Machine Learning Model Management using Cloud-Native Technologies for IoT

Hieu Nguyen

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 31.12.2020

Supervisor

Senior university lecturer Vesa Hirvisalo

Advisors

M.Sc. Jussi Judin

Dr. Sumanta Saha

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Hieu Nguyen

Title Machine Learning Model Management using Cloud-Native Technologies for IoT

Degree programme Master's Programme in Computer, Communication and Information Sciences

Major Machine Learning, Data Science and Artificial Intelligence **Code of major** SCI3044

Supervisor Senior university lecturer Vesa Hirvisalo

Advisors M.Sc. Jussi Judin, Dr. Sumanta Saha

Date 31.12.2020 **Number of pages** 53 **Language** English

Abstract

Thanks to many breakthroughs in neural network techniques, machine learning is widely applied in many applications, including IoT. However, the challenge is to run a machine learning pipeline in production without introducing a huge technical debt. Edge computing and IoT have become popular thanks to the advancements in networking and virtualization. Cloud-native technologies, in general, and Kubernetes, in specific, can manage a large scale cluster to run a massive fleet of containers. Thus, running machine learning pipelines on Kubernetes for IoT is a great match.

However, any abstraction layer would introduce overhead. The question is how large the overhead is to run a machine learning model workflow on Kubernetes. Also, cloud-native technology enables easy management of Edge computing resources. The question is how efficient it is to update machine learning models used by a serving server at an Edge node.

To address the first research question, we measure the execution time overhead of Kubernetes jobs compiled from Jupyter notebook running an E2E machine learning application. The experiment is conducted with different dataset sizes and machine learning model sizes to investigate how they affect the overhead. To address the second research question, we measure and compare the machine model uploading time to an object-storage service in two setups: deployed as a Kubernetes service and a service running in a virtual machine. The measurements are conducted with artificial network latency to simulate serving machine learning model at the Edge.

In our experiments, we observe the overhead introduced by the Kubernetes and the pipeline applications; however, the overhead becomes less significant as working with bigger datasets. Because of the expensive model training job, the overall overhead is much less comparing the training job computation time. In the context of update machine learning models at the Edge node, to upload the model to an object-storage Kubernetes service shows no significant difference to another service serving at a virtual machine without Kubernetes. On the other hand, the experiments also show that cloud-native technologies offer excellent abstractions and automation to manage cloud resources for containerization.

Keywords cloud-native, machine learning, iot, model management, container

Preface

I want to thank my supervisor Dr. Vesa Hirvisalo and my instructor M.Sc. Jussi Judin for their excellent guidance and patience with me throughout the thesis work. I also thank Dr. Sumanta Saha for his topic suggestion and early guidance. Furthermore, the work would not be possible without Ericsson's computing resources. Moreover, I appreciate my friend Teemu Kokkonen to revive my motivation to finish this thesis. Finally, I want to thank my parents Nguyen Phu Trung and Nguyen Thi Bach Yen to support my education in Finland.

Otaniemi, 31.12.2020

Hieu Nguyen

Contents

Abbreviations

5G	5th generation
API	application programming interface
BGP	Border Gateway Protocol
CI	continuous integration
CD	continuous delivery
CPU	central processing unit
DNN	deep neural network
E2E	end-to-end
GB	gigabyte
i.i.d	independent and identically distributed
IoT	Internet of things
K8S	Kubernetes
kB	kilobyte
KF	Kubeflow
MB	megabyte
MEC	mobile Edge computing
ms	millisecond
MSE	mean squared error
RAM	random access memory
RWM	Read-Write-Many
RWO	Read-Write-Once
LB	load balancing
PV	persistent volume
PVC	persistent volume claim
SVC	service
TF	Tensorflow
TF-Serving	Tensorflow-Serving
UI	user interface
VM	virtual machine

1 Introduction

1.1 Overview

Internet of Things is an emerging market thanks to the recent advancements in networking, cloud, and machine learning technologies. High capacity mobile network, such as 5G, enables the concept of Mobile Edge Computing (MEC), where the micro-data center is located outside the core cloud and closer to the devices. Furthermore, container technology allows an easy workflow to create a software package with all dependencies and deploy it in production. As a natural development, cloud orchestration technology, such as Kubernetes, is built on top of the container technology to abstract away the underline networking to reduce the manual configuring labor. Besides, machine learning has gained the industry trend thanks to its capability to perform highly accurate data prediction in several applications such as computer vision, speech recognition, and natural language processing. Thus, machine learning enables complex automation, which is commonly called smart application.

Even though machine learning is studied extensively in academia, to build and deploy a machine learning application in production is not easy. One of the problems is to manage the life cycle of the trained model: how to schedule the model training workload and deploy a trained model to a serving server efficiently without resulting in a huge technical debt. In the industry, machine learning systems have been built on a massive scale, such as Uber's machine learning system Michelangelo [1] or Youtube's recommendation system [2]. However, similar to any massive software system, a machine learning system may also contain huge underlying technical debts if not designed carefully. Sculley et al. 2015 [3] report the major challenges:

- **Broken abstraction isolation** Because of the nature of machine learning application, its internal logic always depends on the external data that it serves. This may result in difficulty to keep the abstractions isolated or, in other words, the CACE problem: **Change Anything Changes Everything** due to the components' tight dependencies.
- **Data dependency problem** Machine Learning system may suffer from unstable data dependency when a component receives inconsistent input from another system.
- **Feedback loop** One example of this problem is video recommendation system recommends new videos based on earlier recommended watched videos. These offering videos may end up to be repetitive or center around inappropriate topics.
- **Configuration management** Configuration management may happen to any massive software project, meaning managing the system settings and deployment parameters goes out of control.

On the other hand, a machine learning application is commonly developed together by data scientists and software engineers. Data scientists are often more competent

in the mathematical and computation side, while software engineers are often more competent in the networking and computing resources side of the application. The problem is to build a framework that can help two communities to work in harmony, meaning that a developer doesn't need to study all the gritty details of the other side to do their job efficiently.

1.2 Research Problem

Kubernetes ecosystem, in specific and cloud-native technologies in general, provides a framework to manage networking and computing resources to orchestrate the container efficiently. Containerization has proven to be a great way to package software, thus suitable to package machine learning applications. However, convenience typically comes with a cost. Two questions are raised in this thesis. Do we inherit a huge overhead cost to manage the machine learning model using cloud-native technologies? Also, how do cloud-native technologies perform in updating the machine learning model serving at the Edge? To address the first question, we set up a Kubeflow pipeline to have an E2E machine learning workflow to measure the extra computation time to study the overhead introduced by Kubernetes and Kubeflow applications. To address the second question, we measure the uploading time for a new machine learning model to an object-storage with artificial latency to the simulated Edge node to study its behavior.

1.3 Results

The author of this thesis deploys a Kubernetes cluster on an OpenStack cloud, create a Jupyter notebook to run an E2E machine learning application, and compile it into Kubernetes jobs using the mentioned Kubeflow Pipelines [4] and Kubeflow Kale [5]. From a public dataset, he generates datasets with different sizes. He also creates neural network models with different sizes by varying the hidden layers' numbers and sizes. Then, he measures the execution time overhead running the Kubeflow pipeline with different configurations to analyze the trends when the dataset and model size change. To measure the model uploading time, the author of this thesis deploys two setups: uploading the models from a pod to a Kubernetes Minio [6] service and uploading the models from a VM to a Minio service running on another VM. Then, he measures the uploading with different simulated network latencies using Netem [7].

From the experiments, we learn that cloud-native technologies provide great abstractions for resource management and networking to manage different parts of a machine learning model management system easily. Porting a Jupyter notebook to run on Kubernetes, we observe the additional overhead. It becomes less significant as we work on a bigger dataset because of the expensive training job. Our experiment to upload a new model from a Kubernetes job and a MinIO object-storage service on Kubernetes display normal behavior as setting up a data transfer connection from a VM to a VM.

1.4 Thesis Structure

In Section 2, we review the related topics to this work so that the thesis is self-contained, which includes IoT 2.1, Edge computing 2.2, machine learning 2.3, virtualization technology 2.4, container management system 2.5 and machine learning model management 2.6.

In Section 3, we summarize the interesting related projects at the time of writing to highlight where this project positions and what is its contribution to this research area.

In Section 4, we explain the author's approaches to answer the research problems, which consist of two parts: **(1)** measure the additional overhead of running Jupyter notebook as jobs on Kubernetes in Section 4.1 and **(2)** measure the characteristics of updating machine learning models at a node managed by a Kubernetes cluster at the Edge in Section 4.2.

In Section 5, we explain in details the author's work to provision the infrastructure in Section 5.1 and bootstrap a Kubernetes cluster in Section 5.2. In Section 5.3, we explain the author's work to develop a Jupyter notebook to produce machine learning a sensible model for an IoT dataset, which is then served as the base to generate testing models and datasets. In Section 5.3.3, the author presents the Kubeflow pipeline generated from the Jupyter notebook to run the experiments.

In Section 6, we present the results and analyses of the experiments, applying the explained methodologies.

Finally, in Section 7, we recap the results and provide our view about the project.

2 Background

In this section, we provide brief introductions to different topics concerned in this thesis. In Section 2.1, we give an overview of the Internet of Things (IoT) and its enabling elements. In Section 2.2, we discuss Mobile Edge Computing (MEC) and the reasons it is suitable for IoT. In Section 2.3, we introduce machine learning by describing problems that machine learning aim to tackle. In Section 2.4, we discuss the relevance of hypervisor and containerization virtualization in software life cycle management. In Section 2.5, we introduce the container orchestration system and the elements of Kubernetes, the most popular container orchestration system at the time of writing. Finally, in Section 2.6, we discuss the motivation to have a machine learning model management system and its requirements.

2.1 Internet of Things (IoT)

Internet of Things (IoT) is defined as the network, on the Internet, of connected devices, such as sensors and smartphones capable of collecting and sharing data. The data collected at the servers are processed to extract the semantics to serve applications in domains such as transportation & logistics, healthcare, smart environments, and other futuristic applications [8].

Al-Fuqaha et al. 2015 [9] have suggested that IoT was enabled by various technological advancements in different areas, which can be grouped into elements:

- **Identification:** In an IoT application, we need to have an identification and addressing scheme for all of them within the network. Electronic product code (EPC) and ubiquitous code (uCode) [10] have been used for object's IDs. IPv4 [11] and IPv6 [12] has been used for in Internet communication. 6LoWPAN [13] provides the IPv6 header compression scheme to enable IP communication in a low-power wireless network.

- **Sensing:** This element is where the data is generated. Sensors, actuators, the camera had been used even before the term Internet of Things was coined. The main advancement is the reduction in size, weight, and power consumption of those devices. In an IoT application, a single-board computer (SBCs) is typically used to connect to sensing devices, process and send data using TCP/IP stack to servers. Common commercial SBC products may include Raspberry Pi [14] and Arduino Uno [15].

- **Communication:** In an IoT application, heterogeneous devices with the low-power constraint often operate on a lossy network. There is a wide range of protocols to support different radio requirements such as WIFI, Bluetooth, LTE, RFID, NFC, and Ultra-wide bandwidth (UWB).

- **Computation:** In an IoT application, the computation can happen either at the edge or at the core of the network. At the edge, we can have microcontrollers running embedded Operation System such as TinyOS [16], RiotOS [17], and

Android [18]. To run computing servers in the cloud, we can have machines running GNU/Linux [19] in bare-metal, VMs on the top of a hypervisor or in containers managed by a container orchestration system such Kubernetes [20].

- **Semantics:** Extracting the semantics out of the IoT data means to infer the relationship and knowledge of the generated data. Many proposals have been introduced to support this task, such as Resource Descriptor Framework (RDF) [21] and Web Ontology Language (OWL) [22], which are parts of Semantics Web. But it has not caught the trend since it does not provide economic incentives to most vendors. Another approach is to use machine learning, which has hyped up the industry in recent years thanks to many breakthroughs in Deep Learning [23].

2.2 Edge Computing

In Section 2.1, we discussed that the computation may happen either at the center or at the edge of the cloud. At the edge of the cloud, the data is processed quickly because we don't have to send it through multi hops of the network. However, the devices here are typically resource-constrained, thus are limited to light-weight computing capability. At the center of the cloud, we afford more powerful computation and complex pipeline, but the data needs to travel for a longer time before it gets processed. Naturally, we can think of a solution where we move part of the computation pipeline near the edge of the cloud to reduce the load from the central servers and increase the response time. That is the idea of Edge Computing. Premsankar et al. 2018 [24] report a list of reasons that Edge Computing architecture is suitable for IoT:

- **Low latency communication** Real-time interactions between the data collection stream and decision-making response is required in applications such as smart traffic (involved connected vehicles), smart health (critical monitoring device)...
- **Geographical distribution** Using a network of sensors, which can be far from the central server, enables applications such as smart agriculture and smart weather prediction. Edge computing enables offloading the computation from the servers at the central cloud to the servers near to the devices. Thus, this process helps reduce the latency to serve the client.
- **Device mobility** Easy new device migration is featured in some application such as the smart home. Edge computing enables the deploy additional virtualized resource to serve the new device's traffic.

Mahdavinejad et al. 2018 [25] report that along with machine learning and data mining techniques, IoT technology realizes many intelligent applications at the Edge such as smart traffic, smart health, smart weather prediction, smart human activity control, and smart agriculture.

2.3 Machine Learning

Pattern recognition in a dataset has been an important problem in many scientific areas to discover laws of nature or verify a theory. Since the age of the computer, the field of machine learning concerns with using the computer algorithm to recognize the patterns in data to perform tasks such as classifying data or predicting the label of new data points. In this section, we introduce three typical machine learning problems: classification, regression, and clustering.

Firstly, classification is a supervised learning technique to classify the data points. Supervision means that we train the model with labeled training data. The predicting data should be in the same space as the training data. For example, Figure 1 shows the classifications of flower types based on two features: sepal width and length from the IRIS dataset (Fisher 1936 [26]) using Multimodel Logistic Regression.

