Automation and Orchestration for Machine Learning Pipelines

A Study of Machine Learning Scaling: Exploring Micro-service architecture with Kubernetes

Master's thesis in Complex Adaptive Systems

FILIP MELBERG

VASILIKI KOSTARA

DEPARTMENT OF PHYSICS

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Master's thesis 2024

Automation and Orchestration for Machine Learning Pipelines

A Study of Machine Learning Scaling:
Exploring Micro-service architecture with Kubernetes

FILIP MELBERG

VASILIKI KOSTARA

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2024

Automation and Orchestration for Machine Learning Pipelines
A Study of Machine Learning Scaling:
Exploring Micro-service architecture with Kubernetes
FILIP MELBERG, VASILIKI KOSTARA

Supervisors: Hamid Ebadi, Infotiv Technology Development &
Giovanni Volpe, Department of Physics
Examiner: Giovanni Volpe, Department of Physics

Master's thesis 2024
Department of Physics
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Chalmers digital printing
Gothenburg, Sweden 2024

Automation and Orchestration for Machine Learning Pipelines
A Study of Machine Learning Scaling:
Exploring Micro-service architecture with Kubernetes
FILIP MELBERG, VASILIKI KOSTARA
Department of Physics
Chalmers University of Technology

Abstract

Although Machine Learning (ML) has been around for many decades, its popularity has grown tremendously in recent years. Today's requirements show a great need for the development and management of ML projects beyond algorithms and coding. The aim of this thesis is to investigate how a minimal team of engineers can create and maintain a ML pipeline. To this end, we will explore how a Machine Learning Operations (MLOps) pipeline could be created using containerization and container orchestration of micro-services. After relevant research, the result is a minimal, on-premises Kubernetes cluster set up on physical servers and Virtual Machines (VMs) running the Ubuntu Operating System (OS). The cluster consists of a master and two worker nodes, which are used for two main ML frameworks. Populating the cluster with more nodes is straightforward, which makes scaling a simple task. Additionally, a locally shared folder on the network is mounted in the cluster as an external storage and the cluster is configured to access either a local or a cloud-provided container registry. Once the cluster is set up and running, an application is launched to train the YOLOv5 model on a custom dataset. Later, Distributed Data Parallel (DDP) training is performed on the cluster using PyTorch, TorchX, PyTorch Lightning and Volcano.

Keywords: DevOps, Docker, Kubernetes, Micro-service, ML, MLOps, PyTorch, YOLO

Acknowledgements

The research for this master thesis was carried out within the SMILE IV project, financed by Vinnova, FFI, Fordonsstrategisk forskning och innovation under the grant number 2023-00789 [75]. We would like to express our deepest gratitude to our project supervisor Dr. Hamid Ebadi, Senior Researcher and Competence Leader at Infotiv Technology Development, for his invaluable guidance and his constant engagement in our thesis project. We would also wish to extend our gratefulness to Maria Kindmark Alemyr, Consultant Manager at Infotiv Technology Development, as well as the company personnel, for providing beneficial advise and access to crucial resources, such as software tools and technical infrastructure. Finally we want to thank our supervisor and examiner at Chalmers University of Technology Giovanni Volpe, Senior Lecturer at Institution of Physics at Gothenburg University, for his direction concerning administrative processes.

Filip Melberg and Vasiliki Kostara, Gothenburg, June 2024

Contents

List of Figures

List of Tables

List of Abbreviations

Chapter 1

Introduction

The use of Artificial Intelligence (AI) by professionals and communities is becoming increasingly prevalent, encompassing a wide range of applications, from complex challenges to everyday entertainment purposes. A study conducted by Stack Overflow based on their 2023 developer survey [54] revealed that all types of professional developers are either planning to or are already utilizing AI tools in their projects. Furthermore, the overall sentiment towards AI is overwhelmingly positive [38]. With the rapid growth of AI, however, an important drawback is overtime maintenance and evolution. Sculley et al. [51] discusses how realization of ML projects bear technical debts for various reasons, aiming to raise awareness about the necessary commitments and best practices regarding ML projects. These issues rise due to challenges that consist of more than developing ML code, when developing and operating ML systems. In fact, according to the same publication, only a small fraction of the applicable ML projects consist of ML code, marked by the dark box in Figure 1.1, while the rest of the components can be quite vast and complex. Instead, as the representation in Figure 1.2 suggests, a ML project may actually require the adoption of many roles with different skills, such as data science and Development and Operations (DevOps) engineering among others [29]. Therefore, MLOps are practices that strive to narrow the distance between these roles and potentially combine the required skills of an ML project. MLOps are similar in principle to DevOps, with focus on ML. They incorporate automation and monitoring of ML pipelines, often end-to-end, and help optimize workflow and minimize errors [47]. Despite this, MLOps is a fairly new subject and poses limitations, which long existing DevOps surpass.

