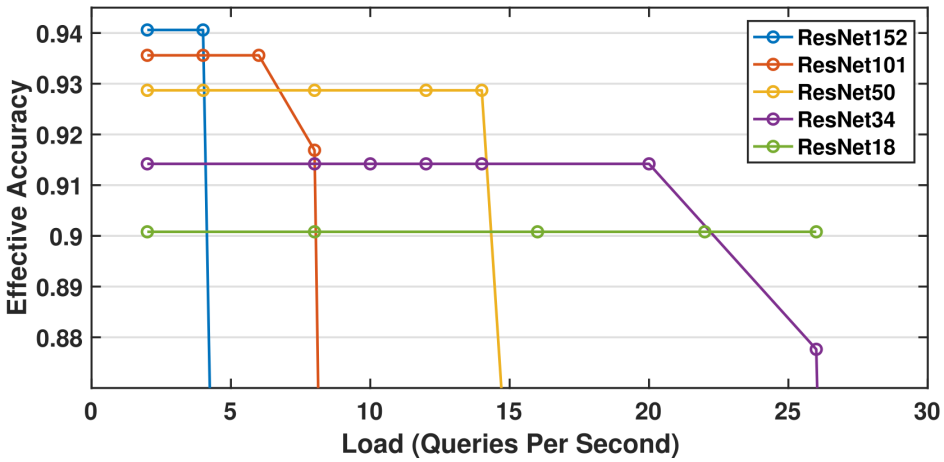


Figure 2.3: Effective accuracy ResNet found in Model-Switching [33]

for this project.

Last is the INFaaS project [26], presumably the current state-of-the-art. It was awarded Best Paper at USENIX ATC '21, with its source code made available to the public. INFaaS tries to solve the problem of selecting the best model versions given SLO requirements in a request. Developers usually have to make decisions in this search space of trade-offs for each request. INFaaS automates this model selection.

Additionally, INFaaS promises a system that can generate model variants from already trained models on the developer's behalf. They do this to create the search space across heterogeneous target hardware. It claims it can generate around 50 different variants for the underlying hardware resources CPU, GPU and Inferentia. Inferentia is a service of AWS dedicated and optimised for DNN inference. Using the characteristics of different hardware resources, INFaaS can create a broad search space of model variants. They do this by profiling latency for different batch sizes of each model variant on its underlying piece of hardware. In this search space, they can leverage trade-offs between accuracy, latency and costs of inference requests and the total demand. e.g. at low loads, INFaaS decides to use the lowest cost hardware, CPU. At high loads, INFaaS uses higher-cost hardware with more capabilities, GPU or Inferentia. Integer Linear Programming (ILP) with slack is used for the model selection and triggering a scaling-up or scaling-down event of deployed resources.

Later on, subsection 5.2.3 shows the problems encountered with INFaaS during the thesis. To defend the selection of INFaaS as the state-of-the-art baseline for this thesis, I would like to rephrase the conclusions of their paper.

They claim they built an automated model-less distributed inference serving system, which outperforms state-of-the-art inference systems in reducing costs, better throughput and fewer SLO violations.

3

DNN-Porting

This chapter discusses the DNN-porting part of the thesis. DNN-porting is the process of porting a DNN model to smaller versions of itself. These smaller versions can limit the CPU- or memory footprint.

In contrast to INFaaS [26], where they accomplish different model variants by optimizing a base model for different hardware platforms (CPU, GPU, Inferentia) and compilers (TensorRT, Neuron). This work creates different model variants by using model compression techniques. This results in more model variants for each hardware platform with greater variety in model complexity.

This chapter shows the complexity of the creating process of NN architectures. It involves lots of trial and error to find a network design and choose hyper-parameters which work well for the given dataset. Why use model compression techniques in the first place? The orchestrator in this thesis requires multiple model variants which differ in resource requirements. Thus if creating these variants is not automated, a neural network engineer must design all these networks. This extra step could make the adaptation of the system in an industry setting unfeasible.

3.1. Hardware constraints

One of the most critical choices for tailoring models is the type of resource constraint. There are two main resource constraint types: memory usage and computational. The latter often expresses itself in the number of Floating Point Operations (FLOPs) or Multiply-Accumulate operations (MACs) used for inference. The following equation is one MAC:

$$a + (b * c)$$

Whereas a FLOP represents any floating operation, which means this one MAC operation counts as two FLOPs. Memory usage involves the necessary storage in and out of memory. The amount of FLOPs concerns the required mathematical operations on the CPU or GPU. In this thesis, the focus will be on computational resource constraints. The industry often expresses the CPU's and GPU's capability in Floating Point Operations per Second (FLOPS). The literature quantifies the computation resource heaviness of neural network architectures in the number of parameters, layers, or FLOP. Later chapter 5 shows why using the theoretical number of FLOP is not sufficient for the goals of this thesis and which metric is necessary for the accuracy/resource usage trade-off.

3.2. Neural Networks

Before one can compress models, a finished and trained base model is necessary. A good source for the essentials of NN and Deep Learning (DL) is the online book "Dive into deep learning" [32]. This section briefly introduces creating and training NN models. The following key components are of the essence for any ML solution.

1. Data - To learn from
2. Model - Transforming the data to a result
3. Loss function - Quantify how accurate the result is
4. Optimisation - Adjusting the model's parameters

Data is an essential part of training an NN. A data set consists of an arbitrary number of samples. The two main methods of training models are supervised and unsupervised learning. With supervised learning, each input sample has a corresponding correct result. The model will try to perform the given task and immediately gets feedback on whether the generated result is correct.

For example, in a classification image data set of dogs and cats, each image with a dog has a corresponding class result, "dog", and each cat has the corresponding "cat" result. In the case of unsupervised learning, the data set does not contain the correct results for a given task but solely the samples themselves.

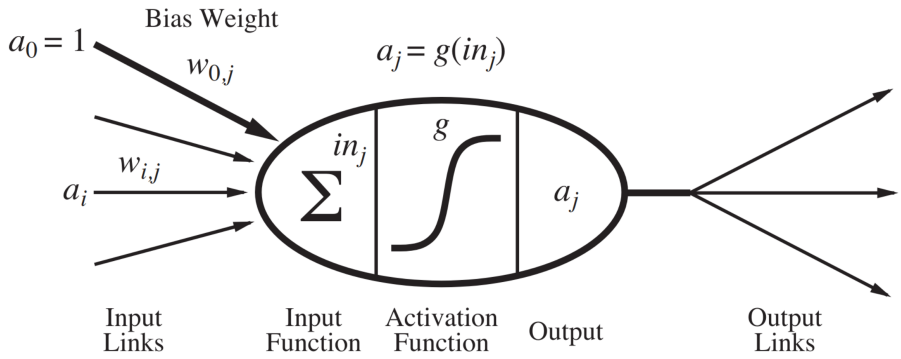


Figure 3.1: Simple mathematical model for a neuron, figure 18.19 from Artificial Intelligence A Modern Approach[28]

With DNNs, the architecture consists of multiple layers, nodes and their connections. Each node represents a neuron. In Figure 3.1, a single node is depicted. The a_i values are inputs for the neuron. These could be from a direct sample or another node in a previous layer. Each input is multiplied by the corresponding weight, $w_{i,j}$. The summation function takes the sum of these multiplications. The bias component b_j gets added to the result of the summation function before being used as input for the activation function $g(x)$. There are many different activation functions, each with different properties, e.g. Binary Step, Linear, ReLU, and sigmoid. Each node's input weights and bias update during the model's training process.

$$in_j = \sum_{i=0}^{n} a_i * w_{i,j}$$

$$a_j = g(in_j + b_j)$$

One could develop an unlimited amount of different architectures by changing the number of hidden layers, the number of nodes per layer, which nodes are connected, and different activation functions. Over the years, scientists have developed building blocks which consist of predefined node structures and operations. DNN models such as ResNet, DenseNet and AlexNet consist of many smaller building blocks. The model architecture also depends on the data set samples' shape and the result's desired shape. For example, one could train on a data set with images of 256 pixels width and height or images of 128 pixels width and height. If one wants to use the same model architecture for both data sets, the 256*256 samples need downscaling, or the 128*128 samples need upscaling to the appropriate size.

The loss function quantifies how well the model's parameters (weights and biases) function on samples from the data set. For example, in Figure 3.2, a sample with a shape of 3 features is fed into the network. During the forward pass, the values propagate through the neurons until they reach the output layer. The output vector is notated with $\hat{y}$. In the case of this simple NN, the output layer consists of one node, which, e.g. could represent how confident the model is about a given sample. The loss

[1] https://www.datasciencecentral.com/a-simple-neural-network-with-python-and-keras/

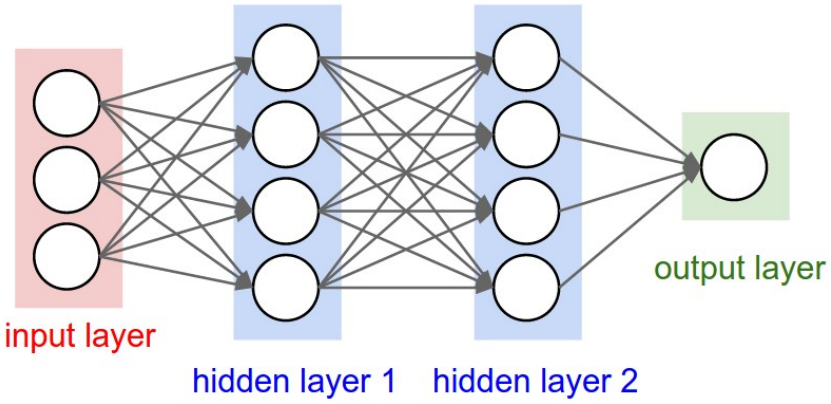


Figure 3.2: Simple NN [1]

function calculates the accuracy of the model's outcome by comparing it to the known correct result for the sample. One can choose from many different loss functions, e.g. squared error loss, cross-entropy loss, logistic loss, and hinge loss. The loss function is crucial for the optimisation part. The model must understand which output nodes need the most adjustment for a single sample. After calculating the loss, the backward pass is performed. The backwards pass calculates the loss gradient for all the model's parameters, which involves a considerable number of calculations.

$$\frac{\partial}{\partial w} Loss(\hat{y}, y)$$

Since this process requires many calculations, the backwards propagation algorithm [27] is used. It runs for each sample in the data set and calculates the average desired gradient for each model's parameter, indicating the desired change. To speed up this process, Stochastic Gradient Descent (SGD) can be used. Meaning one does not calculate the gradient of all samples in the data set but only a subset of the samples, also known as batches. This results in a gradient step in the right direction for the samples in a given batch, reaching a local optimum much faster. This process of updating the model's parameters on batches of samples happens for all samples. A complete training pass over the samples is called an epoch. Usually, one defines a maximum amount of epochs before starting the training process. However, mechanisms can be implemented, like early stopping, to ensure the training stops when the model stalls improving its performance.

When training a model, it is crucial to divide the dataset into the following subsets: train-set, validation-set and test-set. The training process only uses the training set for learning the model weights with the backwards propagation algorithm. The model directly learns from these samples. One uses the validation set to find a suitable model architecture, identify appropriate model hyper-parameters, and implement early-stopping mechanisms during training. The model does not directly learn from these validation set samples, but these samples can influence the model's design decisions, like when to stop training. One uses the test set to validate the final model. The test set samples should not influence the model in any way. It tests how well the model will generalise on similar unseen samples. This test set quantifies the model's accuracy by checking the percentage of correct predictions. The model's accuracy on the validation and test set should be similar. Suppose the validation accuracy is significantly higher than the test accuracy. In that case, the model's design is too specialised for the validation samples and will likely not generalise nicely to unseen data.

3.3. Compression techniques

3.3.1. Unstructured pruning

One can use unstructured pruning to compress NN models [3]. The typical approach for unstructured pruning is weight pruning. Weight pruning first prunes the connections between the nodes with the most negligible weight to zero. One can set a percentage sparsity goal of the total desired weights it should prune. As a result, it creates a sparse model, meaning the model contains a certain percentage of zero

[2] https://neuralmagic.com/blog/pruning-overview/

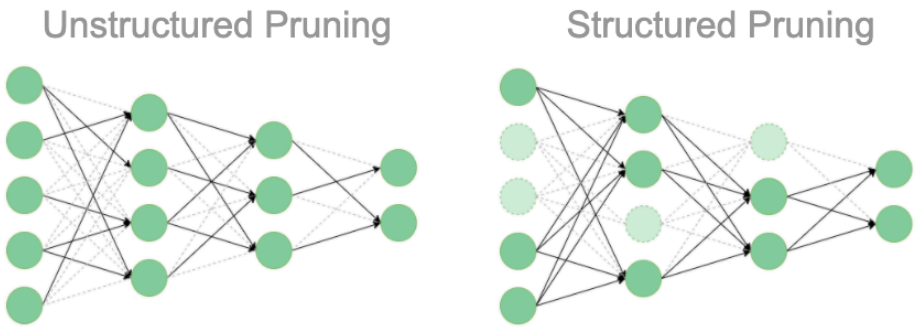


Figure 3.3: Structured pruning vs unstructured pruning [2]

weights. Figure 3.3 on the left shows an unstructured pruned network. In this network, the greyed-out arrows indicate the pruned weights. Note that the model architecture does not change. After pruning these weights, the model trains again for a certain amount of epochs to try and regain most of the lost accuracy, also known as fine-tuning the model.

Sparse models have beneficial characteristics. File compression can significantly reduce the storage size of the model, as the zero weights are all the same and do not need to be stored independently. This compression is helpful for devices with limited storage capacities and network transfers. One could skip the zero weights multiplications during inference for a speedup. Currently, most frameworks and hardware cannot benefit from sparse models in this way. There do exist implementations of sparsity accelerating run times, but these are out of the scope of this research.

3.3.2. Structured pruning

With structured pruning, one does not remove single weights but full nodes, filters, channels or intra-kernels and their corresponding input and output connections. This results in a new network architecture. Figure 3.3 on the right shows a structured pruning example. Here all greyed-out nodes and weights are pruned. The structured sparsity is directly beneficial for savings in computational resources [1]. Again, one must fine-tune the new model to ensure reasonable accuracy, like unstructured pruning. This thesis will use well-established and openly implemented structured pruning techniques.

3.3.3. Quantization

A simple yet effective way of compression is quantization. Figure 3.4 illustrates the difference between pruning and quantization. PyTorch models usually use 32-bit Floating Point (FP32) precision to represent weights and activation values. Quantization uses lower-bit representations for these values and the associated computations. This approach reduces the memory footprint and memory bandwidth requirements [12]. In this thesis, the focus has been put on the structured pruning approach, as many model optimizers already make heavy use of quantization. As a future extension of the DNN-porting module, quantization can be performed on the new variants to generate even more models for a better-optimized search space.

3.4. NNI

The Neural Network Intelligence (NNI) toolkit [22] is an open-source project from Microsoft. It is at the time of writing well maintained and makes a variety of AutoML techniques easily accessible in a plug-and-play manner. They provide implementations for techniques such as Model Pruning and Quantization. The Pruner module of NNI contains many implementations from popular pruning papers. During the selection process of the to-be-pruned parts of the model, NNI uses masks over the model to simulate a compressed model. After the pruning process is over and the pruner has made a selection, it creates the new model resulting in a genuinely compressed model with a new NN architecture.

[3] `https://nni.readthedocs.io/en/stable/compression/overview.html`

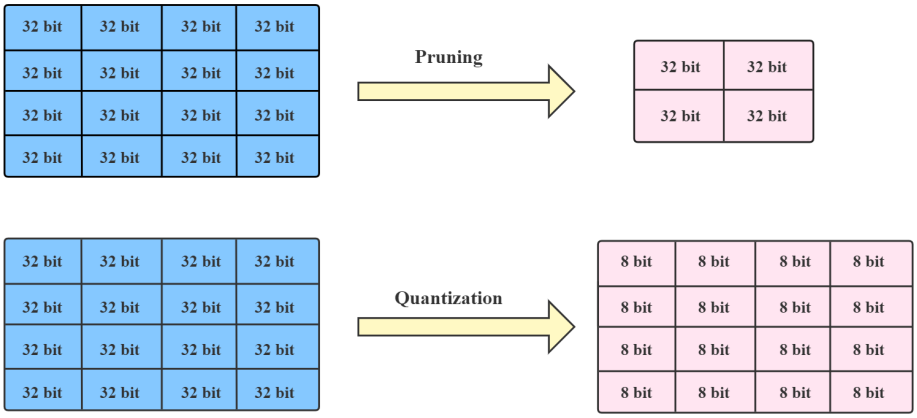


Figure 3.4: Difference Pruning vs Quantization [3]

3.5. Implementation

The DNN-porting part of the thesis is implemented using the NNI-toolkit used for model pruning. The use of this toolkit in the implementation allows the user to switch between the popular implemented pruning techniques easily. The user has to supply the DNN-porting module a base trained model with its corresponding training, test and validation dataset to generate smaller model versions. After the creation of the model variants fine-tuning is necessary. To ensure loose coupling between the different model backbones with the dataset for a given task, PytorchLightning (PL) was used. There were implementation attempts before the discovery of PL, but it became apparent that achieving the decoupling was not so straightforward.

Additionally, PL had many valuable extra functionalities. For example, PL enables the user to easily save all associated hyper-parameters with the model in a single file for a given training run, enabling straightforward training continuation and replication of training runs. With loose coupling between the data modules and model backbones, one can quickly implement their own model backbones and data modules. Further, combining any of them to train a new base model is easy.

A significant difficulty in implementation was the architectural differences between model variants. Here it became clear why most pruning solutions only deal with masks and do not compress the models. When saving a PyTorch model, one saves the weights of the model. When loading the model in the future, one has to initiate the model via its model class supplying it with the corresponding weights. This part is painfully tricky when the model classes, which include the model architecture, are dynamically generated in the past. Giving an extra PL hyper-parameter for each model backbone solves this problem. This extra hyper-parameter contains the serialized model class, to which the corresponding weights match in shape. This solution might not be the most elegant, and future work could improve this.

Once the pruner has processed a base model, the module fine-tunes the resulting new models for several epochs. Then the model variants are ready to be exported, analyzed and profiled. The models are exported to the widely adopted Open Neural Network Exchange (ONNX) format[4], an open format built to represent machine learning models. Many tools have adopted ONNX. As a result, it enables interchanging models between various ML frameworks and tools. The model's computational resource need was initially quantified based on the theoretical number of FLOPs needed for an inference request. Expectations were that there was a strong correlation between the theoretical amount of FLOPs and actual computation resource usage. This quantification differs from what we have seen in INFaaS [26], where they profile based on the latency of actual inference requests in different batch sizes on its target hardware. The profiler of this thesis calculates the theoretical amount of FLOPs by the NNI-toolkit $count\_flops\_params$[5] function. ONNX inference runtime[6] profiles the inference latency on the

[4] https://onnx.ai/

[5] https://nni.readthedocs.io/en/stable/reference/compression/utils.html#nni.compression.pytorch.utils.count_flops_params

[6] https://onnxruntime.ai/

exported ONNX models taking the average of 1000 latency inference samples. The latency profiling is performed with ONNX runtime because this can run directly from the Python code in the DNN-porting module. Otherwise, one must use the inference server in the resulting system, adding complexity. This thesis's experimental setup uses Nvidia's Triton Inference Server[7], which cannot run directly from Python, this choice is further discussed in the thesis design in section 4.3.

The results in section 5.3 show that the theoretical amount of FLOPs nor the inference latency are good metrics for quantifying CPU resource needs. It shows that profiling the maximal throughput on the target hardware gives different results. Therefore, the implementation also profiles the throughput running on a specific piece of hardware. A manual process profiles the throughput by indicating the location of the target hardware to the DNN-porting module, which runs the Triton performance analyzer[8] on the target hardware. In future work, one would want to do this through a cluster-wide procedure on all different target hardware. The accuracy profiling uses the supplied data-modules validation results. With the resulting profiled models, a Pareto optimal frontier of models based on the respective chosen metric can be extracted and used for version decision-making in the orchestrator.