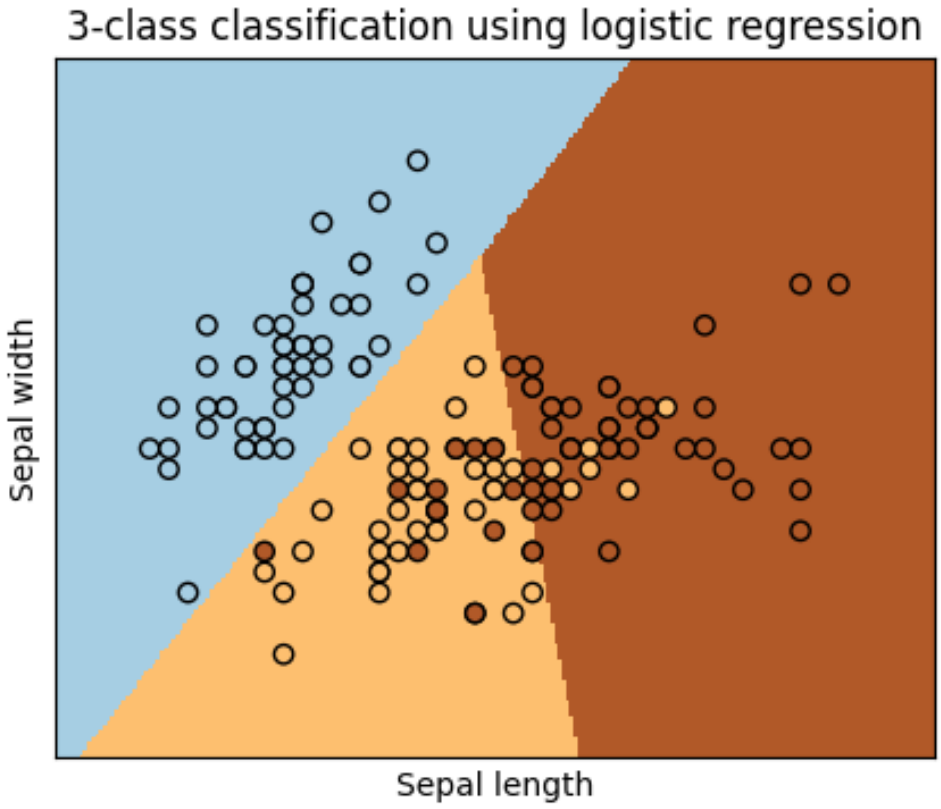

Figure 1: An example of a machine learning classification problem. In this example, we have 3 classes to classify from the data points, depicted as 3 colors. The boundaries between classes are lines between the cut areas. The figure is taken from the example by Varoquaux [27].

Secondly, regression is a supervised learning technique to predict the value of

unseen data points. For example, Figure 2 shows an example of regression trained using Gaussian processes methods. The blue dots represent the means of the simulated training observations. The blue bars represent the variances of the observations. The red line shows the prediction of the trained model. The green area shows the uncertainty of the prediction.

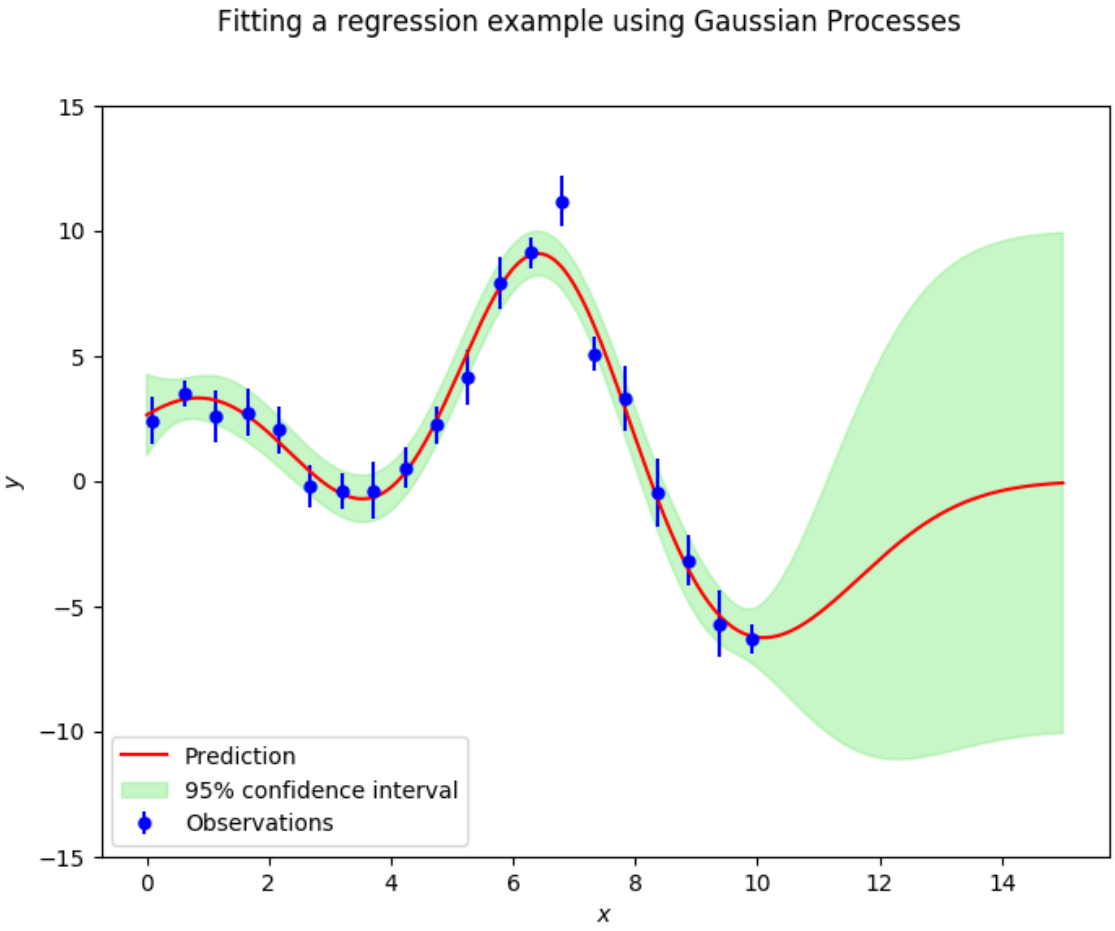

Figure 2: An example of a machine learning regression problem. The code is originally written by Dubourg et al. [28]. The author modifies it to emphasize the uncertainty of the prediction. Each blue dots represent the means of the observations. The blue lines show the variances of the observations. The red lines show the predictions. The green area shows the uncertainty of the predictions.

Thirdly, clustering is an unsupervised learning technique to categorize the data points into groups. Compare to classification, to cluster a dataset, we don't have the labels of the data points as training information. The clusters are determined by the characteristics of the data points such as how close they are to each other using a specified metric such as Euclidean distance. For example, Figure 3 shows an example of clustering a dataset into four groups: red, yellow, green, and pink using the K-mean algorithm.

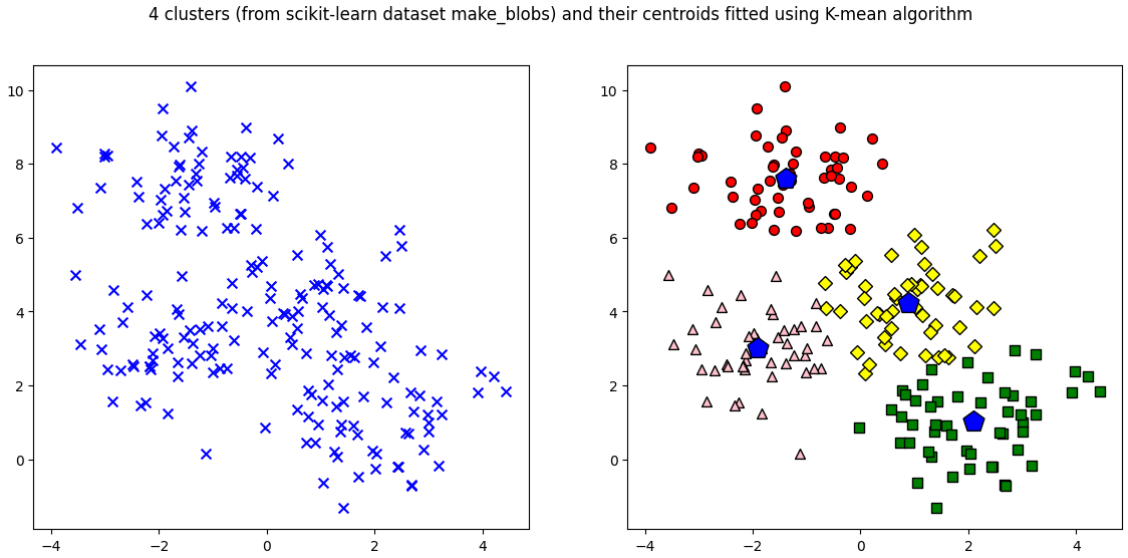

Figure 3: An example of a machine learning clustering problem. The data is generated from **sklearn.datasets.make_blobs** [29]. The data can be classified into 4 classes depicted with different markers and colors. The blue pentagons show the centroid of the clusters.

The examples are chosen to be very simple. In a real application, the data can be much higher dimensional, and its structure can be much more complicated. Thus, there are advanced techniques to deal with such problems. Pattern Recognition and Machine Learning by Bishop [30] is a classic overview of different topics of machine learning.

2.4 Virtualization Technology

In the context of software delivery, a commercial software application is typically delivered as a package. The reason is that it is not economical for non-trivial software to be written from scratch without linking or using other libraries. The package ensures that all dependencies are available during the installation and the runtime. Another reason is that each software delivery should be tested carefully, either through an automatic pipeline (Continuous Integration) or manual verification. A software package is a way to encapsulate and tag each of those testing cycles.

In cloud orchestration, the software may be packaged as a VM image or a container image.

2.4.1 Hypervisor and Container

One of the key technological enablers of Edge Computing is virtualization. In this technology, one physical machine can host multiple independent applications. One way to look at is that multiple tenants can pool dynamically resources from the same provider. This means this technology needs to provide a mechanism to isolate the

resources between tenants to ensure the availability, security, and quality of service. Bernstein 2014 [31] implies that two major virtualization technologies in the market are hypervisor and container.

1. **Hypervisor:**

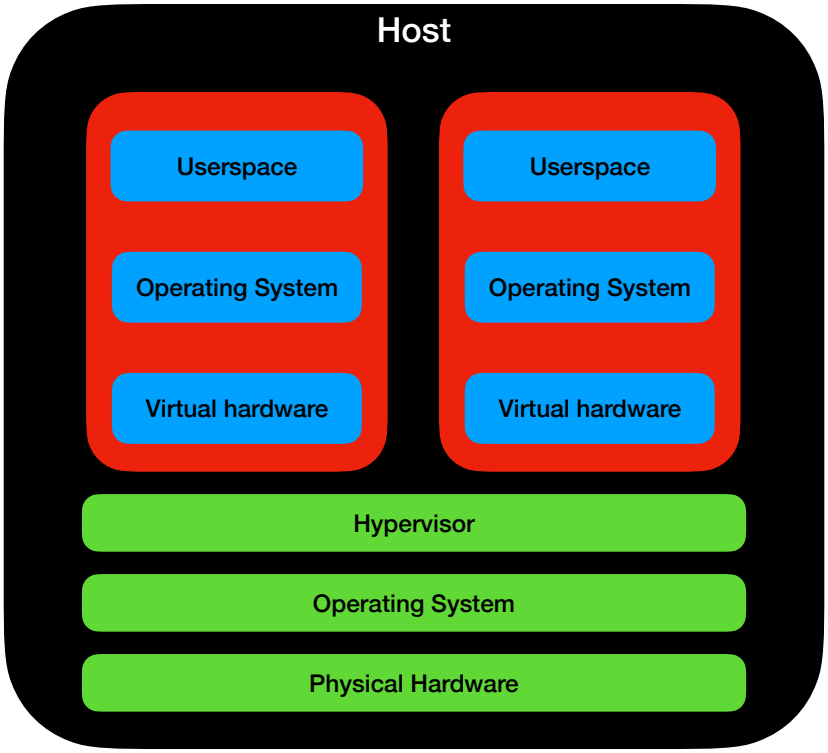

Figure 4: A hypervisor scheme. Each VM uses the virtual resources provided by the Hypervisor.

Thanks to modern hardware and operating system support, hypervisor is the software/firmware that enables creating Virtual Machines (VMs). A native or bare-metal hypervisor (type 1) interacts directly with the physical hardware. On the other hand, a hosted hypervisor (type 2) interacts with the physical hardware indirectly through an operating system layer.

A VM running under a hypervisor is often called a guess machine, which is run with emulated/virtual hardware and an isolated OS (see Figure 4). This level of abstraction allows the users to package their applications along with all dependencies built on a target OS. This package is often called an image, which can be easily deployed in another system with the same supported hypervisor.

Eder 2016 [32] reports that modern hardware also allows VMs to access the host's CPUs in an unprivileged mode. This feature provides performance

without compromising the security of the host machine. The virtual machine also adds another layer of security because the scope of resources exposed to the attackers is limited.

Bernstein 2014 [31] reports some examples of hypervisors in the market including ESXi (Enterprise VMware), Hyper-V (Microsoft), XEN hypervisor (developed by University of Cambridge & then supported by Linux Foundation) and KVM (Kernel-based Virtual Machine), which was started by Qumranet and since included as part of Linux Kernel.

2. **Container:**

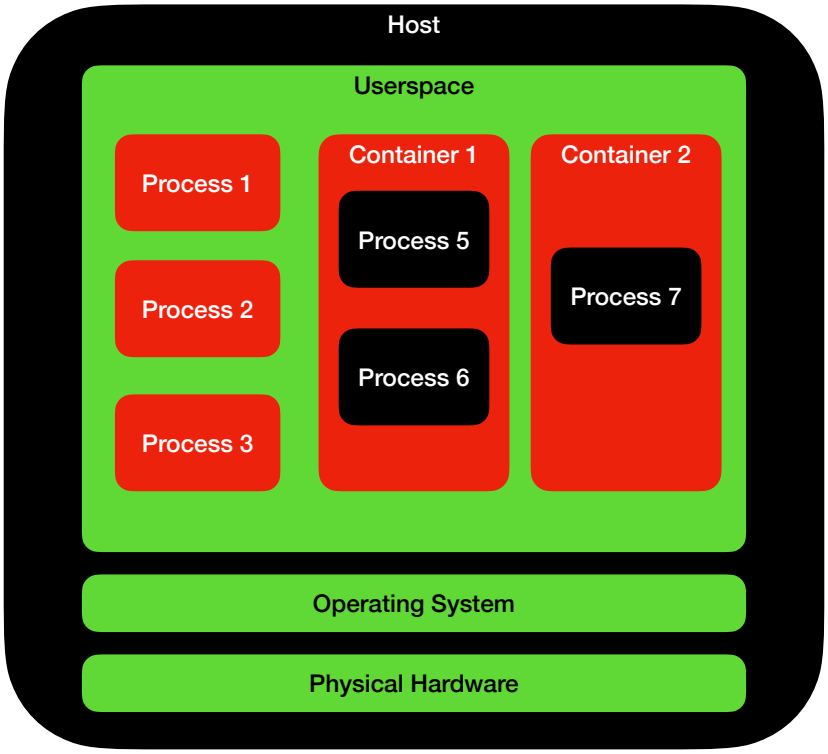

Figure 5: A container scheme. In this Figure, we can see containers can start processes as any other process in the Operating system. This is why it is more light-weight than a full-ledge VM.