Currently, many software development researchers and engineers are executing Continuous Integration/Continuous Deployment (CI/CD) processes manually, including data analysis, model training and validation [20]. This approach is prone to inefficiencies, errors and delays in a ML pipeline, such as varying performance

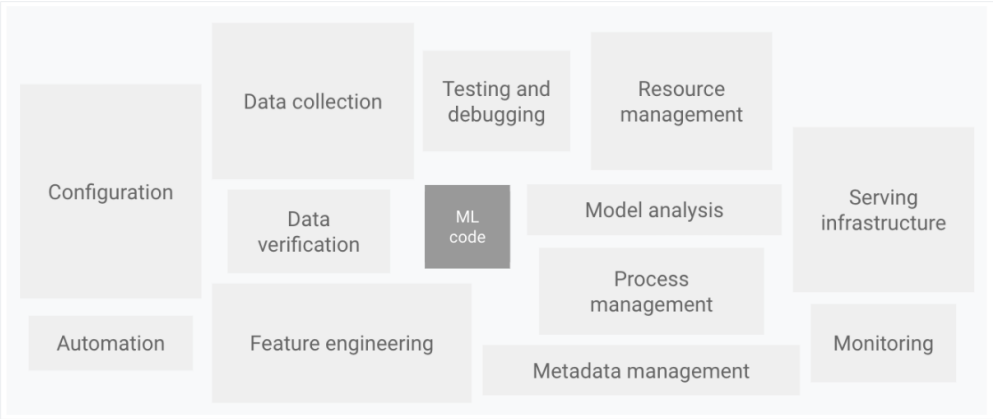

Figure 1.1: Components of an ML project according to [20] and [51].

during training and deployment. In response to these issues, we proposed utilization of DevOps and MLOps as a robust, systematic and automated approach to managing the life cycle of an ML project. By including containerized training and testing stages in an ML pipeline, we implemented the foundations of a seamless process characterized by reliability and can be further developed and enhanced in the future. Inspired by the vast variety of options, we researched literature and software solutions that are applicable in both an academic and a corporate environment. Our main implementation includes a pipeline with containerized applications using Git, Docker and Kubernetes. The use of these popular open source DevOps tools ensures the universality of our thesis and potential benefits to future implementation. To determine which methods would be suitable for our thesis, we also experimented in learning about relevant open-source ML projects. Although successfully configuring and containerizing a ML project can be time-consuming and challenging, it is ultimately worth it.

The main goal of this thesis is to learn and engineer the parts of an ML project that does not involve developing algorithms and code (see Figure 1.1). Instead we wish to gain a global overview of what a full ML project can entail and gain experience assuming different roles (see Figure 1.2). With this objective, we explored and questioned: *Is a small team of engineers sufficient to achieve a fully functional end-to-end ML pipeline and, if so, what are the high-level criteria it should fulfill?*

This report is sectioned in five additional chapters. Chapter 2 discusses primary definitions and theoretical background of material relevant to this thesis, followed by Chapter 3, which presents the realization of the material. Afterwards, Chapter

4 shows the results of our methodologies, while Chapter 5 discusses the limitations posed during the thesis and suggests possible material for future consideration. Finally, Chapter 6 provides a conclusion to this thesis report.