3.6. Complexity Arch

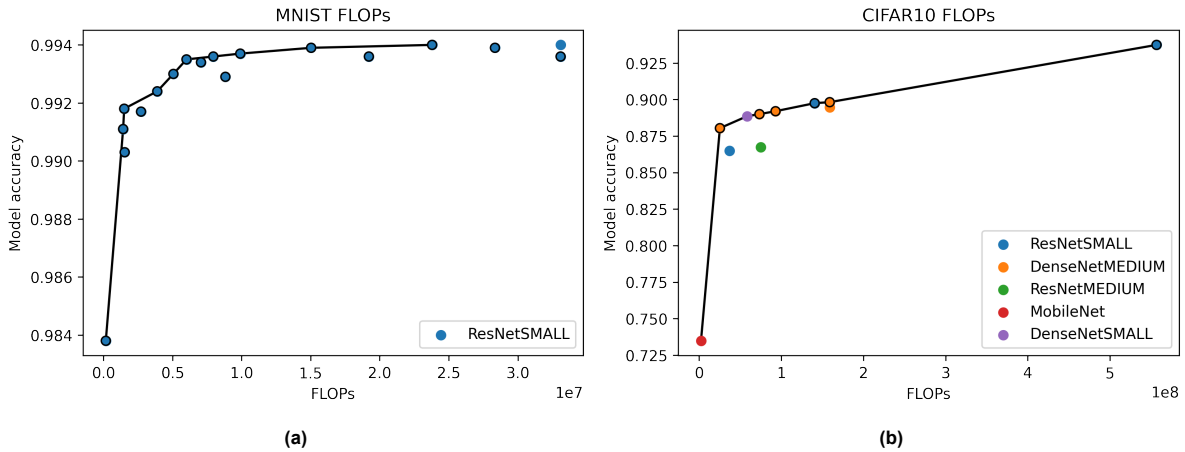


Figure 3.5: Complexity archs

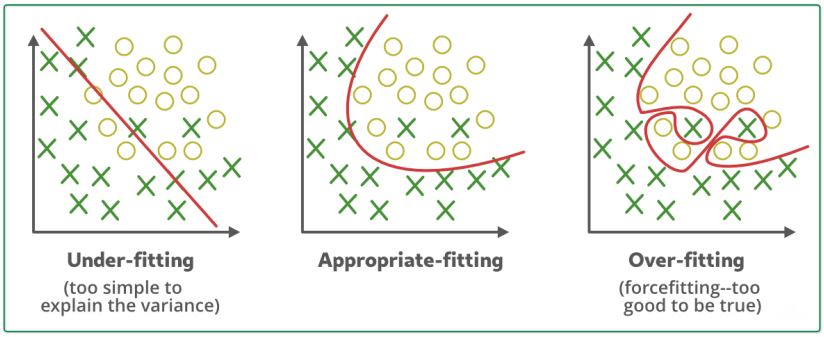


Figure 3.6: Consequence model complexity [9]

The complexity arch is a phenomenon found in many research results [2]. Firstly it comes down to the fact that it is often the case that adding extra complexity to NN models increases the accuracy of the task. For example, by adding extra hidden layers or architectural blocks, provided appropriate mechanisms are in place to prevent over-fitting. This practice is one of the main reasons why DNN

[7] https://docs.nvidia.com/deeplearning/triton-inference-server/

[8] https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/model_analyzer.html

[9] https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/

models are becoming more and more computationally expensive to train. In many tasks, there is a certain complexity threshold below which the accuracy of the task begins to decrease significantly. The intuition behind this is that if the model complexity is too low, under-fitting can happen. Figure 3.6 illustrates under-fitting. A linear function will not be able to fit the training data. Too much complexity can result in over-fitting. In both plots of Figure 3.5, the under-fitting phenomenon can be seen in real tasks. The DNN-porting module has generated these smaller models based on model ResNet-18 as indicated with ResNetSMALL. All points are individual models, and the points on the black line are in the Pareto-optimal frontier. The points with a black outline are pruned models. The complexity of the CIFAR10 task is a lot higher compared to the MNIST task. As a result, the base model ResNetSMALL is overkill for MNIST as smaller networks achieve similar results.

One can hypothesise that the achievable performance for a specific task has an upper bound given a limited amount of time and computational resources. In Figure 3.5, this theoretical upper bound could be significantly higher than the found results with the ResNet architecture, which means that another NN architecture might achieve substantially higher accuracy given fewer resources. The problem with finding these architectures is that one needs vast domain knowledge and a deep understanding of the dataset and its features. One needs to be able to design very efficient architectures that inherently profit from how the task manifests itself in the data, e.g. convolutional layers instead of a fully connected architecture for image data. Such architecture decisions immensely reduce the model resource needs and the amount of data needed to prevent overfitting.

DNN-porting should be combined with a NN engineer. The NN engineer should first create efficient base model architectures which achieve desired model accuracy. After which, the DNN-porting module generates smaller model versions with minimal accuracy degradation. In this way, the NN engineer reasonably attempts to get the determined complexity arch as close to the upper bound as possible, given the resources available.

These characteristics of model accuracy degradation can be used when computational resources are scarce, e.g. in an inference system that can solve 20 different tasks, given an equal demand for each in combination with computational resource deficits. In this situation, one would probably like to downgrade the models for which the accuracy impact is the smallest per FLOP saved. The next chapter discusses how we can utilise this idea.

4

System Design

4.1. Opensource AI/DNN serving solutions

To start on the system design, I wanted first to find frameworks in which the DNN-porting would fit. Designing the system with such a solution in mind should help the ease of using the thesis results in a real-world scenario. This process identified the following frameworks.

Firstly, Clipper [5] is a low-latency prediction-serving system used in related works. It is a framework released in 2017 to bridge the gap between having one uniform low-latency REST API given the growing number of machine learning inference frameworks. Figure 4.1 shows the general architecture of Clipper. It uses a modular approach using model containers to enable model deployment across different frameworks. It introduced extra features such as caching, batching and adaptive model selection techniques for reduced latency and increased throughput. The model selection layer can dynamically select and combine predictions across competing models to reach better accuracy and robust predictions. The recent INFaaS paper [26] uses a derivation of the Clipper framework as a state-of-the-art comparison baseline. Unfortunately, the authors stopped maintenance of Clipper in 2020.

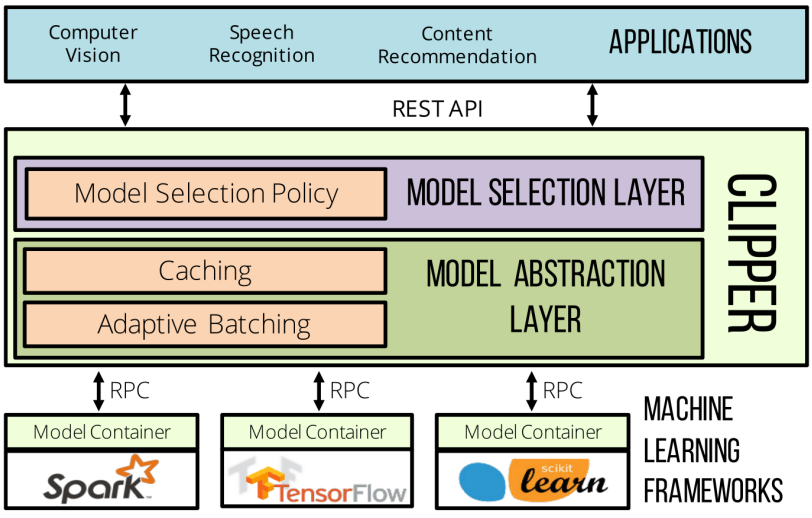


Figure 4.1: Clipper architecture [5]

Two promising maintained open-source MLOps orchestrator frameworks focused on ML deployments are KServe [20] and Seldon core [29]. These frameworks can run on a Kubernetes cluster [21]. Kubernetes is a container orchestration platform, which means it can automate containerised applications' deployment, scaling and management. It is an industry-standard, robust, scalable infrastructure for deploying and running containerised software.

An early concern in this thesis was whether it is better to deploy models in the *'one model, one server'* fashion or in an inference server per node fashion. The first would mean that each model is in

its own container or pod in Kubernetes terms. The latter means that each machine used for inference in our cluster will have an inference server running. In the *'one model, one server'* approach, one can easily use Kubernetes native structure and utilise features like auto-scaling. The expected downside of this approach is the possible extra computational overhead. KServe and Seldon Core use the *'one model, one server'* approach.

The following three main limitations exist in KServe. Firstly a limitation thanks to the extra computational overhead of the *'one model, one server'* approach. The computational overhead was around 0.5 CPU and 0.5GB Memory per model replica. Secondly, within Kubernetes, a best practice is a maximum pod limitation such that a node should not run more than 100 pods, with a default limit set to 110. Thirdly, an IP address limitation, each Kubernetes cluster has an IP address limit which can cause scalability problems.

These scalability limitations became a problem for IBM Watson, which concurrently deploys thousands of models, and thus they developed an in-house deployment solution called ModelMesh. IBM Watson open-sourced this project at the end of 2021 within the KServe project. Specific to this thesis, model version switching will frequently happen. In a *'one model, one server'* setting, the overhead of model switching is probably more significant than switching model versions within the same model server. The KServe documentation recommends using ModelMesh in high-scale, high-density and frequently-changing model-serving use cases [1].

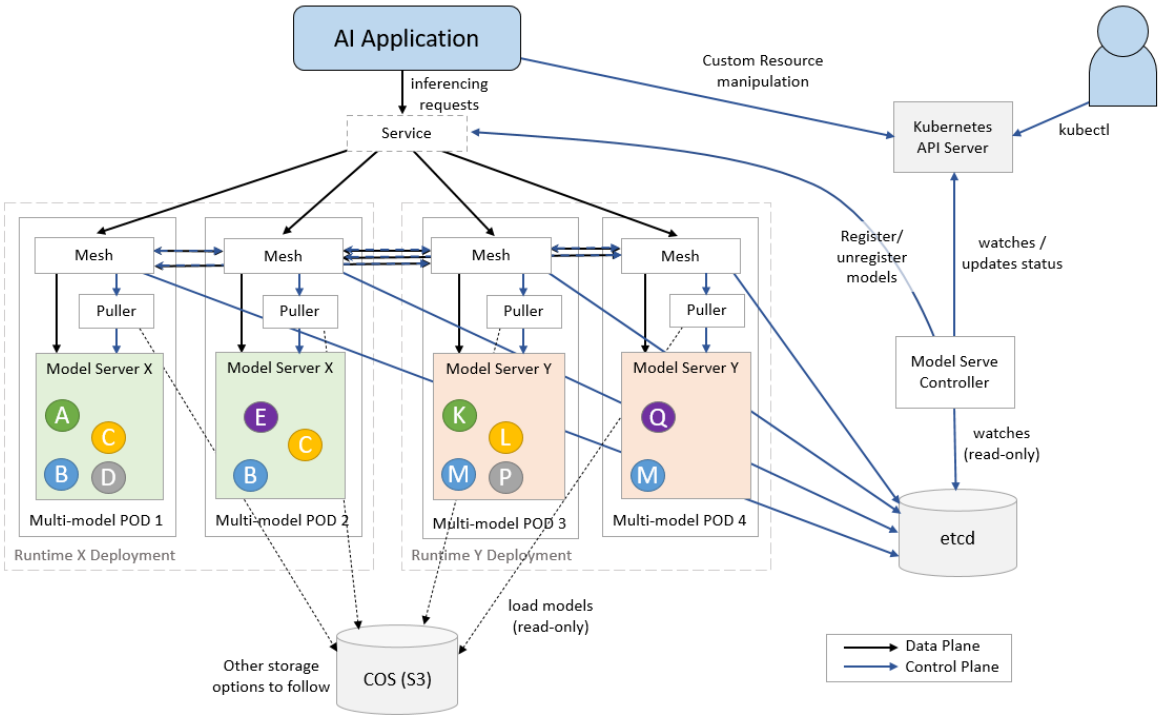


Figure 4.2: ModelMesh Serving Overview[2]

In Figure 4.2, the architecture of ModelMesh can be found. ModelMesh loads and unloads models across the cluster based on model usage over time by using a Least Recently Used (LRU) cache. The servers which deploy the models are called model serving runtimes. The runtimes supported out-of-the-box are Nvidia's Triton Inference Server[3], Seldon's MLServer[4], and OpenVINO Model Server[5]. The Model Serve Controller orchestrates which model serving runtimes should load and unload models. The Service accepts the inference requests and redirects them to one of the Mesh pods, which acts as a routing layer spanning all model serving runtimes. This routing layer also ensures that models

[1] https://kserve.github.io/website/0.8/modelserving/mms/multi-model-serving/
[2] https://kserve.github.io/website/0.8/modelserving/mms/modelmesh/overview/
[3] https://docs.nvidia.com/deeplearning/triton-inference-server/
[4] https://mlserver.readthedocs.io/en/stable/
[5] https://docs.openvino.ai/latest/ovms_what_is_openvino_model_server.html

load in the right places at the right time. S3-compatible object storage hosts all models. In most cases, each model serving runtime has a separate puller container within its pod. This puller retrieves the necessary models from the S3 object storage and saves them to a Pod-local volume accessible from the serving containers. The ModelMesh etcd is a reliable key-value store used by ModelMesh to keep track of active Pods, registered models, and which models are currently running and on which runtimes. The Kubernetes API Server is an internal component of Kubernetes itself. This server exposes the API interface for all interactions with the Kubernetes cluster. Engineers can manage the cluster via this interface. In the case of ModelMesh, one can register new models or edit existing model configurations by using this interface.

4.2. AI workloads

There are different kinds of application timing workloads [10]. The main distinction between application timings is whether the requests are RT or Non-RT. RT requests usually have a latency SLO within which the client must receive a response, e.g. accident detection for autonomous driving. There is no tight latency constraint for Non-RT requests; they can process whenever the system has time to process the requests. Often batching queries are Non-RT. This thesis will focus on RT requests, as the Non-RT requests can wait for available hardware resources. However, they could become interesting when deploying sub-optimal model versions for the RT requests over an extended period. In this case, one has to downscale further the models for the RT requests such that hardware resources become available for the processing of the Non-RT request, and doing this without compromising any latency SLO. As seen in INFaaS [26], requests can have accuracy SLO's, accuracy critical applications might only want to compromise a certain amount of accuracy.

4.3. Thesis design

This work made several design decisions to enable a system capable of serving RT requests. Looking at the related works, INFaaS has the shortcoming of being vendor-locked into the AWS ecosystem [26]. Additionally, subsection 5.2.3 shows INFaaS camps with severe inference throughput problems on their CPU-based nodes and policy priority flaws. Their policy aims to serve requests as close as possible to request SLOs instead of maximising SLOs performance given the available resources. INFaaS also uses the *'one model, one server'*. The model-switching work of Zhang et al. [33] has the shortcoming of having all model versions loaded at all times in the same model per server approach. As discussed in section 4.1, this approach leads to significant resource overhead, especially in settings with many different models like these works. Model-switching also has the pain of using the current demand for a task combined with the pre-profiled effective accuracy as their policy decision metric. This metric becomes erroneous in scenarios where the target hardware resource capabilities vary over time, e.g. if competing processes run on the hardware, shared vCPUs in the cloud or any hardware capability changes. In section 5.3, the system's capabilities change in such scenarios, making their decision policy sensitive to the exploding queue problem.

Firstly, the decision of how to process the inference requests had to be made. As subsection 5.2.3 shows, it would be very suboptimal to follow INFaaS's choice of implementing the CPU inference via its own containerised Python script compared to industry standard inference servers used by, e.g. KServe. Using the *'one model, one server'* approach, it does make sense to implement a simple inference script without much server functionality overhead, as each additional model will bring this overhead. However, the results show a throughput speedup of **7.28x** using the Triton inference server with only half of the CPU resources. Combining this with the additional overhead of the *'one model, one server'* approach indicated by the KServe research in section 4.1, and the fact that an inference server like Triton is capable of serving multiple models, the choice for the inference server per node approach is straightforward. Choosing the inference runtime server technology was less punishing as KServe defined an inference protocol that inference servers must adhere to[6], which various inference serving runtime technologies adopt. As the experimental setup includes an NVIDIA GPU and NVIDIA's Triton inference server can run CPU and GPU-based instances and supports various inference backends, the experimental designs use Triton. One can easily interchange Triton with any other inference runtime servers that implement the KServe V2 gRPC protocol mentioned. Furthermore, if the desired runtime

[6] https://github.com/kserve/kserve/blob/master/docs/predict-api/v2/required_api.md#grpc

does not provide the protocol, the possibility exists to implement it as needed. A vital functionality necessary is loading and offloading model versions on demand. This functionality was available in Triton by changing the model directory in the inference server. After corresponding with the team, they implemented it for gRPC following the KServe protocol[7].

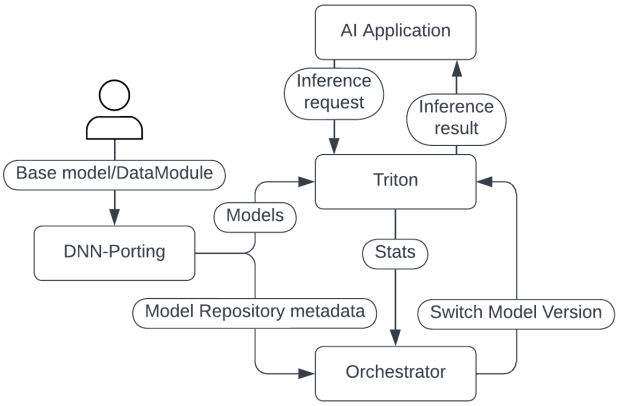


Figure 4.3: Experimental design

The system needs an orchestrator with decision-making capabilities to choose model versions. This work chose Python to implement the orchestrator as it has excellent prototyping capabilities. Additionally, it made functionality from the DNN-porting module reusable and convenient for running and processing results of all experiments. With these design decisions, Figure 4.3 shows the first simple single-server experimental design. The DNN-porting module provides the model repository metadata to the orchestrator to make informed decisions about the model versions and provides the Triton inference server with the actual models for inference. Triton provides the orchestrator's inference and resource stats via a Prometheus endpoint and the docker daemon. Based on these stats combined with the model meta-data and a decision policy, the orchestrator can make model version decisions considering the true CPU usage. In section 5.3 and section 5.4, the results of this experimental design answer the first two research questions.

The single server design is extended with multiple Triton inference servers to show the system's applicability in a distributed setting. Figure 4.4 shows the distributed experimental design. For simplicity, each inference server has an orchestrator. In this way, the orchestrator implementation did not need refactoring. In this setup, requests must be load balanced to multiple inference servers. This design uses the industry standard NGINX load balancer[8]. The load balancer serves the request to the upstream server with the least active connections. In section 5.5, this distributed experimental design is used to show the effectiveness of the system trade-off capabilities between accuracy and resource demand.

For future work, a step to integration within an open-source project like ModelMesh KServe could look like the design in Figure 4.5. It shows the envisioned design for integrating with KServe ModelMesh, and Figure 4.2 displays the standalone ModelMesh design. The ModelServe Controller in the integration has more functionality than the original, as it is also responsible for orchestrating model versions, given the system usage and demand. Figure 4.5 displays the serving runtimes as a combined Serving Runtime Mesh, and the Kubernetes API Server is left out of scope for simplification. The DNN-porting module provides the Pareto optimal models to an S3 object storage. After the upload of the models, the orchestrator receives the updated Model Repository metadata. This metadata contains all information regarding the Pareto optimal models available for orchestration. The orchestrator watches the etcd key-value store to know which models are available and ready within the Server Runtime Mesh. It also receives information regarding the resource utilisation of the individual Multi-model serving runtime pods. Given a particular orchestration policy, the orchestrator can send model ver-

[7] https://github.com/triton-inference-server/server/issues/4416
[8] https://github.com/nginx/nginx

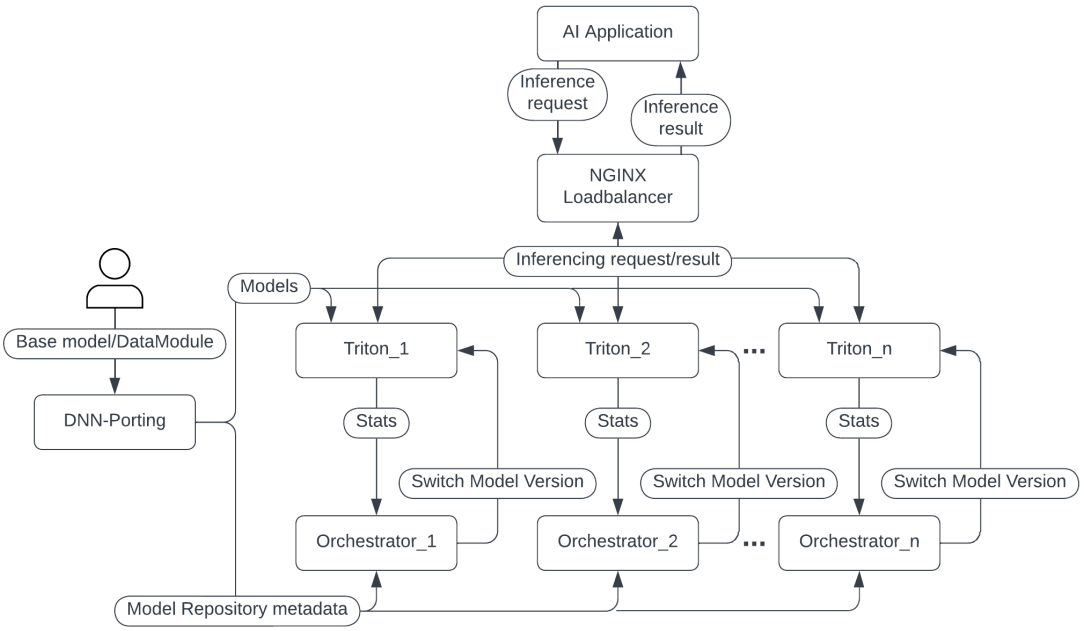


Figure 4.4: Distributed experimental design

sion switch commands to the applicable Multi-model serving runtime via the ModelMesh and monitor changes accordingly.

4.4. Orchestrator policies

The policy of the orchestrator has to decide when and which actions have to be taken, given its inputs. For this thesis, a trivial policy has been composed. Like the Model-switching paper, the latency SLOs are chosen so that a response from the largest task model is in time [33]. An important factor which has to be taken into account by the orchestrator is the additional resource overhead necessary to perform a model-switching action. Thus, a certain trade-off exists in reserving resources for switching overhead or using them for inference. The incoming RT-request stream can fully congest the system's throughput if it reserves insufficient resources for model switching. This congestion results in an exploding queue problem, as later shown in the results. The policy has to be able to switch models without causing such a resource deficit. Otherwise, many requests' latency SLO could be violated.

The trivial policy can use any of the three profiled metrics: inference latency, number of FLOPs, or maximal throughput on the specific target hardware. The implementation uses the inverse of the maximal throughput value for the maximal throughput. In this way, the metric is similar to the other metrics, such that a low value indicates a small model, and a high value indicates a large model. Based on the chosen metric, it generates a model version allocation chain for a given Model Repository metadata and a demand per task. This chain consists of all intermediate model version steps from the highest to the lowest resource usage allocation. For example, in Figure 4.6a, the accuracy and FLOPs of an example Model Repository metadata are shown. In the example, there are three different tasks. Each task has its accuracy/FLOPs Pareto optimal frontier displayed.