Even though the hypervisor technology provides excellent isolation for cloud applications, its overhead may result in a long booting time and a huge cost in configuration management, thus making it difficult to maintain. To overcome such caveats, containerization virtualization is introduced as a lightweight alternative to the hypervisor in the hope to improve cloud applications' life cycle management.

The key difference is that containers' processes run in the host's userspace (see Figure 5), thus does not require to create of emulated hardware and boot up a whole OS. Comparing to a VM-based application, a container image packaging the application has a smaller size and requires less time to start. Eder 2016 [32] reports that the container's processes' isolation is realized by kernel's support including chroot (changing root), namespacing, cgroups (control group), and MAC (Mandatory Access Control).

2.4.2 Docker

Docker, an open-source project developed by Docker Inc., is a widely-adopted implementation of containers. Docker simplifies the workflow for the users to build, distribute, and run an application without going deep into lower levels of the Linux kernel. Docker is implemented using server-client architecture (see Figure 6).

The developer uses the Docker client, which has memorizable commands, to submit tasks with containers such as building (docker build) and running (docker run).

The Docker client interacts through the UNIX socket using Docker daemon to perform the container-related tasks.

The Docker daemon also communicates with a Docker registry, either private or public, to pull or publish Docker images.

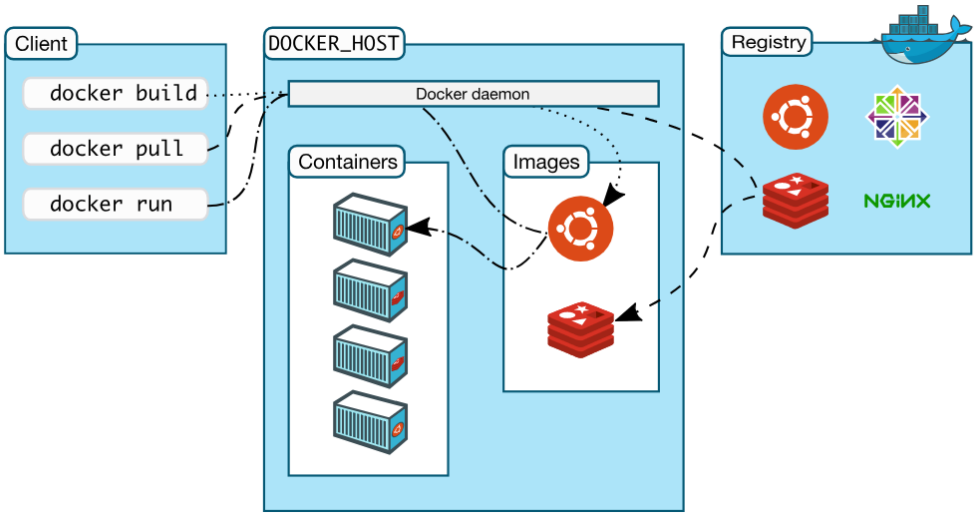

Figure 6: Docker Architecture [33]. Docker host is where the container is run. The Docker images may be fetched from a public Docker registry and stored in a local registry. The process is handled by the Docker daemon. Docker provides a client to interact with the daemon.

2.5 Container Management System

2.5.1 Container Orchestration

Bias 2016 [34] suggests two approaches to manage cloud computing resources: pet vs. cattle. The pet approach means that the servers are treated as unique entities, which are supposed not to go down and to run forever. To overcome disasters, the servers typically run with redundancy, meaning having two or more instances all the time. On the other hand, the cattle approach means that an array of servers are assumed to go down at any time without impacting the whole system. Restarting from failures and data replication are managed by automation.

We can see the analogy that maintaining a bare-metal server or a VM is similar to raise a pet. The bare-metal server or a VM is supposed to run all the time because it takes a huge amount of time to spin up a new instance. Typically, people run multiple instances of servers or VMs to prevent service downtime during a failure or an upgrade. As discussed in Section 2.4.1, this approach inherits a huge configuration management cost, enormous system overhead, and the inflexibility to redeploy. These architectural debts make the pet approach's life cycle management difficult.

To overcome such difficulty, the industry has moved to manage their applications using containers thanks to its lightweight approach. Without the huge overhead of a VM, a containerized application can be redeployed much quicker, thus enables much better life cycle management. However, the next challenge is how to manage containers in massively distributed systems running many hosts. This is the reason why another technology, namely container orchestrator, emerges in the industry.

Hausenblas 2018 [35] suggests that a container orchestrator are responsible for:

- **Organizational primitives** provides sensible/memorizable units to be managed by the administrator.
- **Scheduler** schedules containers to run in the system's machines.
- **Health checking** is crucial to orchestrator automation. This provides the status of containers and executes automation tasks such as starting to receive traffic or restart on failure.
- **Service discovery** provides a systematic way for services to discover each other such as using DNS (domain name service).
- **Upgrades** provides the mechanism to update configuration and restart container using different strategies such as canary or A/B deployment.
- **Scaling** provides the mechanism to scale out the services on demand.

2.5.2 Kubernetes

Kubernetes [20] is an open-source container orchestrator maintained by Google and the community. Kubernetes was predated by two other container management system namely Borg and Omega [36].

Borg was developed to manage container-based long-running services and batch jobs on sharing machines, which was enabled by the container support in the Linux kernel. Borg was used widely inside Google and provides many features such as resource requirement prediction, dynamic configuration update of running processes, etc. However, it has become so sophisticated and requires a too broad scope of expertise to maintain.

Thus, to provide a more principled and consistent system, Google created another container orchestration system named Omega. Omega introduced many innovations such as storing the cluster's state in a centralized transaction-oriented store and breaking the responsibilities of Borgmaster to smaller components.

As Google entered the public-cloud market with Google Cloud Platform (GCP) [37], they needed to provide the customer a framework to easily deploy their applications on GCP and also on other public-cloud platforms such as Azure [38] or AWS [39]. Thus, Google started Kubernetes as an open-source project.

Burns et al. 2016 [36] report that Kubernetes was influenced by both Borg and Omega such as having a highly available key-value store. However, it does not expose the cluster control API's to all the components in the control plane like Omega. Kubernetes provides versioning, semantics, and access control policy to fit the variety of the users' needs for their applications.

In Kubernetes, bare-metal or virtual machines in a cluster are categorized into two groups: master and worker.

Master nodes serve as the control plane of the cluster. They provide a variety of services:

- **kube-apiserver** provides the REST API's to control the cluster.
- **etcd** is a highly-available database storing the states of the cluster and its components.
- **kube-scheduler** schedules which host a newly computing unit (pods) is locates.
- **kube-controller-manager** provides different kinds of controllers such as node controller (actions on node's status), replication controller (number of replications of organizational units), endpoint controller (linking services with pods), and so on.
- **cloud-controller-manager** provides the interface to the underlying public-cloud provider.

Node machines provide the computing services to the cluster. We can also configure the master component to run computing services, but this is not recommended. Worker services include:

- **kubelet** is run in every node to make sure that a scheduled container running in the right node.
- **kube-proxy** provides traffic forwarding and abstracts away the underlying networking.

- **container runtime** provides the runtime to run containers. Supported container runtime communicates with Kuberbernetes through Container Runtime Interface (CRI). This is the effort to make Kubernetes available to different container runtimes such as Docker, containerd, cri-o and rktlet.

The most powerful feature of Kubernetes is, perhaps, its extendability. Users can write their own add-ons such as networking plugin to fit their system's requirements [40].

2.6 Machine Learning Model Management

A typical machine learning pipeline involves collecting and processing the data, training, validating, and serving the model. Figure 7 depicts a typical machine learning application's components. In research, the main focus to correctly interpret the data and to find the best model fitting the data. Therefore, organizing the code in a clean architecture to avoid technical debts is typically ignored.

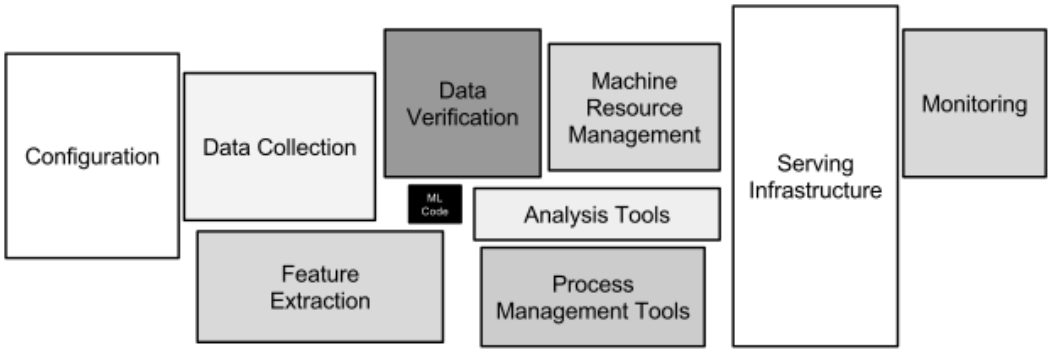

Figure 7: Machine Learning application components suggested by Sculley et al. 2015 [3]. We can observe that even though the machine learning algorithm is the most important part of the system, to run a machine learning application in production, many more components are required.

However, Schelter et al. 2018 [41] imply that a commercial machine learning application has more technical requirements than just obtaining the best model. Building infrastructure around the machine learning pipeline requires well-thought engineering efforts to avoid the technical debts discussed above. Furthermore, the machine learning application may need to be updated as the training data or its assumptions change. Thus, the developers may face many challenges such as **metadata tracking**, where the information about the data needs to be handled automatically instead of relying on error-prone manual efforts; **model definition**, where the model's inputs, outputs & its algorithm may be updated; **model validation**, where the trained models need to be tested properly before serving at large; **multi-language codebase** where the developer may need to develop toolings to support different frameworks. Figure 8 summarizes the loop of how a machine learning model is created and updated.

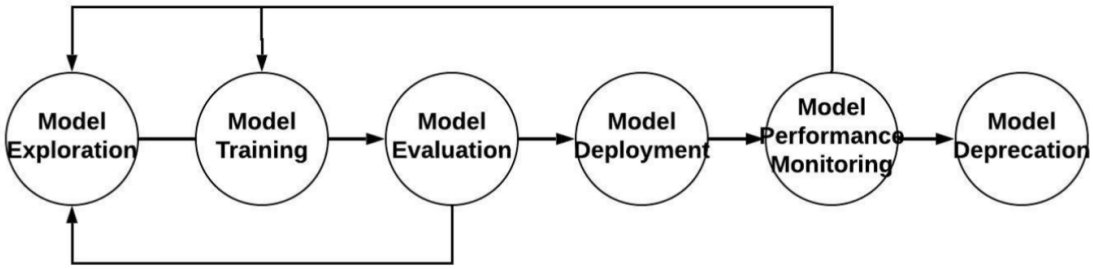

Figure 8: Machine Learning Model life cycle suggested by Sun et al. 2020 [42]. The model starts with exploration meaning trying with different algorithms and data processing. Then the model is trained to fit the dataset. This is typically the most expensive task. Then it is evaluated to check the performance generality on new data points. This can be fed back to the exploration step to define a better model. The model is then deployed and monitored in production. The feedback here can go back to define a better model or improve the training process. In the end, if a new model is introduced, the old model can be deprecated.

Therefore, the industry needs a systematic way to manage the machine learning model. The machine learning model management system Gallery [42] at Uber provides a list of features that a typical machine learning model management should have: **saving & loading models**, **providing metadata notation**, **supporting searching**, **serving the models**, **collecting metrics** and **supporting jobs orchestration**. We can observe that many systems in Section 3 share similar features, which are subsets of the list above.

3 Related Projects

In this section, we discuss the related projects to this thesis work. In Section 3.1, we discuss the recent development in the machine learning framework, which provides the foundation for the machine learning model management system. In Section 3.2, we give an overview of open-source projects and commercial solutions to manage the machine learning model. In Section 3.3, we discuss the related projects to our experiment to measure the computation time overhead to run a machine learning pipeline on Kubernetes. In Section 3.4, we discuss machine learning model serving systems related to the experiment to update the machine learning model serving at the Edge.

3.1 Machine Learning framework

Machine learning has been around for a long time. In academia, the researchers have developed their mathematical models in interpreted programming languages such as Matlab [43], Python [44], R [45] for fast prototyping, and C++ [46] for high performance. For data mining, we have framework such as Hadoop [47] and Apache Spark [48], implementing MapReduce, proposed by Dean and Ghemawat 2008 [49], to provide a paradigm to process mega-volume data stream. Thanks to many breakthroughs in Deep Learning, there was a sudden high demand for parametric computing frameworks to define complex machine learning models. Starting with research projects, we have frameworks such as Scikit-learn [50] and Theano [51], which already support hardware computation acceleration such as GPU. Followed that, there was a boom of production-ready machine learning open frameworks such as Tensorflow [52], PyTorch [53], Caffe [54], XGBoost [55] and MXNet [56] backed by big corporates or large opensource foundations. The main feature that these frameworks provide is Automatic Differentiation (autodiff) [57], to compute the derivatives numerically of the matrices to update the parameters of machine learning models. This computation must take advantage of the accelerated hardware such as GPU and provide parallelism to support the requirements of large models. This advancement also created a boom in GPU and cloud computing market. As the machine learning technology progresses, the commercial interest also increases leading to a demand for scalable managing systems to bring the machine learning models to production.

