A Joint Model Provisioning and Request Dispatch Solution for Mobile Inference Serving at the Edge

Anish Nagendra Prasad

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2021

Committee:

Yang Peng

Munehiro Fukuda

William Erdly

Program Authorized to Offer Degree:
Computing and Software Systems

University of Washington

Abstract

A Joint Model Provisioning and Request Dispatch Solution for Mobile Inference Serving at the Edge

Anish Nagendra Prasad

Chair of the Supervisory Committee:
Assistant Professor Yang Peng
Computing & Software Systems

With the advancement of machine learning (ML), a growing number of mobile clients rely on ML inference for making time-sensitive and safety-critical decisions. Therefore, the demand for high-quality and low-latency inference services at the network edge has become the key to the modern intelligent society. This thesis proposes a novel solution that jointly provisions inference models and dispatches inference requests for reducing the latency of mobile inference serving on edge nodes. Unlike existing solutions that either direct inference requests to the nearest edge node or balance the workload between edge nodes, the solution we propose provisions each edge node with the optimal type and the number of inference serving instances under a holistic consideration of networking, computing, and memory resources. Mobile clients can thus utilize ML inference services on edge nodes that offer minimal inference serving latency.

In this work, we implement the proposed solution using TensorFlow Serving and Kubernetes on a cluster of edge nodes, including Nvidia Jetson Nano and Jetson Xavier. We further demonstrate the proposed solution's effectiveness in reducing the overall inference latency under various system parameters and practical system settings through simulation and testbed experiments, respectively.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to firstly thank my advisor and committee chair, Dr. Yang Peng, for his guidance during this work. Without his support and genuine interest, this work would not have been possible.

Secondly, I extend a word of thanks to my committee members, Dr. Munehiro Fukuda and Dr. Bill Erdly. They offered much-needed advice and asked all the questions that needed to be asked.

A further thank you to the CSS department for their academic and financial support during this process.

Last, but by no means least, I would like to thank my family for their love and support over the years and for creating an environment in which I could succeed. Without them, I would not be here today.

Chapter 1

INTRODUCTION

Machine learning is an innovation that is driving technological advancement. As new solutions are built that tackle societal issues and make life better for everyone, the intelligent processing or inference that machine learning provides will be instrumental. Traditionally, machine learning inference is carried out in large cloud data centers with hundreds of server blades or expensive on-site machines. These machines offer acceptable performance for traditional mobile applications that use ML [8]. However, the world is moving toward building smaller and more mobile devices. For example, a self-driving vehicle might want to analyze some road data to plan its route or avoid hazardous road conditions in real-time, or a person with disabilities may rely on smart cognitive devices to help them avoid obstacles that they might not be able to navigate. For these emerging applications, machine learning inference will need to be carried out quickly and in close proximity to the mobile clients [9]. This new operating paradigm has given rise to the idea of mobile inference serving or running inference at the network edge.

Autonomous vehicles or other powerful mobile devices might contain all the hardware needed to perform high-computing inference tasks. However, less powerful mobile devices such as smartphones might need to offload more demanding workloads to the cloud in traditional solutions. In this case, edge nodes equipped with GPU cards can be deployed in common locations, such as high traffic public spaces and cellular towers [15]. Throughout this paper, we will use the term "edge nodes" to refer to these smaller GPU-equipped devices deployed at the network edge. When needed machine learning models are installed on these edge nodes, mobile clients can obtain inference results with shorter communication delays.

In addition, these pervasive edge nodes bring the benefit of serving inference requests in relatively close proximity and minimizing response time, a critical requirement for safety or latency-sensitive applications. When deployed correctly, these devices offer performance comparable to or even more significant than cloud-based solutions [13].

The average inference time of many machine learning models is relatively low. This makes it possible to deploy these models onto edge nodes and still get a result in a reasonable amount of time. For example, the Mobilenet Convolutional Neural Network's average inference time for a single image is approximately 27.4 milliseconds on edge representative hardware [23]. Other researchers have shown that neural networks such as this one can be executed quickly with fast results. For example, running CNNs on Jetson TX2 devices for 100 images yielded a time of around 1.35 seconds, with each image taking around just six milliseconds [31]. The increasing prevalence of powerful yet small hardware has led to the introduction of models that allow for rapid inference at the edge, necessitating novel provisioning solutions to take full advantage of the opportunity.

Existing solutions leveraging edge nodes for inference serving approach provisioning primarily from two different directions, and each has its benefits and drawbacks. From the edge cluster's perspective, some solutions focus on balancing the performance of the edge servers by distributing similar workloads across multiple nodes so that queuing delay for resources can be reduced across edge servers. However, these solutions may affect the serving latency experienced by some mobile clients. On the other hand, some existing solutions prefer to select nodes for the client based on the client's physical proximity to the server and simply assume that the closest edge server will have the least latency. It is, however, entirely possible that the edge node physically closest to a mobile client may not actually be the edge node that will respond with minimal latency. Multiple factors, such as the current deployments on the edge nodes, the memory utilization, and the varying quality of the network link, can all affect the overall latency. This makes relying on location alone a flawed method.

To address the lack of an appropriate solution, this thesis presents a joint model provisioning and request dispatch solution for serving mobile inference requests at the network

edge, a novel scheme for managing the deployment and balancing of inference execution on distributed edge nodes with a focus on minimizing latency. The key goal of this work was to develop a solution that could not only direct a client to use the best inference node but also provision a node with the most desired inference resources to reduce the inference latency and resource consumption. Attempting to perform inference provisioning when a request is received is untenable since provisioning takes time, and the system must deal with numerous simultaneous clients. Our work was designed to bring together multiple elements into a cohesive system to achieve this goal.

Our solution takes the form of two interconnected software modules. The first module is a front-end service that handles user requests and collects metrics about the frequency of user requests and the models they request. Given a model, the service can locate the optimal endpoint for the specific model. The metrics it collects are integral to the functionality of the second module, the back-end service. This service acts as an orchestrator, performing real-time provisioning operations to optimize the system and reduce latency. The key component tying the two modules together is a Non-Linear Programming (NLP) solver, which is used to take the front-end's metrics and calculate an optimization function that attempts to minimize latency. The back-end service uses the solver's results to perform proactive provisioning tasks.

The novelty of this work lies in the system approach, enabling low-latency mobile inference services on edge servers. The solution's core components incorporate widely used software systems such as Tensorflow Serving and Kubernetes, making the overall solution highly maintainable and extensible. Furthermore, the solution's back-end integrates an NLP solver, which can automatically adapt to varied system parameters. Most importantly, to the best of our knowledge, this solution achieves the first functional system that performs joint inference provisioning and request dispatch for mobile inference servers on edge servers.

The contributions of this work are summarized below.

- This work studied a new problem in mobile edge computing, namely, joint inference model provisioning and request dispatch. This problem has been formally presented

as a non-linear integer programming problem.

- This work implemented a software solution that consists of both front-end and back-end services to accomplish the goal of efficient and latency-aware provisioning. This solution is fully containerized and can be ported to various container-friendly platforms.

- The complete solution has been evaluated through both simulation and testbed experiments. The achieved results demonstrated that the proposed solution could effectively reduce the inference serving latency compared to existing solutions.

The following sections of this thesis are organized as follows. Chapter 2 considers related work in this field. Chapter 3 formally presents the studied problem, while Chapter 4 examines the system architecture and design. Chapter 5 delves into the implementation details of the prototype system, while Chapter 6 analyzes the evaluation results obtained through simulation and testbed experiments. Finally, Chapter 7 explores potential avenues for future improvement of the proposed solution and Chapter 8 concludes this thesis.

Chapter 2

RELATED WORK

The central goal of edge computing is to move computation closer to the user to minimize latency and distribute computational load across many devices. This desire is directly driven by the increased generation of data at the edge and the need for machine learning to analyze and use this data [32]. Edge computing is a comparatively new field, with devices that are both powerful enough and cheap enough to be deployed at scale arriving on the market within the last few years. However, making efficient edge computing feasible is a difficult endeavor, leading to a significant amount of research in edge computing and intelligence provisioning. This chapter will compare and contrast our work with existing research.

2.1 Container Migration

One often-used technique when placing services at the edge is container migration. This relies on the layered architecture of popular containers and container runtimes, especially Docker [11]. The core idea behind this technique is to freeze a container, transfer its required components, files, and execution state, and resume it on a new node [11]. The goal is to preserve the current state of the execution while changing nodes and minimizing downtime. Researchers at San Jose State University proposed a system for edge computing with service placement by container migration. In their system, a client uses a web app interface to issue commands to deploy applications on an edge node through AWS [5]. While this approach also attempted to minimize latency, it differs in key ways from our solution. Their solution relies on a cloud orchestrator to issue commands, as well as a focus on freezing and migrating models [5]. Their evaluation found that migrating an existing container could take up to a minute or more depending on network speed. Our solution omits a cloud orchestrator entirely

since communication between edge and cloud may introduce non-deterministic latency.

Other approaches have also made use of the cloud for container migration but in a different way. Rather than having a cloud orchestrator issue migration commands, one method is to have the mobile client request a migration itself from the system [20]. This approach allows the user to control migration, thus eliminating the need for the system to continuously monitor each client's position and make a migration decision. This solution is very close to ours, but it still relies on the cloud to perform the actual migration and lacks the efficiency of optimized provisioning [4].

A paper published in MDPI discusses the potential benefits of container migration in fog computing [21]. Their implementation focused on migrating VMs and experimented with different migration schemes, some of which did not require all the data to be transferred before resuming the container. They recorded a migration time around 100 to 200 ms depending on what was copied [21]. While such a design might be necessary if a job must be moved before completion, we instead focused on minimizing client latency first. Our solution prefers to create multiple replicas of a model and serving a new replica to a future connection. Though our approach requires more memory, it minimizes the latency experienced by the client. Also, it eliminates the extended downtime required for container migration. Our solution is also less dependent on the internet speed since commands to spin up a container require far less data transmission than moving an entire container, its execution state, and the input data of the current job.