As one can see, the models differ significantly in resources needed and accuracy. The trivial policy uses normalised accuracy for each task. Normalised accuracy means normalising the accuracy performance of all model versions of a given task on the maximal accuracy achieved by the best model of the task. In Figure 4.6b, the normalised orchestrator view is shown. The intuition behind this decision is that if there is no resource shortage, all tasks can perform with maximum possible accuracy. Thus the system performance is maximised. It cannot go any higher without the introduction of higher accuracy models. We say that the normalised system performance is 100%. To optimally downgrade the system, the task for which the next model version loses the least normalised accuracy per chosen

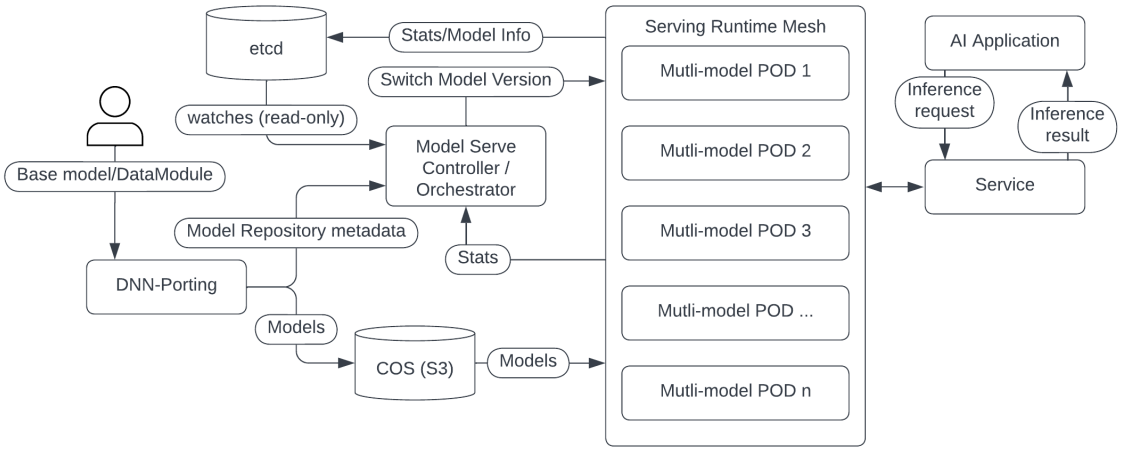


Figure 4.5: ModelMesh integration

resource metric must downgrade first. Figure 4.6c shows the estimated system performance based on downgrading the model versions in this fashion. This scenario assumes an equal demand of one request per task.

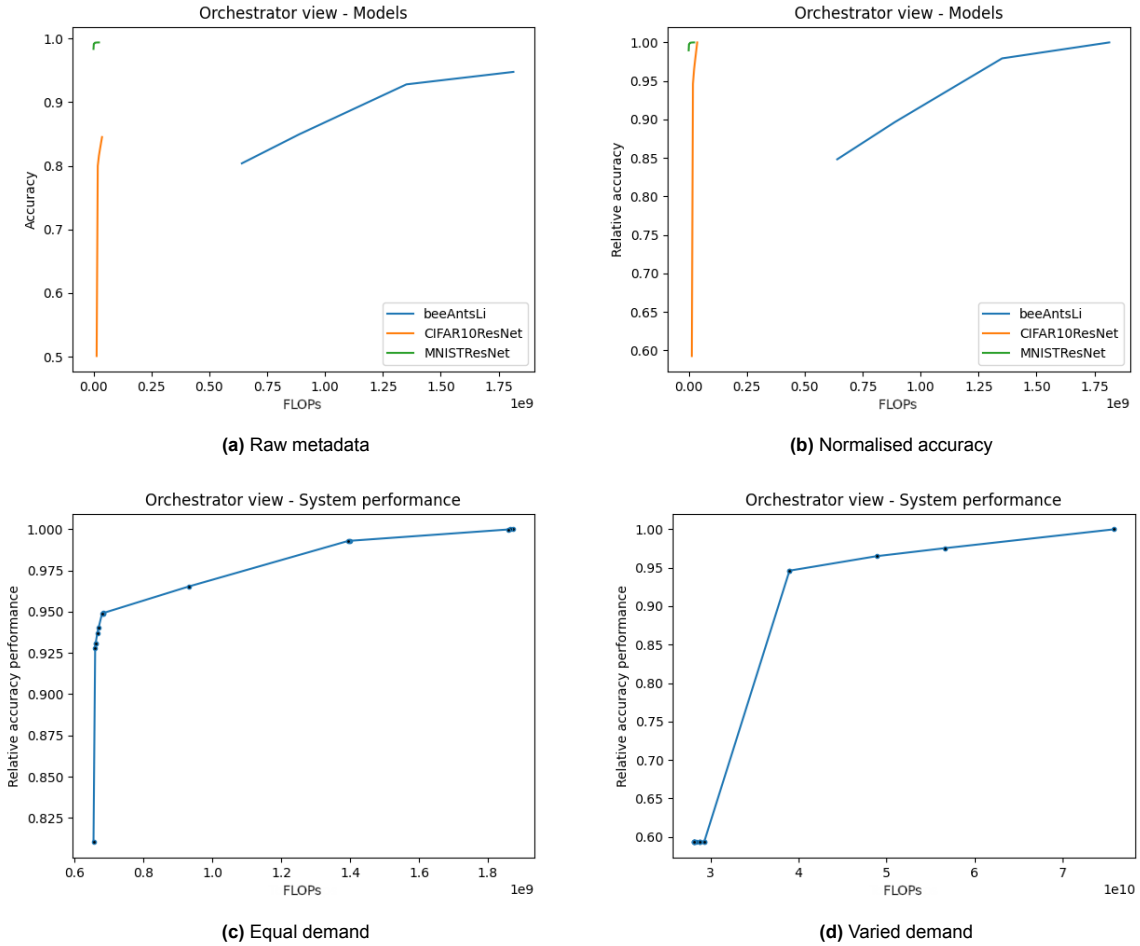
However, such a situation is not representative of scenarios with changing demands. The system performance should depend on the current demand of each task. The trivial policy multiplies the chosen resource metric needed for a model version by its current demand. This multiplication stretches a single task's accuracy/metric curve, making the slopes between model versions less steep. The policy starts with the highest accuracy models. If necessary, the policy takes a step down for the task with the flattest slope. This way, it finds a fitting allocation for optimal overall normalised system performance.

For example, in Figure 4.6d, the CIFAR10 task gets a demand of 2000 requests per second, while the other two tasks only get one request per second. It shows that if only 60 GFLOPS are available, the CIFAR10 task has to downgrade 1 model version. The system shows the most significant loss of system performance at 30 GFLOPS. At this point, the system must deploy the smallest CIFAR10 model version. Lower than this point, there is almost no normalised system performance difference, as 2000 of the 2002 requesters will get a normalised CIFAR10 accuracy of 59%. This example is a very extreme case but clearly shows how the trivial policy reacts to different demand distributions.

In an online setting, the trivial policy uses the current resource usage of the inference server for its decision-making. It logs the docker container CPU utilisation every second and pushes it into a deque with a maximum length of 20. Meaning after 20 seconds, it pushes the CPU utilisation value out of the queue. Firstly, the orchestrator checks if the last allocation update was over 15 seconds ago. This time check ensures the system has time to propagate the resource usage effect of any allocation changes. If not, the orchestrator takes a step down if the mean CPU utilisation value of the last 10 seconds is higher than 70%. If not, the orchestrator checks whether the maximal CPU value of the last 10 seconds is lower than 50%. If that is the case and the last allocation update was more than 30 seconds ago, the orchestrator takes a step up. Lastly, if the last one is not the case, but the last allocation update was more than 30 seconds ago, the system checks whether there is a more optimal current allocation given the current demand. If that is the case, the allocation gets updated correspondingly.

As one can see in Figure 4.6b, not all model updates are equally beneficial for reducing the number of FLOPs. Thus, a step-down or a step-up does not always have the desired impact. The policy chooses a proper step down of at least a given percentage smaller than the current allocation's metric usage to combat this problem generically. The minimal reduction set is a 40% reduction of the chosen metric.

In case the metric used is not the inverse throughput, thus latency or FLOPs, each step-up in allocation is done in the smallest possible resource increment to mitigate the risk of the exploding queue problem. The policy uses more sophisticated step-up logic if the inverse throughput metric applies. In section 5.3, it shows the correlation between the inverse throughput metric and the CPU



(a) Raw metadata

(b) Normalised accuracy

(c) Equal demand

(d) Varied demand

Figure 4.6: Orchestrator views

resource usage. The step-up logic can utilise this information to find a more appropriate new allocation. The intuition is that one can approximate the CPU utilisation of a new allocation by multiplying the inverse throughput metric for the model version with the demand, which translates to expected CPU usage for that task. By summing this usage for each deployed task, an CPU usage approximation is possible. However, a problem with this approach is when the inverse throughput metric does not directly correlate with CPU usages, such as in co-allocation cases or models with very high throughput, as shown in the results of research question 1.

Firstly, consider the case where the throughput metric correlates with CPU usage and CPU usage is lower than the defined 50%, thus needing a step-up action. The policy has a predefined step-up CPU utilisation goal of 65%. The policy should not select a new allocation if this allocation is going to shoot past this CPU goal. For example, going from a model version that can handle 50 req/s to 10 req/s with a demand of 20. The CPU usage before the allocation switch will be $(20/50) * 100\% = 40\%$, however with the new allocation it would become $(20/10) * 100\% = 200\%$. This policy decision would harm the system's stability and manifest in the exploding queue problem, which is undesirable. As the throughput metric correlates with the CPU usage, the policy can approximate the CPU usage of a new allocation, thus deciding not to take a step up and keep the current version allocation in place. It calculates how much the throughput metric can grow to meet the CPU usage goal. In this case, $65\%/40\% = 1.625$ is the desired metric multiplier. It then tries to find a new allocation in which the throughput metric is smaller than the desired multiplier times the current throughput metric, thus not exceeding the CPU goal of 65%.

However, if this correlation is not guaranteed, comparing the real CPU usage with the expected current CPU usage can be useful. The following calculation attempts to combat this problem while gen-

eralising to situations where resource competition exists or hardware configuration changes. Firstly, it checks whether the theoretical CPU utilisation of the current allocation adheres to the observed CPU utilisation. If this is the case, it can trust the CPU usage approximation. The policy does this by multiplying the calculated theoretical CPU utilisation with a predefined parameter step-up multiplier limit. This limit is always higher than 1. The experiments in this thesis use 1.2 as the limit. If this number is lower than the observed CPU utilisation, the policy determines that the metric does not correlate enough with the observed CPU usage. The system should notify its administrators as it cannot make well-informed decisions and requires action. In this case, the policy uses the step-up multiplier of $65\%/(theoretical\_current\_CPU * 1.2)$.

The values chosen for the trivial policy are based on trial and error. The experimental implementation allows for the implementation of improved policies in future work. For example, an online Reinforcement Learning (RL)-agent could be attractive. Another policy, the maximal metric policy, takes a metric value for each task. This policy only considers models with a lower requirement than that value. This policy is useful when the models are too resource-heavy for the chosen hardware or when the expected demand for a task is sudden and higher than the hardware can handle. For example, a sudden demand from 0 to 20 requests per second is common, and the largest model version can only handle one request per second. In this case, it is beneficial to consider only models capable of handling this sudden demand and preventing the exploding queue problem.

A noteworthy contrast to INFaaS is the different objectives of the used policy [26]. INFaaS requires each request to facilitate its desired minimal accuracy and a maximal latency SLO. Their policy objective minimises latency, whereas this thesis's objective maximises accuracy. As a result, INFaaS process requests without strict accuracy SLO by the worst accuracy model variant possible. This policy might be beneficial in the cloud setting for cost minimisation. An interesting hypothesis is whether a combination of the two would be a better policy for the cloud. In this case, a policy like INFaaS can find the most cost-effective hardware selection necessary for the current demand. Next, the thesis's trivial policy can run to maximise the accuracy given this hardware-constrained set of machines.

5 Evaluation

The evaluation in this chapter provides insights into the three research questions by evaluating the experimental design in different settings. Firstly, the evaluation shows how DNN-porting techniques can enable trade-off capabilities between accuracy and resource availability. Secondly, the study investigates the effectiveness of DNN-porting in coping with dynamic demand from RT clients in resource-constrained settings. Showing the effectiveness of DNN-porting in handling fluctuating workloads and maximising resource utilisation. Lastly, the work examines how DNN-porting can be leveraged to orchestrate model versioning in distributed hardware-constrained settings.

5.1. Models and datasets

This work uses the following datasets as tasks for the evaluation:

- BeeAnts[1]
- MNIST [7]
- CIFAR10 [19]
- CIFAR100 [19]

The BeeAnts dataset originates from a PyTorch tutorial and consists of around 200 images for both classes, ants and bees. MNIST is an image data set of handwritten digits. The CIFAR10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. The CIFAR100 dataset is similar to CIFAR10, with 100 classes containing 600 images each.

The experiments use base models generated by combining the following NN architecture backbones with the datasets:

- MobileNet [16]
- ResNet [15]
- DenseNet [17]

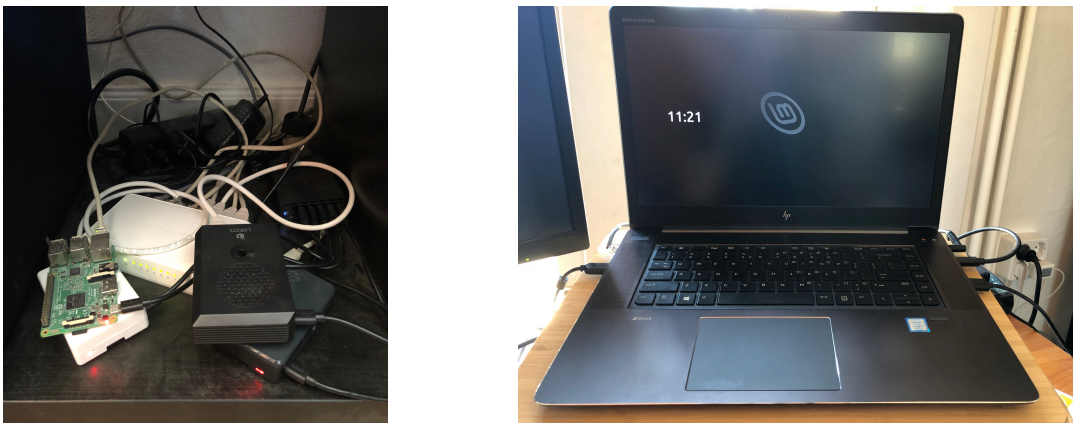
These different backbones differ in complexity and resource needs. The model versions derived from these base models are generated with the DNN-porting module as discussed in section 3.5.

[1] https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

5.2. Experimental setup

5.2.1. Hardware

The testbed for the experiments consists of two Raspberry Pi 4 Model B's, which run the triton inference servers and metric collectors. The inference servers run in a docker container, utilising one dedicated CPU thread using docker CPU-set[2]. The orchestrators, NGINX Loadbalancer, Prometheus logging server, and Grafana Dashboard run on an HP ZBOOK STUDIO G4 I7-7700HQ. Figure 5.1 shows the hardware setup.



(a) Rasberry pi's, power, and network switch

(b) HP ZBOOK STUDIO G4 I7-7700HQ

Figure 5.1: Experimental hardware

For the INFaaS evaluation, AWS instances had to be used. Here the primary and secondary nodes used the general purpose t2.medium instances[3].

5.2.2. Stress testers

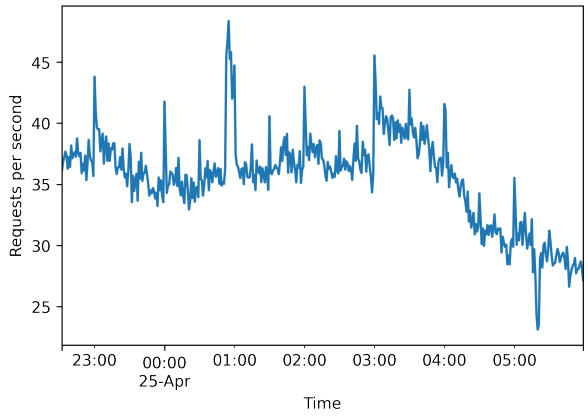
A variety of stress tests assesses the system's performance. Firstly, the constant stress tests are very straightforward and send a certain number of requests per second for the whole experiment. Secondly, linearly growing or descending stress tests are used. These have a specific request per second starting point and linearly increase or decrease after a specific interval duration. In the case of an increasing experiment, it does not stop until requests are significantly out of time or if responses contain a server error. In the case of a decreasing experiment, the test does not stop until the tested request per second is smaller or equal to 0. Then there is a similar implementation for exponentially growing experiments.

Lastly, a stress test was created based on the real-world test used in INFaaS. This stress test is created based on the timing information from Twitter trace data [31]. It is not fully clear how INFaaS did implement and used this data. This work aggregates the data by counting the tweets per second, resulting in a trace of seven and a half hours. In Figure 5.2a, the seven-and-a-half-hour trace is plotted and resampled to each minute's mean value. This plot shows the overall long-term change in demand for the Twitter service. Figure 5.2b shows a random sample of 5 minutes from the total sample without resampling. Meaning the demand change per second is shown. As can be seen, the demand is volatile, with a mean value of around 30. In order to make a helpful stress test from this data without having to run the experiment for 7.5 hours, the experiment can use a data manipulation method. Figure 5.2c shows the result of the manipulated 5-minute sample. The manipulation method has an option for a warm-up period. This method scales the demand linearly to the desired mean value by scaling the volatile trace character. After the warm-up period, a custom function can add or remove demand per second. This plot uses a sinus function.

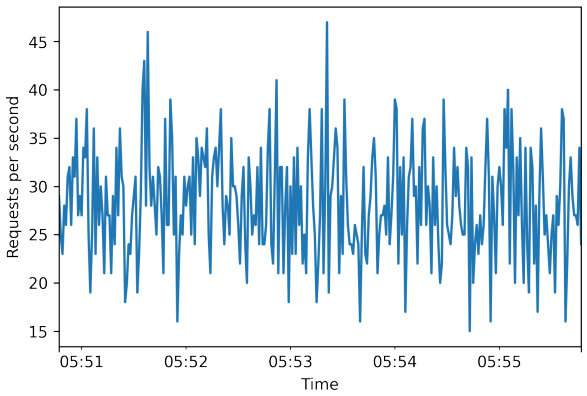
For generating random macro trends, Figure 5.3 shows the result of a custom function which combines a certain amount of sinus functions with a random amplitude, frequency, and phase. The figure and experiments with multiple tasks use a combination of 8 random sinus functions. This manipulation wave always starts at 0 and generates a random wave signal, suitable for simulating macro-trends.

[2] `https://docs.docker.com/config/containers/resource_constraints/`

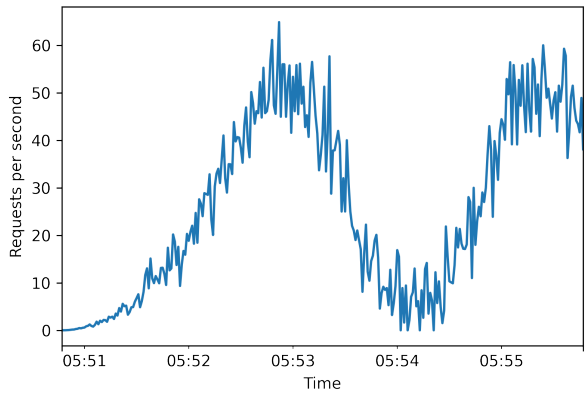
[3] `https://aws.amazon.com/ec2/instance-types/t2/`



(a) Twitter trace 1-minute resolution full 7.5 hours



(b) Twitter trace 1-second resolution random 5-minute sample



(c) Twitter trace 1-second resolution random 5-minute manipulated sample

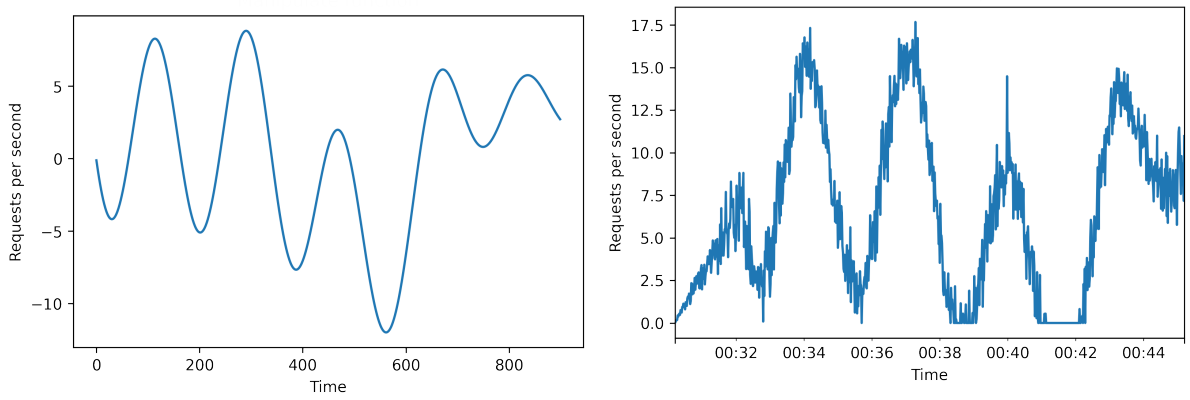
Figure 5.2: Twitter traces

One can generate a new manipulated sample when desired. The randomly generated sample with different means and deviations across tasks simulates an appropriate combination of real-world unstable short-term and simulated macro demand changes.

5.2.3. INFaaS Baseline

The work of INFaaS was selected as baseline [26]. The implementation and results in the paper were very promising, as discussed in section 2.3. However, unfortunately, the implementation did not live up to the claims made in the paper. The authors did not implement the mentioned Variant-Generator and automatic Variant-Profiler. It turned out the variant generation was a manual operation, not included in the source code. Moreover, the profiling was a bash script required to be run manually on each model variant on each target hardware. The “variants” generated for CPU-based systems only differentiate by the number of CPU cores the target hardware uses. This way of generating models comes down to the fact that the model architectures between variants for CPU are the same, but the target hardware used was different. The variants generated for the other hardware targets came down to the same idea. Except for GPU target hardware which uses TensorRT [23] to quantise the models such that they use 16-bit Floating Points. Additionally, developers can specify the maximal cost requirement for their queries. However, the requirement is not used anywhere in the source code.