3.2 Machine Learning Management Systems

To offer a sound E2E machine learning solution, many projects have been launched with different focuses. **MLFlow** [58] is an open-source project developed and maintained by Databricks. It has 3 components. MLFlow tracking is an API for logging parameters, files, and version and a web server for visualization. MLFlow project is a format to package machine learning applications for reusability. MLFLow model is a convention to version control machine learning models, by offering a list of metadata such as build ID, creation DateTime, running environments (e.g. container), library

(e.g scikit-learn), calling functions, and serialized names. **DVC** [59] manages machine learning models and experiments by offering commands and metadata scheme to operate top of Git. It provides the concept of the stage, which is command along with its inputs and outputs. A sequence of stages is called a pipeline. A data file is not version-controlled, linked as references, supports local disk files, and different kinds of network storage backend such as Amazon S3, Google Cloud Storage, SSH/S-FTP... Metric files store the evaluation data of the pipelines. **Gallery** [42], which is developed and used internally at Uber to manage their machine learning models. It is part of their bigger system called Michelangelo [1], which is claimed to serve 1 million machine learning models in different kinds of applications. Gallery offers features such as saving/loading models, notating metadata, searching, archiving metrics, and orchestrating tasks. The project has helped Uber's data scientists and software engineers to get rid of the manual work to manage scripts and files. To offer the **machine learning as a platform**, we have Microsoft's **AzureML** [60] and Amazon's **SageMaker** [61]. Their internal architecture is closed-source and seems to be specific to their system. Google's **TFX** [62] offers a platform to run Tensorflow applications in production. Even though Tensorflow is an open machine learning framework, TFX does not support other machine learning frameworks such as PyTorch and XGBoost.

3.3 Cloud-native Overhead

ModelDB [63], originally developed at MIT CSAIL and now maintained by Verta.ai, offers a microservice application wich a client in Python and Scala and a multi-component relational database backend. It offers a list of entities such as project, experiment, experimentRun, model, and other metadata. ModelDB was started as a Master thesis and was one of the earliest projects to address and tackle machine learning model management. The thesis offers a comprehensive analysis of the main aspects of a model management system. However, the work does not concern cloud-native technologies. Thus, it inspires us to provide an analysis of the overhead of machine learning workflow on Kubernetes. **Kubeflow** [64], initiated by Google, is an open-source framework to develop and deploy machine learning application on Kubernetes. The project tries to void the gap between machine learning development and application delivery. It consists of both own components (such as Pipelines [4]) and other open-source frameworks (such as Jupyter Notebooks and Istio) to manage an E2E machine learning pipeline. The pipeline includes saving/loading models (through Minio and relational DB MySQL), orchestrating jobs (Pipeline), serving models(KFServing, TF-Serving, Seldon....), multiple training frameworks (PyTorch, Tensorflow, MXNet...), and many others. Because the project is open-source and easy to use, we utilize a subset of Kubeflow's components to perform our overhead measurement experiments, described in detail in Section 4.1.

3.4 Updating Machine Learning model

Regarding serving machine learning models, **Clipper** [65] is an open-source project developed at Berkeley which offers a low-latency machine learning prediction server running in a container. It supports multiple machine learning frameworks, such as PyTorch, Tensorflow, and XGBoost. To optimize the latency, it has features such as caching to serve frequent queries efficiently, batching to process concurrent queries in batch to improve the throughput, and adaptive model selection to improve the prediction performance. **TF-serving** (Tensorflow-serving) [66] is an open-source machine learning serving framework developed and used at Google. TF-serving supervises the filesystem storing the trained model code to get new models and their versions. It provides a concept Source Adapter, which creates model loaders from the model's metadata. Apart from Source Adapter, all the components in TF-serving treat the real machine learning models as black boxes called **servables**. The adapter handles the implementation details to load the model. TF-serving has Source Router routes the model version streams to their corresponding adapters. In the context of serving the machine learning model at a micro-data center or a computing edge node, TF-serving is a suitable choice. Thus, we use it to run the serving server at an Edge node in our updating machine learning model experiment.

4 Methodology

As stated in the introduction 1, we study two problems in the thesis. The first problem is to understand the overhead introduced by Kubernetes abstractions and automation. The approach is to measure the extra computation time introduced by Kubernetes and Kubeflow services in addition to the machine learning pipeline tasks. The details are presented in Section 4.1. The second problem is to understand how well to update a machine learning model at an Edge node. In Section 4.2, we discuss the two setups to measure the machine learning model uploading time with simulated network latencies to address the second research problem.

4.1 Cloud-native Overhead

In this experiment, the main goal is to measure the execution time of the Kubernetes jobs in the testing machine learning pipeline. The execution time includes the overall duration and the machine learning task's execution time. The overhead is calculated by subtracting the machine learning task's execution time from the overall duration. The experiment is conducted with different dataset sizes and model sizes to study the overhead trend.

The author creates a Jupyter notebook by adapting a tutorial on the Tensorflow website [67] as a quick start to the Tensorflow Keras API. Jupyter notebook is commonly used by data scientists and academics to document their experiments. The main feature is the program, typically in Python, is run as a web application, composed of blocks of code separated into cells to save the intermediate results as HTML elements. The author wants to have a proper analysis relevant to IoT, thus analyzes the dataset Intel Lab [68]. Even though adapting an existing popular notebook, such as on Kaggle, may suffice, the author thinks that working on every step to produce a sensible model helps understand the tasks better. It also allows easy modification to do other work, such as generate models of different sizes. The notebook elements are described in Section 5.3.

The testing Jupyter notebook consists of 10 cells, which are transformed into Kubernetes jobs. The goal is to measure the time to complete the original cells and the overhead introduced by Kubernetes and Kubeflow.

To run the Jupyter cells as Kubernetes jobs, we utilize two open-source projects named Kubeflow Pipelines [4] and Kubeflow Kale [5]. Kubeflow Pipelines framework provides a platform to run machine learning workflow as Kubernetes jobs. It utilizes the Argo Workflows controller [69] to spawn pods and manage them. It also archives artifacts such as HTML documents and saves metadata to a relational database backend. Plugin Kubeflow Kale offers the ability to annotate Jupyter notebook to compile into Kubeflow Pipelines code. The author's work is to reorganize the notebook in the way that the two plugins require. This process includes putting all the imports in a single cell on the top to fit Kubeflow Pipelines' way of working and labeling the cells to define their dependency.

The whole pipeline does not work out of the box. One limitation is that the plugin Kale assumes the default storage class to have the permission Read-Write-Many

(RWM). However, the storage backend we have for the Kubernetes cluster only supports Read-Write-Once (RWO). Thus, the author needs to manually modify the PVC (persistent volume claim) request to be RWO in the compiled Kubeflow Pipelines code.

The pipeline is run with different dataset sizes and model sizes. Using the dataset to develop the sensible model, roughly 650 kB in size, the author duplicates and concatenates it to generate four dataset sizes: 2 MB, 10 MB, 20 MB, and 40 MB. The author varies the hidden layers to generate four models: $8 \times 8 \times 8$ (185 trainable parameters), $16 \times 16 \times 16$ (626 trainable parameters), $32 \times 32 \times 32$ (2273 trainable parameters) and $64 \times 64 \times 64$ (8648 trainable parameters). Two series are run: model $8 \times 8 \times 8$ against four datasets and the 20 MB dataset against four models. Thus we have seven configurations in total.

From a brief examination, the author identifies two probable sources of overhead: Kubernetes orchestrator and Kubeflow Pipelines. To measure the Kubernetes metrics, the author installs the plugin kube-state-metrics [70] to expose Kubernetes API to Prometheus and installs Prometheus [71] to fetch the metrics. The author writes a Python script to collect four statistics: pod creation time, pod scheduled time, pod starting time, and pod completion time.

The extra computation time by Kubeflow Pipelines comes from the way it handles data passing. The plugin utilizes a dynamic volume to marshal and unmarshals its variables and objects. The Pipelines also save information about the run MySQL backends and archived artifacts to a MinIO object-store.

To track the computation time in the original cells, the author adds to each cell extra lines of code to calculate the execution time. A console print **[JT] cell end** is added to signify where the original cell execution code ends and the corresponding result is collected from the line right before it. For example, the **dnnfit** cell is defined as followed:

```
start = time.time()
history = dnn_model.fit(
    train_features, train_labels,
    validation_split=0.2,
    workers=4,
    use_multiprocessing=True,
    verbose=1, epochs=20)
plot_loss(history)
end = time.time()
print("[JT] dnnfit time(s) {}".format(end - start))
print("[JT] end cell")
```

The author writes a bash script to collect the logs from the pods in each run. The computation time overhead is calculated by subtracting the pod's lifetime duration (collected from Prometheus) by the cell execution time (extracted from the log files).

The measurement for each configuration is the average of 6 runs. Thus, we have 42 runs in total.

4.2 Update Machine Learning Model at The Edge

In this experiment, the main goal is to measure the uploading time for machine learning models with different sizes, having different networking latencies added in two setups: from a VM to a VM and from a Kubernetes pod to a Kubernetes service. The reason is to study how well to update a machine learning model stored on an Edge node managed by Kubernetes.

We don't go further with machine learning serving performance, which has been covered in other work such as Crankshaw et al 2017 [65]. Go further in that direction may put the focus off the track. The experiment is similar to a benchmark of uploading data on a noisy network.

The author first creates a working machine learning serving prototype consists of two services: TF-Serving and MinIO [6] object-storage services. The prediction is verified with the command line tool Curl [72] over HTTP to ensure the system can serve the selected model. The author writes YAML manifests to install the two services. To simulate an Edge node in a Kubernetes cluster, the author labels one of three worker nodes as **iot=edge** and the other two as **iot=core**. The label **iot=edge** pin where the MinIO and TF-Serving containers run. The Edge worker node is attached with a 200 GB local volume to simulate the local disk. To simulate the distance from the Edge node to the rest of the system, the author uses Netem [7] to simulate network latency. Four latency values: 0 ms, 50 ms, 250 ms, and 400 ms are configured to the network interface of the Edge node.

To generate models with different sizes, the author varies the neural network model's (developed in Section 4.1) hidden layer number and size to be arbitrarily large. Four models are created: 49 MB (around 12.6 million trainable parameters), 113 MB (around 30 million trainable parameters), 177 MB (around 46.1 million trainable parameters), and 241 MB (around 63 million trainable parameters).

Uploading models are run in two setups: from a pod to a Kubernetes MinIO service and from a VM to another VM. Artificial network latencies are added to the Edge node's network interface in both setups. In the first setup, the author builds a custom container packed with the models and uploads it to a Docker registry. The testing pod pulls the image from the Docker registry. The result is collected by reading the pod's logs. In the second setup, the author writes a shell script to upload the models, report the time, and write the result to files.

The measurement for each configuration is the average of 10 runs. Thus, we have 320 runs in total.

5 Experiment Setup

Before working on the research problems, we need to have a testbed to conduct our experiments. The author installs a Kubernetes cluster on top of a research Openstack cloud using Kubespray [73]. With the provided Terraform templates and Ansible playbook from Kuberspray, we do not need to develop the installation scripts, but the work involves network planning, dimensioning, and troubleshooting. The details of this process is explained in details in Section 5.1 and Section 5.2. The result is a fully functional Kubernetes cluster with selected internal services exposed to the Internet using floating IPs. In Section 5.3, we describe the work to develop a Jupyter notebook analyzing an IoT dataset for the overhead experiment.

5.1 Infrastructure

The experiments are conducted on an on-prem Openstack cloud. The author uses Terraform [74] and Kubespray [73] templates to provision Openstack router, network, subnet, floating IPs, and compute servers. Terraform resolves the dependency graph of the resources internally and offers the preview (command **plan**), installation (command **apply**), and uninstallation (command **destroy**) functionalities, which is quite convenient to quickly set up and modify the resources. Because we don't focus on the security aspects in the experiments, the author disables the port security of all the interfaces after the installation.

There are three groups of resources:

1. **Kubernetes cluster (named micro)** consists of 1 master (4 vCPUs and 16 GB of RAM each) and 3 worker nodes (8 vCPUs and 16 GB of RAM each). We use Ubuntu 18.04 LTS as the base operating system for the VMs. The number of masters needs to be odd because they run ETCD [75], because ETCD internally requires the majority of nodes, a quorum, make the verdict about the cluster state.

2. **Benchmarking host** is a virtual machine (8 vCPUs and 16 GB of RAM) which has the access to cluster **micro**.

3. **Edge host** is a virtual machine (4 vCPUs and 8 GB of RAM) which is in the same subnet as the benchmarking host. This is used for the model uploading experiments in Section 4.2.

The network plan between the Kubernetes cluster and the benchmarking machine is straightforward. All the nodes in the Kubernetes cluster share the same subnet **micro-internal-network**, within the same network **micro-network**, connected to the Internet through the router **micro-router**. The benchmark machine locates in the subnet**benchmark-internal-network**, within the network **benchmark-network**, behind the router **benchmark-router**. We connect the benchmark machine to the Kubernetes cluster by creating an interface in the **micro-router** to the **benchmark-network** subnet and add a route from the benchmark machine to the Kubernetes cluster's master IP via that interface. The network plan is shown in Figure 9.