2.2 Service Placement

Service placement is the core function of our system. Attempting to place services intelligently is not a new idea, and there are several pre-existing approaches. One is to focus on user mobility. In this scheme, the idea is to create a "service profile" for the client, which contains all the information related to the jobs being executed for the client [6]. This implementation requires the system to be careful about the latency perceived by the client and the load on each edge node. However, this solution still depends on the movement of existing

jobs, which has a baseline latency cost regardless of the network's speed. It is possible to redirect the user to a new node and delete unused models in our solution. As before, this method consumes more memory, but it reduces latency and saves on complex placement calculations.

Queues are another commonly seen technique for placing services [24]. This scheme assigns each node a queue within which jobs can be added. The node then handles provisioning and serving clients in the order of the queue. This type of system also allows nodes to cooperate by deciding if they or a neighboring node would be better suited to handle a job and offloading it by moving it to another node's queue [24]. The queue idea can also be used differently: batching. Batching inference provisioning and then executing them all at one time optimizes the processing time of the provisioning process [30]. Our solution reflects this idea as well. It also performs provisioning tasks together in a batch rather than spontaneously, cutting down the time where resource availability is in flux.

2.3 Latency-Based Designs

Focusing on latency as a guide for service placement is a standard technique. Minimizing the latency experienced by the client has been a priority when building remote services since before edge networks were even feasible. It is important to note that focusing on latency alone is an impractical solution due to its volatility as a metric and the possibility that clients may have more concerns than just latency. Some researchers have chosen to introduce the idea of a Service Level Agreement (SLA) to the edge domain [16]. In this type of solution, the system makes a trade-off between latency and model accuracy to decide where to run mobile inferences. In contrast, our solution chooses to minimize latency only so far as model accuracy will allow. We do not compromise the quality of the deployed service, instead choosing to consume more resources to serve more clients, thus reducing latency by reducing load instead of accuracy.

2.4 Orchestrator-Based Designs

One orchestrator-based technique is to use a system operator to decide which node a client should connect. In [17], the researchers used a time-slot model to allocate services. Clients were assigned to a node by the operator in order of arrival. The model allowed for mobile users to be quickly moved between nodes as they moved within the coverage area of the system since a central operator was performing the placement [17]. As our solution lacks a central operator of this type, it cannot assign clients in this fashion. However, it is important to note that by allowing the client choice, as we do, they are more likely to receive satisfactory service since they can choose which they prefer, proximity or performance.

A slightly more common design is to take the orchestrator architecture and transform it into a hierarchical structure [28]. This design assigns a tier to each level of the device, from the client device itself to edge nodes and up to the cloud. The benefit of this design is the availability of cloud offload for running heavier inferences that cannot be handled on edge nodes [12].

The architecture most similar to ours is the uncoordinated access model [2]. In this design, a central orchestrator makes available a pool of edge computing resources in the form of lambdas. The orchestrator also acts as a repository of lambda images that edge nodes can pull from to deploy a new resource. The nodes themselves are tasked with opportunistically choosing which resources to deploy. The system then allows clients to choose a resource to use based on the latency that the client measures, which is very similar to our design [2]. Where the design differs is in the provisioning of resources and the role of the orchestrator. In our design, the orchestrator makes predictive provisioning decisions to optimize client latency. It also does not act as a repository of models. In our solution, each node acquires the models it needs from the internet and otherwise stores its available models in local storage. This enables our solution to perform rapid provisioning and deletion of models as per the orchestrator's instructions.

2.5 Comparison to Our Work

Our work differs from existing research in a few key ways. Firstly, it does not use a cloud orchestrator to manage edge nodes instead of moving all orchestration tasks to the edge cluster itself. While we use client-reported latency for service placement, we do not do it by way of SLA, QoS, or queues. We instead use that data to make proactive decisions about our system's deployments. Another key difference is the decision to omit a container migration system. Our system instead creates more replicas of a given model on other nodes and directs traffic to it while allowing in-person jobs to complete where they are. While this approach consumes more resources, it also eliminates the latency and network link saturation imposed by migrating an entire container across nodes. Since we can spin up a new replica, serving new clients on a different node is much faster. It also means our solution is not dependent on a specific container runtime architecture and can be implemented with any industry-standard tooling. In contrast to many of the designs in the existing body of research, our solution can also be containerized for portability.

Chapter 3

SOLUTION ANALYSIS

This chapter formally presents the studied problem of joint provisioning and request dispatch for performing mobile inference jobs over distributed edge servers.

3.1 System Model

In practice, edge networks will be built primarily with smaller servers operated in a cluster. These servers might be physically distributed and found at network access points, public infrastructure, or even within mobile units such as buses or trains. Distributed deployment strategies allow edge devices to perform operations while taking into account the status of their mobile clients and leverage their combined computing power to serve clients faster than any single node could.

The system under consideration consists of three key components: request, server, and provisioner.

- A request denoted as r_i, is generated by a mobile client for some inference job (machine learning model). Different requests may use the same model for the inference job. After communicating with a nearby edge server, a client decides which edge to execute its inference jobs. N represents the total number of requests.

- An edge node, denoted as s_i, executes an inference job upon request. An edge node has limited memory; therefore, it can only host a limited number of model instances in memory. Nevertheless, it can adjust the provisioned inference models based on the provisioner's decisions. S represents the total number of nodes.

- A provisioner determines what machine learning models will be provisioned on an edge server. The provisioner runs allocation algorithms to find the best servers for each request to minimize the overall inference latency, including communication and computing time. As a result, the communication and computing time can be estimated either in advance or during the run time with relatively high stability over a certain period.

3.2 Problem Statement

Suppose an edge network running machine learning models is to be deployed. Given a situation in which multiple mobile clients will request inference services from the edge network, how do we design a system that can serve clients as quickly as possible and proactively manage the system to reduce latency and optimize system functionality? Such a system must consider various factors, including latency, available system resources, and inference resource availability.

Based on the system model, we can formally define the model provisioning and request dispatch problem as follows:

- Objective:
  - $\min\{U\}$, where $U = \sum_{i=1}^{N} \sum_{j=1}^{S} I_{i,j}(\tau_{i,j} + \theta_{i,j})$
- Given:
  - $\theta_{i,j}$: execution time of r_i's inference job on s_j
  - $\tau_{i,j}$: round-trip communication time between request r_i's originating client and edge node s_j
  - m_i: memory requirement of r_i's requested model
  - c_i: computing requirement of r_i's requested model
  - $\mathrm{M_i}$: available memory resources on node s_i

  - C_i: available computing resources on node s_i

- Output

  - $I_{i,j}$: indicator that request r_i executes on s_j

- Subject to:

  - Memory constraint:
    for each node s_j, $\sum_{i=1}^{N} I_{i,j} m_i \leqslant M_j$
  - Computing constraint:
    for each node s_j, $\sum_{i=1}^{N} I_{i,j} c_i \leqslant C_j$
  - Server selection constraint:
    * for each request r_i, $\sum_{j=1}^{S} I_{i,j} \geqslant 1$
    * $I_{i,j} = \{0, 1\}$

Essentially, this is a non-linear integer programming problem. In the formulation, indicator $I_{i,j}$ is an integer value. When a request i is scheduled to execute on node j, $I_{i,j} = 1$; otherwise, $I_{i,j} = 0$. Ideally, a request r_i must be scheduled to some server s_j for the best performance. Although the node selection constraint may allow a request to be scheduled to multiple servers, the optimal solution will only select one node for the lowest latency. In this formulation, the execution time of an inference job is assumed static when the computing constraint is satisfied.

Based on this problem's formulation, either heuristic or NLP solver-based solutions can achieve a useful result in a reasonable amount of time. Chapter 4 and Chapter 5 will present the details of the NLP solver-based solution developed in this work.

Chapter 4
SOLUTION ARCHITECTURE AND DESIGN

This chapter presents the design and architecture of our proposed solution to the problem laid out in Chapter 3.

4.1 Architecture Overview

As described in the preceding chapters, the system's key goal is to minimize latency. It aims to do this by serving clients the optimal ML resource when requested and proactively provisioning ML models based on predicted request patterns.

The following features are required to solve this problem:

- A way to understand and serve incoming requests
- A method of collecting the metrics required to perform predictive provisioning
- A system that utilizes the collected metrics and makes provisioning decisions
- A scheme for performing provisioning tasks

From a software perspective, there are three modules. These consist of a user-facing front-end service, a back-end service, and a Non-Linear Programming (NLP) solver. The front-end is designed to handle user requests, while the back-end performs ML model provisioning. Finally, the NLP solver ties everything together and acts as the bridge between the front-end and back-end. We decided to decouple user-facing functionality from back-end functionality since each deals with a specific set of operations with little overlap. Additionally, separated modules made it possible to create a system design without dependence

on a specific technology stack. Instead, each module could be developed using whatever industry-standard technology stack that the developer prefers.

Figure 4.1 shows an example of the considered system.

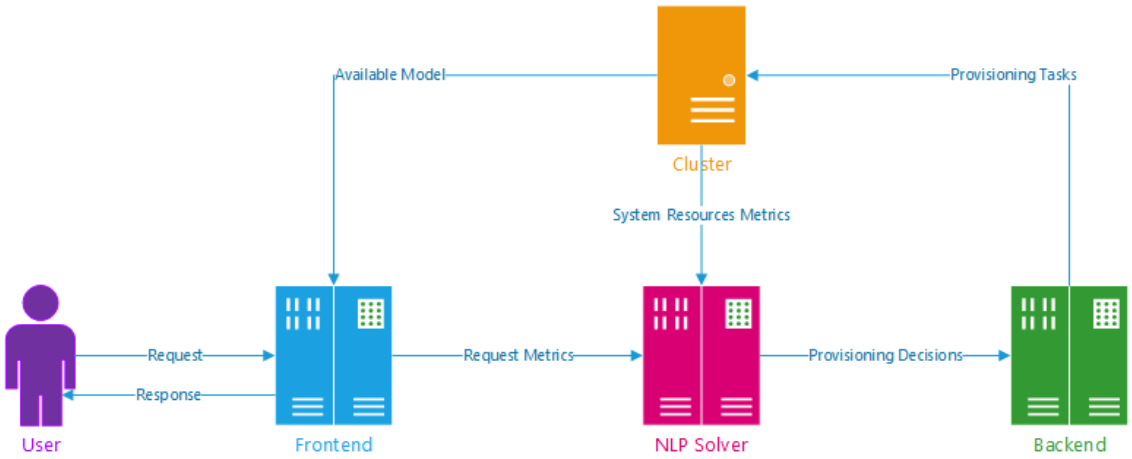

Figure 4.1: Overview of the Prototype System Model

In Figure 4.1 we can see the three modules as they relate to the edge node cluster. We can also observe that the client's only point of communication with the system is through the front-end. While all three software modules run on the control node in our prototype system model, in practice, these modules could be implemented to run whatever combination of machines desired. These three modules interact with the edge node cluster through the control node, which connects to the rest of the cluster over a network interface. The number of workers and the choice of Ethernet for the network interface is arbitrary, as in practice, these details would be unique to a specific deployment.