Without major implementation adjustments, the INFaaS system is incompatible outside the AWS domain. As a result, the use of AWS was necessary for comparison experiments. These experiments focused on the CPU-based system without the other target hardware. The paper states that their policy selects the model variant that achieves the closest target accuracy and/or latency. However, this is blatantly incorrect, as the source code clearly shows it strictly needs both minimal accuracy and



(a) Random generated manipulate function

(b) Twitter trace 1-second resolution random 15-minute random manipulated sample

Figure 5.3: Twitter traces manipulated with random manipulate function

maximal latency SLOs to select a model variant given a specific task.

Additionally, whenever the requester chooses a combination of these two SLOs not in or too far from the available variants, the system will return an unenlightening error message that the user has to provide different SLOs as no model was available. A significant flaw to their claimed policy is that it selects the closest variant to the given SLOs. However, practically, no requester wants to be closest to these SLOs. A requester does not want minimal accuracy and/or maximal latency. It would motivate the requesters to select higher accuracies and lower latencies than necessary, putting unnecessary load on the system in moments of resource scarcity. Changing the policy would be the best solution, in my opinion. For example, one can consider the given minimal accuracy and/or maximal latency in combination with the system's current state. Based on these values, the policy should select the best possible model without increasing the overall costs of the system. Additionally, the requester could choose whether to prioritise maximising the accuracy SLO or minimising the latency SLO. In this way, the requester can clearly define his strict SLO constraints while getting served better queries if resources allow for it.

Overall, the system turned out to be highly lacking compared to the claims made in the paper. Figuring out how the system worked, or better, did not work, was very time-consuming. To give the authors some slack, it does seem like they only show the effectiveness of their approach in a heterogeneous hardware setting. However, the claims are much broader and should apply to the homogeneous CPU setting. It was the best state-of-the-art paper that implemented a similar system to this thesis's implementation. As mentioned, it was awarded the Best Paper at USENIX ATC '21. Here will follow some of INFaaS results regarding their reduced costs, better throughput and fewer SLO violations compared to previous state-of-the-art baselines. The comparisons in this thesis will use these results.

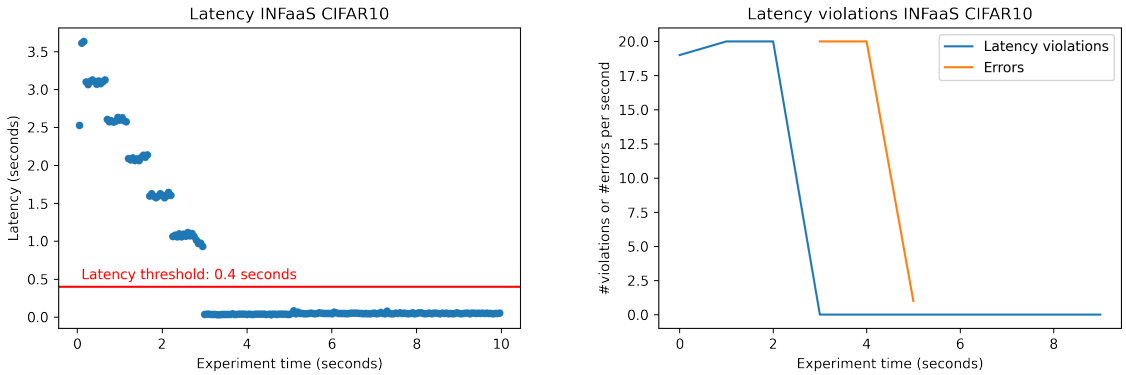


Figure 5.4: CIFAR10 - INFaaS cold start problem 20 req/s

The initially designed stress testers did not work during the first analysis as they failed in the first few seconds of the experiment. These first requests were significantly too late compared to the latency SLO, or even by returning an error within the latency SLO. Figure 5.4 shows this problem. The requests made in the first 2 seconds of the experiment all violated their latency SLO, as seen on the left. After this initial 2 seconds, requests return within the latency SLO. However, the server does respond with errors until the fifth second, as seen on the right. These returns were problematic for the stress testers, as these logically stopped when latency grew to an unacceptable amount or responses were invalid. This problem is mainly caused by starting the model variants only when the first request arrives. As discussed in section 4.1, this is done in the *'one model, one server'* fashion, which is quite an unusual choice given their high focus on cost-effectiveness. KServe showed that this approach has a significant extra computational overhead of around 0.5 CPU and 0.5 GB Memory per model replica in the KServe setting. The stress testers were modified to give the INFaaS system some slack. The modification does not cancel the experiment based on the latencies and responses from requests sent in the first 10 experiment seconds.

The next part of the analysis regards the capability of the system to function under higher loads. In Figure 5.5, the results are shown for an exponential growth experiment. For comparison, the figure in the bottom right is the inference server used by this thesis. The experiment started with one request per second and grew exponentially, with each step taking 40 seconds. The vertical green dashed lines indicate the moment the stress tester takes a step up. As seen in the upper left figure, a small latency spike exists at the start of 5 requests per second. The resources plot in the bottom left correlates with the starting up of a model replica in a new container. This correlation indicates the resource-heavy process of the *'one model, one server'* fashion. After this point, the CPU utilisation shows that the load nicely divides between the two replicas. From 9 to 33 requests per second, it seems like the response latency has a larger variety. Possibly one of the replicas was slower.

At the start of 65 requests per second, a dangerous resource spike shoots up for both containers, and here, the lines overlap in the plot. As one would expect, the response latency spikes up as well. What is noteworthy is that the error response also spikes up. This spike indicates that the requests were immediately returned by the INFaaS system, possibly without ever reaching the worker node. It is unclear what causes this extreme resource spike, but it consistently occurs across runs and correlates with the error responses. After the resource spikes, the utilisation drops to around 50% for each replica. Considering the resource utilisation progression in the last step, we can approximate that the maximum possible throughput on this hardware set-up would be around 130 requests per second. It has two vCPUs meaning maximal container utilisation of 200% is possible. However, validating this maximal throughput has not been possible due to the unstable nature of the INFaaS system.

The minimal accuracy SLO of 0 has been used for these experiments. INFaaS policy selected the MobileNet architecture as a result. This model is the worst-performing model variant concerning model accuracy. This assumption generalises for INFaaS, given the assumption that the model variant with the lowest latency also has the lowest model accuracy. This selection was expected, given their choice of policy implementation. As a result, our baseline for system accuracy performance is equal to that of the worst-performing model in the selection pool. Otherwise, INFaaS would probably have selected the model with the smallest request latency, disregarding the accuracy of the model variant.

The performance difference is shown by comparing the three INFaaS results with the bottom right plot in Figure 5.5. Here both implementations use the smallest CIFAR10 model available, MobileNet. INFaaS can access both vCPUs, whereas the triton inference server can only use the second vCPU. The Triton experiment stopped due to a networking bottleneck instead of resource scarcity. Triton performance analyser profiled the model from the worker node to establish an accurate maximal throughput. This profiling was possible with minimal resource interference as the Triton inference server only used the second vCPU. In the experimental process profiling the model versions metadata happens with the same process. The maximal throughput found was 947 inferences per second. This result shows that an industry-standard inferencing server can achieve a $947/(65*2) =$ **7.28x** speedup regarding throughput performance over the state-of-the-art INFaaS system. This calculation gives INFaaS some slack as it assumes the maximal potential throughput, deduced based on the resource usage and only half the CPU resources available for the Triton inference server. Triton had no latency violations until the last step of 513 requests per second.

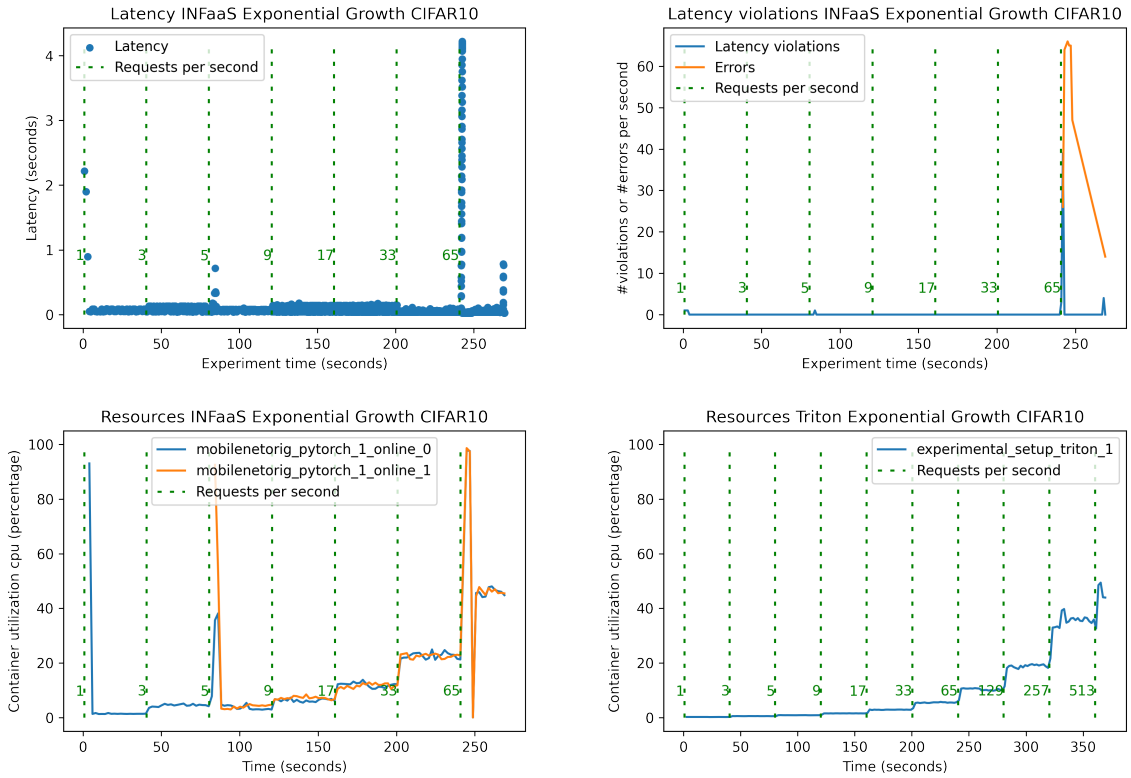


Figure 5.5: CIFAR10 - INFaaS throughput problem for exponentially growing requests per second

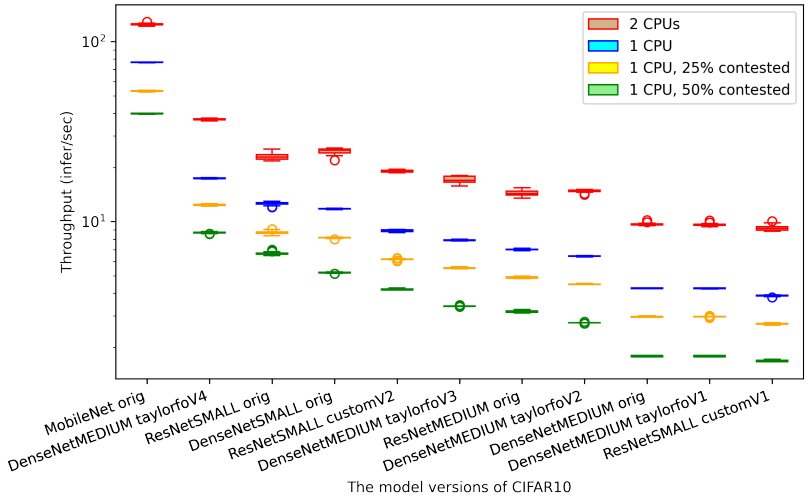
5.3. RQ1. Trade-off between Model Accuracy and Resources

How can DNN-porting techniques be used to make trade-offs between model accuracy and resource availability?

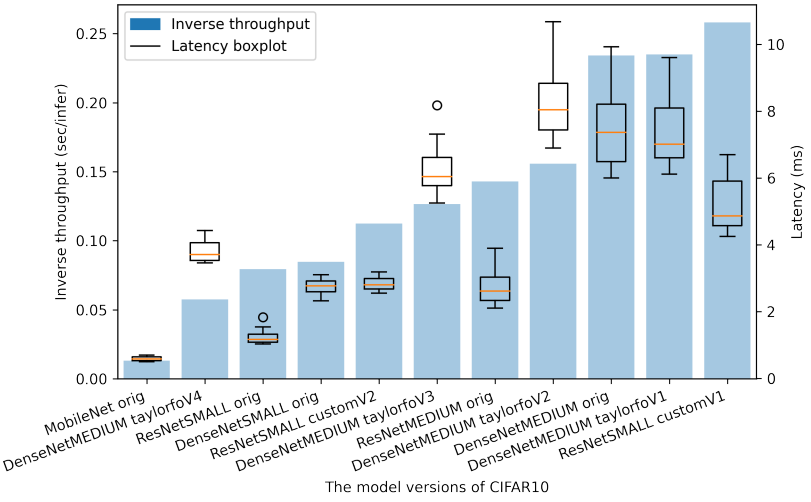
The following results explore three metrics to answer this question: the theoretical amount of FLOPs, the ONNX runtime latency for inference, and the maximal inference throughput on the target hardware. Part of section 3.5 discusses the profiling of these metrics. The maximal inference throughput on the target hardware shows from which point a hardware bottleneck exists. The model version decision-making regarding resource availability needs this data. However, this metric adds significant complexity as it needs to add profiling model versions on specific target hardware available to the system. On a heterogeneous cluster, this set-up needs to perform multiple profilings. Thus, it could be beneficial to research the two other metrics to identify a valid resource availability metric independent of the target hardware.

Firstly, inference latency was not an appropriate metric [9] because latency in this way is dismissed with the following analogy. Getting to a destination using a free highway with a strict speed limit takes time. Between the model versions, this travelling time can vary based on the length of the highway. A maximal throughput case, where a resource bottleneck has been reached, can be seen as a highway with all its lanes filled with cars driving the maximum speed limit. In this case, the latency remains unchanged and thus does not directly correlate with the number of total resources used. However, when even a single extra car wants to take the highway, it will have to wait as there is no more space, increasing latency. One can think of the number of highway lanes as the number of resources available on the experimental hardware target. As a result, the latency increases only when the demand exceeds the maximal throughput.

The measured inference latency was quite unstable between runs, even when results averaged over 1000 inference requests in contrast to the profiled maximal throughput, which was remarkably stable across 20 repetitions. Figure 5.6a shows the stability of the profiled throughputs. These results also include a situation where the 1 CPU server is contested by request to a different model on the same server. In Figure 5.6b, the instability of the latency metric is shown, given the spread of the



(a) Throughput boxplot for all CIFAR10 model versions for (contested) 1 CPU core and 2 CPU cores



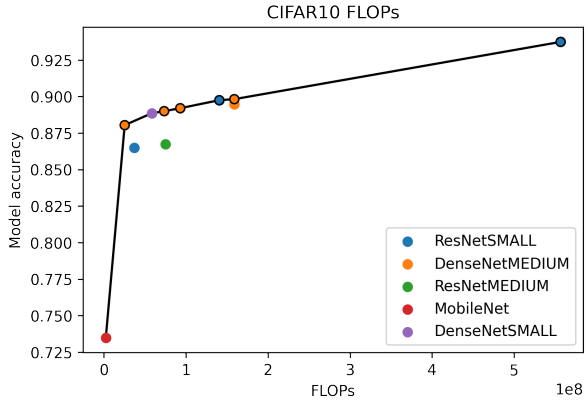
(b) Latency variations for all CIFAR10 model versions

Figure 5.6: CIFAR10 model metrics

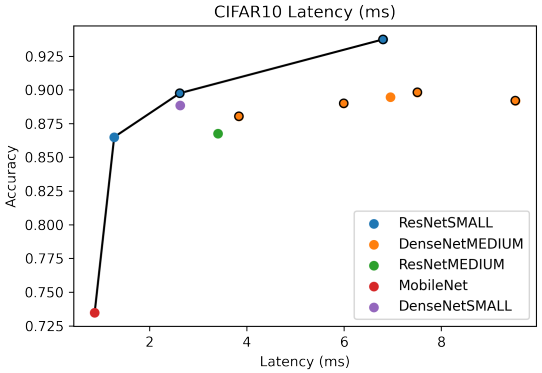
boxplots. These results show that the latency is not correlated with the maximal throughput between different model architectures, as the latency metric varies wildly compared to the measured throughput. These results show why one should not use latency as a model version decision metric.

Figure 5.7 shows the results for models trained on CIFAR10. The orange dots with a black outline in these figures are the Taylorfo pruned DenseNetMEDIUM models. The largest pruned Densenet model has the same architecture as the original but with double the training time. This model is an artefact of the pruning process, which outputs the original model as one of its results. The number of theoretical FLOPs does not correlate one-to-one with the inverse throughput. For example, in Figure 5.7a, the ResNetMedium model has a slightly higher theoretical number of FLOPs compared to the third largest version of the taylorfo pruned models. However, the ResNetMedium model has a much worse throughput than the third-pruned version in Figure 5.7b.

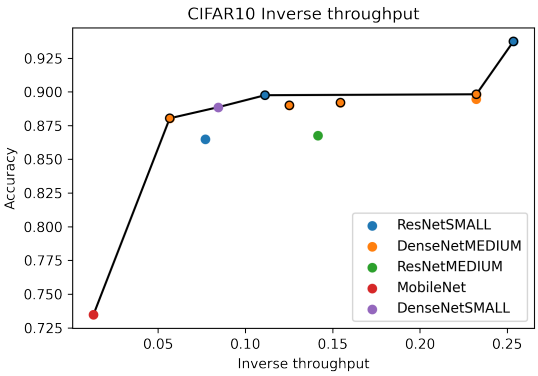
To make this problem even more apparent and follow the suggestion in section 3.6 of creating a custom architecture based on the task. Modifications were made to the ResNet architecture attempting to utilise the image size of CIFAR10 better. These models are the blue dots with black outlines in Figure 5.7. Figure 5.7d shows the Pareto frontier of the FLOP metric in the throughput domain. In Figure 5.7e, this is done for the Pareto frontier of latency. The latency metric shows a sub-optimal frontier, resulting in sub-optimal overall system performance when used for the model version decision-making. Using the theoretical FLOP metric shows another problem: a version step that reduces the



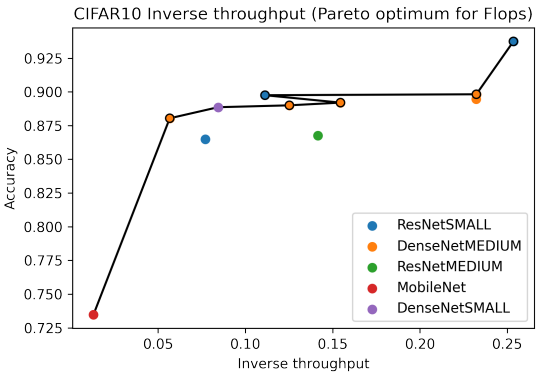
(a) CIFAR10 - FLOPs



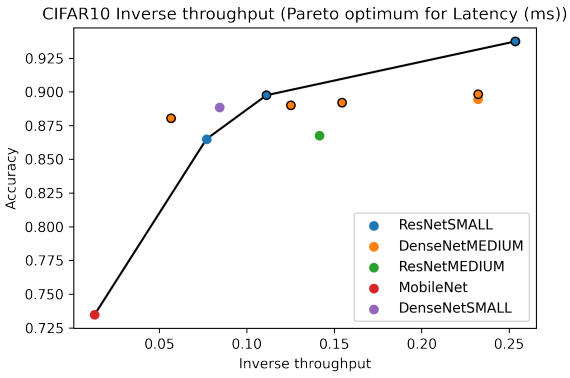
(b) CIFAR10 - latency



(c) CIFAR10 - inverse throughput



(d) CIFAR10 - inverse throughput - FLOP Pareto frontier



(e) CIFAR10 - inverse throughput - latency Pareto frontier

Figure 5.7: CIFAR10 model metrics

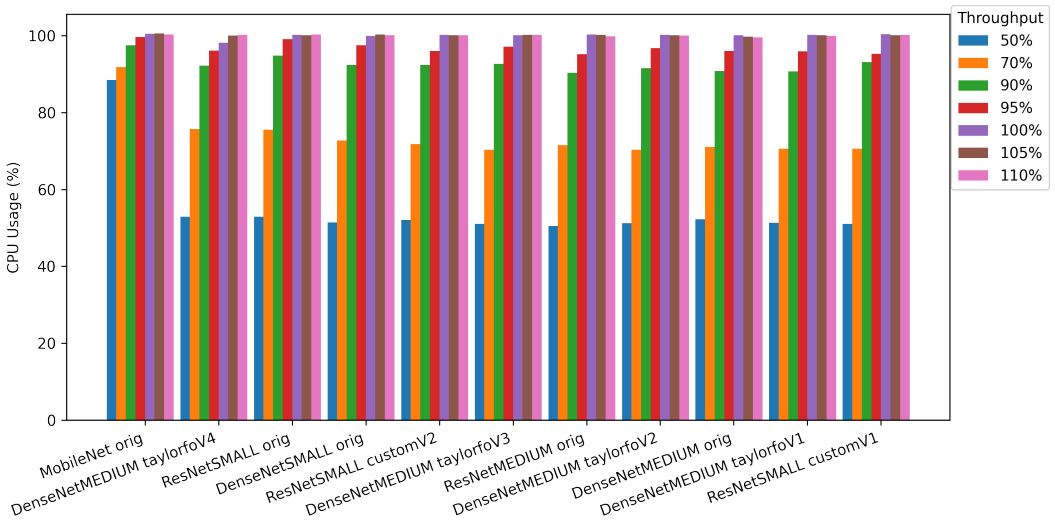


Figure 5.8: CIFAR10 - Median CPU Utilisation for varying percentages of corresponding maximal throughput for each model version

maximal throughput. Such a misstep can lead to a system overload resulting in an exploding queue problem.