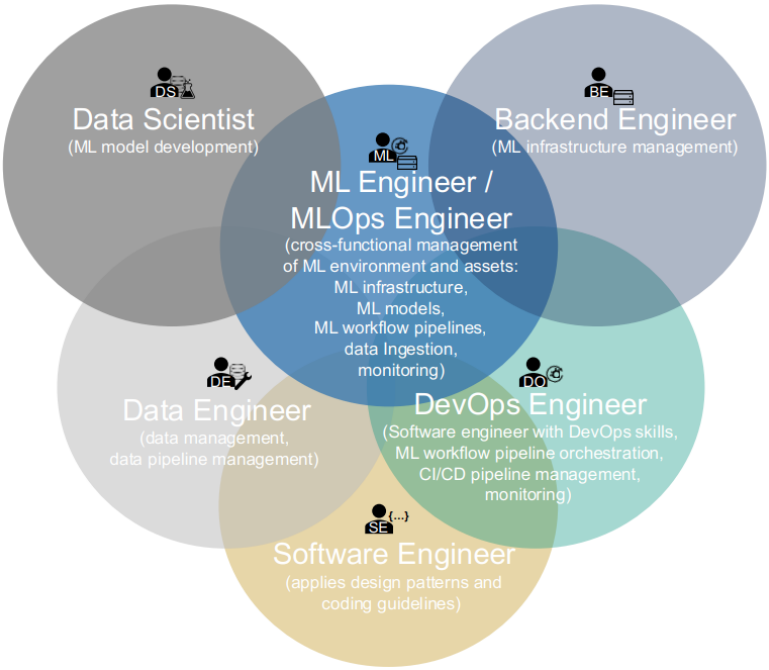

Figure 1.2: Intersections between ML/MLOps engineers and contributing roles as illustrated by Kreuzberger, D., Kühl, N., and Hirschl, S. [29]

Chapter 2

Background

In this chapter, definitions and theoretical background are provided regarding the principles and software relevant to the scope of this thesis. In particular, this chapter mainly discusses MLOps and DevOps, VMs, containers and their orchestration with particular focus on Kubernetes and, finally, file sharing.

2.1 Machine Learning

ML comprises the greatest subset of AI. Inspired by the human learning processes, ML algorithms have become a key component to many solutions both in academic research and corporate applications. From Large Language Models (LLMs) to object detection and many more applications, AI projects can employ a big variety of ML and Neural Networks (NNs) to achieve results tailored to contemporary demands. What is more, entire online communities, such as Hugging Face [23] and Kaggle [25], are constituted with sole interest in AI and ML. As a result, open source AI software is readily available by many corporations, while ML libraries and platforms like PyTorch [39] and TensorFlow [1] offer algorithms and code to suit a majority of ML projects.

During this thesis, ML work published in online communities often brought inspiration and new knowledge. Initially, a framework called Continuous Machine Learning (CML) was tried, which offered the option of integrated CI/CD with ML. However, the main part of this thesis consists of object detection of forklifts and people in relation to the SMILE IV project [75], which hosts code for a warehouse simulation. As a last interesting and advanced ML practice, DDP training has been implemented for a Kubernetes cluster, a software tool that will be discussed in Section 2.5. These parts use mainly PyTorch, as well as other tools, which will be described in more detail below.

Continuous Machine Learning (CML)

CML is an open source software tool utilized for implementing CI/CD pipelines in machine learning projects [14]. In particular, CML is a library of functions used inside CI/CD runners to make ML compatible with popular distributed version control platforms. It can be used to develop ML workflow, such as model training and testing, compare ML models and monitor changes in datasets.

You Only Look Once (YOLO)

You Only Look Once (YOLO) is an object detection algorithm that employs deep learning in computer vision and, since it was first introduced, it has been developed to offer high speed and accuracy, while being simple and straightforward. Specifically, YOLO is a one-stage detector [55] using a Convolutional Neural Network (CNN) architecture, that improves the result with each iteration and predicts spacial association of bounding boxes with class probabilities [48]. Until today, several versions of YOLO have been released, however, the fifth version YOLOv5 is of particular relevance to this thesis. YOLOv5 uses several convolutional layers including a Spatial Pyramid Pooling Fast (SPPF), which ensures computational efficiency and captures information at various scales, followed by upsampling to increase resolution [74]. This fast and powerful version uses PyTorch and is maintained by Ultralytics [18].

Distributed Data Parallel

The process of training ML algorithms on several machines is called distributed training and can be done with different methods. In this text the focus will lie on the DDP method where the data is partitioned among several different workers in a Kubernetes cluster. Common reasons for using distributed training include:

- performing training on enormous data sets, which take too much time to go through
- when either all the data cannot be accessed or stored on a single machine
- when the data is only accessible on different locations
- when the data should fit entirely in the memory to be preprocessed

A typical training iteration may consist of three steps. First, in forward propagation the NN outputs a best guess, generating losses and labels. Second, in backpropagation weights and thresholds are optimized using gradients generated by taking into account values and losses from the best guess. Finally, an optimizer applies gradients to update these parameters. What is specific about DDP training [30] is that the gradients are communicated across nodes before the third step.