4.2 Front-end Design

The front-end module is responsible for all features that deal with handling user requests. This module has two primary functions. Firstly, it responds to user requests for ML models with the optimal endpoint for the client to access a requested inference. We define an endpoint as the IP address of the node hosting the model and the port number corresponding

to the model. It is also responsible for collecting metrics related to when and how often a model is requested. This data is made accessible to the back-end by a web request. This design was chosen primarily to enable the front and back-end services to operate on different hardware. Even if the two services are operating on the same machine, this design prevents any issues from two different services accessing the same files without any guarantee of synchronization. Finally, this design also allows the front-end to clean up or otherwise modify the data if needed before sending it to the back-end. Another benefit is that while the front-end depends on the back-end for provisioning, it does not explicitly depend on the presence of *our specific back-end* to function. Any resource provisioning solution could be running, and the front-end would continue to function as designed.

4.3 Back-end Design

The back-end service module's core function is performing real-time provisioning operations. It uses the provisioning decisions made by the NLP solver to decide where to provision ML models. The back-end module has two operating modes.

- **Mode 1** is the primary mode where the system prioritizes minimizing latency after provisioning. In this mode, the system provisions new models based on the decisions of the solver but does not do so when deleting models. Instead, the system deletes models based on how recently they have been requested and how often. It also uses the available memory on each edge node when executing the optimization calculation.

- **Mode 2** optimizes the system's performance during the stable time but at the potential cost of some instant latency. When performing provisioning tasks in Mode 2, the back-end service strictly follows the solver's decisions. Instead of deleting models using the time they were last used, they are immediately deleted if the solver does not decide to place them on an edge node. In addition, instead of using the **available** memory on each node for the solver's optimization function, the **total** amount of memory on each node is used.

We designed two modes since a potential implementation could prioritize either client latency or system stability.

4.4 Non-Linear Programming Solver

The NLP solver is arguably the most important component of our system. We supply it with an optimization function with constraints and various metrics that reflect the various requests received and the current state of the cluster. Given all of that information, the solver executes a minimization function that assigns a probability to each model on each node, effectively acting as a decision for where each model should be deployed. The solver allows the system to use past events to make effective predictions about what models might be needed in the future. Without this functionality, the system would need to modify ML model deployments based only on the system hardware's current status rather than clients' needs.

In Section 4.1, we referred to the NLP solver as a bridge between the front and back-end modules. Without the NLP solver, the front and back-end modules would have no connection with each other. They would still operate on the same hardware cluster but would be otherwise disconnected. As a result, the front-end would have no way to influence provisioning decisions. The back-end would be unable to take educated provisioning actions. The NLP solver forms this key connection by taking the front-end's data and, after performing the optimization calculations, offering the back-end provisioning decisions. The back-end uses these decisions to provision models, which in turn affects the front-end and the models that it can serve, creating a cycle through which the system can continuously make use of updated data to better provision available resources.

4.5 Practical Issues

The formulated problem in Chapter 3 assumes that node resources are sufficient to serve all requests. However, in practice, several issues may waste the limited resources available to edge nodes. Many of these stem from the environment within which the system must operate.

This environment presents new challenges compared to largely immobile data centers. The most critical of these issues and solutions are as follows:

- **Client movement during inference execution** [4]. Suppose the client moves out of the communication range of the edge node that is currently serving it. In that case, the system should move on to serving another client. If the client should return, they will be served again.

- **Edge node resource exhaustion**. Unlike large data centers, edge nodes have a relatively low amount of resources available to them. Since an edge network does not have access to many machines to provision when serving more models, they must instead intelligently provision the models available to them. Dealing with this issue is primarily a matter of prevention. The system should never provision models to the point where a node would run out of resources. Once a request has been serviced, the system's provisioning service should decide evicting models to conserve resources. Suppose a client request should arrive that cannot be serviced without preempting existing client workloads. In that case, it will be rejected in favor of those inferences that are already in progress.

- **Client device failure**. Suppose the client device was to fail or disappear for any reason. In that case, the system must handle having no recipient for the inference it has been performing. Once the inference is complete, the resource that the missing client had requested may be terminated by the provisioning system to minimize resource consumption. Should the client return, they would be served again as if they were a new client.

- **Edge node failure**. Over time, edge node hardware failures are inevitable. When they occur, the system should be resilient and direct requests to the remaining nodes without crashing or directing a client to an unavailable endpoint. Any services that the failed node was hosting should also be redeployed onto other nodes when needed.

This process must also consider the resource envelope of the remaining nodes, as their memory space may be full, necessitating existing services to be terminated.

- **Network or power failure.** Similarly to node hardware failures, network and power failures generally occur at some point in time. When edge nodes return to operational status, they should contain the last set of deployed resources before they went down. From that point on, any redeployed services while the node was down might be destroyed by the system to free memory.

- **Resource requested that is not deployed**. If the client was to request a model that is not deployed, the system must make a quick decision as to whether the model can be deployed, and if so, where. The system must consider the model's requirements, the client's information (such as physical location), and the resource consumption of the available nodes. If the model cannot be deployed, the client's request will be rejected.

- **Unknown resource requested**. If the client requests a model that the system does not have, it must communicate this fact to the client swiftly and reject its request.

Chapter 5

SYSTEM IMPLEMENTATION

This chapter explores the design of the software implementation of the solution outlined in Chapter 4.

5.1 Overview of Software Components

Figure 5.1 shows the hardware and software design of the prototype system.

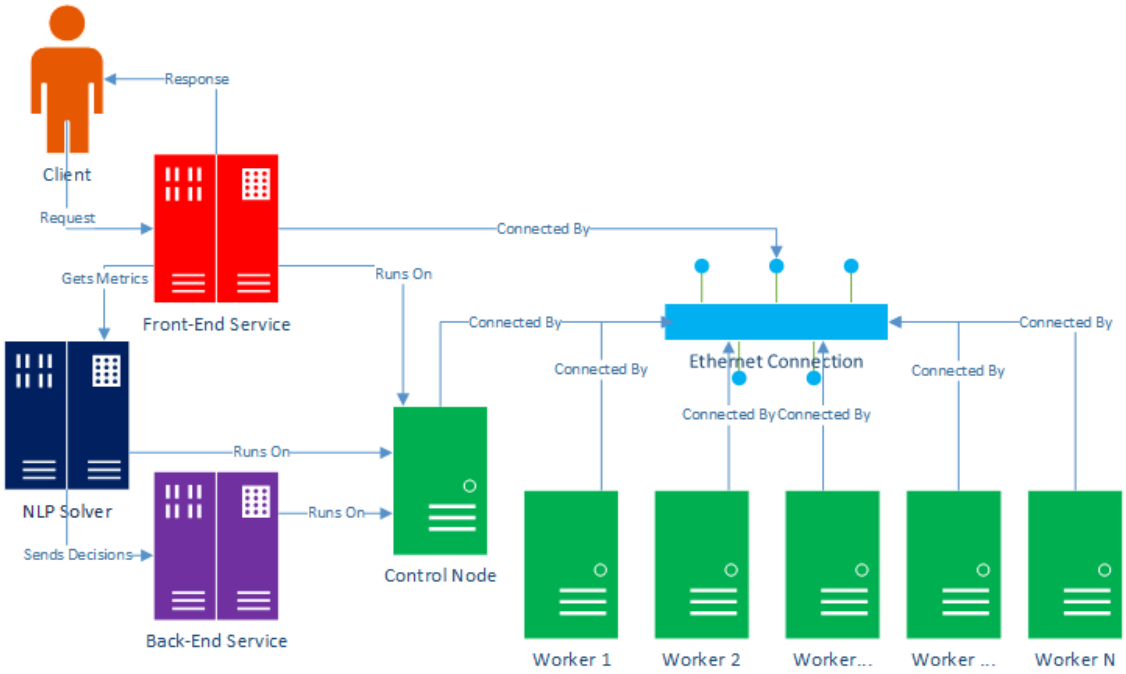

Figure 5.1: Overview of the Prototype Implementation Design

At a high level, the system is comprised of two semi-connected software modules written in Python running within Docker atop a Kubernetes cluster. Kubernetes is a container management and orchestration platform originally developed at Google for container workloads

either locally or in the cloud [27]. Evaluations of Kubernetes have shown it to be quite efficient, making it an appropriate choice for this implementation [19]. The system's various modules employ the singleton design pattern, with one instance of the front-end, NLP solver, and back-end, respectively.

- The front-end service, which is a RESTful API implemented with Python Flask [18]. It is responsible for accepting user requests, returning a specific endpoint or list of endpoints, and proxying requests. It is also tasked with collecting metrics to be used by the back-end service.

- The back-end service is also written in Python. It performs provisioning and optimization tasks by acquiring metrics from the front-end and invoking the NLP solver to obtain probability values for each model on each node. It performs these tasks periodically as defined by the system administrator. For solving the NLP optimization problem, we made use of the Ipopt solver [29] from the COIN-OR suite [3]. We invoked Ipopt through AMPL, a modeling tool for solving NLP problems [1]. Using Ipopt in conjunction with AMPL allowed us to dynamically generate configuration files in code based on templates when needed.

The front-end expects the client to make one of three types of requests.

- Requesting a list of endpoints for every model.

- Requesting an endpoint for a specific model.

- Requesting the system to proxy and return an inference result executed by a known endpoint.