Unfortunately, the number of FLOPs and the ONNX inference latency show sub-optimal results within the target hardware throughput domain. Thus, the target hardware dependency complicates profiling the model's decision metric. This dependency means the DNN-porting module will need access to the target hardware to make valid trade-offs between accuracy and resource availability.

In Figure 5.8, the CPU utilisation is shown with the number of requests per second based on different ratios concerning the maximal inference throughput of each model version. It shows that sending smaller percentages of the maximal inferences results in similar percentages in CPU utilisation, except for the MobileNet architecture, which sits significantly higher. The overhead work for each inference request could be the cause of this. This overhead becomes more prominent relative to the computational load of inference as the number of inferences increases and model complexity decreases. When sending more than the expected maximal inference throughput, the CPU utilisation maxes out at 100%. This maximal CPU utilisation is expected as the target hardware should not be able to exceed 100% CPU utilisation. Figure 5.6a showed the maximal throughput values for which the CPU became a bottleneck of the system in this experiment. Table 5.1 shows the exact throughput speedup for each model version achieved by doubling the available CPU cores. These results are based on 20 repetitions profiling the throughput. The median speedup takes the median results. The min/max speedup takes the minimal throughput of the one CPU core experiment and the maximal throughput of the corresponding experiment. The max/min speedup does this vice versa. These results show that the achieved speedup does not precisely double for each model version. Some achieve lower speedups, and some higher speedups. This difference shows the importance of profiling the model versions against the specific target hardware as these could lead to a different optimal decision Pareto frontier. Important to note is that this is already important regarding doubling the CPU cores on the same hardware device. This target hardware profiling will be even more critical when using different CPUs or other hardware like GPUs or TPUs.

The literature often mentions the adverse effect of co-allocating multiple models on target hardware. The INFaaS paper shows the effect of co-allocation for GPU hardware targets [26]. The contested columns show the effect for CPU hardware targets in Table 5.1. Here the BeeAnts model version 2 was deployed to serve the contesting requests. During the experiment, inference requests were continuously sent while profiling the CIFAR10 model versions. For the 25% column, a quarter of the maximal throughput of this model version was sent and half for the 50% column. The results show that the profiled maximal throughput is usually worse than expected. For example, for the 25% contested results, one would want to see a 0.75 multiplier. However, results show that generally, overhead loses 5% extra throughput. Looking at the CPU resource usage for the BeeAnts models in Figure 5.9, it is

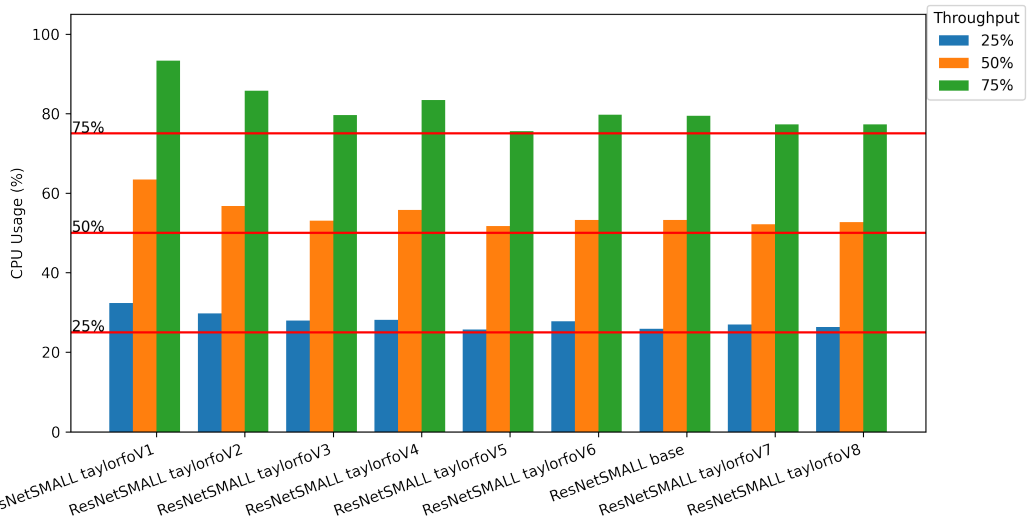


Figure 5.9: BeeAnts - Median CPU Utilisation for varying percentages of corresponding maximal throughput for each model version

Model version	**2 CPU**	**1 CPU 25% contested**	**1 CPU 50% contested**
MobileNet orig	1.62 (1.58 - 1.69)	0.69 (0.68 - 0.70)	0.52 (0.51 - 0.52)
DenseNetMEDIUM taylorfoV4	2.12 (2.06 - 2.18)	0.71 (0.69 - 0.73)	0.50 (0.49 - 0.51)
ResNetSMALL orig	1.80 (1.67 - 2.10)	0.69 (0.64 - 0.75)	0.53 (0.50 - 0.58)
DenseNetSMALL orig	2.12 (1.85 - 2.20)	0.69 (0.67 - 0.70)	0.44 (0.43 - 0.45)
ResNetSMALL customV2	2.14 (2.07 - 2.25)	0.69 (0.67 - 0.72)	0.47 (0.46 - 0.49)
DenseNetMEDIUM taylorfoV3	2.14 (1.98 - 2.31)	0.70 (0.68 - 0.72)	0.43 (0.42 - 0.44)
ResNetMEDIUM orig	2.05 (1.90 - 2.24)	0.70 (0.68 - 0.72)	0.45 (0.44 - 0.47)
DenseNetMEDIUM taylorfoV2	2.31 (2.18 - 2.38)	0.70 (0.69 - 0.71)	0.43 (0.42 - 0.44)
DenseNetMEDIUM orig	2.25 (2.22 - 2.40)	0.69 (0.69 - 0.71)	0.42 (0.42 - 0.69)
DenseNetMEDIUM taylorfoV1	2.26 (2.19 - 2.41)	0.70 (0.68 - 0.71)	0.42 (0.42 - 0.43)
ResNetSMALL customV1	2.35 (2.26 - 2.65)	0.69 (0.68 - 0.72)	0.44 (0.43 - 0.45)

Table 5.1: CIFAR10 - Throughput speedup for two CPU cores and congested server with model BeeAnts version 2 compared to free one CPU deployment. Values are Median (Min-Max) Multipliers.

clear that a non-co-allocation set-up already loses CPU resources due to overhead. The models are in descending order based on their profiled maximal throughput. Each experiment shows a median CPU usage equal to or higher than the percentage of maximal throughput, indicating overhead resource usage. These results show that it is impossible to assume the following policy-making rule: Taking the sum of each task demand divided by the maximal throughput of the allocated model version fits on the target hardware if it is smaller or equal to one.

$$\sum_{i=0}^{len(tasks)} demand\_task_i / throughput\_model\_version_i <= 1$$

Thus the orchestrator policy will have to use actual CPU utilisation to make model-switching decisions. The overhead per model version does not seem consistent in the CPU resource usage. This variation could result in sub-optimal allocation for some cases where profiled throughput is very similar. For example, take two model versions on the Pareto frontier, one with maximal throughput of 99 and the other with 100. If the one with 99 has no CPU overhead on 50% of its maximal throughput, but the 100 version has 5% overhead, then this will mean that at 50% CPU utilisation, the ”slower” model with 99 inference throughput is the optimal choice. This problem is mitigated with the implementation of the minimal step-down size when switching to smaller versions of models. This functionality ensures that

when a step down is necessary, the act of stepping down has the desired result of a less resource-heavy allocation.

Figure 5.10 visualises the resulting exploding queue problem because of the CPU bottleneck. It shows how from a certain overload throughput threshold, the latency starts growing linearly. With these results, one can clearly see that a combination of maximal profiled throughput and accuracy can be used to make trade-offs concerning CPU resource availability. Which in these cases was a bottleneck of the system. However, due to computational overhead, especially in co-allocation cases, as shown in Table 5.1, the orchestrator switching policy must consider the actual CPU utilisation. Additionally, in Figure A.1, the exploding queue problems are plotted for all CIFAR10 model versions. In all but one case, the queue explodes using 10% more requests per second over the profiled maximal throughput. Furthermore, in seven out of eleven cases, the queue explodes when handling 5% more requests per second over the profiled maximal throughput. Small latency spikes can sometimes be observed, possibly due to temporary extra computational activity from the Triton inference server or other processes running on the target hardware.

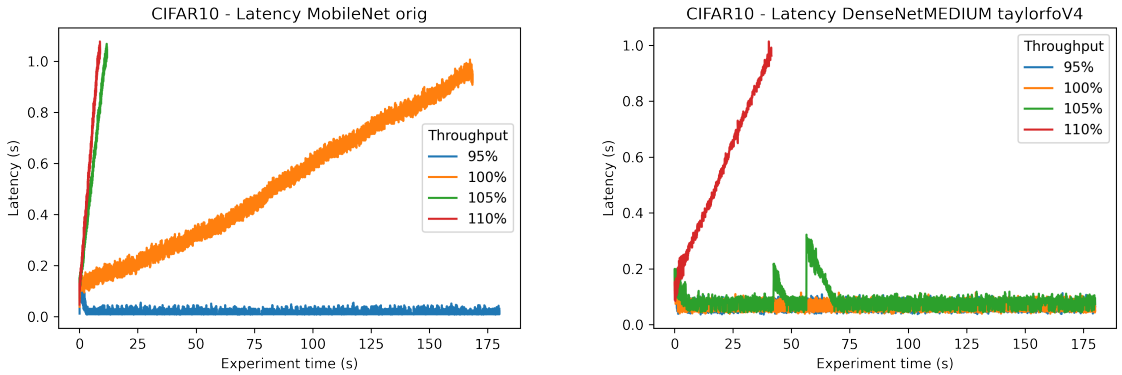


Figure 5.10: Exploding queue problem

These results clearly show the impact of model architecture on maximal throughput. It shows how structured pruning DNN-porting techniques can create a diverse Pareto frontier of model versions regarding maximal throughput concerning model accuracy without manually designing all the architecture versions. For example, pruning down the DenseNetMedium model with an original accuracy of **89.82%** and profiled throughput of **4.30 inferences/second** results in a throughput speedup of **4.09x**. This pruning creates a new model with an accuracy of **88.04%** and a maximal throughput of **17.62 inferences/second**. The results in Figure 5.8 show how the maximal throughput correlates with the CPU utilisation.

5.4. RQ2. Dynamic Demand in Resource-Constrained Settings

How can DNN-porting be used for coping with the dynamic demand of RT clients in a resource-constrained setting?

The proposed trivial policy is analysed for different workloads to answer this research question. As discussed in section 4.4, this is a trivial policy implemented to maximise normalised system performance while trying to avoid the exploding queue problem. This policy takes customisable parameters to optimise performance for different workloads and settings. The parameters set for this policy are conservative, as this work optimised the parameters by trial and error to work on all the stress tests discussed in subsection 5.2.2. This research question and policy focus on orchestrating vertical model version scaling in a setting with one inference server. The orchestration metric uses inverse throughput for all these cases because, as shown in the previous research question, the simpler metrics did not find the optimal Pareto Frontier. Firstly the results for version scaling will be shown when only one task, CIFAR10, is deployed, followed by version scaling results in settings where multiple tasks are co-allocated on the same server.

5.4.1. Single task

Figure 5.11 plots the model-chain of the policy in the inverse throughput domain and the throughput domain. These model-chain plots help to interpret the following results. Firstly it should be clear that V6 is the ”smallest” model, and V1 is the ”biggest” model regarding resource usage and maximal throughput, as shown in research question 1.

In all the following result plots, the x-axis indicates the experiment time in seconds. Each group of plots display a different aspect of the same experiment run. During the experiment, the number of requests per second sent to the inference server changes. The green dotted lines indicate these changes in demand, and the green number indicates the new demand up until the following green dotted line. The opaque blue boxes indicate the allocated model version at a given time. In the bottom-left corner of the box, one can find the associated model version of the box. Each model version box has a certain height to help visualise the change in inference performance in terms of accuracy corresponding to the values on the right y-axis. The horizontal red line in the latency plots shows the latency SLO of 0.4 seconds.

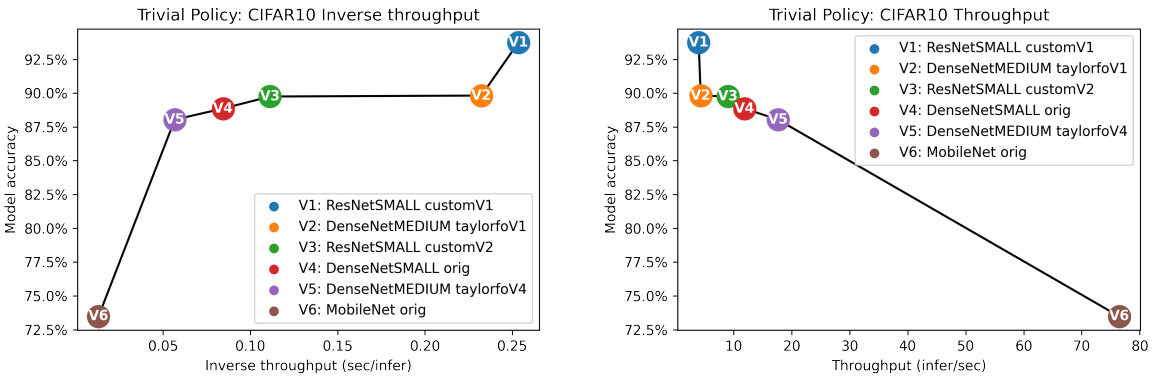


Figure 5.11: Trivial policy - CIFAR10 Model-chain

Figure 5.12 shows the results of the static allocation of version V1. The resource usage quickly grows with each increase in demand, reaching full utilisation at four requests per second. Almost all requests made during the four requests per second demand violate the 0.4 second latency SLO. However, the system does not yet experience the exploding queue problem until a demand of five requests per second. Figure 5.13 shows the results of static allocation of version V6. As discussed in subsection 5.2.3, this static allocation represents the model allocation of INFaaS when one provides a low minimal accuracy SLO, e.g. 1%. This allocation ensures no latency SLO violations up until the throughput limit of 76 req/s for version V6. However, the accuracy of all requests is minimal.

In Figure 5.14, the results for the trivial policy are shown with an exponential growth stress test, starting at one request per second and increasing up to 129 requests per second. The biggest model, V1, was already allocated by the orchestrator before the start of the experiment. In Figure 5.14a, the

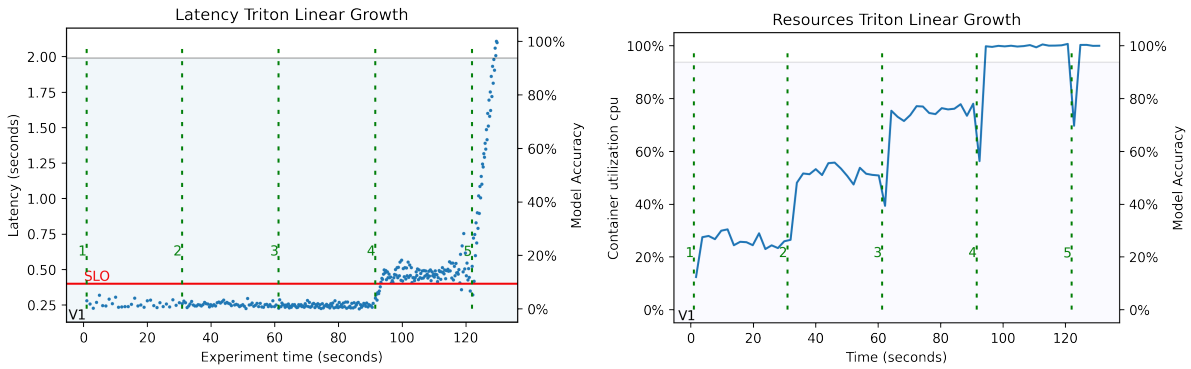


Figure 5.12: Static allocation with V1 - single task - Linear growth

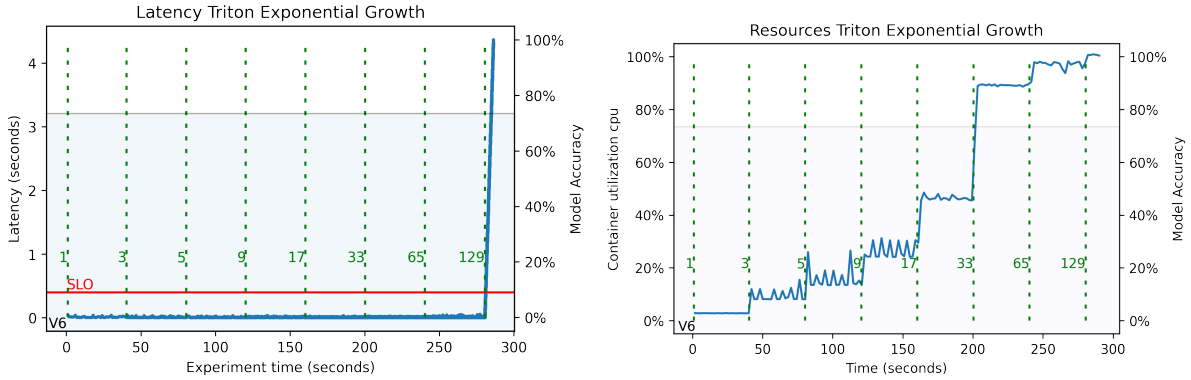


Figure 5.13: Static allocation with V6 - single task - Exponential growth

latency trails of each request are plotted, each request is represented by a dot. From these latency trails, it becomes clear that a change in model allocation causes the sequential request latencies to increase. As observed, each change in model allocation violates the latency SLO for a short time and drops back down quickly. Figure 5.14b shows the number of violations. As expected, this corresponds exactly to the latency plot. The number of violations increases whenever requests have exceeded the latency SLO. In Figure 5.14c, the percentage of requests violated per second is plotted. This plot clearly shows that, especially in the first change in allocation to version V3, all the requests are out of time for multiple consecutive seconds. Figure 5.14d shows the results regarding the CPU utilisation for the inference server. Like the latency plot, there are very significant spikes in CPU utilisation around changes in version allocation.

Additionally, as expected, CPU utilisation shoots up with each increase in demand. The resource plot clearly shows how model allocation changes help reduce resource usage on the inference server. The changes in model allocation show that the scaling-down process skips model versions 2 and 4. This skip happens since the policy requires a minimal percentage of reduction in model metric. In this case, a minimal decrease of 40% in the inverse throughput metric. This metric decrease correlates to a reduced CPU resource usage for all models except the most miniature model V6 MobileNet as shown in Figure 5.8 research question 1.

Figure 5.15 shows the results for the trivial policy for two linearly decreasing stress tests. In plots 5.15a and 5.15b, the experiment starts from 51 requests per second and scales down with five requests every 30 seconds. In plots 5.15c and 5.15d, the experiment starts from 11 requests per second and scales down with three requests every 50 seconds. As seen in the experiment starting from 51 requests, no new model allocations happen until it reaches a demand of 6 requests per second. This allocation choice is sub-optimal, as when the model chain plotted in Figure 5.11 is considered, version 5 should be able to serve a demand of 11 requests per second with a CPU utilisation under 65%. This problem is due to the resource correlation irregularity of model version V6, MobileNet. When taking a step up, the trivial policy tries to approximate the new resource utilisation of each allocation change based on

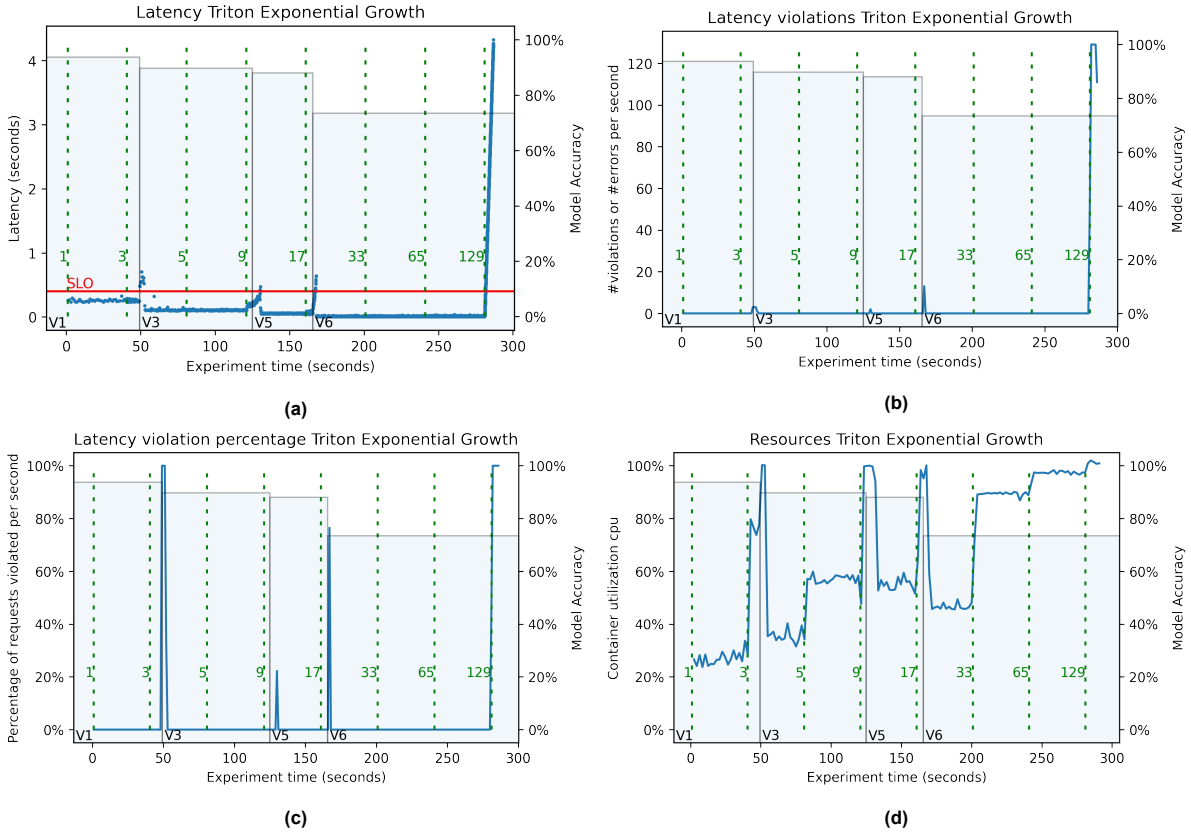


Figure 5.14: Trivial policy - single task - Exponential growth

the inverse throughput metric, current demand, and resource usage.