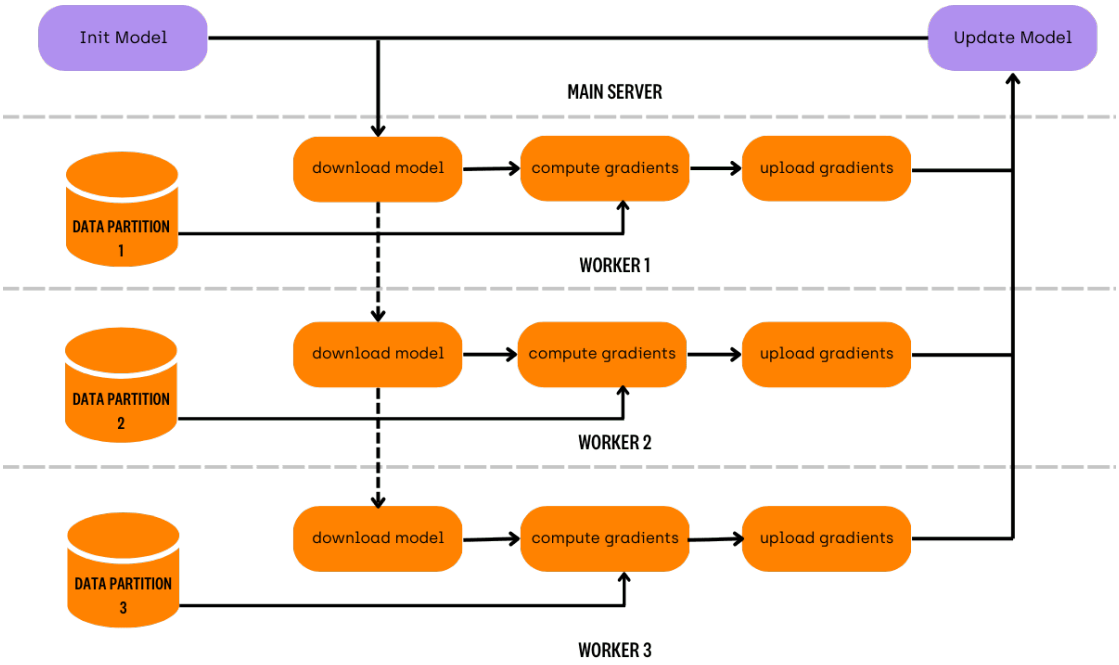

Figure 2.1: During the DDP process the dataset is split and each partition is processed by a separate worker. The workers compute gradients which are synchronized in a main server [41].

Therefore, all model partitions are updated according to the same trend and rate of change. Finally, a local optimizer is applied to each partition.

Specifically, during the DDP process the model to be trained is first initialized on a main server. Each Kubernetes worker is responsible for a partition of the training data and downloads a copy of the model, from the main server. At the worker nodes, gradients of the loss function are calculated on the batches of training data for which they are responsible. The calculated gradients are then sent to the main server where the model is updated. The model update can happen either synchronously or asynchronously. During synchronous updates the main server will wait for all workers to finish calculating and sending their gradients. This may result in slower training because the process is limited by the slowest worker. By updating asynchronously, the main server will update the model as soon as it receives a gradient from a worker, eliminating the limitation by the slowest worker. However this method comes with its own trade-offs in the form of potentially worse convergence due to race conditions[41]. Race conditions are

flaws that occur due to dependence on timing or sequence of events in a program, leading to various unwanted consequences [49]. In the context of DDP training, a potential race condition can occur when a Kubernetes worker performs an update based on an obsolete model and overwrites the updates of more recent models computed on other workers. This can result in issues with model performance and convergence.