The system implementation is also very accessible since it can be packaged up as a container and deployed on any Kubernetes cluster. However, the system design does not depend

on Python, Kubernetes, Ipopt, or any other specific toolchain. The system design described in Chapter 4 could be implemented using any tech stack. We chose the hardware and software for our prototype system due to their extensive use in academia and industry. Furthermore, we decided that using well-known technologies would also help with accessibility to our work. This choice simultaneously proves that our design is realistic and can be implemented with the same hardware and software being used today by millions of developers worldwide.

5.2 Front-end Service Design

The front-end service is written in Python. It is designed around a Flask server which acts as a RESTful API, handling and responding to user requests [18]. For the remainder of this paper, when we refer to a server when discussing the implementation, we refer to the Flask server that defines the front-end. A separate Python file contains a set of functions available to the front-end service. The Flask server checks the request body that comes as part of any user request and passes it to the appropriate function based on the requested information. The front-end service has three types of client-facing requests that it handles across two endpoints: getting endpoints or proxying a request. Additionally, the front-end collects and stores the model requested. The endpoint returned for every request and the number of times a model has been requested overall, plus the last time it was requested. These metrics are made available to the back-end via three special endpoints used by the back-end service to acquire these metrics for background provisioning. Once invoked, the front-end service continues running until it is stopped or the machine stops functioning. Figure 5.2 depicts the logical steps taken by the front-end service when serving a client request. The front-end service is also robust enough to handle worker failures. When a worker node goes down, the front-end will continue to serve endpoints from the remaining nodes. The front-end is also scalable. If a new node were to come online and start serving models, the front-end would begin serving the new endpoint.

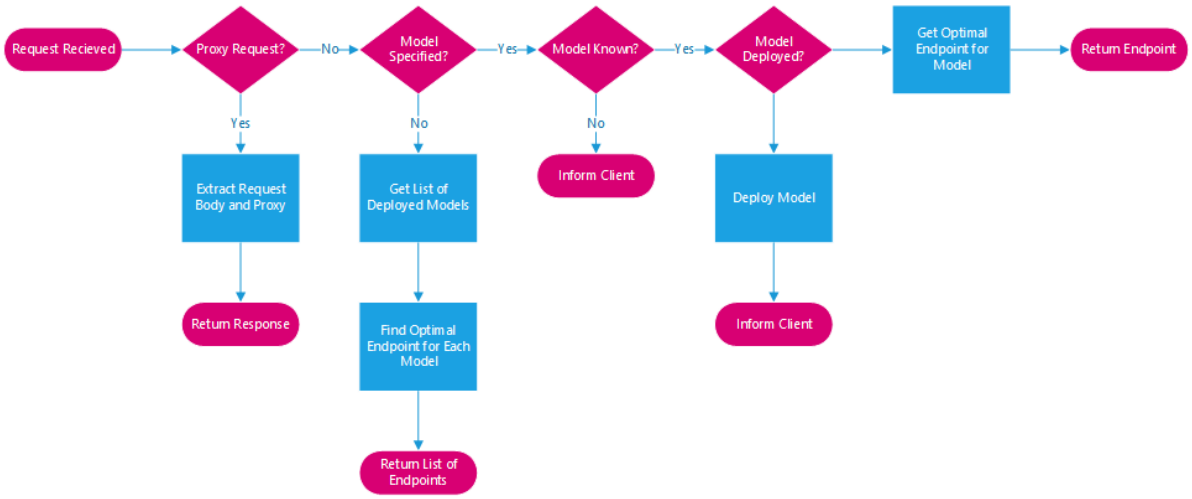

Figure 5.2: Overview of the Front-End Execution Path

5.2.1 Services Client Endpoint

This endpoint is responsible for returning the client a list of available endpoints for each model or one specific endpoint. There are several steps that the system performs to accomplish this.

- The server will first check to see if the client has included a model request in their request body. If there is no request body, the system will return a list consisting of one endpoint per model. Otherwise, the system will return an endpoint only for the specific model requested. The functions used to accomplish this are essentially the same for both operations. The only difference is that returning a list requires the functions used to get one endpoint to be run once per model.

- Once the model to locate is known, the system will first query the cluster to get a list of running pods. Pods are how Kubernetes encapsulates Docker containers [26], which are running the models. The system then checks the pods to find out which ones are running the requested model.

- Once the list is shortened to the pods running the desired model, the system then

queries the cluster for the CPU usage of the nodes those pods are running on. From that list, the node running the required model with the lowest CPU usage is selected. Finally, the system gets the port for that model and then assembles that into the standard IP address and port format. It then returns this endpoint to the client.

- If a list is required instead, the system will first obtain a list of all the models currently running from the cluster and then run the aforementioned functions for each known service. For example, if the system is currently hosting five models, the functions required to obtain an endpoint will be run five times. All five endpoints would then be serialized into JSON and returned to the client.

It is important to note that the system considers each model as a running service irrespective of the number of replicas. This means that each model could have an arbitrary number of replicas running. However, the system will see them all as one service and pick one of these replicas to serve the client. Additionally, suppose the client requests a model that the system is aware of but is not currently serving. In that case, the front-end can provision the model on the cluster independent of the back-end to serve the client. The next time the client requests a model that was not originally provisioned, they will receive an endpoint to the newly provisioned model. Any models provisioned this way will be considered by the back-end when it performs its periodic optimization operations.

5.2.2 Proxy Client Endpoint

Proxying requests is a slightly more involved process than simply returning an endpoint. However, it reuses much of the same functionality. The client informs the server that it wants a request proxied by including the TensorFlow Serving model path in the request endpoint [25]. For example, instead of just making a request on */services*, they may request */services/model1/v1/model1b/add*. The service will then understand how to proxy this request. It employs the same functionality as requesting a specific service to obtain an appropriate endpoint. However, instead of simply returning the endpoint to the client, the

system then extracts the request body meant for the model from the original request. It then performs a POST request on the model itself. The response from the model is then proxied back to the client.

5.2.3 Request Stats Back-end Endpoint

The back-end service uses this endpoint. The front-end records the model requested, the latency value, and the node eventually chosen for every request received that asks for a specific model. The back-end uses this data to decide on its provisioning tasks. This endpoint is made available to allow each module to remain semi-disconnected. Using an endpoint allows the front-end and back-end to run on different machines. It avoids the problems that may arise from both services accessing the same files if they run on the same machine.

5.2.4 Model Stats Back-end Endpoint

Similar to the request stats endpoint, this endpoint provides metrics to the back-end. In this case, the front-end records the last time a model was requested and the number of times a model has been requested since it was provisioned. The back-end uses this data to remove unnecessary services.

5.2.5 Node Memory Stats Back-end Endpoint

Similar to the request stats endpoint, this endpoint is used by the back-end to acquire the current RAM usage of each node in the cluster. This information is critical for deciding which node to place a model on, on perhaps which node to remove a model from.

5.3 Back-end Service Design

The back-end service is also written in Python. All of the external information needed, such as thresholds for deletion, are provided by a command-line argument. The back-end service is invoked by a cron job, which runs periodically at the specified time. Figure 5.3 visualizes

the primary operation mode of the back-end service. When the Non-Linear Programming solver is run, it outputs a 2D array of probability values corresponding to each model on each node. Kubernetes uses YAML files to define deployments [26], so these probability values are then used to generate YAML files for each deployment that needs to be performed. Once the files are generated, the Kubernetes commands are issued through the Kubernetes Client Library [10].

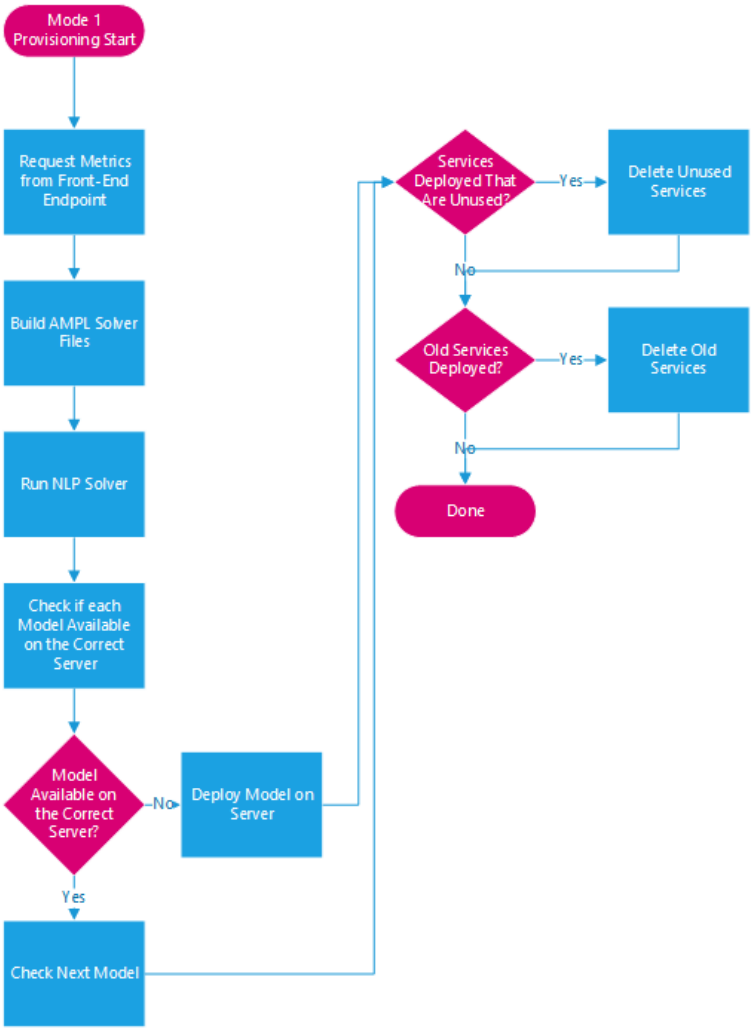

Figure 5.3: Overview of the Back-End Execution Path in Mode 1

In the primary operation mode (Mode 1), the back-end services prioritize latency over

other considerations when performing optimization and provisioning tasks. However, it might be preferred for the system to prioritize resource availability at the cost of latency. In this case, a mode choice can be specified as a command-line argument when invoking the service. Mode 1 corresponds to the implementation described in this chapter. Mode 2 changes the functionality of the provisioning operation. When running the NLP solver, the system uses the total amount of RAM on each node rather than how much is currently available. Furthermore, the operation that deletes models is also modified. Instead of being based on how much and how often they are requested, the solver's output is followed strictly. If the solver does not place a model on a node and the model is present on the node, it will be deleted, and similarly, if a model is placed and the model is not present, it will be provisioned. In Figure 5.4, we can see a representation of Mode 2, as well as how it differs from Mode 1.