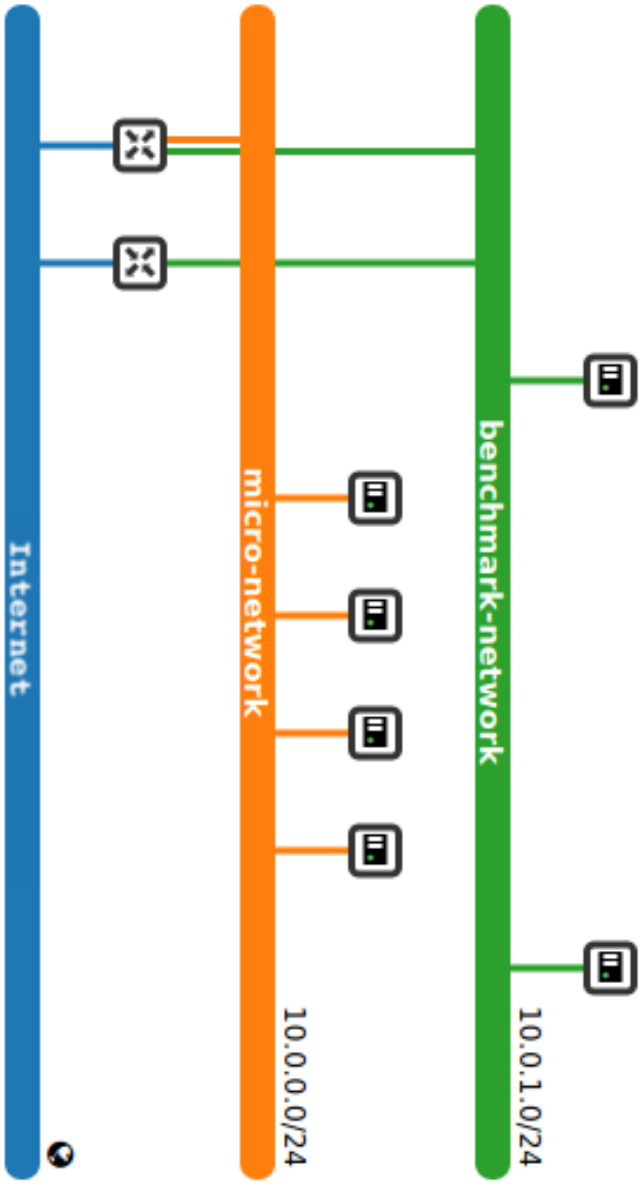

Figure 9: Testing network plan. The network **micro-network** is where the Kubernetes cluster's nodes located, which consist of one master and three worker nodes. In the **benchmark-network**, we have a host to run commands and collect data and another host to host a MinIO server to be tested in the model uploading experiments.

5.2 Kubernetes Bootstrap

To bootstrap the Kubernetes cluster, the author uses the Ansible [76] playbook provided by Kubespray, which utilizes Kubeadm [77]. Ansible is widely used to manage infrastructure as code. It provides a systematic framework to configure multiple hosts instead of logging in them individually and run commands. Kubeadm offers all the steps needed to set up a minimal Kubernetes cluster including provisioning the TLS certificates to all the nodes to secure the cluster internal communication; bootstrapping ECTD service running in the master node; installing container runtime (Docker in our case); installing controller, API server, scheduler, kubelet to the master node; installing and configuring Container Network Interface (CNI) plugin and kube-proxy to the worker nodes. To get more details of the Kubernetes bootstrapping procedure, Kelsey Hightower's tutorial "Kubernetes The Hard Way" [78] is a good introduction. The version of Kubernetes we install is **1.18.10**.

Kubernetes provides a command-line tool called Kubectl to interact with the cluster to perform operations, administrations, and management (OAM). We install kubectl to the benchmark machine and configure the credentials using the provisioned root and API server certificates generated by kubeadm. We configure the API endpoint with one of the master IP address, which has been routed via the router **micro-router**'s interface discussed in Section 5.1.

The next step is to expose internal services to the Internet. When deploying a web application on a public cloud such as Amazon Web Service (AWS) cloud or Google Cloud Platform (GCP), the user typically requests a load balancer, which receives the HTTP/HTTPS traffic directed from their public IP, to spread the traffic to the backend servers. In a production Kubernetes cluster, the user can deploy an Ingress [79] to expose the internal services externally with an URL or a public IP. The HTTP/HTTPS traffic from the ingress is spread to the corresponding services, which in turn, is spread to the corresponding pods running one or multiple containers (see Figure 10) Most public cloud providers provide their Ingress controller, which can be deployed to the Kubernetes cluster to interact with their load balancing API.

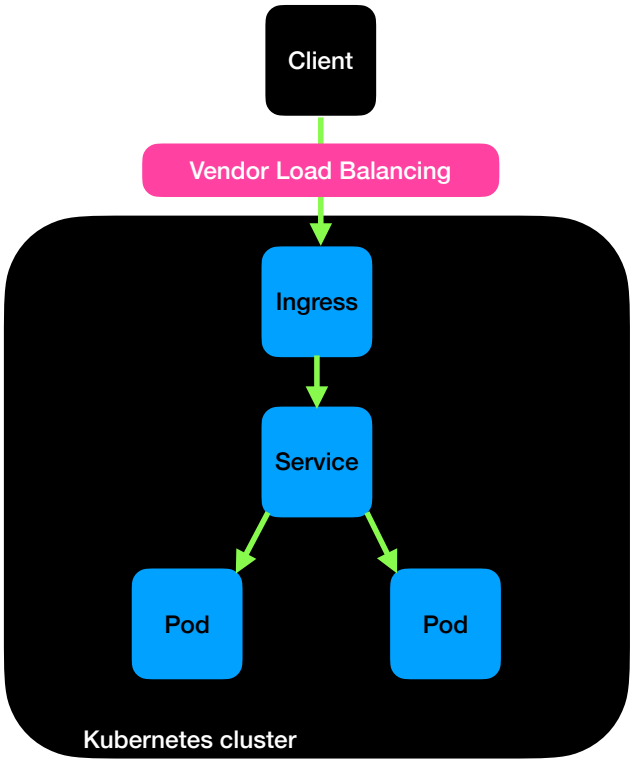

Figure 10: The Kubernetes traffic flow spreads from a client to pods.

The OpenStack project provides Octavia [80] and Neutron LBaaS (v1 or v2) [81] to offer the load balancing as a service similar to the load balancing API that public cloud providers offer. However, these plugins are not installed in our Openstack cloud, thus we choose to expose the internal services as they do for bare-metal Kubernetes cluster using a plugin called Metallb [82]. Currently, there are two ways to configure Metallb: layer 2 and BGP (Border Gateway Protocol) mode. To use BGP mode is more complicated because it requires a router supporting BGP, which may be deployed as a virtual machine running a routing suite such as FRRouting [83] or installing an Openstack routing plugin such as BGP dynamic routing. Thus, we choose to go with the layer 2 mode as it is simpler. Metallb's layer 2 mode replies ARP (Address Resolution Protocol) for IPv4 requests and NDP (Neighbor Discovery Protocol) for IPv6 requests. Behind the scene, one worker node advertises the service in the local network (between the nodes) and spreads the traffic to pods through kube-proxy. This means that one worker node serves as the load balancer and can be the bottle-neck. In fact, in the networking point of view, the traffic flow looks exactly the same as using service with type NodePort [84]. What Metallb layer 2 mode offers is the ability to set an IP sub-range in the subnet (in our case: **micro-internal-network**) to the LB's IP list and the failover capability (having another worker node to serve as the LB after the current LB worker node goes down).

Another benefit of using Metallb layer 2 comparing to using NodePort service is to specify a common port number, such as 8080 for HTTP and 443 for HTTPS, instead of the typical NodePort's high range (30000-32767). NodePort service is not meant for production service anyway.

After Metallb is installed and configured, we pin the LB's range to a single IP and associate it with a floating IP to expose the internal service to the Internet.

5.3 Jupyter notebook development

5.3.1 Dataset

To demonstrate the relevance to IoT, we use the Intel Lab dataset [68], which consists of humidity, temperature, light, and voltage readings from 54 sensors from February 28th to April 5th, 2004. The readings are polled at a 31-second interval. The location of the sensors is described in Figure 11. Totally, the dataset has 2.3 million readings. The author filters only node 54's readings using the Unix AWK command to develop a sensible model. Incompleted data points are also dropped. The author also assumes that the data points are identical independent distributed (i.i.d), meaning that we ignore the chronicle relationship between data points.

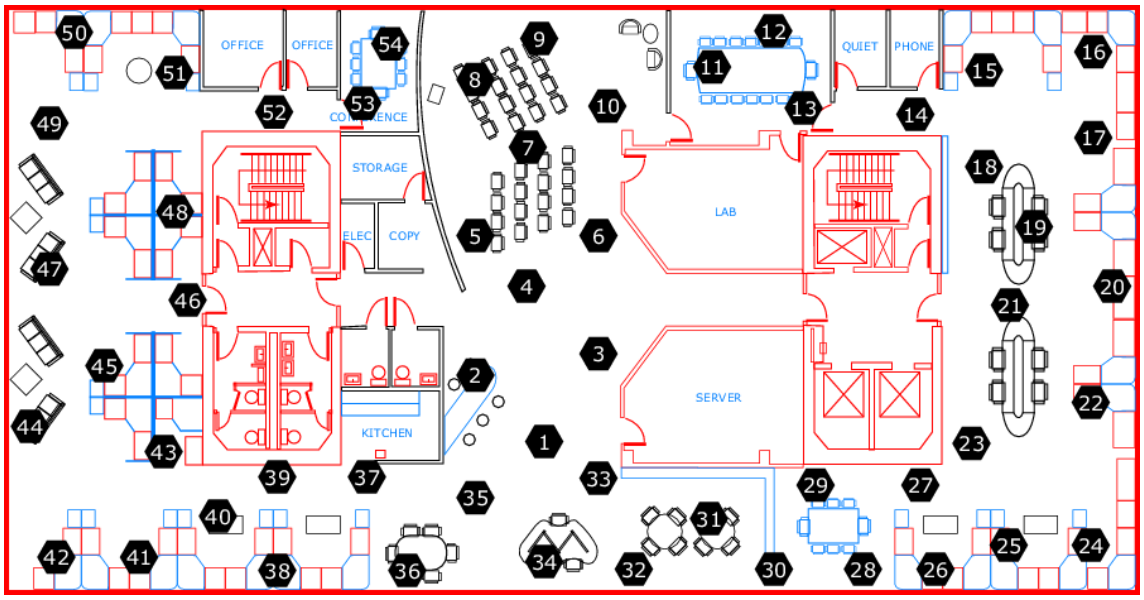

Figure 11: Intel Lab sensor map. The dataset is collected from 54 sensors' data stream. [68]

In summary, we have 28.7K i.i.d data points. We define a regression objective for the machine learning objective that uses humidity, light, and voltage data to predict the temperature. We split into 2 parts: training set (80 %) and testing set(20 %), having 3 numeric features (humidity, light, and voltage) and a numeric target value (temperature).

5.3.2 Machine Learning Models

Tensorflow Keras [85] is our framework of choice. Jupyter notebook is used as the development environment.

The model is adapted from the Predict fuel efficiency tutorial [67] which is used as a quick start to Keras API. The author first tries out a small **feed forward neural network** with different layer architecture to develop a sensible model on the Intel Lab dataset [68]'s node 54. The training algorithm is stochastic gradient descent based algorithm called Adam [86], with a learning rate 0.001 and mean absolute error as the loss function. After the model performs relatively well, it is used as a base to generate testing models by increasing the number of layers, increasing the hidden layer sizes, or both.

The dataset is also used to generate bigger datasets by duplicating and concatenating.

5.3.3 Kubeflow Pipelines

The Jupyter notebook is organized into cells as described in Table 1. This notebook resembles a simplified machine learning model workflow.

Cell/Job	**Functionality**
import	Import libraries
preprocessing	Download and split the dataset.
plot	Pair plot the training data.
createdict	Create a dictionary to store results.
dnncompile	Define and compile the neural network model.
dnnfit	Fit the neural network model to the training set.
dnneval	Evaluate the neural network model with the testing set.
modelerrorsummary	Print the evaluation errors of the model.
dnnpredict	Make predictions using the neural network model.
dnnpredictsummary	Plot the prediction errors of the neural network model.

Table 1: This table shows different cells in the testing Jupyter notebook. These are the original computations we have for the application. Kubernetes may introduce extra overhead in addition to this list of tasks.

The cells are labeled to compile into Kubeflow pipeline code. The plugin Kubeflow Kale offers a GUI extending Jupyter notebook to label the cells and define the dependency between them. For example, in Figure 12, we annotate the **dnnfit** cell and it depends on the cell **dnncompile** as the direct parent. Figure 13 depicts a successful run of a whole generated pipeline. We don't have the job **import** in the Kubeflow generated pipeline because it is prepended to all the other jobs to simplify the library importing procedure. A new job **kale-marshal-volume** to create a PVC to handle data passing between the jobs.

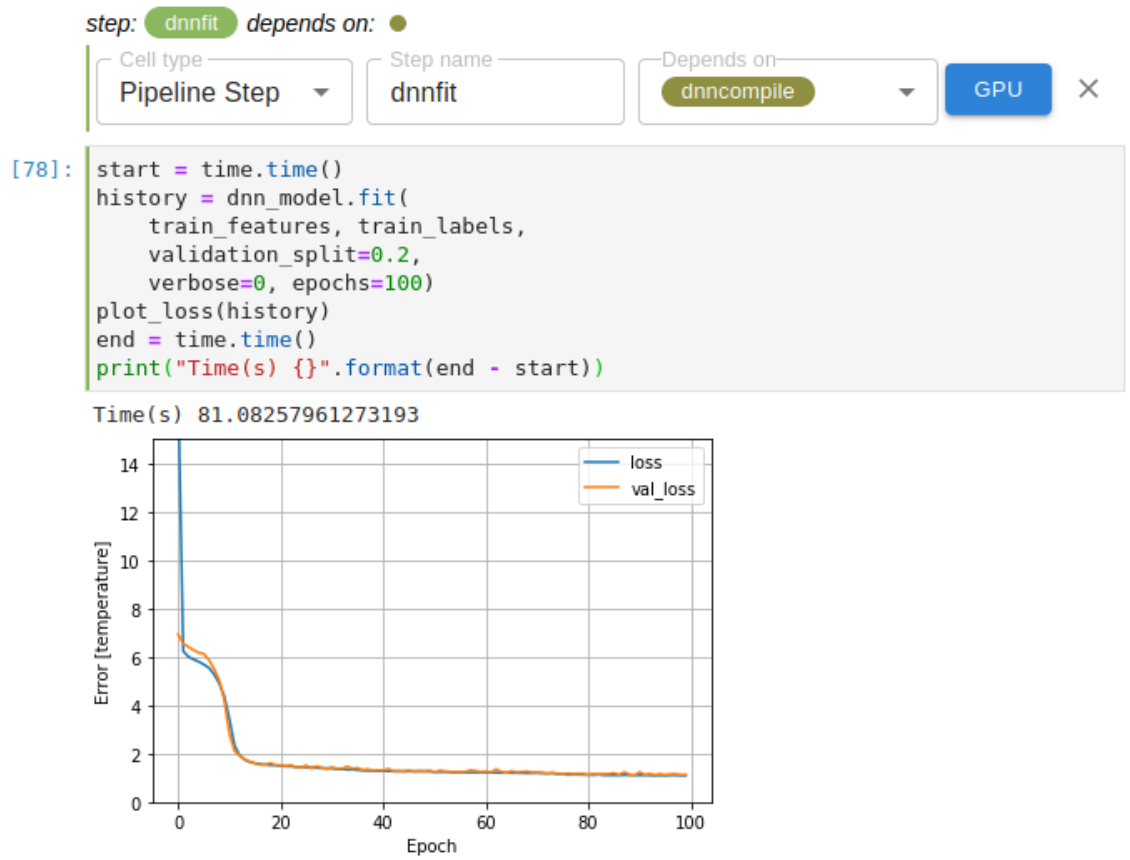

Figure 12: This Figure depicts the web frontend to annotate the cell **dnnfit** label and define its dependent cell **dnncompile**.