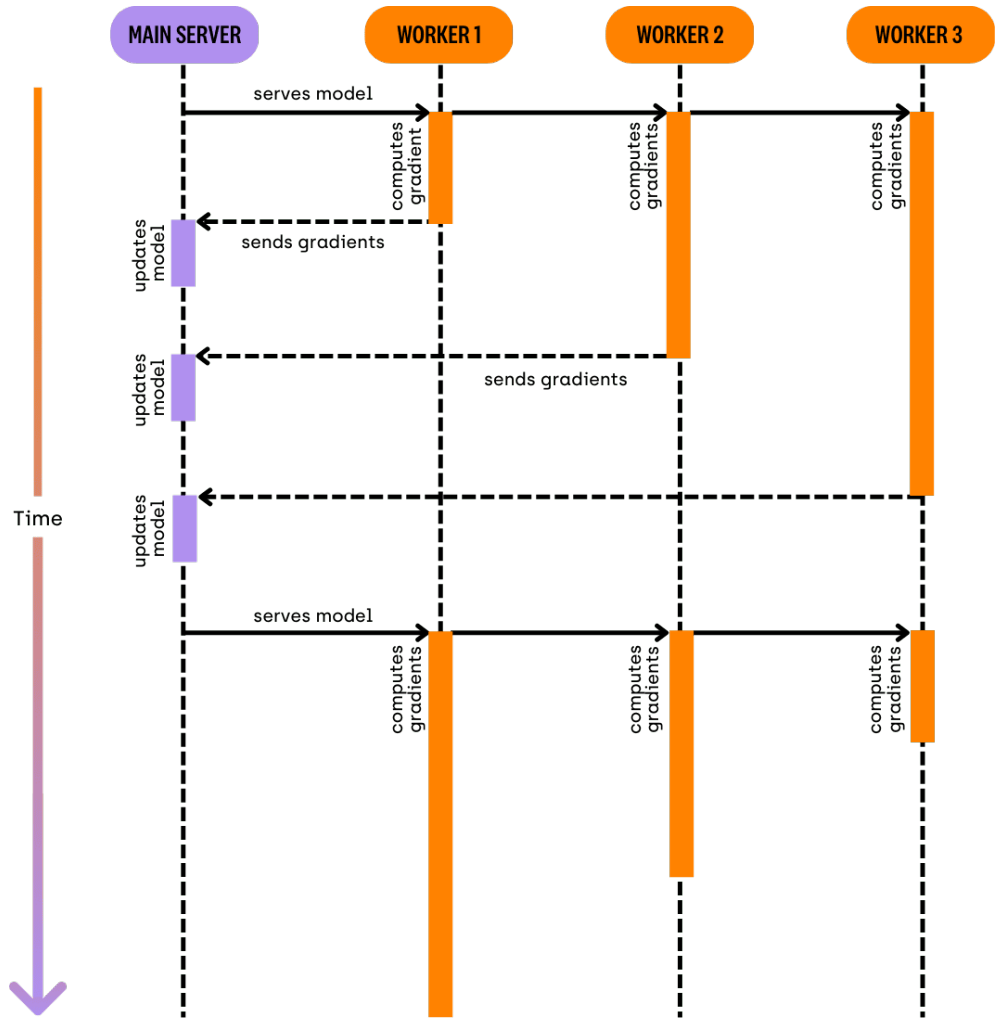

Figure 2.2: During synchronous updating the main server waits for all workers to complete their calculations [41].