With the step-up approximation calculation, the policy uses a multiplier limit as discussed in section 4.4. One can define the policy resource multiplier limit. So, for example, 35% resource utilisation is used with version V6, with a demand of 11 requests per second. This usage would mean that if the policy had no multiplier limit, it would set a multiplier of $65\%/35\% = 1.857$, where 65% resource utilisation is the maximal goal of the new allocation. It would try to find a new allocation with a total inverse throughput metric that is as close as possible but lower than 1.857 times the inverse throughput metric of the current allocation. This decision makes the policy too conservative when stepping up from version V6. Thus the policy takes a multiplier limit, e.g. these results use a limit of 1.2, which ensures the policy only considers the actual resource usage up until this limit. In such a case, the system should notify its administrators, which means the expected throughput metric is no longer correct. This incorrectness could be due to resource competition with other processes on the target hardware, a model version with unexpected overhead in production, or the co-allocation of models with unexpectedly large resource overhead.

When comparing these results with static version allocation results, found in Figure 5.12 and Figure 5.13. It becomes apparent what kind of gains the proposed trivial policy achieves. When the demand is low, it can deploy more complex models with better model accuracy. When the demand is high, it can select more simple models and increase the maximal throughput of the system by trading off model accuracy. However, the static allocation of V1 can outperform the trivial policy in case the demand of the task never exceeds three requests per second. As in this case, the trivial policy would take a version step down, resulting in responses with sub-optimal model performance in model accuracy. In terms of latency violations, static allocation can also outperform the trivial policy, as changes in model allocation can lead to increased latency.

Figure 5.16 shows the Twitter trace stress test plots. There are a small number of latency violations when CPU utilisation spikes up to 100%. However, the trivial policy manages to handle the dynamic load nicely. It switches to smaller model versions when the demand gets high and switches back to the larger models when demand declines. A difficulty with this case is that the largest CIFAR10 model,

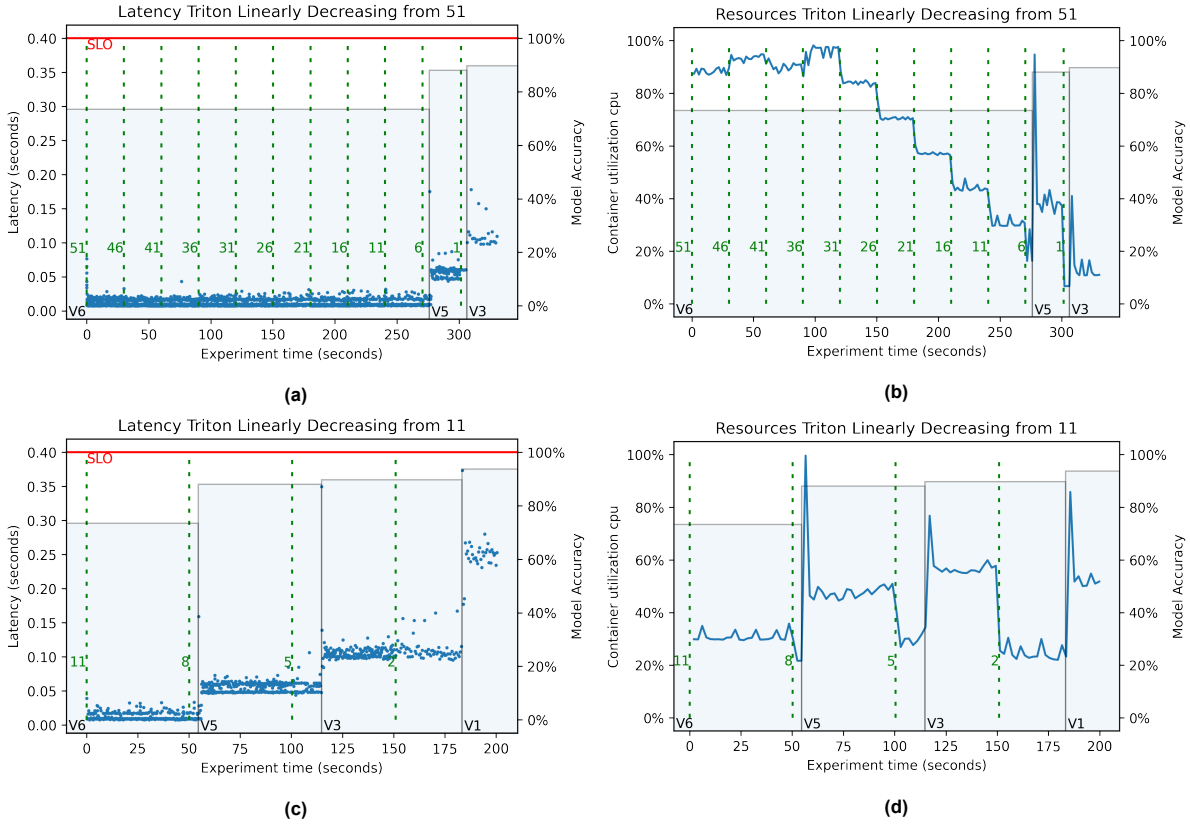


Figure 5.15: Trivial policy - single task - Linearly Decreasing

V1, can only handle a throughput of four on the hardware used. The orchestrator can switch to a smaller model in time, thanks to the gracious warm-up period of the stress-tester. When using a less gracious warm-up period, the largest model causes problems because CPU resources cannot handle the low demand and switching the model versions without causing an exploding queue. As discussed in section 4.4, the maximal metric policy discards the low-throughput models necessary for the following co-allocation test.

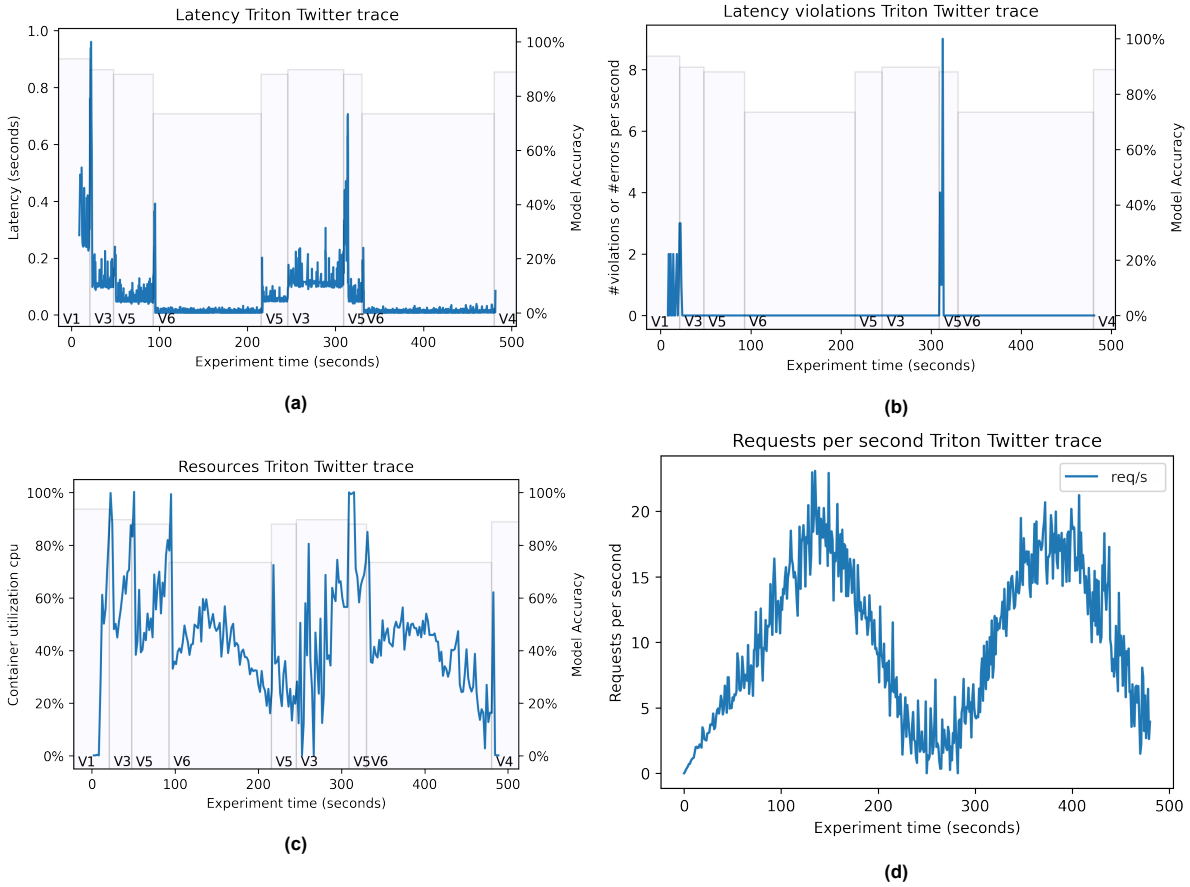


Figure 5.16: Trivial policy - single task - Twiter Trace

5.4.2. Co-allocation of tasks

With the co-allocation tests, CIFAR10, CIFAR100 and BeeAnts get simultaneous dynamic demand. In this experiment, a single triton inference server provides all models. Figure 5.17 shows the version chains used for this experiment. As discussed in the previous section, the following test uses the maximal metric policy for the CIFAR10 model with a value of $1/4 = 0.25$ for the inverse throughput metric. As a result, the orchestrator discarded the two model versions incapable of providing more throughput than four inferences per second under the upper CPU threshold of 70%. Figure 5.18 shows the experiment's latency, model switching, demand and resource results. The opaque green lines in the latency plots depict the demand.Figure 5.18f shows the systems latency violations and Figure 5.18e shows the CPU utilisation. The orchestrator achieved a mean CPU utilisation of around 57.5%, with 0.66% of the responses violating the latency SLO. Table 5.2 shows each task's overall model accuracy performance. Additionally, the normalised accuracy per task is shown. This metric would achieve 100 % when there are enough resources to serve all requests with the best and largest model V1. These values do not consider the responses that violated the latency SLO, meaning the actual accuracy performance is a fraction lower. Figure A.2 shows the result from another 10-minute experiment iteration. Here a mean CPU utilisation of 56% was achieved.

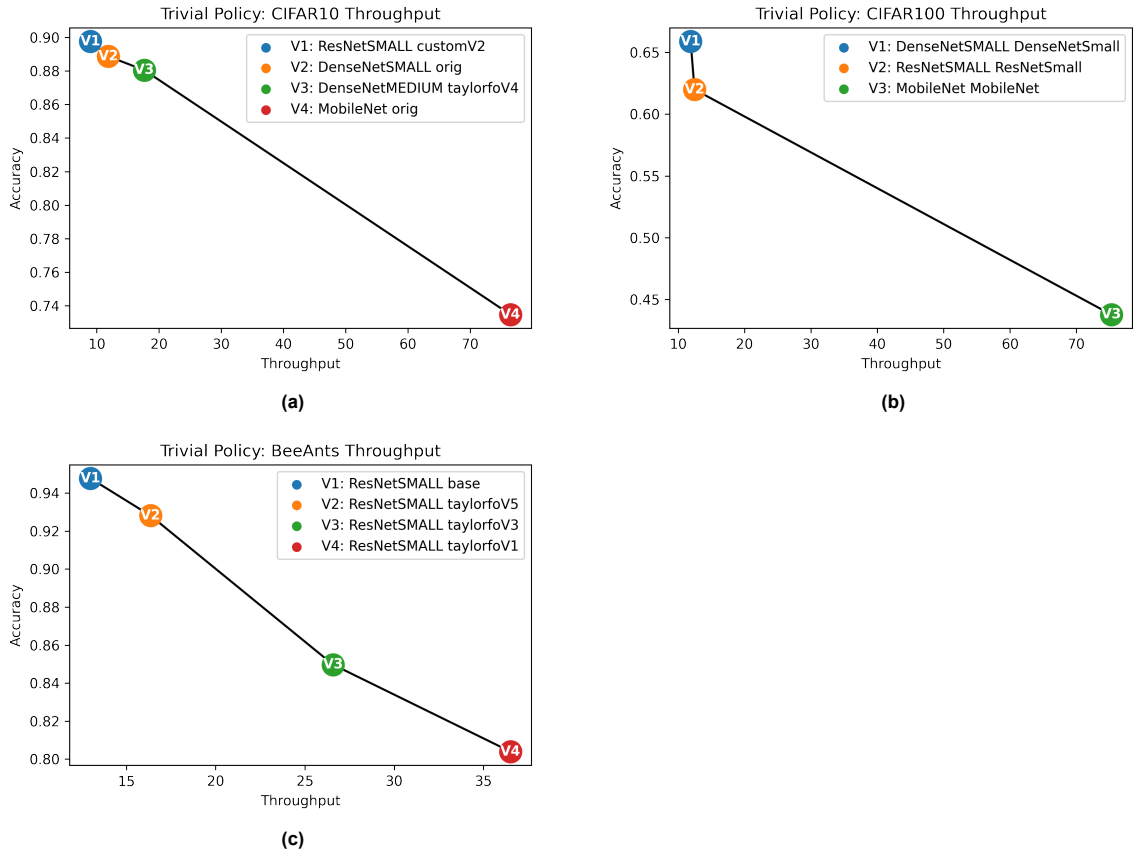


Figure 5.17: Trivial policy - Model version chains

Task	Model accuracy	Normalised accuracy
BeeAnts	81.47%	85.96%
CIFAR10	73.97%	78.91%
CIFAR100	44.52%	67.56%

Table 5.2: Coallocation test - Accuracy performance

These results show that co-allocating multiple tasks on one inference server does not have to be a problem. The orchestrator is capable of handling the dynamic demand following the trivial policy.

As a result, the overall accuracy of requests is higher than only serving the smallest model versions, and the system throughput is higher than only serving the largest model versions. However, there are latency SLO violations. Using more conservative parameters for the trivial policy can probably lower the number of violations. Another approach for decreasing the violations is by giving the inference server more CPU resources. Figure A.3 shows the results the system can achieve by allowing the inference server to use two CPU cores instead of one. This experiment doubled the demand just as the CPU resources did. It shows that model version switches have much less impact on CPU resource utilisation, thus causing fewer latency spikes. Figure A.4 shows the results of the system where the inference server has access to half a CPU. However, these results are not as comparable due to the resources not being limited to physical CPU threads but being limited by the operating system CPU scheduler across all available CPU resources. For example, this occasionally causes the utilisation in Figure A.4f to rise above 100% utilisation.

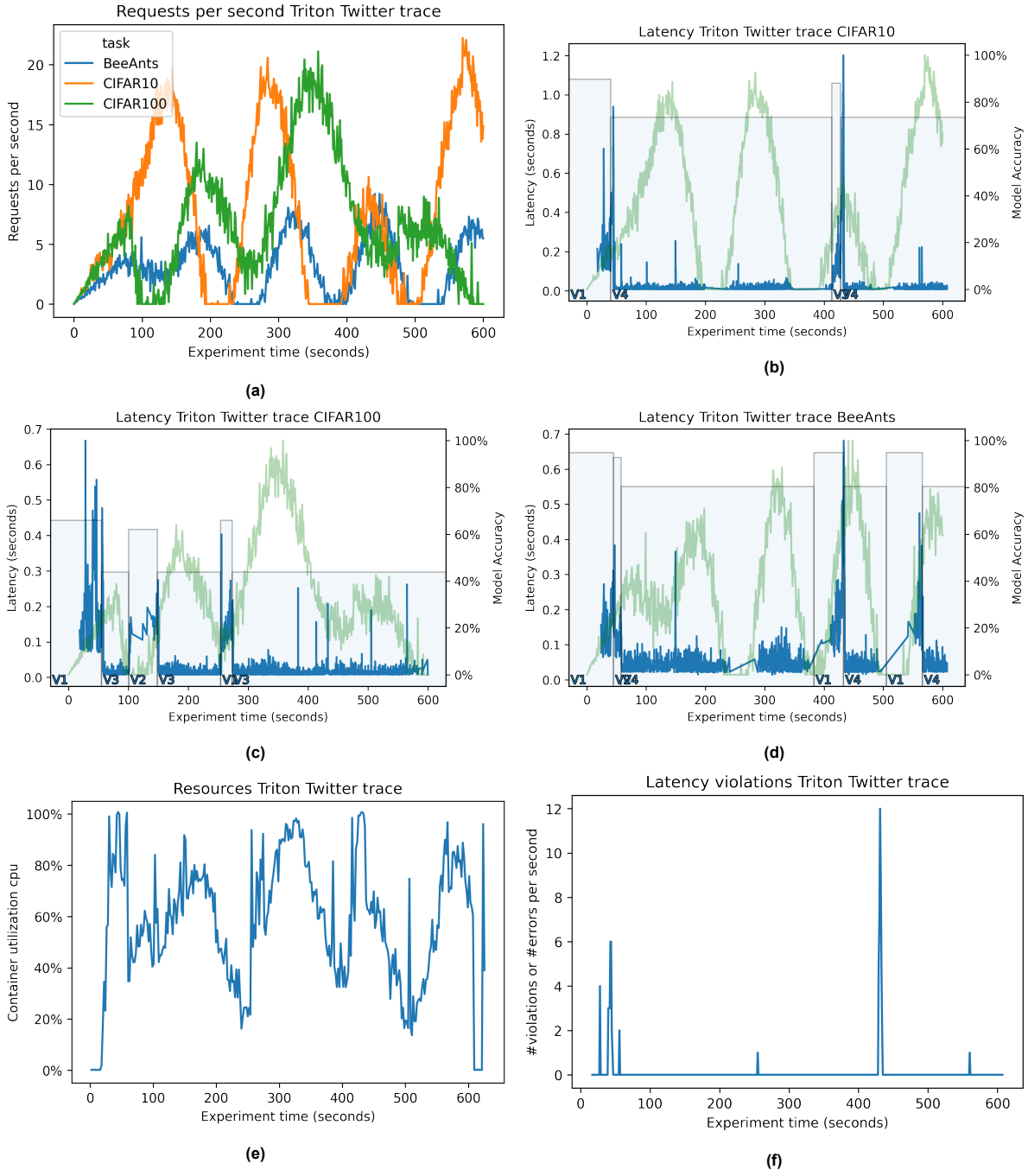


Figure 5.18: Trivial policy - Coallocation test - Twiter Trace

5.5. RQ3. Distributed Resource-Constrained Settings

How can DNN-porting be used to orchestrate model versioning in a distributed hardware-constrained setting?

Figure 5.19 shows how the trivial policy performs in a distributed setting. This specific setting divides the three tasks over three inference servers, where two inference servers serve each task. Figure 5.19f shows the legend for the inference servers used in the other plots. This experiment uses the same manipulated Twitter trace as in subsection 5.4.2. Like the plots in the previous research question, the opaque boxes indicate the accuracy performance at a given time. In the bottom left of the performance boxes, the number shows the version switch, and the colour of the number indicates the inference server. As two servers serve each task, the performance boxes overlap. To illustrate the difference in performance between the servers, they are distinguishable by the markers and colour. As expected, the results show similar model version switching behaviour. There is a notable difference in the latency SLO violations to research question two. In a one-server setup, the latency violations were more concentrated together. In these results, there tend to be more spread-out high latency violations. These unusually high latencies result from a technical limitation of the gRPC implementation combined with the NGINX load balancer. This has been validated by implementing HTTP requests instead of gRPC; the results are found in Figure A.5. This is most probably due to the long-lived connection nature of gRPC, which requires more sophisticated load balancing [4]. Unfortunately, these HTTP results cannot be used for comparison as there is a performance difference between the two protocols in the implementation of the Triton inference server.

Despite the latency violations by the technical issue, only 0.72% of the responses violated the latency SLO, a slight increase compared to the 0.66% violations of the single server co-allocation experiment. Table 5.3 shows the mean CPU usage of each inference server under the low load results. They all dip under the policy's lower CPU threshold of 50%. These are relatively low compared to research question 2. In Table 5.4 shows the accuracy and normalised accuracy performance under the low load experiment.

	Mean CPU	
Instance name	**Low load**	**High Load**
k3s-rpi1_triton_1	43.53%	63.32%
k3s-rpi1_triton_2	34.52%	52.50%
k3s-rpi2_triton_1	41.71%	57.51%

Table 5.3: Mean resource utilisation per server

The intuition for the next experiment is that compared to the results in subsection 5.4.2, this setup has three times the amount of CPU resources, as it uses three inference servers instead of one. One would expect similar results if the demand is also three times as high, showing the scalability of the setup.

Figure 5.20 shows the same experiment results for the same Twitter trace, but the demand is three times as high. Table 5.3 shows that the CPU utilisation has become more comparable with the mean utilisation from the results in the co-allocation experiment, which was 56%. The achieved task performance is also very similar when comparing the high load with the research question 2 results in Table 5.4. Notable is the high latency violations spike in Figure 5.20g, which is a result of the k3s-rpi1_triton_1 and k3s-rpi2_triton_1 switching model versions for CIFAR10 at the same time. Figure 5.20e shows that both servers experience resource scarcity at this point, as indicated by 100% resource utilisation for multiple seconds. This resource scarcity consequently leads to high latency violations comparable to the high latency spikes in the single server results when version switches occur with resource scarcity.