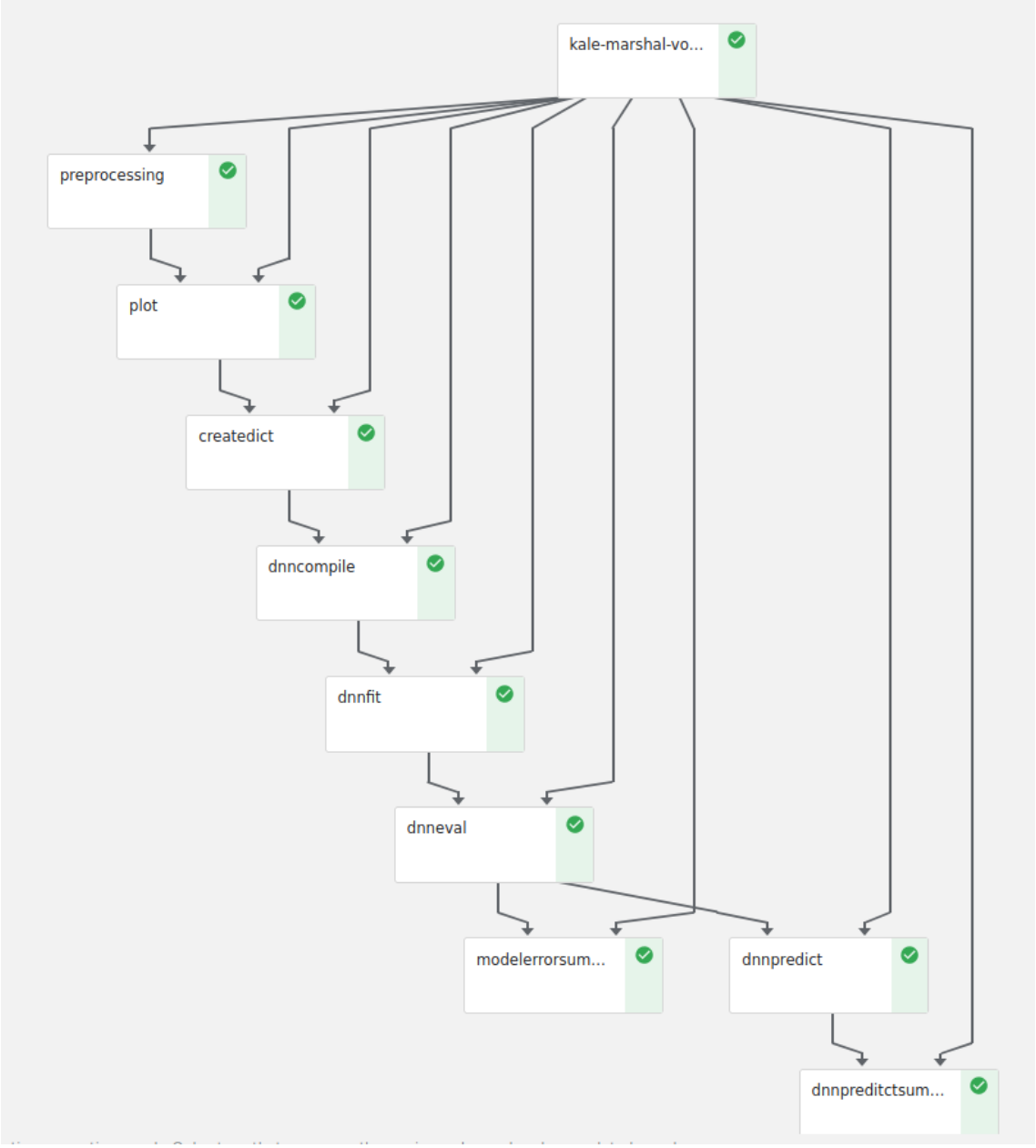

Figure 13: This Figure depicts the dependency graph of the generated Kubeflow Pipeline. Each box presents a job to be orchestrated to run in Kubernetes. **kale-marshal-volume** in the root shows the marshaling requirement to a PVC to pass the data between Jobs.

6 Results

In this section, we discuss the results of our experiments. In Section 6.1, we show and analyze the results of the experiment described in Section 4.1 to quantify the overhead running machine learning pipeline on Kubernetes. In Section 6.2, we show and analyze the results of the experiment described in Section 4.2 to measure the machine learning model uploading time to a serving node at the Edge.

6.1 Cloud-native Overhead

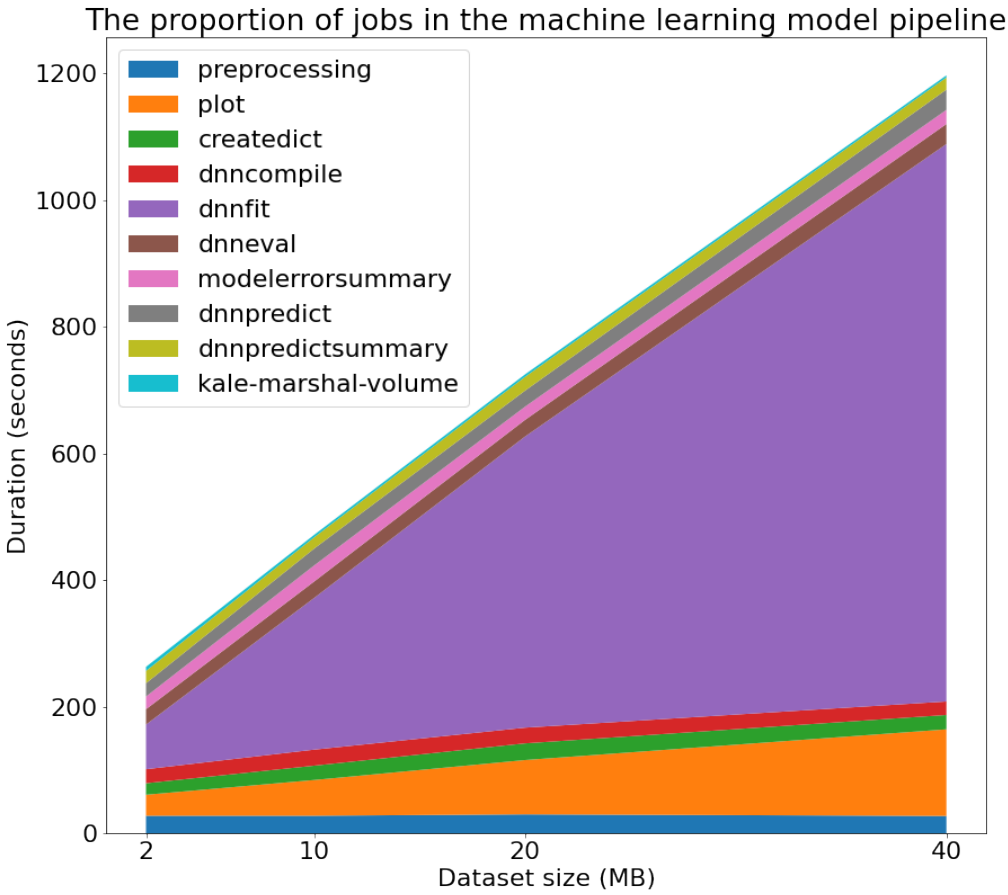

Figure 14: This Figure depicts the total durations for the orchestrated jobs in the pipelines for a run in different dataset sizes. The measurement for each dataset size is the average of 6 runs.

Figure 14 depicts the proportion of the jobs' durations in the experimenting workflow when the dataset size increases. In this experiment, the machine learning model used is $8 \times 8 \times 8$-hidden-layer neural network described in Section 4.1.

The most expensive job **dnnfit** to fit the machine learning model grows from 70.5 seconds for the dataset 2 MB to 880.6 seconds for the dataset 40 MB. The second most expensive job **plot** to pair-plot the training data grows from 33.2 seconds for the dataset 2 MB to 136.8 seconds for the dataset 40 MB.

The job **dnneval** and **dnnpredict**' durations grow from 24.3 to 31.7 seconds and from 21 to 32 seconds respectively. However, the growths of these two jobs are quite small compared to the two most expensive jobs.

Table 2 shows measurements of the four discussed jobs in the experiment.

Job/ Model size	2 MB	10 MB	20 MB	40 MB
dnnfit	70.5	240	460	880.6
plot	33.2	56.5	85.8	136.8
dnneval	24.3	26	26.3	31.7
dnnpredict	21.0	26.7	25.3	32.0

Table 2: The table summarizes the durations in seconds of jobs **dnnfit**, **plot**, **dnneval**, and **dnnpredict** in the experiment with model $8 \times 8 \times 8$ vs. dataset sizes 2 MB, 10 MB, 20 MB, and 40 MB.

Other jobs' computation time is roughly the same throughout the trend comparing to the two mentioned expensive jobs. Table 3 summarizes the fluctuating range of those jobs.

Job	Fluctuating range (seconds)
preprocessing	from 27.7 to 29.8
createdict	from 18.2 to 22.7
dnncompile	from 21.3 to 25.3
dnnpredictsummary	from 17.8 to 18.5
modelerrorsummary	from 20 to 25.3
kale-marshal-volume	from 3.3 to 6.3

Table 3: The table summarizes the jobs' durations fluctuating ranges in seconds in the experiment with the model $8 \times 8 \times 8$ vs. dataset sizes 2 MB, 10 MB, 20 MB, and 40 MB.

It is evident that the **dnnfit** job, which trains the machine learning model's parameters to dominate the computation time. In fact, it is the most expensive job in any non-trivial machine learning pipeline. In our experiment, the average time to train the testing model peaks at around 15 minutes for the dataset 40 MB.

The pair plot function **plot** also becomes expensive over time. But in our experiment, we plot every data point in the training set, which can be improved by sampling a subset of the training set instead.

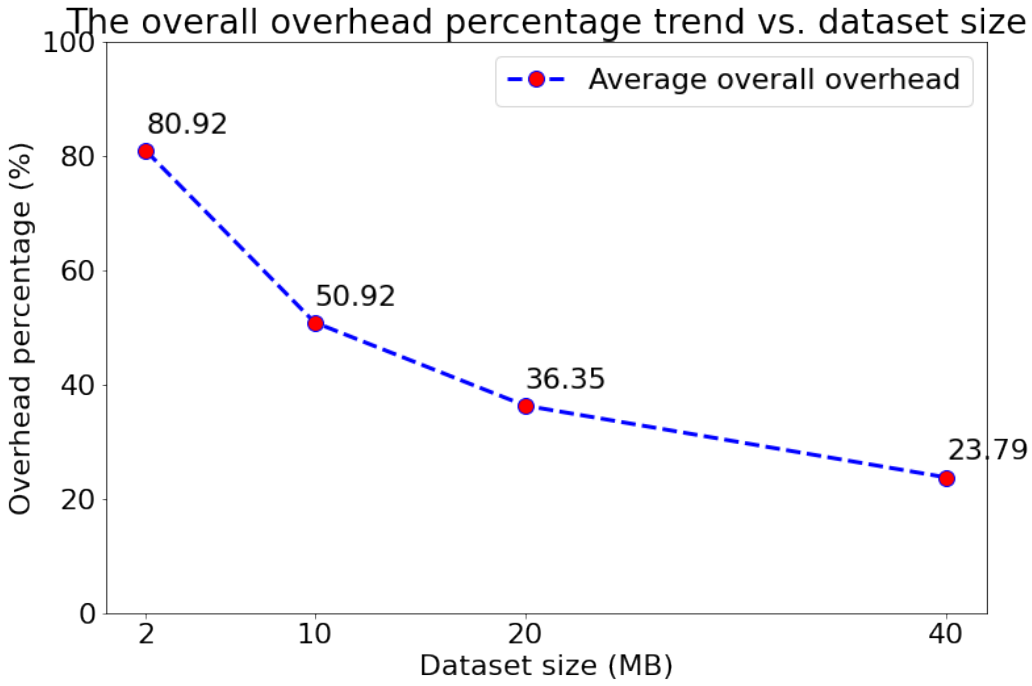

Figure 15: This figure depicts the percentage of overhead for the pipelines running with different dataset sizes. The $8 \times 8 \times 8$-hidden-layer model is used for all the runs. The measurement for each dataset size is the average of 6 runs.

Figure 15 depicts the average overall overhead percentage trend to decrease as the dataset becomes bigger. When the dataset is small, at 2 MB, the overhead is up to around 81 %. However, as the dataset becomes bigger, the overhead becomes less significant, for example, 23.8 % for the 40 MB dataset.