5.3.1 Non-Linear Programming Solver Implementation

While logically a separate component, the NLP solver is implemented as part of the back-end service within our prototype system. AMPL allows for defining an NLP optimization problem using a model file, which defines the optimization problem and the constraints that bind it. A data file defines the data used in the calculations. These files are generated by Python code from a template file that defines the required format. The number of nodes, number of requests, latency values, the execution time of a model on a node, the amount of memory needed by a model, and the amount of available memory on each node are written into the data file. The number of requests and nodes are similarly written into the model file. The system then generates a run file, which is simply a method of storing all of the command-line commands issued to AMPL if the solver were to be run manually. AMPL is then invoked in the command-line through the Python *os* module. Once the provisioning is complete, the system deletes any temporary files to avoid any chance that previous results might contaminate future solver runs.

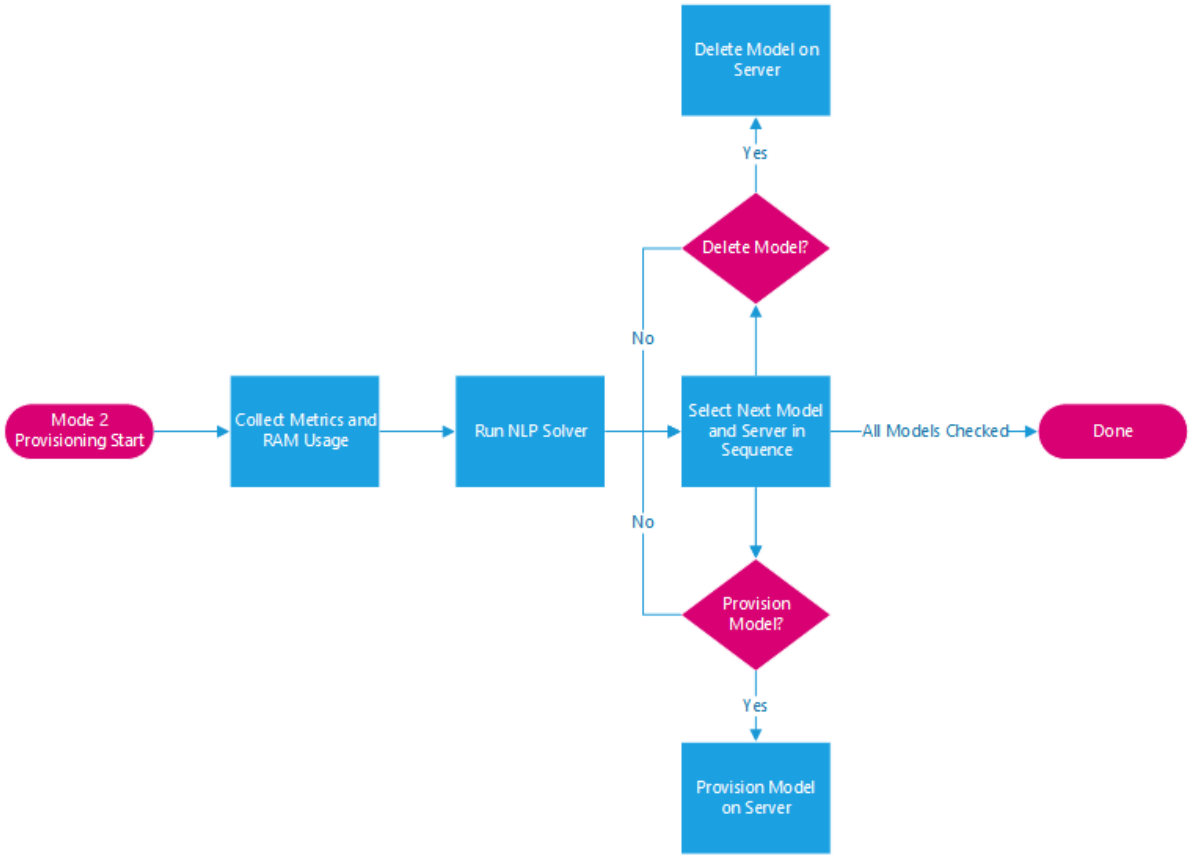

Figure 5.4: Overview of the Back-End Execution Path in Mode 2

5.4 Serving Models

Machine learning models are served using TensorFlow Serving (TFS) containers [25]. We chose TFS because it is explicitly designed to serve ML models and can be accessed through standard REST API formats. TensorFlow Serving's developers offer pre-compiled Docker images that can be run in Kubernetes. However, their images are only available for x86-based hardware. Since Nvidia Jetson hardware is ARM-based, the official Docker images could not be used as-is. To mitigate this issue, a freely available Docker image that can be run on an ARM was used instead [7]. The system uses four existing example docker containers to demonstrate how multiple models can be served.

5.5 Model Provisioning

Each model to be deployed requires a template file containing information such as the Docker image's location, the name of the credential file needed to access a private Docker repository, etc. Using a similar method as the generation of files for AMPL, the system can fill in critical info such as the node's name to deploy to create a new file. This file is then used to invoke the deployment function via the Kubernetes client library. The function to create a Kubernetes service, which exposes the pods for use, is also invoked using another YAML file that defines the service parameters in the Kubernetes-required format.

It is also possible to delete a deployment and its related service through the client library when given the name of a model. Since deployments and services have a longer name with strings of letters and numbers, the system will first request a list of pods from the cluster and check to see if the specified model is running. It will then use the deployment and service's full name to instruct the cluster to delete both. The cluster will then immediately delete the service and terminate the associated pods before deleting the deployment. Once complete, the front-end will no longer offer the deleted model when a list of endpoints is requested.

5.6 Testbed System Hardware

The testbed system consists of six Nvidia Jetson development kits. One Nvidia Jetson Xavier NX as a control node, four Nvidia Jetson Nano units as standard workers, and one Nvidia Jetson AGX Xavier as a more powerful worker. All boards are running Nvidia JetPack [14] based on Ubuntu 18.04. We chose Jetson boards for our prototype system because they are popular devices for machine learning, edge computing, and cluster computing applications. The system can be run on any hardware supporting Kubernetes, Docker, and Python. However, Nvidia Jetsons were the ideal devices to simulate a real-world edge deployment. Since they are well-known devices, they were also fully supported by the software toolchain we used to build our prototype system, eliminating any potential obstacles that may have arisen from using obscure or incompatible hardware. Figure 5.5 depicts the physical devices

as deployed for the prototype system. Devices labeled with "A" are Jetson Nanos, Device "B" is the Jetson Xavier NX, and Device "C" is the Jetson AGX Xavier.

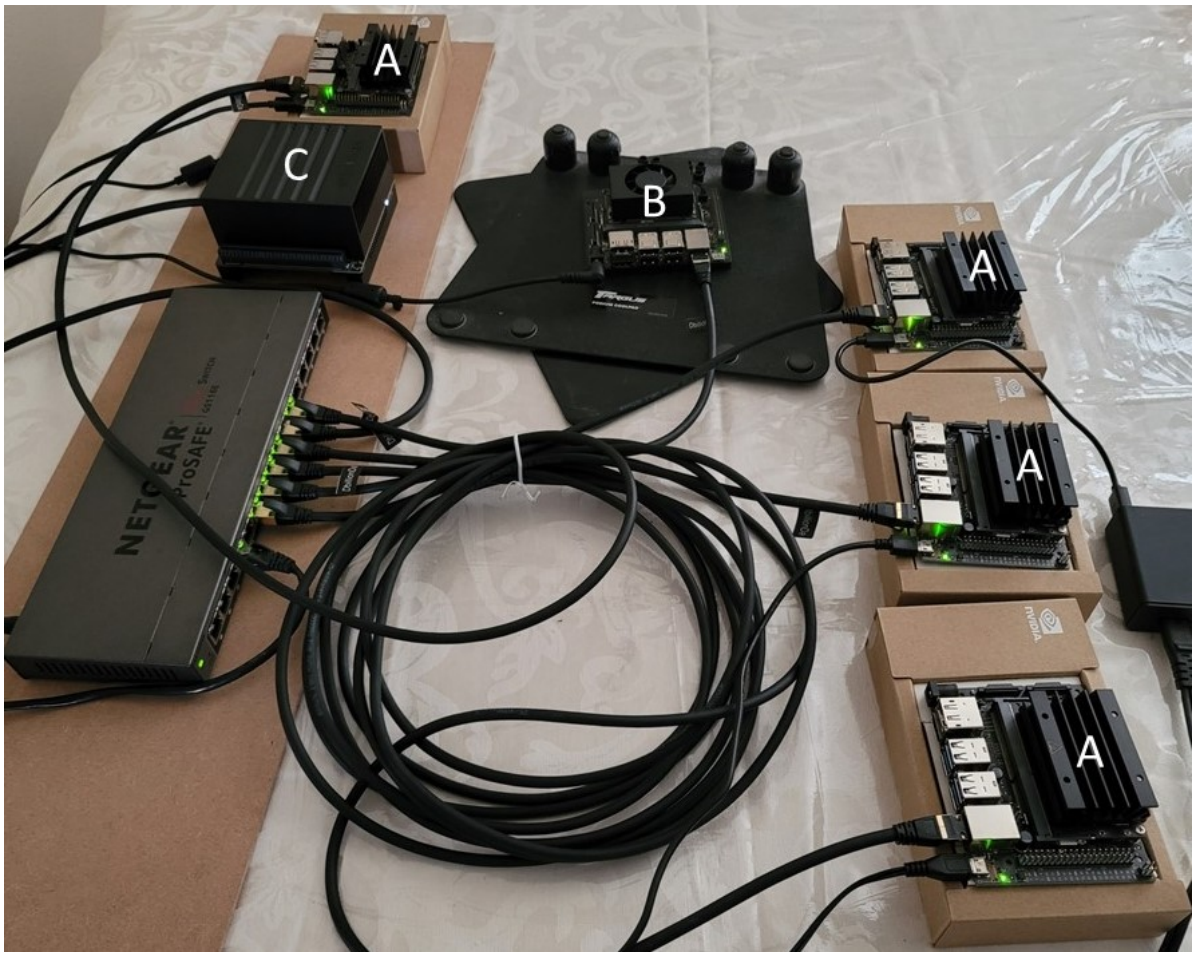

Figure 5.5: The Prototype Edge Cluster

5.7 Challenges

There were several challenges encountered when implementing the system.