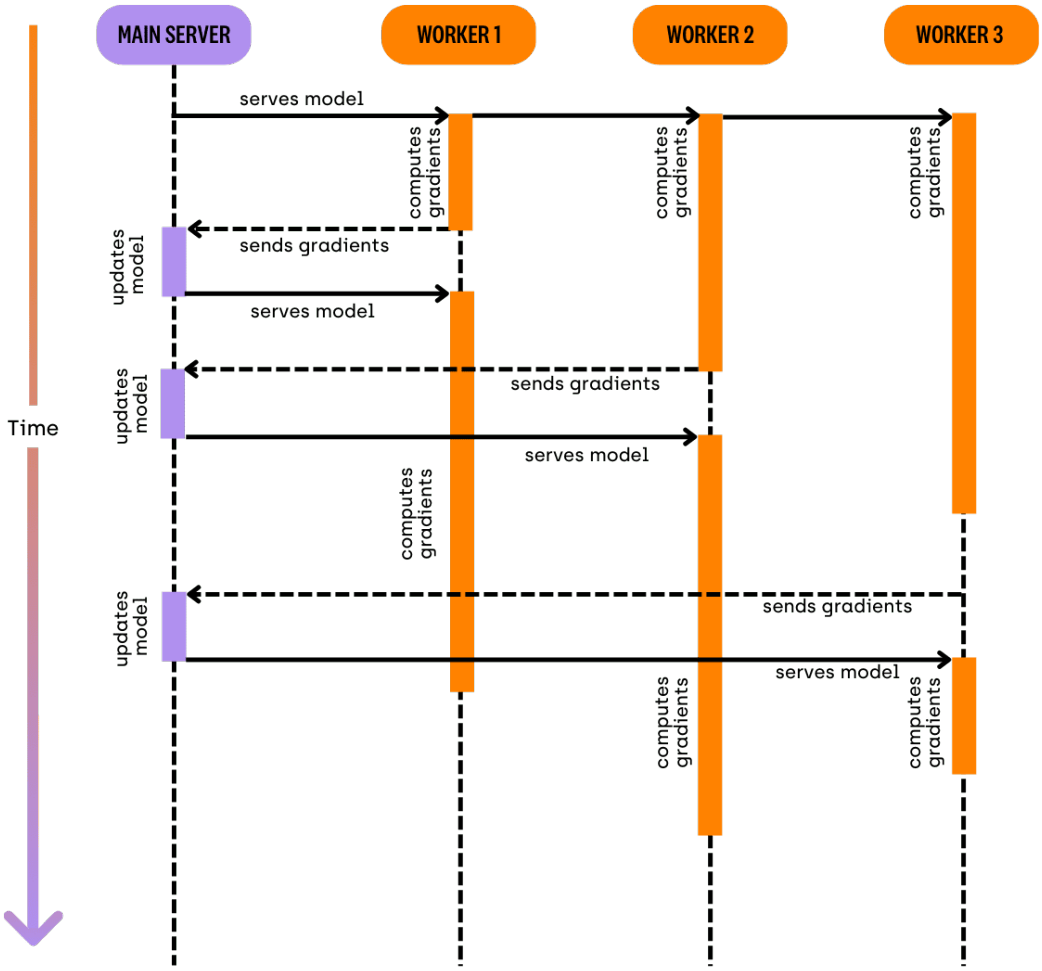

Figure 2.3: When updating asynchronously the main server updates the model instantly when receiving a computed gradient [41].

2.2 DevOps

Applications, after they are created, need to be made available for clients to use. This involves several steps that can be categorized as either development or operations. During development the first step could be to plan and define an application's purpose as well as functionality. The next step is to code and create the actual application followed by building and testing to ensure that it functions as

intended. However making the application available to customers usually involves going through several steps to deploy it in a different environment, such as a Linux server, and then making sure that it runs without problems. These steps are part of operations and can include preparing servers by installing necessary tools and packages, opening ports and performing network configuration and monitoring the application by examining user data.

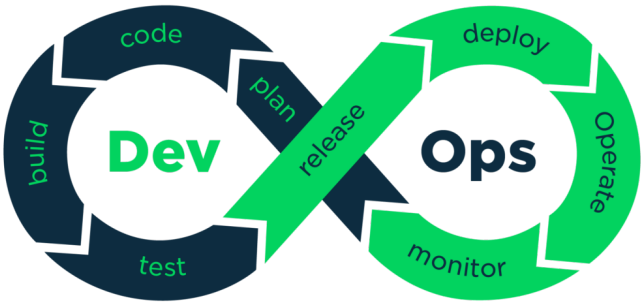

Figure 2.4: The figure illustrates different DevOps steps in an infinite loop [10].

DevOps attempts to merge development and operations to increase the speed of delivering high quality code, rapidly update deployment and improve collaboration and overall product quality. DevOps is hard to define but is often said to be a set of practises, tools and cultural philosophy. At the center of DevOps is CI/CD which is about automating all the required steps between planning a new application and deploying it.[6]

2.3 MLOps

MLOps is the process of automating ML using DevOps methodologies, according to Noah Gift and Alfredo Deza [17]. In practice, MLOps combines ML application development with system deployment and operations to create standardized and automated processes along the ML lifecycle, depicted in Figure 2.5.

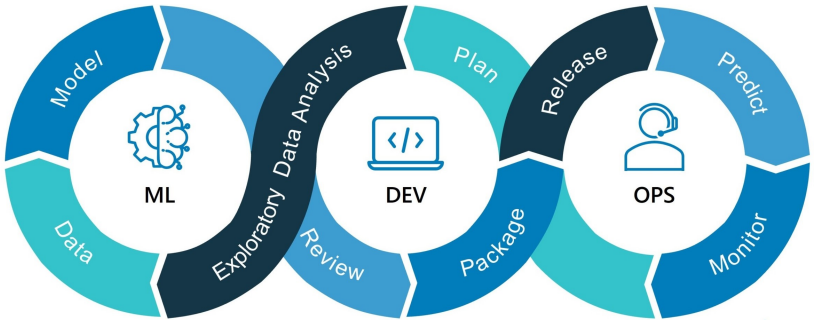

Figure 2.5: ML lifecycle as illustrated by Neil Analytics [35].