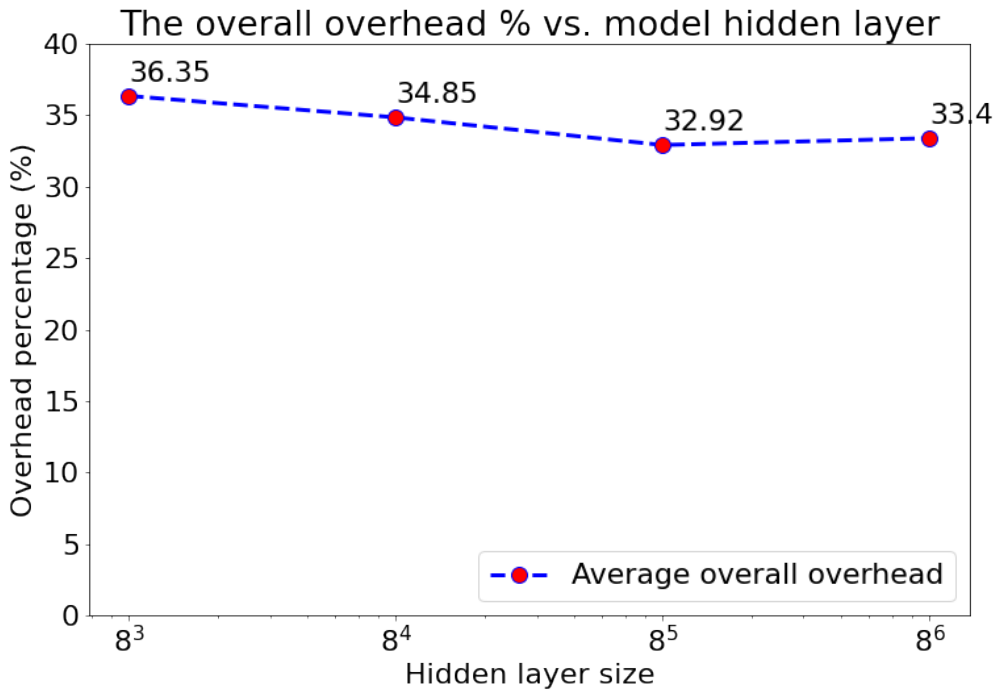

Figure 16: This figure depicts the percentage of overhead for the pipelines running with different machine learning model sizes. The same dataset 20 MB is used for all the runs. The measurement for each model size is the average of 6 runs.

Figure 16 also shows the overhead percentage trend running the pipeline to train models with different sizes on the same dataset 20 MB. The trend shows a decrease from 36.4 % for the model $8 \times 8 \times 8$ to around 33 % for the model $64 \times 64 \times 64$. Even though there is a jump at the end, it is relatively small: 0.5 % from 32.9 % to 33.4 %. The reason is that we may choose the size step between the models' sizes too small, thus the measurement variance is susceptible to noise.

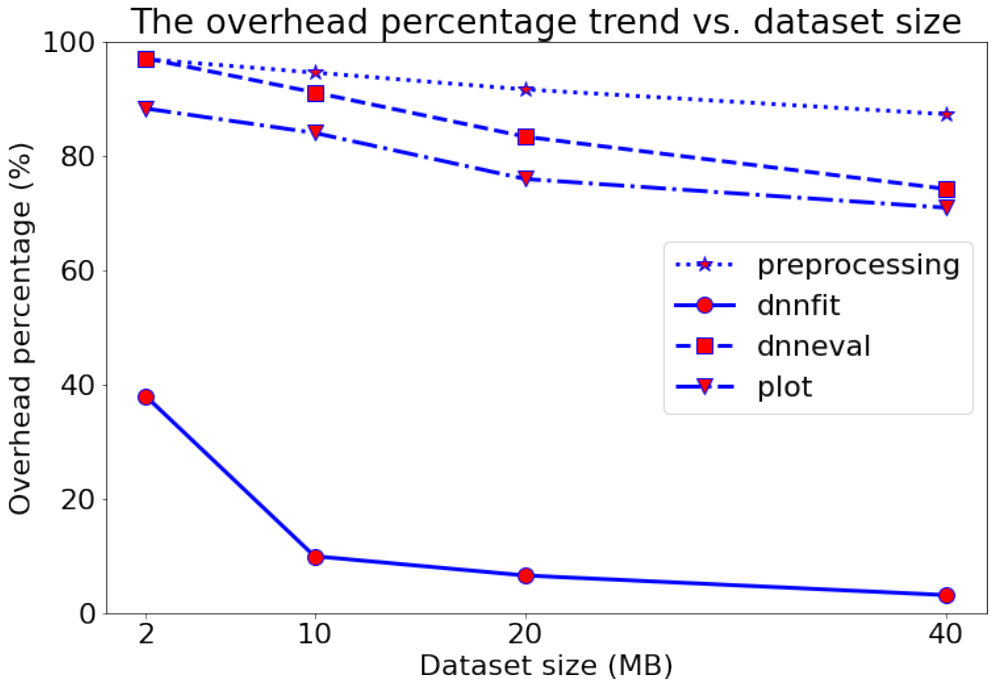

Figure 17: This figure depicts the percentage of overhead for selected interesting jobs running with different machine learning model sizes. The same $8 \times 8 \times 8$-hidden-layer model is used for all the runs. The measurement for each dataset size is the average of 6 runs.

Figure 17 depicts the trend of computation time overhead of selected jobs.

We find that most trivial jobs (in term of computation time) such as **createdict**, **dnnpredictsummary**, **modelerrorsummary**'s overhead always stay relatively high around 99 %, which means that they can be optimized by being included into other significant jobs. Because these jobs show no interesting changes, we exclude them from the figure.

The training job **dnnfit**'s overhead becomes insignificant as the dataset becomes bigger. The overhead percentage decreases from 38 % for the dataset 2 MB to 3.2 % for the dataset 40 MB. This result aligns with the duration for this job increases as the dataset size increases described in Table 2.

The pair-plot job **plot**'s overhead percentage decreases from 88.3 % for the dataset 2 MB to 71 % for the dataset 40 MB. This workload increases due to we plot all the training data points. This job can be improved by only plot a sample of the training set.

The evaluation job **dnneval**'s overhead percentage decreases from 97.1 % for the dataset 2 MB to 74.3 % for the dataset 40 MB. The workload increases because the evaluation set is proportional to the training set.

The **preprocessing** job's overhead percentage decreases from 97 % for the dataset 2 MB to 87.3 % for the dataset 40 MB. As described in Table 3, the

job **preprocessing**'s duration fluctuates around 28 seconds. The execution time increases from 0.84 seconds for the dataset 2 MB to 3.5 seconds for the dataset 40 MB resulting in the overhead percentage decrease (see Table 4). The reason for this trend is the increase in dataset downloading time.

Job/ Dataset size	2 MB	10 MB	20 MB	40 MB
dnnfit	43.5	215.9	429.5	852.4
plot	3.7	10.9	20.5	39.7
dnneval	0.68	2.3	4.1	7.8
preprocessing	0.84	1.7	2.4	3.5

Table 4: This table summarizes the execution times in seconds of the machine learning jobs excluding the overhead in the experiment with model $8 \times 8 \times 8$ vs. dataset sizes 2 MB, 10 MB, 20 MB, and 40 MB.

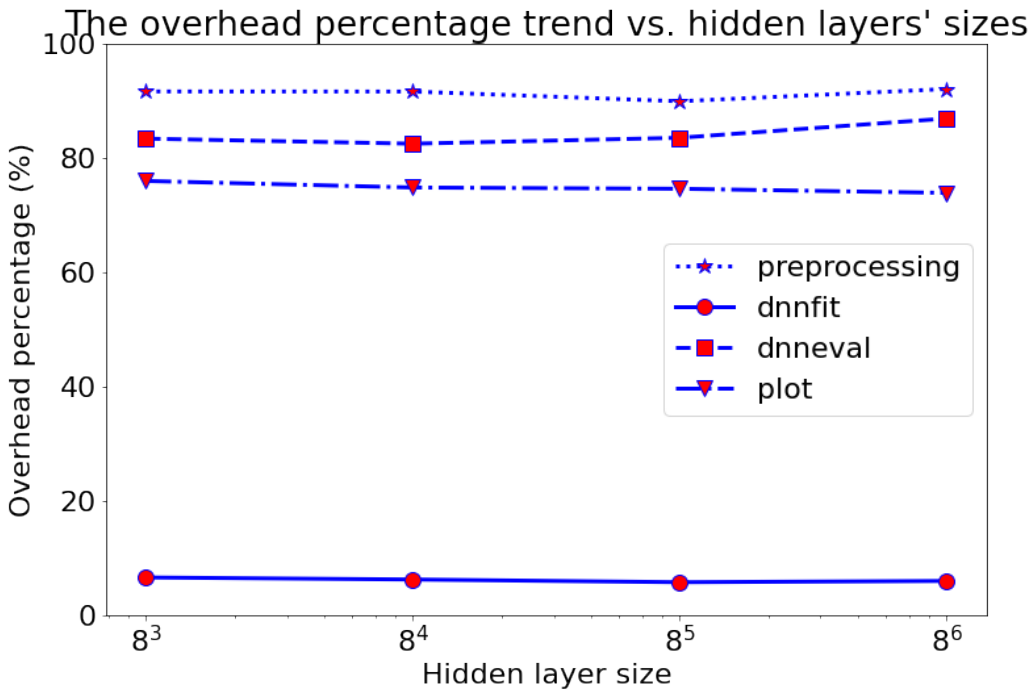

Figure 18: This figure depicts the percentage of overhead for selected interesting jobs running with different machine learning model sizes. The same dataset 20 MB is used for all the runs. The measurement for each model size is the average of 6 runs.

Figure 18 evidence our poor choice of selecting too small a step between the model sizes.

The job **dnnfit**'s overhead percentage fluctuates around 6 %. The job **plot**'s overhead percentage ranges from 73.9 % to 76 %. The job **dnneval**'s overhead

percentage ranges from 82.5 % to 86.9 %. The job **preprocessing** ranges from 90 % to 92.1 %.

Besides small fluctuations, which are probably noise, there are virtually no meaningful changes in overhead percentages in the selected jobs.

6.1.1 Analysis

Container downloading time may also contribute to computation time overhead. It is worth mentioning that using Kubeflow Pipelines, the jobs are run with the same container to run the Jupyter notebook that triggered it. As the author pack much useful software in this image, the image is about 3 GB compressed. However, after a container is downloaded to the local registry, it is cached and the next job running the same container doesn't have to download it again (except it is specified to do so). Thus, we don't consider this a major factor for the overhead.

The pod starting time may also contribute to the overhead. But examine the Prometheus metrics, most duration between creation time to the scheduled time and from scheduled time to running time falls within one second (we cannot get a finer measurement because second is the unit of the metrics). Manaouil and Lebre 2020 [87] report that these durations don't introduce big delays in case no latency is added to the underlying network. Thus, we consider them not a major factor for the overhead.

Checking the underlying processes, we notice the overhead comes from many sources. The major one is the way the data is passed around between the jobs. At the beginning of a pipeline run, a PVC (1 GB) is claimed with the job **kale-marshal-volume**. This takes roughly 3 seconds. We can notice it as a very thin stripe on top of the stack plot 14. During the job run, the data is (de)serialized to the PVC. The process results in the execution time for the containers to attach to the PVC and for the networking traffic to the Cinder storage backend. Another source of overhead is the cell **imports** code is prepended to every job, which takes around 3 to 4 seconds.

Provided by the Kubeflow framework, the artifacts of the jobs such as figure and metadata are archived in multiple backends, including a MinIO object-storage and MySQL database. The time to create the client and upload the artifacts to these backends contributes to the overhead.

6.2 Updating machine learning model at the Edge

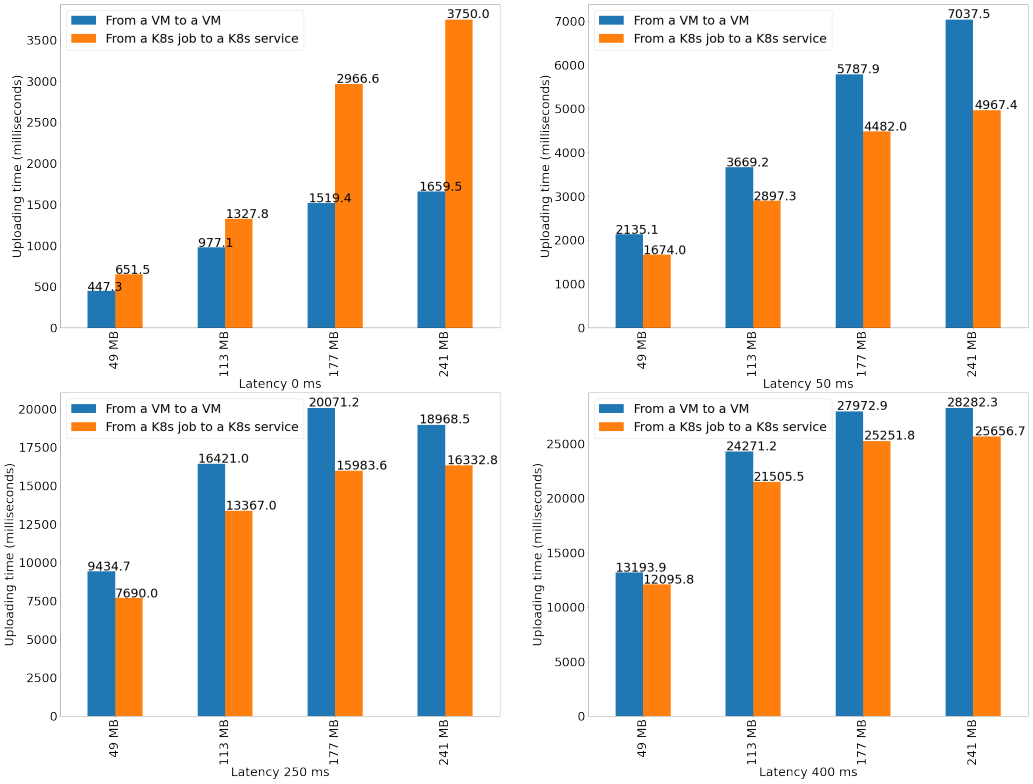

Figure 19: This Figure depicts the model uploading time in milliseconds to object-store MinIO service endpoints in two setups: between two VMs (blue) and from a Kubernetes job to a Kubernetes service (orange). Different model sizes are tested: 49 MB, 113 MB, 177 MB and 241 MB. Artificial networking latencies: 0 ms, 50 ms, 250 ms, and 400 ms are added in the runs, which are depicted in the subplots. The measurement for each configuration is the average of 10 runs.

Figure 19 shows a machine learning model uploading time to an object-store service.

For both setups and all model sizes, it is evident that the bigger the network latency, the bigger the uploading time. For instance, in the case of uploading the 241 MB model in the Kubernetes setup, the uploading time increases from 3.8 seconds for no network latency to 25.7 seconds for network latency 400 ms.