- An important architectural challenge that arose early on was designing a way for the front and back-end services to both have access to the metrics collected by the front-end concerning requests. The front-end needed access to update the metrics, and

the back-end needed them to run the NLP solver. However, it would be a poor and limiting design to simply put both services on the same hardware and access the same files. Doing so would have required complicated synchronization. More importantly, it would have required both services to operate on the same hardware and file system. Our implementation instead has the back-end access the front-end through the same mechanism clients already do, REST calls. Having the back-end make a GET call on the front-end means only the front-end access the files, and both modules can run on different hardware or even outside the cluster.

- Later on in development, we found out that TensorFlow did not provide official TFS Docker images for the ARM architecture that Jetson devices use. To work around this, we had to find and acquire a freely available image [7] that we were then able to package models into in order to deploy them.

- Along with the issue of incompatible architectures, we also had problems setting up local Docker repositories in the cluster to store our models, as they did not appear to function on our Nvidia devices. We solved this by setting up a secured private Docker repo in Docker's cloud hub and linking our devices to it by setting the pull location for images in Kubernetes using YAML configuration files.

Chapter 6
PERFORMANCE EVALUATION

This chapter analyzes the methodology and results of the simulation and testbed experiments we conducted to evaluate our work.

6.1 Simulation Experiments

Through multiple simulation experiments, we have evaluated the latency of the inference provisioning research system using the following schemes:

- Varying the number of requests to the system
- Varying the number of nodes in the system

Each operational mode is evaluated through each scheme. An iterative search solution is also evaluated using the same simulations to provide a point of comparison. These schemes were chosen because requests and nodes constitute the most likely points of variance in both the research system and potential real-world implementation. Increasing the number of requests represents an increase in the number of clients. An increase in the number of nodes represents the changing hardware landscape typical to edge deployments.

Simulation Setup

Each simulation varies one parameter while holding the others constant. When varying the number of requests, the number of nodes, models, the execution time on each node, and all other metrics and information remains constant. Similarly, while varying the number of nodes, we fix all other values, including the number and content of the requests. Each simulation was conducted against two implementations. For all tests, the content of the

Table 6.1: Default Simulation Values

Parameter	Default Value (Units)
Number of Requests	25
Number of Nodes	10
Number of Models	4
Execution Time	Random 0.1 - 0.8 s
Round-Trip Time	Random 0.1 - 0.5 s
Available RAM (Mode 1)	Random 10,000 KB - 500,000 KB
Total RAM (Mode 2)	Random 30,000 KB - 800,000 KB

requests (the model requested and the latency experienced by the client) is randomized within fixed bounds, as is the amount of memory available to each node. All other values are preset and remain unchanged in all simulations. Table 6.1 provides a list of default values for simulation parameters. As defined in the table, some parameters were randomly generated within a controlled interval to obtain useful results.

We simulated and compared the results of the solver and iterative search solution as follows:

- The key part of the system is the NLP solver that the provisioner uses to determine which tasks to undertake. For the simulation, we isolated the solver and ran each simulation against it. To measure the solution's latency, we recorded the objective value reported by the solver after it completed the minimization and optimization calculations.

- The iterative search solution does not use an NLP solver, so we built a function that produced a similar objective value by different means. We instead summed each node's latency and execution time on each node to obtain a list of values, one per node. We then took the minimum value and used it as the objective value for the iterative search solution.

6.1.1 Simulation Results

Number of Requests

Firstly, we evaluated the latency of the system as the number of requests changes. We ran four intervals for this simulation, beginning from 20 requests and increasing by ten each time until reaching 50 requests. This spread of request counts represents the number of requests the NLP solver would be asked to consider in the average case. Figure 6.1 depicts the results of all four runs of the simulation.

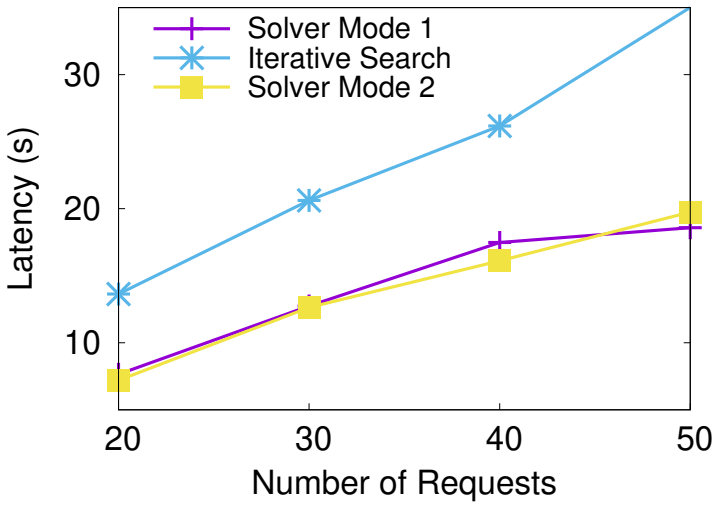

Figure 6.1: Latency Result of Solver and Iterative Search Implementations when Varying Number of Requests from 20 to 50 With All Other Variables Fixed

We can see from Figure 6.1 that in every run, the Solver Mode 1 solution resulted in lower latency than the iterative search solution. As the number of requests increased, the number of latency values did as well, with a sizable gulf persisting between the two solutions through every run. At 20 requests, we see a latency value of 7.6445 for the solver and 13.6383 for the iterative search. At 50 requests, this difference becomes even more pronounced. The solver solution produces a latency value of just 18.5799 at 50 requests. In contrast, the iterative search solution produces a latency value of 35.0095. From the trend line in the figure, we can see the latency value at 50 requests start to plateau in the solver solution. In

contrast, the iterative search solution continues to increase at a much higher rate. Overall, the solver solution produced an average latency value of 14.1118 compared to the iterative search solution's 23.8578. When we bring Solver Mode 2 into consideration, we can see that it produces an overall lower latency value than even the Mode 1 solver. At 20 requests, Mode 2 produces a latency value of 7.1987 with a value of 19.7490 when we move to 50 requests. Mode 2 produces an average latency value of 13.9244, which is only 0.1874 less than Mode 1. We can see from these results that both solver solutions are more effective at handling higher request counts than the iterative search solution. The Mode 2 solution is almost the same as Mode 1, though we can see Mode 1 start to pull ahead as we increase the number of requests to 50. This result makes sense because Mode 2 prioritizes system balance,

Number of Nodes

Secondly, we evaluated the latency results derived from varying the number of nodes available in the simulation. As with the simulation of the request, the content of the requests and the amount of memory a node was given were randomized. For this test, the number of requests was fixed, and the number of nodes was varied from 5 to 20 in increments of five. We chose these node counts for our simulations because it would be appropriate to find 5 to 20 edge nodes in close proximity working together in a compute cluster in a potential real-world implementation. Figure 6.2 shows us that, as we might expect, increasing the number of nodes reduced the overall latency experienced by the client. We can also see from the figure that the solver solution produced a lower latency value in all runs than the iterative search solution. We can see a latency value of 15.0272 for the Solver Mode 1 solution at five nodes, which is not significantly different from the iterative search latency value of 17.8215. However, at 20 nodes, the latency value produced by the solver solution decreased to just 8.6376, with an average latency of 10.6706. On the other hand, the iterative search solution's latency values decreased to just 16.1020, with an average value of 16.8416. Solver Mode 2, on the other hand, produced a latency value of 13.1386 at five nodes, which reduced to 7.3429 at 20 nodes. Mode 2 produced an average value of 9.5920, 1.1416 less than Mode 1. From these

results, we can infer that both solver solutions will be more successful at taking advantage of the extra resources made available by more significant numbers of nodes. We can also see Mode 2 start with a lower latency value before briefly rising above Mode 1 and then finally lowering again. Mode 2 producing lower latency values as the number of nodes increases is a logical result since Mode 2 is focused on balancing the system and keeping performance high, which in turn contributes to lowering latency.

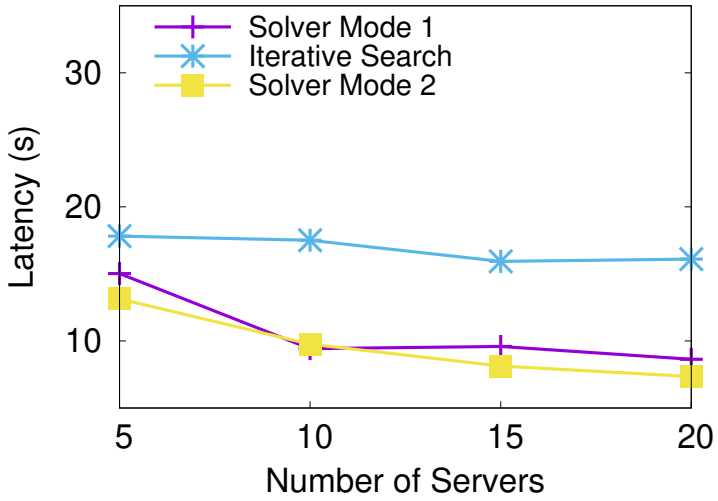

Figure 6.2: Latency Result of Solver and Iterative Search Implementations when Varying Number of Nodes from 5 to 20 With All Other Variables Fixed

Analysis of Results

The simulations show that the solver solution is more effective at minimizing latency than the iterative search solution. This is primarily because the solver solution is explicitly geared toward minimizing latency. After all, it considers the current state of the system when making its decisions. Even though the iterative search solution selects the smallest combination of latency and execution time when calculating its objective function, it does not look at the "big picture". The iterative search solution does not consider any extraneous factors such as memory capacity or node hardware constraints. It only considers the execution time

Table 6.2: Solver Execution Time for Varied Number of Requests

Number of Requests	Mode 1 (ms)	Mode 2 (ms)
20	19.229	23.227
30	20.581	23.532
40	24.380	25.901
50	24.391	19.239

Table 6.3: Solver Execution Time for Varied Number of Nodes

Number of Nodes	Mode 1 (ms)	Mode 2 (ms)
5	17.742	19.239
10	20.417	21.231
15	22.323	22.449
20	26.012	26.546

and latency local to a particular node, limiting its overall efficacy. However, the solver solution can place services and calculate the objective function with respect to the memory, computing, and node hardware selection constraints. This behavior allows it to identify whether or not to place a service even if a particular node does not perform the inference as fast as another, as long as it minimizes the overall latency. Taking all of these factors into consideration enables both solver solutions to look at the "big picture," making them the more efficient solutions.