Adopting this approach, a completed framework requires technical and management principles, that aim to increase velocity and high-quality software creation [17], alongside a strong consideration of best practices. Figure 2.6 depicts these principles linked to fundamental components of MLOps.

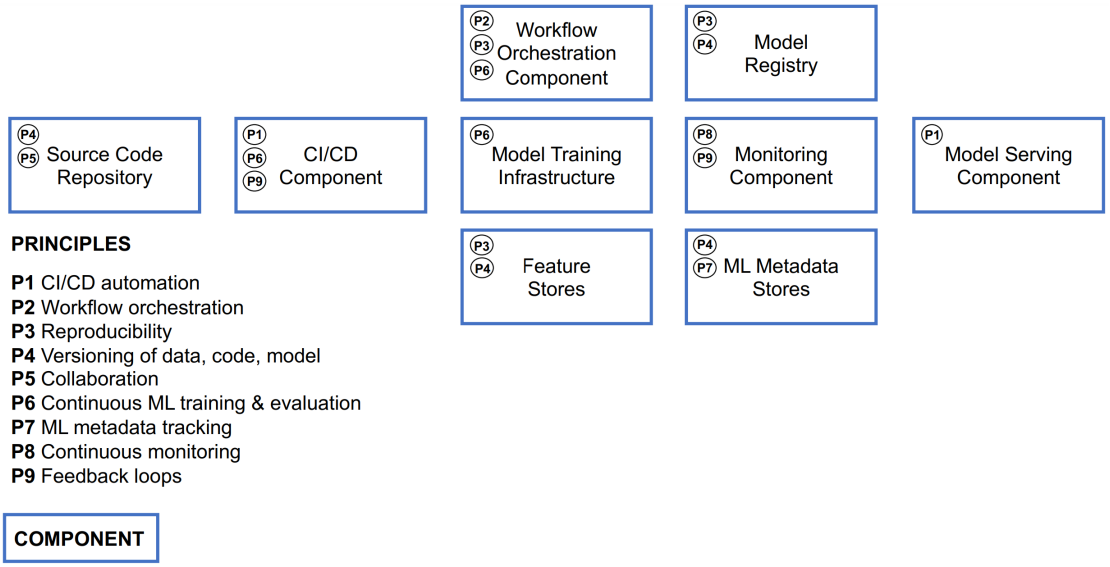

Figure 2.6: MLOps implementation principles and components as illustrated by Kreuzberger, D., Kühl, N., and Hirschl, S. [29].

2.4 Containerization

Containerization is the process of encapsulating applications and their dependencies into a single container that can easily be shipped to other systems [32]. In contrast to VMs, containers share the host kernel and OS and do not require

separate virtual ones [33]. The concept relies on building an image from the application and its dependencies and then pushing it to a remote or local image registry. This image can later be pulled from the registry and used to start the application on another system seamlessly. To automate the build process, a text file named `Dockerfile` that describes the process can be created [32]. Applications inside containers can act as micro services in a larger framework, which are relatively simple to debug and replace.

When it comes to containerization, the most popular and lightweight open-source platform is Docker [13], ranking top tool in the Stack Overflow Annual Developer Survey for 2023 [54]. In the scope of this thesis, a `Dockerfile` will be used as a recipe to build an image for a containerized ML application. The application is then managed by a Kubernetes cluster and registered by either Docker Hub or a local container registry.

2.4.1 Container Runtimes

The container runtime is the software that is responsible for running containers on the host system. It is able to pull images from a registry from which it then starts containers. In more technical terms, a container runtime limits, accounts for and isolates system resources for a collection of processes. Containers operate on top of the operating system, and any virtual machines, and are thus independent of the system hardware. Docker offers one the most famous container runtimes and is widely used by different organizations to enable applications running in different environments, on different hosts and with minimal effort [36]. In this project the Containerd [11] runtime will be used instead of the Docker runtime.

The container execution process consists of several steps. First it creates the container and initializes its environment from an image containing dependencies and the actual application to be run. Next the runtime starts the container and ensures that the application is running inside. Afterwards the container runtime keeps monitoring the container and ensures the application is working by restarting the container if it fails. To be able to isolate the container the runtime utilizes namespaces and cgroups (control groups) and ensures that processes inside containers do not disrupt host or other container processes [36].