For both setups with most of the latency configurations, in general, the larger the model size, the longer the uploading time. For instance, in the case of network latency 400 ms and VM-to-VM setup, the uploading time increases from 13.2 seconds for the model size 49 MB to 28.3 seconds for the model size 241 MB. However, in the

case of latency 250 ms, the uploading time decreases from 20.1 seconds for model size 177 MB to 19 seconds for model size 241 MB. This pattern may have resulted from the randomness from the network emulation and the small uploading time difference between the 177 MB model and the 241 MB model with network latencies 250 ms and 400 ms.

To compare the two setups, in the case no network latency added, the uploading times in the Kubernetes setup are consistently higher than in the VM-to-VM setup. In cases the network latencies added, the uploading times in the VM-to-VM setup are higher than in the Kubernetes setup.

6.2.1 Analysis

As the performance may better for either setup for different network latency cases, we think it may only have resulted from the virtual router connected to the uploading client and the storage endpoint. Also, ten runs for each configuration is probably not enough to minimize the bias. Therefore, the difference in the uploading time between two setups does not inherit from the underlying technology such as Kubernetes and container.

Manaouil and Lebre 2020 [87] show that thanks to the reliable REST-based architecture, Kubernetes has no issue to manage pod at the WAN scale. However, the issue may happen in service discovery when the link becomes unreliable. We don't experience an issue with name resolution in our experiment with network latency up to 400 ms.

7 Conclusion

In this thesis, we answer the research question to quantify the overhead introduced by Kubernetes and other applications to run machine learning pipelines by measuring the task execution time and overall duration. We first create a Jupyter notebook on an IoT dataset, then modify it to generate models and datasets with different sizes. We compile the Jupyter notebooks to orchestrate jobs on a testing Kubernetes cluster. We define the overhead as the extra computation time in addition to the machine learning task's execution time. The results show that the bigger the dataset, the more expensive many important jobs become. These jobs include training the model, evaluating the model, data preprocessing, and plotting. The model training job is the most expensive and dominates the pipeline. Therefore, the overall execution time overhead becomes less significant when the dataset size increases. The overhead trend also decreases when the model size increases. However, it does not show the same change in overhead significance as dataset size increases. The reason is our step size choice between model sizes is too small.

To answer the second research question, we set up a machine learning model serving prototype and benchmark the machine learning uploading time with different sizes to an object-storage service. The object-storage service is deployed in two setups: as a Kubernetes service and a service running in a VM. We run the experiment with different artificial network latencies. The results indicate that the uploading process running on Kubernetes does not show a significant difference from running without Kubernetes. It is like any data transfer over a network, thanks to the reliable REST-based architecture provided by Kubernetes.

From the experiments, we can observe the convenience and easiness provided by the Kubernetes ecosystem. Along with the results discussed previously, we think it is beneficial to use cloud-native technologies to manage a machine learning model pipeline for IoT because we can avoid technical debts thanks to the principled resource managing framework and the overhead cost is not significant when the size of the pipeline increases.

Finally, seeing how the cloud-native community grows and many innovations produced, we are optimistic that it will be the way forward for the IoT machine learning model management and the software industry, in general.

References

[1] J. Hermann and M. D. Balso, "Scaling machine learning at uber with michelangelo," https://eng.uber.com/scaling-michelangelo/, 2018.

[2] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM Conference on Recommender Systems*, New York, NY, USA, 2016.

[3] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 2503–2511. [Online]. Available: http://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf

[4] "Kubeflow pipelines: provides an end-to-end orchestration and managing trials/experiments," https://v1-0-branch.kubeflow.org/docs/components/pipelines/pipelines/, accessed: 2020-11-11.

[5] "Kubeflow kale: A tool to help transform jupyter notebook into kubeflow pipelines," https://github.com/kubeflow-kale/kale, accessed: 2020-11-11.

[6] "MinIO, a High Performance Object Storage," https://min.io/, accessed: 2020-11-11.

[7] S. Hemminger, "Network emulation with netem," *Linux Conf Au*, 05 2005.

[8] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128610001568

[9] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[10] N. Koshizuka and K. Sakamura, "Ubiquitous id: Standards for ubiquitous computing and the internet of things," *IEEE Pervasive Computing*, vol. 9, no. 4, pp. 98–101, 2010.

[11] Information Sciences Institute University of Southern California, "Internet protocol, https://tools.ietf.org/html/rfc791.html," *Darpa Internet Program, Protocol Specification*, 1981.

[12] S. Deering and R. Hinden, "Internet protocol, version 6 (ipv6) specification. https://tools.ietf.org/html/rfc2460.html," *Copyright (C) The Internet Society (1998). All Rights Reserved.*, 1998.

[13] N. Kushalnagar, G. Montenegro, and C. Schumacher, "Ipv6 over low-power wireless personal area networks (6lowpans): Overview, assumptions, problem statement, and goals https://tools.ietf.org/html/rfc4919.html," *Copyright (C) The IETF Trust (2007)*, 2007.

[14] "Raspberry Pi," https://www.raspberrypi.org/, accessed: 2020-11-11.

[15] "Arduino," https://www.arduino.cc, accessed: 2020-11-11.

[16] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, *TinyOS: An Operating System for Sensor Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 115–148. [Online]. Available: https://doi.org/10.1007/3-540-27139-2_7

[17] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "Riot os: Towards an os for the internet of things," in *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2013, pp. 79–80.

[18] "Android, an JVM-based Operating System," https://www.android.com/, accessed: 2020-11-11.

[19] R. Stallman, "Linux and the GNU System," https://www.gnu.org/gnu/linux-and-gnu.html, accessed: 2020-11-11.

[20] "Kubernetes, an open-source system for automating deployment, scaling, and management of containerized applications," https://www.kubernetes.io, accessed: 2020-11-11.

[21] "Resource Description Framework (RDF): Concepts and Abstract Syntax," https://www.w3.org/TR/rdf-concepts/, accessed: 2020-11-11.

[22] "OWL Web Ontology Language Overview," https://www.w3.org/TR/owl-features/, accessed: 2020-11-11.

[23] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani, "Deep learning for iot big data and streaming analytics: A survey," *IEEE Communications Surveys Tutorials*, vol. 20, no. 4, pp. 2923–2960, 2018.

[24] G. Premsankar, M. Di Francesco, and T. Taleb, "Edge computing for the internet of things: A case study," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1275–1284, April 2018.

[25] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. P. Sheth, "Machine learning for internet of things data analysis: a survey," *Digital Communications and Networks*, vol. 4, no. 3, pp. 161 – 175, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S235286481730247X

[26] R. A. FISHER, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, no. 2, pp. 179–188, 1936. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-1809.1936.tb02137.x

[27] Gaël Varoquaux and Jaques Grobler, "Scikit-learn examples: Logistic Regression 3-class Classifier analyzing the IRIS dataset by Fisher 1936," accessed: 2020-11-11. [Online]. Available: https://scikit-learn.org/stable/auto_examples/linear_model/plot_iris_logistic.html#sphx-glr-auto-examples-linear-model-plot-iris-logistic-py

[28] Vincent Dubourg <vincent.dubourggmail.com> and Jake Vanderplas and <vanderplasastro.washington.edu> and Jan Hendrik Metzen <jhminformatik.uni-bremen.de>, "Scikit-learn examples: Gaussian Processes regression: basic introductory example," accessed: 2020-11-11. [Online]. Available: https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpr_noisy_targets.html

[29] "Scikit-learn's make_blobs: Generate isotropic gaussian blobs for clustering," https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html, accessed: 2020-11-11.

[30] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[31] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[32] M. Eder, "Hypervisor- vs. container-based virtualization," *Proceedings Of The Seminars Future Internet (Fi) And Innovative Internet Technologies And Mobile Communications (Iitm), Winter Semester 2015/2016*, 2016.

[33] "Docker architecture," https://docs.docker.com/get-started/overview/#docker-architecture, accessed: 2020-11-11.

[34] R. Bias, "The history of pets vs cattle and how to use the analogy properly," http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle, accessed: 2020-11-11.

[35] M. Hausenblas, *Container Networking*, N. McDonald, Ed. O'Reilly Media, 2018.

[36] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Queue*, vol. 14, no. 1, pp. 10:70–10:93, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2898442.2898444

[37] "Google cloud platform," https://cloud.google.com/, accessed: 2020-11-11.

[38] "Azure, a Microsoft's cloud service," https://azure.microsoft.com/en-us/, accessed: 2020-11-11.

[39] "Amazon web service," https://aws.amazon.com, accessed: 2020-11-11.

[40] "Kubernetes addons," https://kubernetes.io/docs/concepts/cluster-administration/addons, accessed: 2020-11-11.

[41] S. Schelter, F. Bießmann, T. Januschowski, D. Salinas, S. Seufert, and G. Szarvas, "On challenges in machine learning model management," *IEEE Data Eng. Bull.*, vol. 41, pp. 5–15, 2018.

[42] C. Sun, N. Azari, and C. Turakhia, "Gallery: A machine learning model management system at uber," in *Advances in Database Technology - EDBT 2020, 23rd International Conference on Extending Database Technology, Copenhagen, Denmark, March 30 - April 02, Proceedings, OpenProceedings.org, ISBN: 978-3-89318-083-7, Series ISSN: 2367-2005*, 2020. [Online]. Available: https://openproceedings.org/2020/conf/edbt/paper_217.pdf

[43] MATLAB, *9.7.0.1190202 (R2019b)*. Natick, Massachusetts: The MathWorks Inc., 2018.

[44] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[45] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2016. [Online]. Available: https://www.R-project.org/

[46] B. Stroustrup, *The C++ Programming Language*, 4th ed. Addison-Wesley Professional, 2013.

[47] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[48] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud 2010*, June 2010.

[49] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, p. 107–113, Jan. 2008. [Online]. Available: https://doi.org/10.1145/1327452.1327492

[50] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[51] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[52] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[53] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[54] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 675–678. [Online]. Available: https://doi.org/10.1145/2647868.2654889

[55] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785

[56] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015. [Online]. Available: http://arxiv.org/abs/1512.01274

[57] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey," *J. Mach. Learn. Res.*, vol. 18, no. 1, p. 5595–5637, Jan. 2017.

[58] A. Chen, A. Chow, A. Davidson, A. DCunha, A. Ghodsi, S. A. Hong, A. Konwinski, C. Mewald, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, A. Singh, F. Xie, M. Zaharia, R. Zang, J. Zheng, and C. Zumar, "Developments in mlflow: A system to accelerate the machine learning lifecycle," in *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, ser. DEEM'20. New York,

NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3399579.3399867

[59] "DVC, a data versioning, workflow, and experiment management software, that builds upon git," https://dvc.org, accessed: 2020-11-11.

[60] "Microsoft's machine learning platform azureml," https://azure.microsoft.com/en-us/services/machine-learning/, accessed: 2020-11-11.

[61] "Amazon's machine learning platform sagemaker," https://aws.amazon.com/sagemaker/, accessed: 2020-11-11.

[62] "Google's machine learning platform tfx," https://www.tensorflow.org/tfx, accessed: 2020-11-11.

[63] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia, "Modeldb: A system for machine learning model management," in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, ser. HILDA '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2939502.2939516

[64] "Kubeflow: an open project for machine learning workflows deployments on kubernetes," https://v1-0-branch.kubeflow.org, accessed: 2020-11-11.

[65] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. USA: USENIX Association, 2017, p. 613–627.

[66] C. Olston, F. Li, J. Harmsen, J. Soyke, K. Gorovoy, L. Lao, N. Fiedel, S. Ramesh, and V. Rajashekhar, "Tensorflow-serving: Flexible, high-performance ml serving," in *Workshop on ML Systems at NIPS 2017*, 2017.

[67] "Basic regression: Predict fuel efficiency," https://www.tensorflow.org/tutorials/keras/regression, accessed: 2020-11-11.

[68] "Intel lab data," http://db.csail.mit.edu/labdata/labdata.html, accessed: 2020-11-11.

[69] "Argo workflow: a workflow engine for kubernetes," https://argoproj.github.io/argo/, accessed: 2020-11-11.

[70] "kube-state-metics: generate metrics from kubernetes api," https://github.com/kubernetes/kube-state-metrics, accessed: 2020-11-11.

[71] "Prometheus, an open-source systems monitoring and alerting toolkit," https://prometheus.io/, accessed: 2020-11-11.

[72] "Curl: a tool to transfer to data from/to a server," https://curl.se/docs/manpage.html, accessed: 2020-11-11.

[73] "Kubespray, a tool to deploy a production ready kubernetes cluster," https://kubespray.io, accessed: 2020-11-11.

[74] "Terraform: A tool to provision cloud infrastructure," https://www.terraform.io/, accessed: 2020-11-11.

[75] "Etcd: A key-value store solution for distribution system," https://etcd.io/, accessed: 2020-11-11.

[76] "Ansible: A tool to automate cloud deployment," https://www.ansible.com/, accessed: 2020-11-11.

[77] "Kubeadm, a tool to create kubernetes cluster," https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm, accessed: 2020-11-11.

[78] K. Hightower, "Kubernetes the hard way," https://github.com/kelseyhightower/kubernetes-the-hard-way, accessed: 2020-11-11.

[79] "Kubernetes: Ingress," https://kubernetes.io/docs/concepts/services-networking/ingress/#what-is-ingress, accessed: 2020-11-11.

[80] "Octavia: An opensource load-balancing solution designed to work with openstack," https://docs.openstack.org/octavia/queens/reference/introduction.html, accessed: 2020-11-11.

[81] "Lbaas: Load balancer as a service solution for openstack," https://docs.openstack.org/mitaka/networking-guide/config-lbaas.html, accessed: 2020-11-11.

[82] "Metallb: An load balancing solution for bare metal kubernetes cluster using standarding network protocol," https://kubernetes.io/docs/concepts/services-networking/service/#nodeport, accessed: 2020-11-11.

[83] "FRRouting: a free software suite for IP routing," http://docs.frrouting.org/en/latest/overview.html, accessed: 2020-11-11.

[84] "Kubernetes: Service node port mode," https://kubernetes.io/docs/concepts/services-networking/service/#nodeport, accessed: 2020-11-11.

[85] F. Chollet *et al.*, "Keras, a deep learning api running on top of the machine learning platform tensorflow," https://github.com/fchollet/keras, 2015.

[86] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[87] K. Manaouil and A. Lebre, "Kubernetes and the Edge?" Inria Rennes - Bretagne Atlantique, Research Report RR-9370, Oct. 2020. [Online]. Available: https://hal.inria.fr/hal-02972686