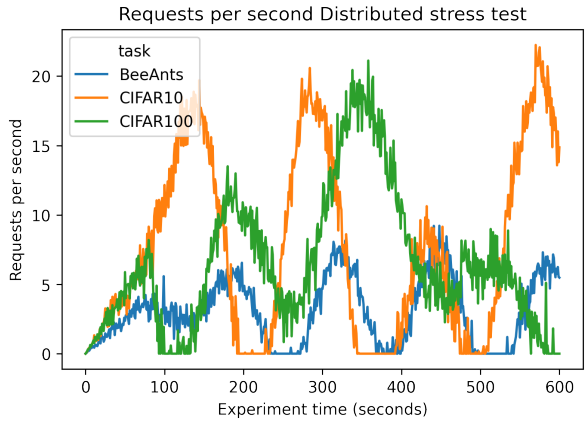
These experiments show that there is still a problem with latency SLO violations when version switches are performed simultaneously. For real use cases, the percentage of violating responses will probably decrease because the macro trends of the demand tend to spread over a longer time and are less explosive, as seen in Figure 5.2a. As a result, the orchestrator would have to perform fewer version switches. A promising way to eliminate these remaining latency violations in the distributed setting is by implementing coordination between the load balancer and the orchestrators. By ensuring

[4] `https://kubernetes.io/blog/2018/11/07/grpc-loadbalancing-on-kubernetes-without-tears/`

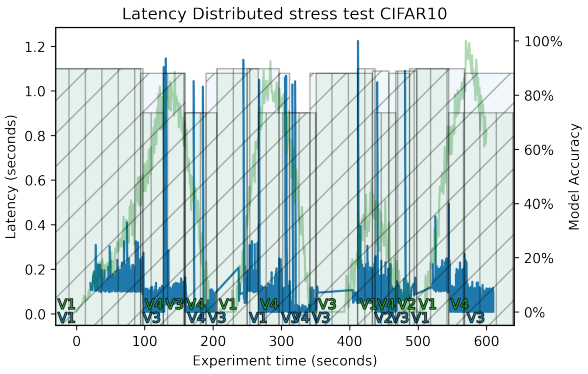
	Low Load		High Load		RQ2 Co-allocation	
Task	**Model accuracy**	**Normalised accuracy**	**Model accuracy**	**Normalised accuracy**	**Model accuracy**	**Normalised accuracy**
BeeAnts	92.01%	97.09%	82.25%	86.79%	81.47%	85.96%
CIFAR10	80.90%	86.31%	74.23%	79.19%	73.97%	78.91%
CIFAR100	56.57%	85.86%	44.73%	67.89%	44.52%	67.56%

Table 5.4: Distributed test - Accuracy performance

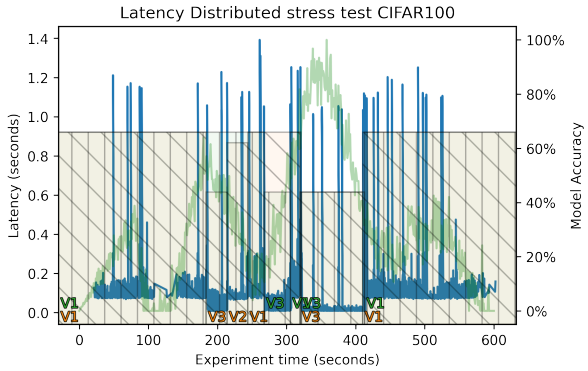
no two servers with overlapping tasks switch model versions simultaneously and flagging the server that will perform a model version switch, the load balancer can send fewer requests to the flagged server. Such coordination is probably easier achieved by utilising one dedicated orchestrator for all inference servers, like the ModelMesh integration design found in Figure 4.5.



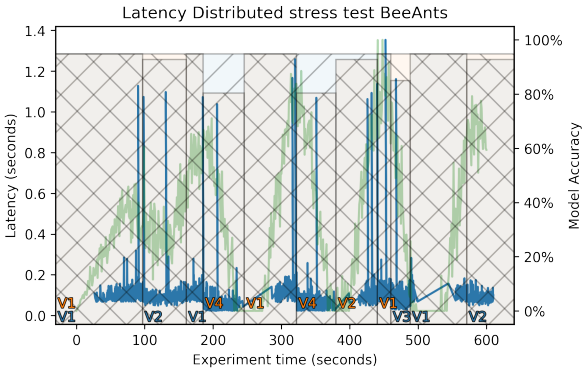
(a)



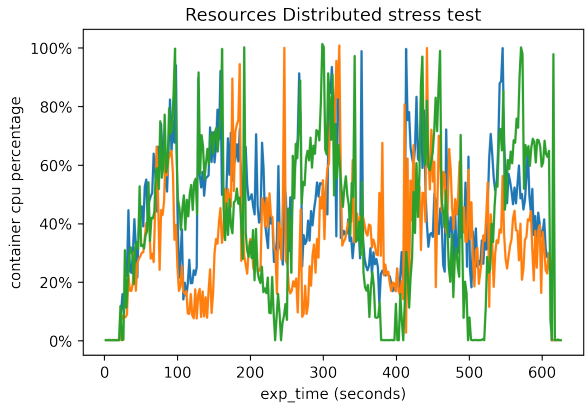
(b)



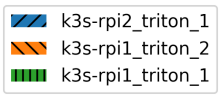
(c)



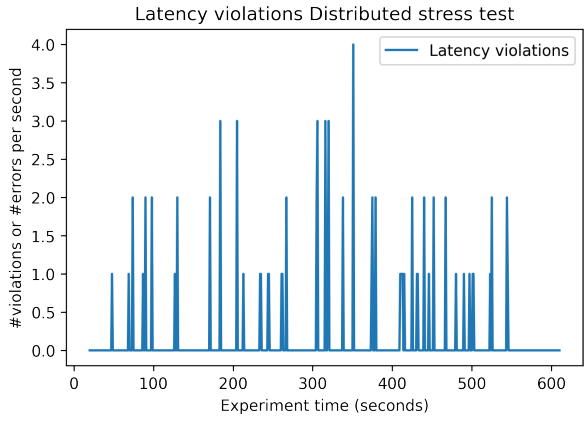
(d)



(e)



(f) Server instance legend



(g)

Figure 5.19: Trivial policy (with a maximal metric for CIFAR10) - Distributed test low load - Twiter Trace

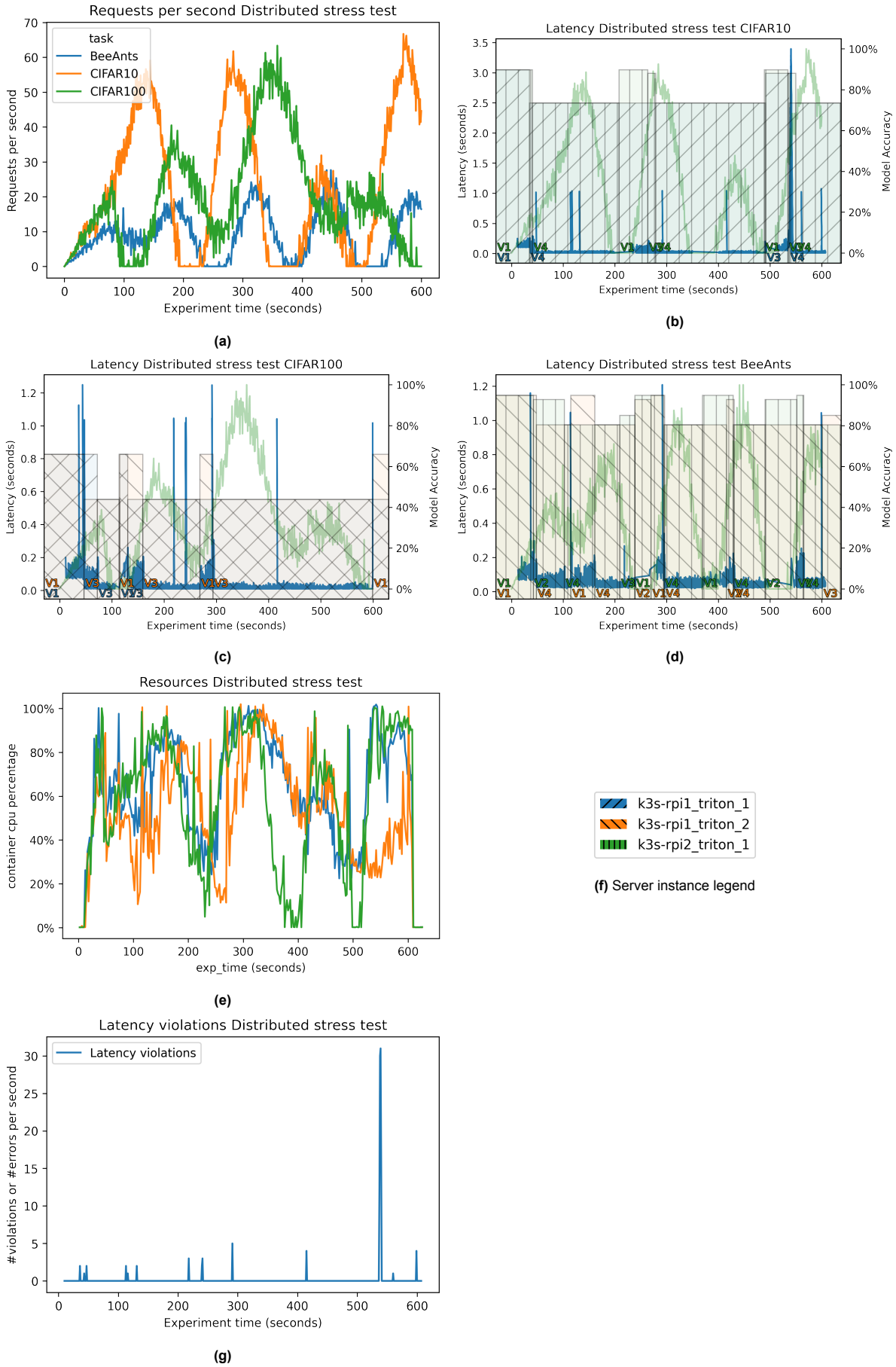


Figure 5.20: Trivial policy (with a maximal metric for CIFAR10) - Distributed test high load - Twiter Trace replicate research question 2, demand times 3

6 Discussion

This thesis showed how DNN-porting techniques, such as structured pruning, can maximise hardware utilisation in hardware-constrained settings like the Edge. The DNN-porting module discussed in chapter 3 can generate models with varying accuracy and resource characteristics based on a trained base model and data module using structured pruning. The system gets its trade-off capabilities between accuracy and resource usage thanks to the generated landscape of model variations. The system design proposed in chapter 4 considers the shortcomings of the state-of-the-art in combination with promising upcoming open-source ML inference platforms.

In research question 1, the metrics FLOPs, inference latency and maximal inference throughput are compared to quantify the resource needs of a model version. The experiments consider the other two metrics as the maximal inference throughput metric brings the complexity of profiling each model version on its target hardware. Unfortunately, the results showed suboptimal Pareto frontiers to the frontier found in the maximal inference throughput domain. The results showed an apparent correlation between the throughput metric and CPU utilisation, except for the case of most miniature models, such as MobileNet. With these smallest models, the server overhead becomes more prominent as the number of requests grows considerably.

Moreover, the results showed the extra resource overhead in model co-allocation cases. Considering these limitations within the decision-making policy and CPU usage, research question 2 showed how the trivial policy proposed in section 4.4 trade-offs accuracy with resources when demand changes. The results clearly show the throughput and accuracy benefits over static allocation policies, which only serve the largest or smallest model version. A problem that became apparent with this thesis model switching approach is the latency violations when model switches occur, especially when CPU resources are scarce. In most experiments, the inference server hardware only had one CPU core. Thus, an important ability of the system is to prevent the detrimental exploding queue problem after model switches by considering CPU resource scarcity.

Luckily, the distributed system design can partly tackle the latency SLO problem, as shown in the results of research question 3. The large latency spikes occur only when model version switches happen simultaneously on multiple servers. Due to the benefit of using the load balancer, when a server is busy performing a model version switch, the incoming requests are routed to another inference server until the queued requests are processed. As discussed, coordination between the orchestrators and load-balancer could probably avoid these remaining violations. Nevertheless, the total latency violations are comparable due to the technical limitation with gRPC requests and the load balancer.

Research question 3 shows the system's scalability in a distributed setting with three inference servers concerning achieved accuracy for lower and higher demands. Table 5.4 shows the accuracy results between the distributed high load setting and the single server research question 2 co-allocation results. The high-load experiment got three times the demand of the single server co-allocation test, as it had three times the amount of CPU resources. The distributed setting achieved similar accuracy and CPU utilisation results.

7

Conclusion & Future Work

7.1. Conclusion

To conclude, this thesis showed the flexibility and possibilities DNN-porting techniques can bring to maximising hardware utilisation in CPU hardware-constrained settings by utilising the trade-off characteristics between accuracy and resource usage. It can provide high-accuracy RT inference responses when resources are abundant and can scale with growing demand by providing lower-accuracy RT responses. This trade-off capability was one of the three challenges that deserved community attention following Dagstuhl's perspective in subsection 2.1.2 [8]. Additionally, the results showed that one could add distributed resources by adding extra inference servers to improve the system's accuracy across tasks.

With the evaluation results in subsection 5.2.3, the limitations of INFaaS caused by their design decisions were clear. Resulting in problems in the performance of their system and the problems with the priority of their policy [26]. Compared to the model-switching paper, this thesis shows the feasibility of not always keeping all model versions in memory and serving multiple models from one container. Further, a decision policy based on live resource metrics for model version-switching decisions solves their system's fixed computing resource challenge. All in all, the trade of capabilities of models generated by DNN-porting techniques can bring significant resource flexibility to hardware-constrained settings like the Edge, resulting in higher and balanced CPU utilisation.

7.2. Future work

As shown in section 4.3, the system used for this thesis can be integrated into promising open-source inference platforms enabling adaptation of the ideas and utilising the benefits in industry-used systems. The implementation within an established open-source inference platform would be an important future work direction for enabling industry adaptation and providing the scientific community with an accessible scalable testbed platform for future research. Such a system would enable research on different orchestrator policies for different use cases. For example, the use of RL in the decision-making process, or as earlier discussed, the reduction of latency by using collaboration between the orchestrator and load-balancer of the system. It would be interesting to complicate this system by adding non-RT requests, requiring to make trade-offs between task performance between the RT and non-RT requests in hardware-scarce situations. This addition might lead to the need for a priority request-based approach, where just like INFaaS, the requestor has to direct the system which trade-offs it will need to make.

The DNN-porting part of the thesis can be extended with other porting techniques, e.g. quantisation, hardware-specific optimisation, and knowledge distillation. Combining all these techniques can result in shorter overall training time and a broader model version landscape, allowing for more fine-tuned version decisions. Additionally, INFaaS's research direction of utilising heterogeneous hardware clusters can enable even better trade-off characteristics. Another exciting research direction is adapting a single model version with inherent different resource versions. These models could provide results by only passing through a portion of the layers when resources are scarce and utilising more and deeper layers when resources are abundant. However, the inference runtime servers must support running such models. Another useful direction would be to continue searching for a resource usage metric

independent of the target hardware. It would dramatically reduce the complexity of using this system in heterogeneous clusters and allow the scientific community to adequately quantify their NN architecture resource needs. It would be interesting to see the shortcomings of the broadly used FLOP metric regarding specific target hardware inference and whether it can be improved. As this thesis focused on CPU-based inferences, a good direction would be evaluating the system on different hardware, e.g. GPU.

References

[1] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. “Structured pruning of deep convolutional neural networks”. In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13.3 (2017), pp. 1–18.

[2] Simone Bianco et al. “Benchmark analysis of representative deep neural network architectures”. In: *IEEE access* 6 (2018), pp. 64270–64277.

[3] Davis Blalock et al. “What is the state of neural network pruning?” In: *Proceedings of machine learning and systems* 2 (2020), pp. 129–146.

[4] Shraman Ray Chaudhuri et al. “Fine-Grained Stochastic Architecture Search”. In: *ICLR Workshop on Neural Architecture Search*. 2020. URL: https://arxiv.org/pdf/2006.09581.pdf.

[5] Daniel Crankshaw et al. “Clipper: A Low-Latency Online Prediction Serving System”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 613–627. URL: https://github.com/ucbrise/clipper.

[6] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[7] Li Deng. “The mnist database of handwritten digit images for machine learning research”. In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[8] Aaron Yi Ding et al. “Roadmap for Edge AI: A Dagstuhl Perspective”. In: *SIGCOMM Comput. Commun. Rev.* 52.1 (Mar. 2022), pp. 28–33. ISSN: 0146-4833. DOI: 10.1145/3523230.3523235. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3523230.3523235.

[9] Philippe Dobbelaere and Kyumars Sheykh Esmaili. “Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper”. In: *Proceedings of the 11th ACM international conference on distributed and event-based systems*. 2017, pp. 227–238.

[10] Zhou Fang, Dezhi Hong, and Rajesh K Gupta. “Serving deep neural networks at the cloud edge for vision applications on mobile platforms”. In: *Proceedings of the 10th ACM Multimedia Systems Conference*. 2019, pp. 36–47.

[11] Julien Gedeon et al. “What the fog? edge computing revisited: Promises, applications and future challenges”. In: *IEEE Access* 7 (2019), pp. 152847–152878.

[12] Amir Gholami et al. “A survey of quantization methods for efficient neural network inference”. In: *arXiv preprint arXiv:2103.13630* (2021).

[13] Peizhen Guo, Bo Hu, and Wenjun Hu. “Mistify: Automating DNN Model Porting for On-Device Inference at the Edge”. In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 2021, pp. 705–719.

[14] Matthew Halpern et al. “One size does not fit all: Quantifying and exposing the accuracy-latency trade-off in machine learning cloud service apis via tolerance tiers”. In: *arXiv preprint arXiv:1906.11307* (2019).

[15] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[16] Andrew G. Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[17] Gao Huang et al. “Densely connected convolutional networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[18] Henning Kagermann. “Change through digitization—Value creation in the age of Industry 4.0”. In: *Management of permanent change*. Springer, 2014, pp. 23–45.

[19] Alex Krizhevsky, Geoffrey Hinton, et al. “Learning multiple layers of features from tiny images”. In: (2009).

[20] KServe. *KServe*. Version 0.9.0. July 2022. URL: https://github.com/kserve/kserve.

[21] Marko Luksa. *Kubernetes in action*. Simon and Schuster, 2017.

[22] Microsoft. *Neural Network Intelligence*. Version 2.7. Apr. 2022. URL: https://github.com/microsoft/nni.

[23] *Nvidia TensorRT*. Feb. 2023. URL: https://developer.nvidia.com/tensorrt.

[24] Rydning Reinsel Gantz. *Data Age 2025: The Evolution of Data to Life-Critical*. Apr. 2017. URL: https://www.seagate.com/files/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf.

[25] Rydning Reinsel Gantz. *The Digitization of the World: From Edge to Core*. Nov. 2018. URL: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

[26] Francisco Romero et al. “INFaaS: Automated Model-less Inference Serving”. In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 397–411.

[27] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[28] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[29] Seldon. *Seldon Core*. Version 1.14.0. June 2022. URL: https://github.com/SeldonIO/seldon-core/.

[30] Alexander Sigov et al. “Emerging enabling technologies for industry 4.0 and beyond”. In: *Information Systems Frontiers* (2022), pp. 1–11.

[31] *Twitter Streaming Traces*. 2018. URL: https://archive.org/details/archiveteam-twitter-stream-2018-04.

[32] Aston Zhang et al. “Dive into deep learning”. In: *arXiv preprint arXiv:2106.11342* (2021). URL: https://d2l.ai/index.html.

[33] Jeff Zhang et al. “Model-Switching: Dealing with Fluctuating Workloads in Machine-Learning-as-a-Service Systems”. In: *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. 2020.