6.2 Testbed Experiments

6.2.1 Solver Timing Simulation

Another simulation we conducted was to capture the execution time of the solver to see how efficient it is and how well it dealt with increasing numbers of requests and nodes. Tables 6.2 and 6.3 show the execution time of the solver in both Mode 1 and Mode 2 when performing the simulations that change the number of requests and the number of nodes. These times are reported in milliseconds and were collected using the Python *time* module.

We can see from these results that, on the whole, the solver takes ever so slightly longer to execute in Mode 2 than Mode 1. Since the only difference between the modes from the solver's perspective is the use of each node's total amount of RAM as opposed to just the amount of RAM available, this result makes sense. Larger memory space values for each calculation result in more significant numbers in each calculation, translating to higher calculation times. However, we can also see from these results that the difference is negligible. Furthermore, we can see that as the number of requests and number of serves increases, the amount of time required for the solver to complete its calculations is still relatively small. This shows us that even with request counts of 50 or more, or node counts of 20 or more, the solver will still complete its optimization calculations in a perfectly acceptable amount of time.

6.2.2 Provisioning Time Comparison Between Mode 1 and Mode 2

As described in Chapter 4, the back-end has two operating modes. Mode 1 prioritizes minimizing latency, and Mode 2 prioritizes system performance. To achieve this, Mode 1 does not delete models simultaneously; it provisions new ones, effectively ignoring the NLP solver for freeing resources. Mode 2 takes the opposite approach, freeing resources immediately as per the solver's output. This difference in operation means that Mode 1 should produce lower provisioning latency at the cost of more system resources being used. In contrast, Mode 2 should be the opposite. To test this, we set up a testbed experiment to measure provisioning latency on the prototype system.

Experiment Setup

This experiment was run directly on the prototype system. There are several experiment conditions to be aware of:

- Provisioning Latency is defined as the amount of time in milliseconds from when the provisioning task is executed to when it completes. It does not include the solver's

Table 6.4: Testbed Provisioning Latency of Mode 1 in Milliseconds

Test	Mode 1 (ms)
No Provisioning Task Needed	495.977
Provision 1 Model	2548.240
Provision 1 Model While Another is Deleting on the Same Node	10499.871
Provision 4 Models	11848.184

Table 6.5: Testbed Provisioning Latency of Mode 2 in Milliseconds

Test	Mode 2 (ms)
Delete 3 Models	818.980
Provision 1 Model and Delete 2 Models	3150.921
Provision 2 Models and Delete 3 Models	5915.636
Provision 3 Models and Delete 2 Models	9350.82

execution time.

- The request metrics are a sample of metrics from real system testing. They are not changed during the testbed experiment.
- All other inputs for the solver as they are in the normal system.
- When starting, there are no models provisioned on the system.
- The back-end is run several times in each mode. The provisioning latency and what provisioning tasks were completed were recorded.

Experiment Results

Table 6.4 lists the results for Mode 1, all collected using the Python *time* module. Table 6.5 similarly lists the results for Mode 2. These results offer us several measurements comparing the difference in provisioning latency between the two modes and insights into what tasks most affect it. From these results, we can see that provisioning models require more time than

deleting them. The provisioning of one model takes more than 2500 ms, whereas deleting one takes only around 272 ms. However, once we combine provisioning with Mode 2's model deletion task, the latency measurements immediately increase. In Mode 1, provisioning a model takes roughly 2548.240 ms. When running the same test in Mode 2, we added on the two deletion tasks the solver added, resulting in a latency value of 3150.921 ms. As we added more models for provisioning and deletion, the Mode 2 latency values increased even further. We only see Mode 1's provisioning latency increase significantly when it has to provision more models than Mode 2, with a value of 11848.184 ms for four models. Rarely, Mode 1 may provision a model on a node simultaneously that same model might be terminating. This can only happen in Mode 1 since the deletion process is not based on the solver. Interestingly, the latency value is very high when this happens, with a recorded value of 10499.871 ms. This is because both the creation and deletion processes are occurring at the same time on the same node for the same service.

These results show that, given the same number of models to be provisioned, Mode 2 has higher provisioning latency than Mode 1. The addition of time-consuming deletions to almost every Mode 2 raises the latency by several hundred milliseconds per model to be deleted. While the latency is higher, the system will run better overall since unnecessary models have already been deleted. Mode 1 is faster, reducing the unstable period where provisioning is occurring. Since old services are still available, latency experienced by the client should be reduced since more replicas are available. Of course, this comes at the cost of available system resources.

6.2.3 Front-End Access Latency Evaluation

This testbed experiment measures the amount of time it takes for the front-end to parse a client's request and return a JSON of the appropriate edge nodes serving the requested models. Once again, we made use of the Python *time* module to measure the amount of time in milliseconds. We measured the response time of the front-end endpoint, which retrieves a list of all models, and the front-end endpoint, which returns a single model. We

Table 6.6: Testbed Front-End Access Latency for All Models Endpoint in Milliseconds

Number of Available Models	Latency (ms)
1	212.954
2	332.679
3	434.875

Table 6.7: Testbed Front-End Access Latency for Specific Model Endpoint in Milliseconds

Number of Available Models	Latency (ms)
1	103.695
2	112.181
3	154.776

collected measurements with one, two, and finally three models available in the system for both endpoints. For each set, we ran the test three times and averaged the resulting latency values. Additionally, any file access tasks related to storing the metrics were omitted from these measurements. Table 6.6 provides the results for retrieving all models. Table 6.7 shows the results for the endpoint providing one model.

Analysis of Results

From these results, we can see that the overall front-end access latency is low. Getting a list of models with only one model deployed in the system resulted in a value of 212 ms, which went up to 434 ms with three models. We can further see a significant increase in the front-end latency when getting a list of all models as the number of models increases. However, when the client requests one particular model, the latency values experienced do not vary drastically even when more models are deployed. In some tests, the latency experienced was as low as 83 ms, despite multiple models deployed on the system at the time. These results point to a specific part of the implementation design. In order to return the client the ideal endpoint for the model they requested, we must find every endpoint serving the requested

model and return the best one. The endpoint that returns a list of all available models must repeat this process once per model, whereas the endpoint that returns a single model only ever needs to operate once. Thus, the latency will increase for getting a list of all models but will vary an insignificant amount for getting a specific model. Despite this, the overall latency experienced by the client is still low, and the back-end's provisioning management processes ensure that only required models are present. This keeps the overall number of deployed models, and thus the front-end latency, low. It is also important to note that each client will not continually access these endpoints. Once a client requests a model or list of models, they will communicate with the node hosting the model directly without needing to request the front-end again. This behavior means that the front-end access latency is now acceptable for maintaining low latencies for clients.

Chapter 7

FUTURE WORK

The joint model provisioning and request dispatch solution for mobile inference presented in this thesis is a novel scheme for efficiently serving inferences at the edge. However, we see some opportunities for improvement in both the solution and the implemented system through potential future work.

As covered in Chapter 4, the NLP solver that forms the bridge between the front and back-end modules requires particular files to exist before it can run. These files contain data that must be known ahead of time, such as the execution time of a model on a node, the amount of RAM a model requires to be deployed, and the IP address of each node. Much of this information is hardcoded into the template files used to generate files for the NLP solver in the prototype implementation. Future work could improve the implementation of configuration files instead of hardcoded values, allowing for these values to be updated and modified. This would allow the system to dynamically handle the addition and removal of nodes and models without requiring the system to be brought down and restarted. The prototype implementation could be modified to periodically poll the file system to look for updated configuration files and automatically use them to update the internal configuration of the system. The underlying clustering system, Kubernetes, already supports dynamic changes in the composition of the cluster.

In Chapter 6 we measured the front-end response latency through a testbed experiment. As it is currently, the implementation requires that the operations for retrieving a single endpoint must be repeated once per model for retrieving a list of all models. Future work could also focus on optimizing this implementation to reduce the time taken to build a list of endpoints.

In our implementation, we created the choice for two different operating modes for the back-end service to prioritize latency or system resource balancing. Currently, the choice of which mode to use must be made when the system is invoked. One avenue for future work would be to design improvements to the solution architecture that define how the system could sense the system's current state and intelligently determine which mode to use during operation. Designing the system to switch intelligently can help avoid any potential problems from overloaded edge nodes and directly contribute to keeping client latency low.

Finally, one important area of focus for future work would be ensuring correctness through internal system oversight. The system is robust enough to deal with missing nodes or deployment files as it is currently implemented. However, it assumes that the values produced by the NLP solver and the latency values reported by clients are correct and make no attempt to verify or cross-check them beyond basic error-checking. Future work should focus on designing a method by which the system can analyze its overall functionality and state and compare that with the values coming in to determine if those values are accurate, and if not, to make the required changes needed to ensure correct operation.

Chapter 8

CONCLUSION

This thesis presented a novel solution for joint model provisioning and request dispatch for mobile inference on edge networks. We explored the reasoning behind the rise of edge machine learning, prior work in edge computing, and the logic behind focusing on latency when trying to efficiently provision inference resources on edge. We formalized the problem as a non-linear integer programming problem where we minimize the objective function. These results were used to define how and where we would place computing resources to minimize the latency experienced by clients proactively. We further defined a solution architecture that laid out our three-module solution, including the front-end service, dual-mode back-end service, and NLP solver collaborating in a semi-interconnected manner to perform predictive provisioning operations. This included exploring the solution architecture, depicting how the system was designed to be resilient and independent of specific implementation schemes. Not only did we propose a solution, but we also implemented a prototype system to verify the feasibility of our solution to the problem statement and ran that solution on real-world edge computing hardware. In contrast to existing solutions that migrate containers or prioritize balancing the system load, our solution prioritizes latency first and foremost. Our solution is unique because it maintains awareness of current latency levels and system state to perform intelligent provisioning.