A

Figures

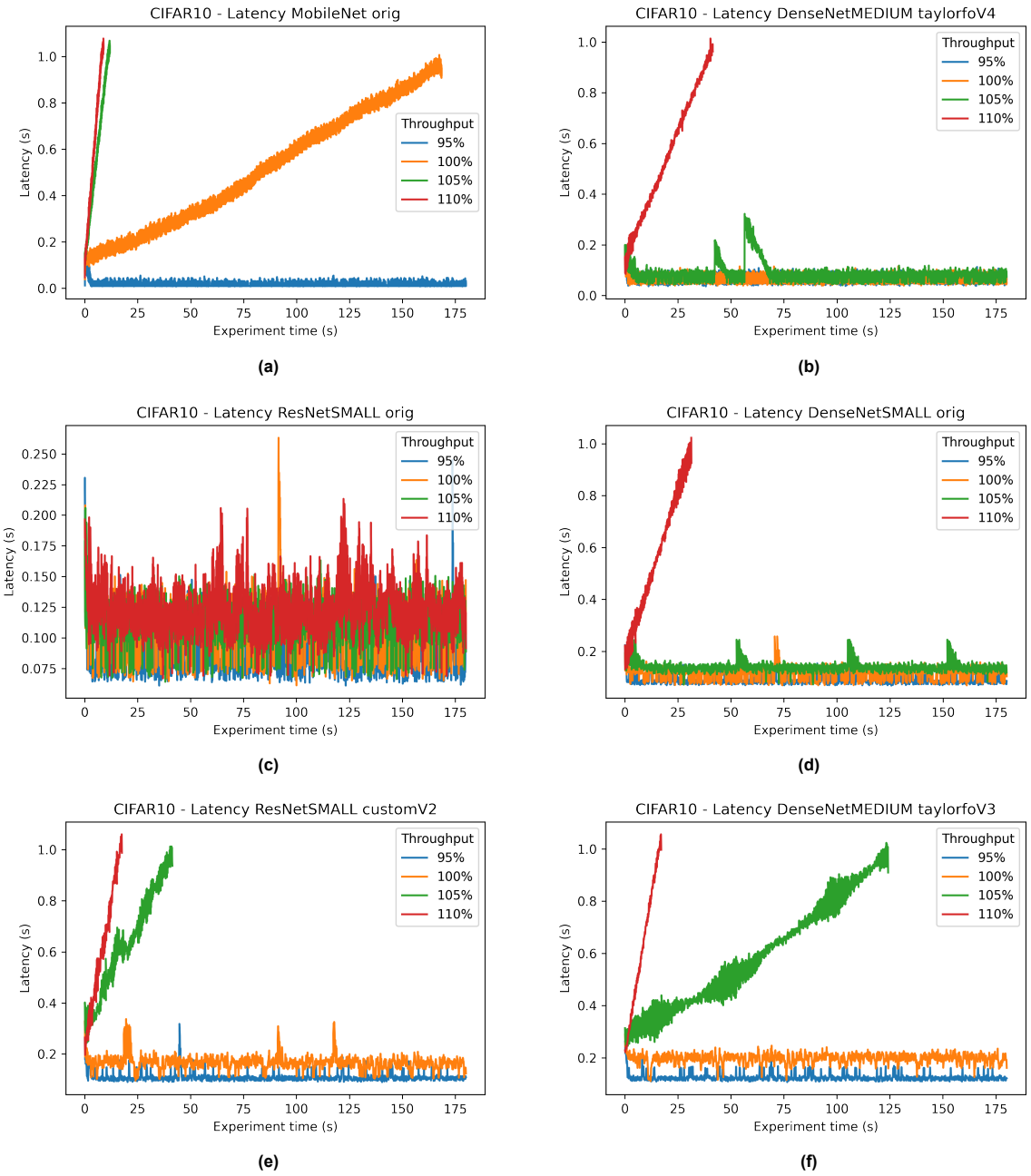
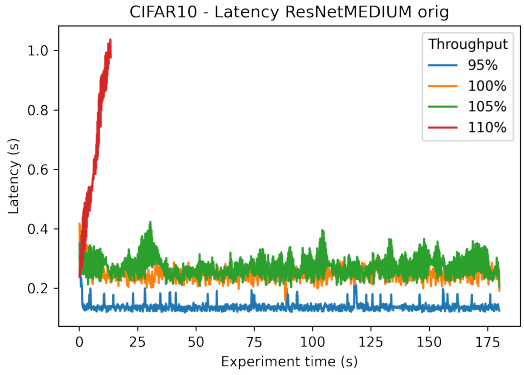
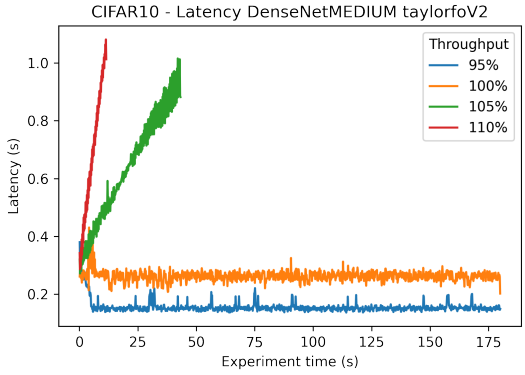


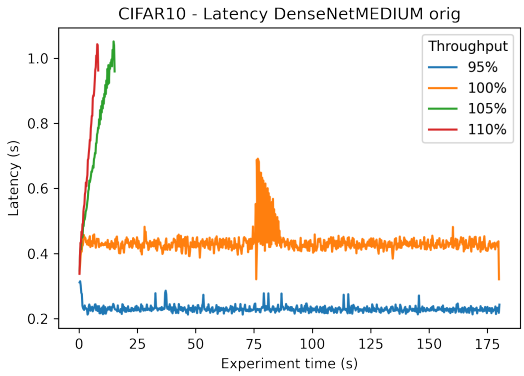
Figure A.1: CIFAR10 - Latency results for varying percentages of corresponding maximal throughput



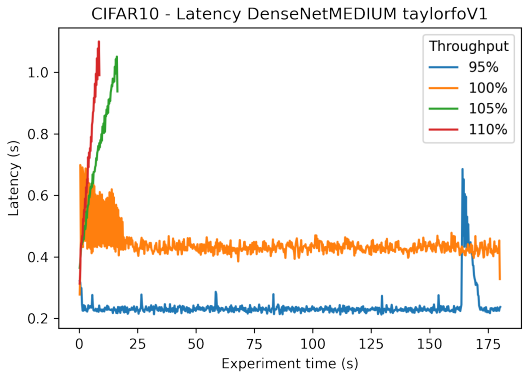
(g)



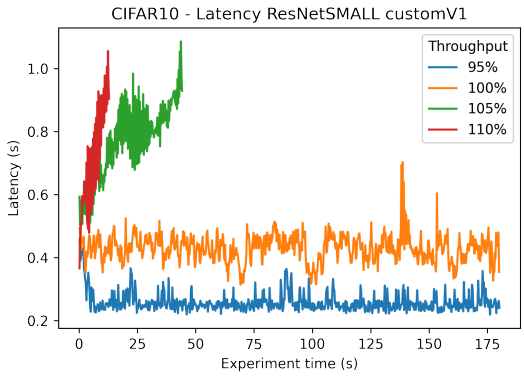
(h)



(i)



(j)



(k)

Figure A.1: CIFAR10 - Latency results for varying percentages of corresponding maximal throughput (continued)

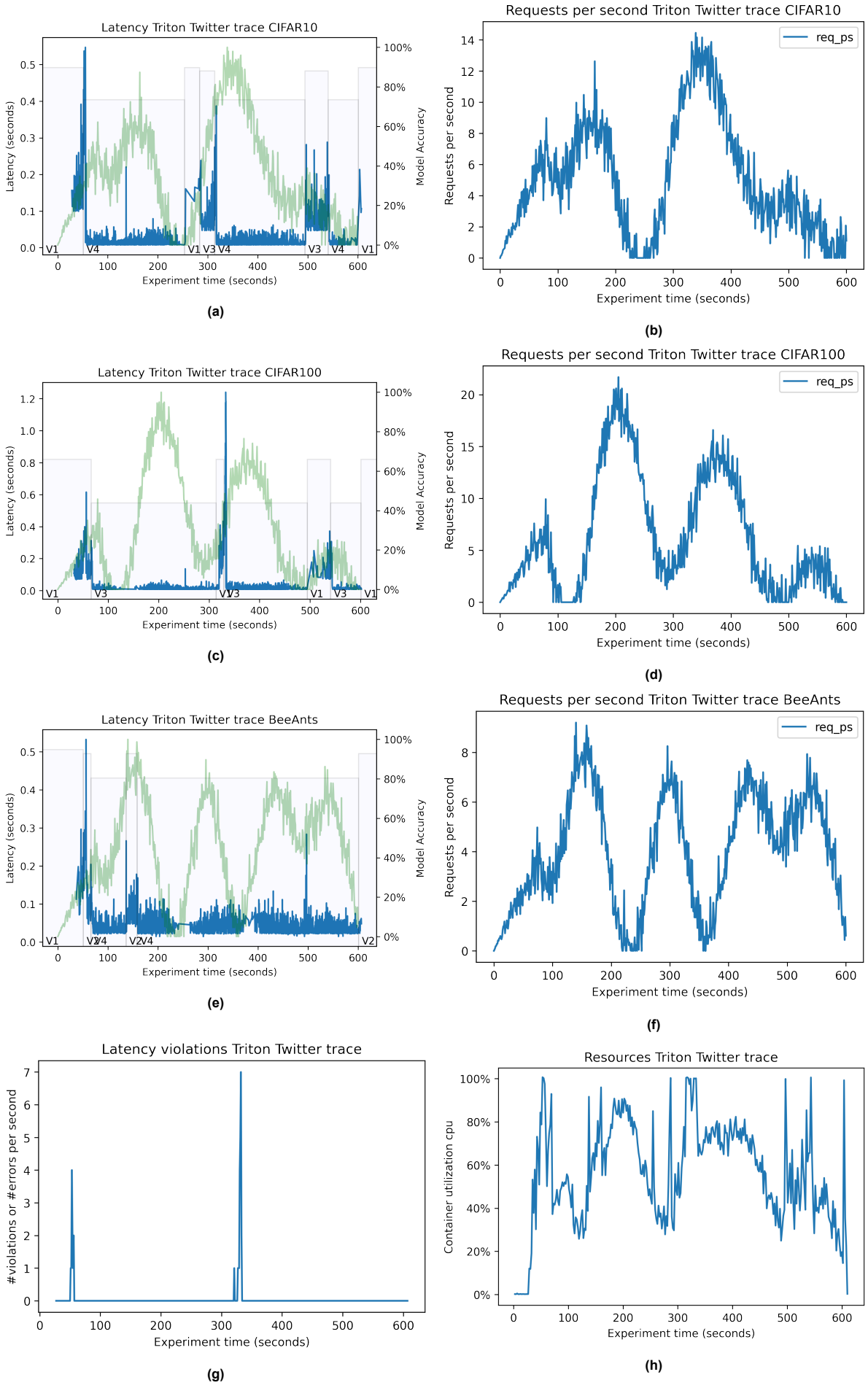


Figure A.2: Trivial policy (with a maximal metric for CIFAR10) - Coallocation test - Twiter Trace

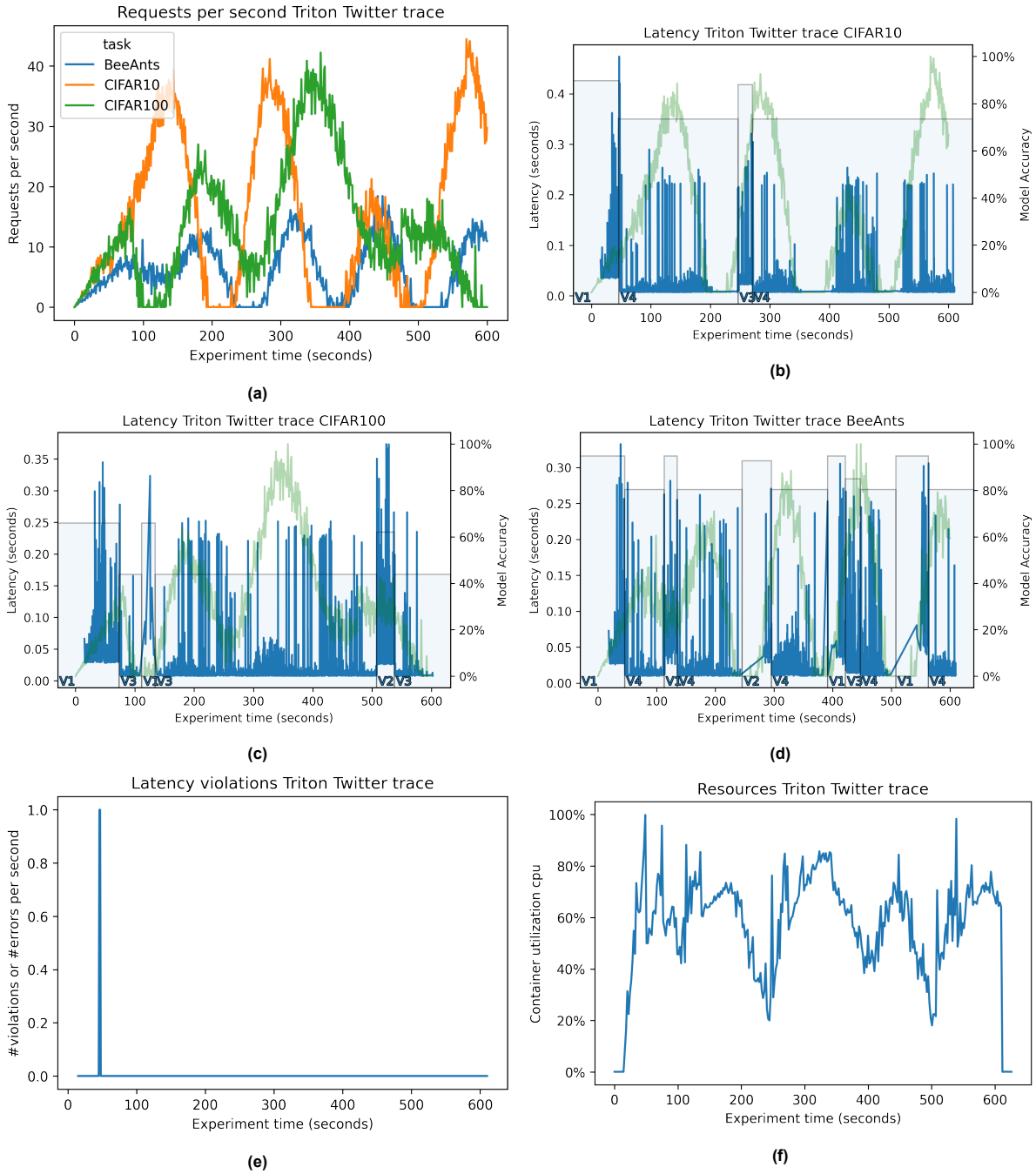


Figure A.3: Trivial policy (with a maximal metric for CIFAR10) - Coallocation test with cpus=2 - Twiter Trace

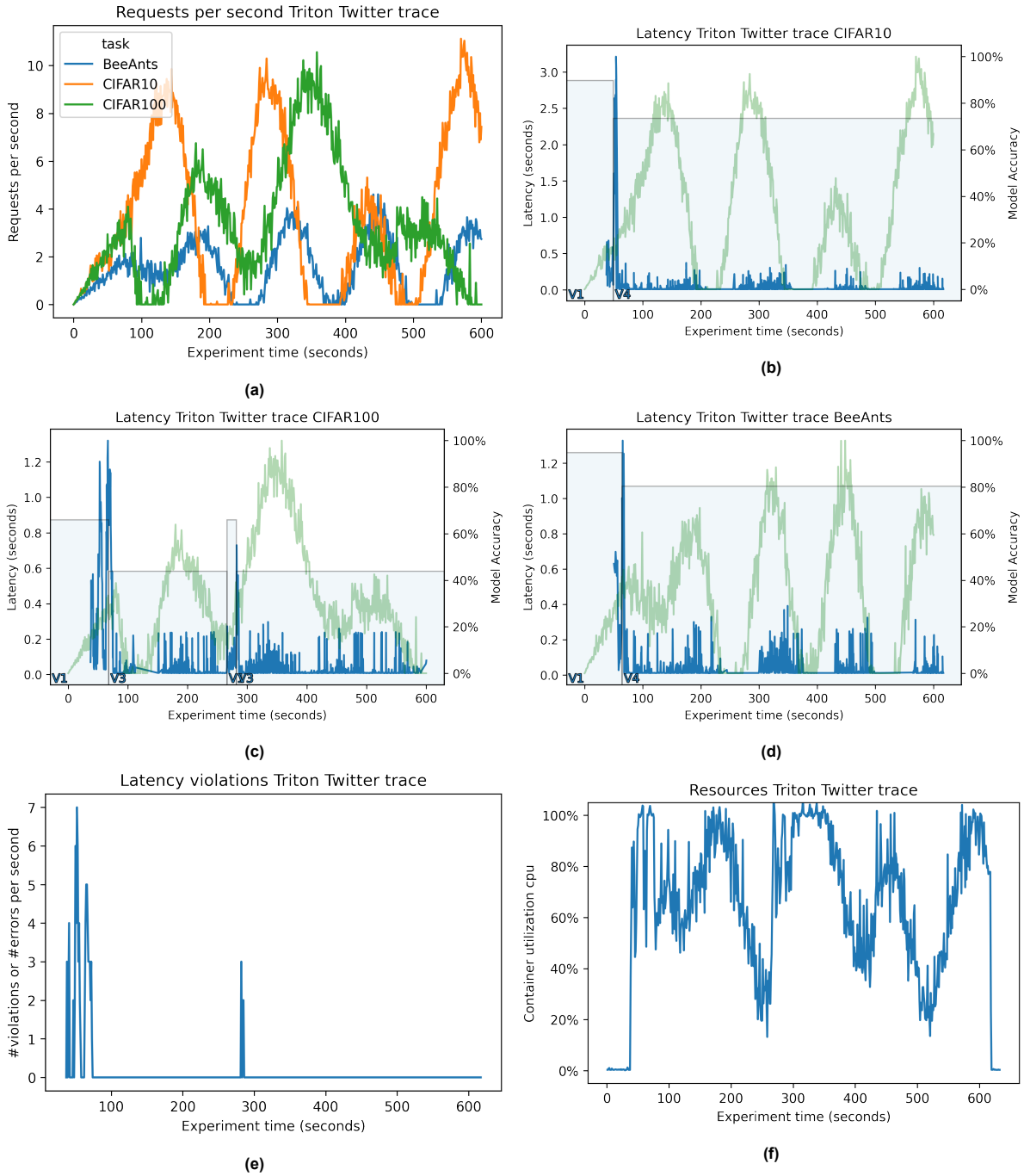


Figure A.4: Trivial policy (with a maximal metric for CIFAR10) - Coallocation test with cpus=0.5 - Twiter Trace

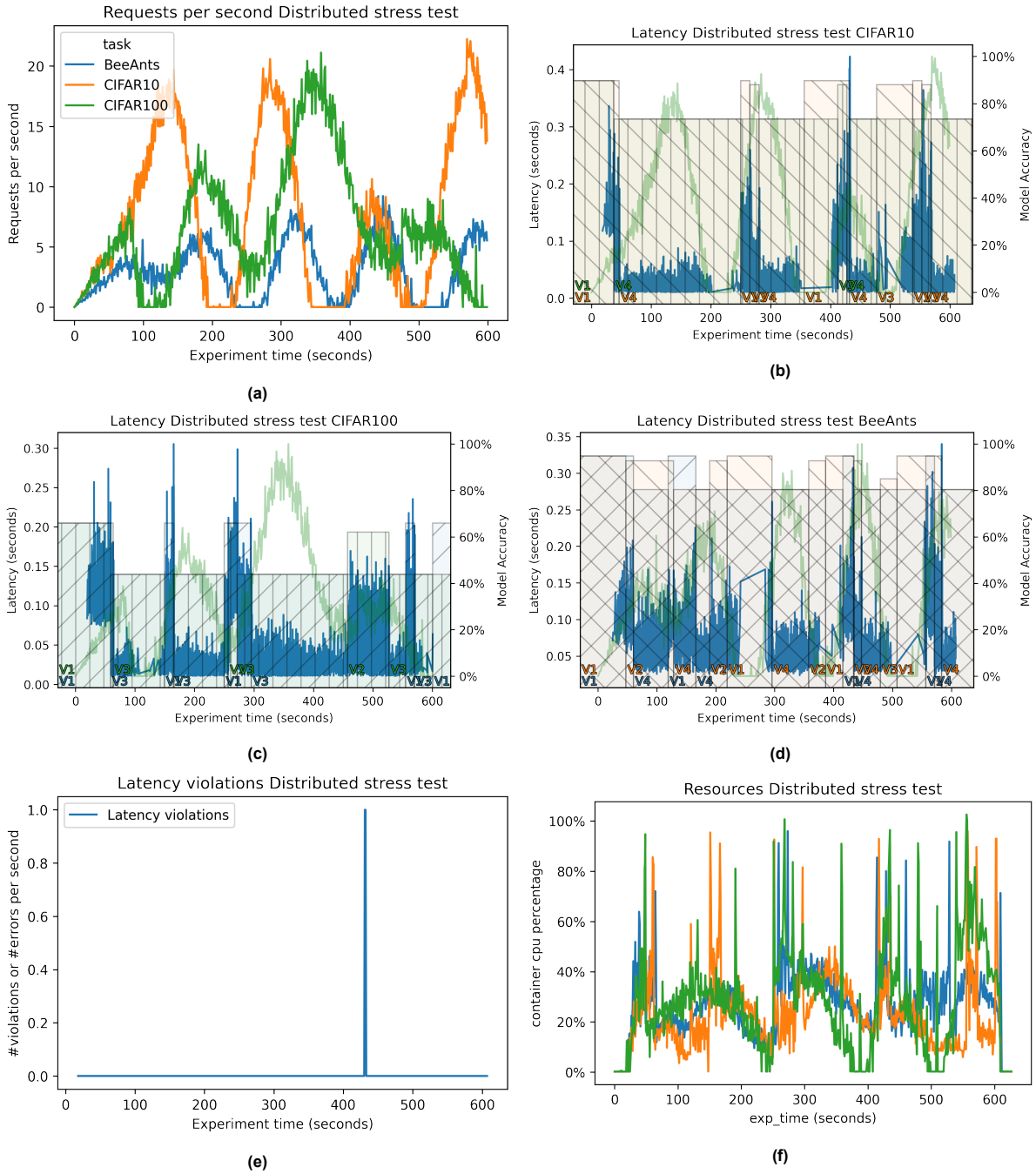


Figure A.5: Trivial policy (with a maximal metric for CIFAR10) - RQ3 with HTTP instead of gRPC

Acronyms

AI Artificial intelligence.
AIMET AI Model Efficiency Toolkit.

CUDA Compute Unified Device Architecture.

DL Deep Learning.
DNN Deep Neural Networks.

FL Federated Learning.
FLOP Floating Point Operation.
FLOPS Floating Point Operations per Second.
FP32 32-bit Floating Point.

IDC International Data Corporation.
IIoT Industrial Internet of Things.
ILP Integer Linear Programming.

LRU Least Recently Used.

MAC Multiply-Accumulate operation.
ML Machine Learning.
MLaaS Machine Learning as a Service.

NN Neural Network.
NNI Neural Network Intelligence.

ONNX Open Neural Network Exchange.

PL PytorchLightning.

RL Reinforcement Learning.
RT Real-Time.

SGD Stochastic Gradient Descent.
SLO Service Level Objective.