Through a series of simulation and testbed experiments, we showed that our solution consistently reduced latency values compared to a solution that did not perform joint provisioning. We showed that our solution produced smaller latency values in either operation mode, with changing counts of requests and nodes. Through testbed experiments, we depicted how our system reacted to varying request arrival rates and the latency values reported

in them and intelligently modified the deployments on each node to handle the predicted load. Our testbed experiments further showed how the system's front-end could scale well and handle requests with minimal response latency. Overall, we demonstrated that the novel solution presented in this thesis keeps processing time and latency low to avoid consuming valuable resources and serve clients as quickly and efficiently as possible.

BIBLIOGRAPHY

[1] AMPL Optimization Inc. AMPL for Students. https://ampl.com/products/ampl/ampl-for-students/.

[2] Claudio Cicconetti, Marco Conti, and Andrea Passarella. Uncoordinated access to serverless computing in MEC systems for iot. *Comput. Networks*, 172:107184, 2020.

[3] COIN-OR Foundation. COIN-OR Suite Documentation. https://coin-or.github.io/index.html.

[4] Vittoria De Nitto Personè and Vincenzo Grassi. Architectural issues for self-adaptive service migration management in mobile edge computing scenarios. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 27–29, 2019.

[5] Labhesh Deshpande and Kaikai Liu. Edge computing embedded platform with container migration. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation, SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI 2017, San Francisco, CA, USA, August 4-8, 2017*, pages 1–6. IEEE, 2017.

[6] Bin Gao, Zhi Zhou, Fangming Liu, and Fei Xu. Winning at the starting line: Joint network selection and service placement for mobile edge computing. In *2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019*, pages 1459–1467. IEEE, 2019.

[7] Helmut Hoffer von Ankershoffen. helmuthva/jetson-xavier-tensorflow-serving. https://hub.docker.com/r/helmuthva/jetson-xavier-tensorflow-serving.

[8] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[9] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. Edge computing: A survey. *Future Gener. Comput. Syst.*, 97:219–235, 2019.

[10] kubernetes.io. Kubernetes Client Library - Python. https://github.com/kubernetes-client/python/.

[11] Lele Ma, Shanhe Yi, and Qun Li. Efficient service handoff across edge servers via docker container migration. In Junshan Zhang, Mung Chiang, and Bruce M. Maggs, editors, *Proceedings of the Second ACM/IEEE Symposium on Edge Computing, San Jose / Silicon Valley, SEC 2017, CA, USA, October 12-14, 2017*, pages 11:1–11:13. ACM, 2017.

[12] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Commun. Surv. Tutorials*, 19(3):1628–1656, 2017.

[13] Sumit Maheshwari, Dipankar Raychaudhuri, Ivan Seskar, and Francesco Bronzino. Scalability and performance evaluation of edge cloud systems for latency constrained applications. In *2018 IEEE/ACM Symposium on Edge Computing, SEC 2018, Seattle, WA, USA, October 25-27, 2018*, pages 286–299. IEEE, 2018.

[14] NVIDIA Corporation. JetPack SDK. https://developer.nvidia.com/embedded/jetpack.

[15] NVIDIA Corporation. Jetson Modules. https://developer.nvidia.com/embedded/jetson-modules.

[16] Samuel S. Ogden and Tian Guo. Mdinference: Balancing inference accuracy and latency for mobile applications. *CoRR*, abs/2002.06603, 2020.

[17] Tao Ouyang, Zhi Zhou, and Xu Chen. Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing. *IEEE J. Sel. Areas Commun.*, 36(10):2333–2345, 2018.

[18] Pallets. Flask Documentation. https://flask.palletsprojects.com/en/2.0.x/.

[19] Arnaldo Pereira Ferreira and Richard Sinnott. A performance evaluation of containers running on managed kubernetes services. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 199–208, 2019.

[20] Carlo Puliafito, Enzo Mingozzi, Carlo Vallati, Francesco Longo, and Giovanni Merlino. Companion fog computing: Supporting things mobility through container migration at the edge. In *2018 IEEE International Conference on Smart Computing, SMARTCOMP 2018, Taormina, Sicily, Italy, June 18-20, 2018*, pages 97–105. IEEE Computer Society, 2018.

[21] Carlo Puliafito, Carlo Vallati, Enzo Mingozzi, Giovanni Merlino, Francesco Longo, and Antonio Puliafito. Container migration in the fog: A performance evaluation. *Sensors*, 19(7):1488, 2019.

[22] Python Software Foundation. Python - Unit Testing Framework. https://docs.python.org/3/library/unittest.html.

[23] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey and benchmarking of machine learning accelerators. In *HPEC*, pages 1–9. IEEE, 2019.

[24] Saurav Sthapit, John Thompson, Neil M. Robertson, and James R. Hopgood. Computational load balancing on the edge in absence of cloud and fog. *IEEE Trans. Mob. Comput.*, 18(7):1499–1512, 2019.

[25] TensorFlow - Google Developers. TensorFlow Serving Documentation. https://www.tensorflow.org/tfx/guide/serving.

[26] The Kubernetes Authors. Kubernetes Documentation. https://kubernetes.io/docs/home/.

[27] The Kubernetes Authors. What is Kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[28] Liang Tong, Yong Li, and Wei Gao. A hierarchical edge cloud architecture for mobile computing. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*, pages 1–9. IEEE, 2016.

[29] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106(1):25–57, 2006.

[30] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 1049–1062. USENIX Association, 2019.

[31] Xingzhou Zhang, Yifan Wang, and Weisong Shi. pcamp: Performance comparison of machine learning packages on the edges. In *HotEdge*. USENIX Association, 2018.

[32] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proc. IEEE*, 107(8):1738–1762, 2019.

Appendix A

SOFTWARE DEVELOPMENT AND ENGINEERING PERSPECTIVE

A.1 Development Process

During this thesis, I followed a standard sprint-based method. Each week would be focused on designing and implementing a specific feature or collecting a particular type of data. This would be punctuated by a weekly meeting in which we would discuss everything that had been worked on up to that point and plan for the next week's work. This workflow worked well for me. It provided an impetus to keep on schedule while also offering numerous opportunities for discussion and feedback. We also used a private Github repository for version control and various standard development tools such as VSCode and Postman. Testing the system was accomplished through the use of Python unit test frameworks [22] as well as by managing Kubernetes resources in the command line and then verifying expected behavior when using the system.

A.2 Experience Gained

This thesis was my first time working with Kubernetes, Docker, TensorFlow Serving, Ipopt, AMPL software, or Nvidia Jetson hardware. While I had prior experience with Python and REST APIs, I had never specifically worked with Flask. Working on this thesis allowed me to learn how to work with these popular industry-standard tools and exposure to mathematical tools that are slightly outside what I have typically worked on. I also had the chance to apply other aspects of software engineering learning in class. The creation of design documentation before implementation is an important process that I learned during my education. I applied it during this work. Figure A.1 depicts a component diagram representing the implemented

research system. Furthermore, I was able to make use of Python unit testing frameworks to test the implementation as well as to build the simulations used to collect the simulation and testbed experimental data presented in Chapter 6. Fundamentally, this thesis was instrumental in helping me learn how to apply a theoretical design to build a functioning implementation, an instrumental skill for future endeavors in software engineering.

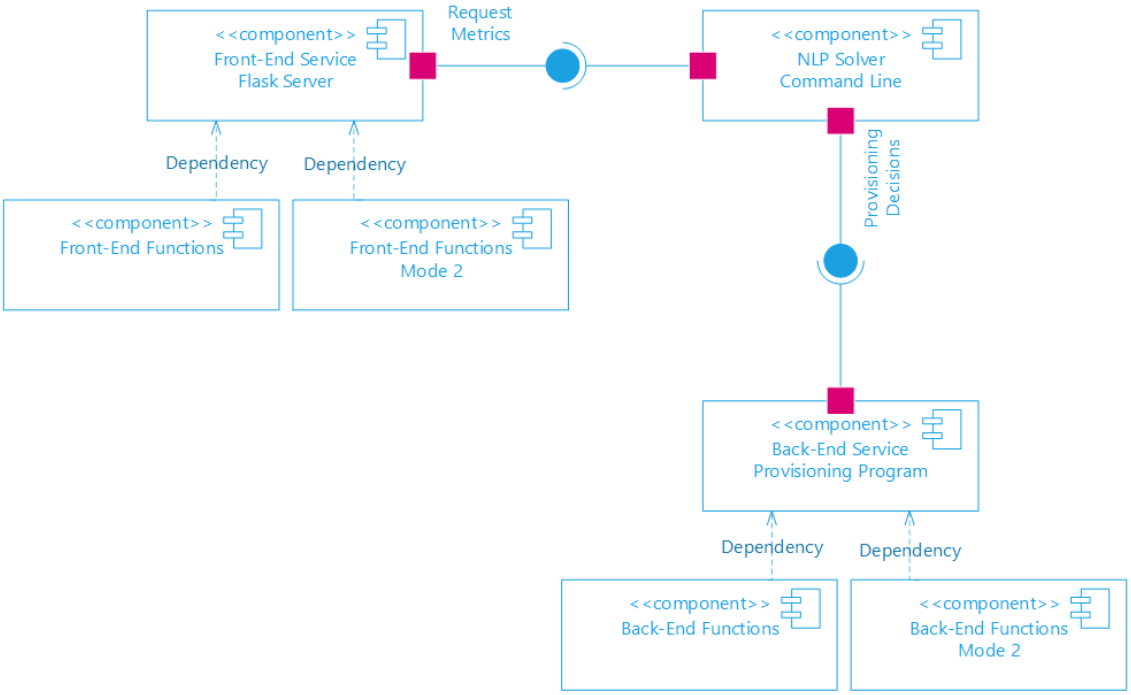

Figure A.1: System Implementation Component Diagram