2.4.2 Container Runtime Interface

A CRI is an interface that enables the Kubernetes cluster to communicate with the container runtime, for instance Docker or Containerd, since it cannot directly operate the container runtime. Kubernetes, and its componenets, will be explained

in Section 2.5. Some examples of container runtimes that provide a CRI for kubernetes include Mirantis Container Runtime, CRI-O and Containerd. At the time of writing Docker engine needs the cri-dockerd adapter in order to work with Kubernetes. Figure 2.7 illustrates an example where the kubelet manages containers by calling the Containerd runtime, through the CRI, which in turn will start up the specified amount of containers.

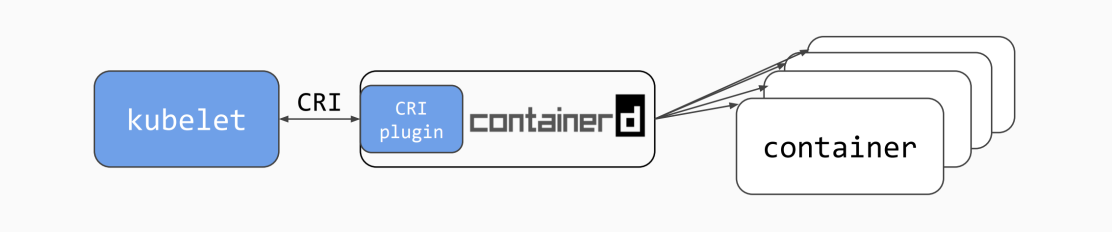

Figure 2.7: The kubelet (Kubernetes node agent) is a componenet that communicates with the container runtime through the CRI [11].

2.4.3 Container Registry

An important feature of containerization is the registry, where one can push and pull their own images [32]. In other words, container registries store any images relevant to a project and can be maintained either locally or remotely. In the context of container orchestration and specifically Kubernetes, a container registry is not necessarily part of the cluster. However configuring communication between a cluster and a registry is a vital component for any containerized project, mainly involving the creation of a Kubernetes secret [66]. These practices will be analyzed more thoroughly within this chapter, as well as in Methods (Chapter 3).

Private Registry

Setting up a private container registry can offer significant control over a developing project. Firstly, this solution offers full control over storing and accessing container images, which can prove useful for those that want to develop private projects instead of hosting them publicly. Additionally, a developer can set up and configure their private registry in full control and according to project requirements. A possible drawback with a private container registry, as with every private project, is the requirement for more comprehensive knowledge and experience in this particular matter.

Docker Hub

Docker Hub is a public container registry platform for finding and sharing Docker images [12]. It is professionally developed and maintained while also offering many pre-built and community provided images that can be used instead of developing from the ground up. However, this solution is unsuitable for projects intended to be developed internally.

2.5 Container Orchestration

Managing large numbers of containers can be difficult. Container orchestrators are designed to handle thousands of containers; automating things like upgrading, starting, stopping, monitoring and scaling. They perform important container life cycle tasks in a small amount of time effortlessly. Container orchestration is based on declarative programming where rather than specifying the steps for the desired output only the output is defined and then the tool achieves it automatically. Container orchestrators can be self built or managed, with this document focusing on the former and its challenges [19].

2.5.1 Kubernetes (K8s)

Kubernetes (K8s) is a versatile open source platform for automating, scaling, managing and deploying containerized applications [62]. It provides services, support and tools and ranks high in popularity and usage by professionals according to the Stack Overflow Annual Developer Survey for 2023 [54].
The Kubernetes container orchestrator manifests itself in the form of a cluster that uses its own abstractions to manage micro services. A cluster consists of a control plane (master node) and worker nodes which are governed by the control plane. Each cluster has one main control plane that can have multiple copies for redundancy in case the main goes down. Should a worker node fail the control plane will automatically reschedule all processes on another available node, guaranteeing that applications keep running.

Applications are managed through pods. A pod is a Kubernetes abstraction which means that it has no meaning by itself and can instead be loosely seen as a reference to a container in the most common cases. The container is the actual object that contains the running application while the pod acts as a wrapper around it. In the context of Kubernetes all applications are put in pods that in turn are scheduled onto nodes. Every pod has its own internal Internet Protocol (IP) address that is used to communicate with it from within the cluster. Pods are the smallest units that can be deployed on the cluster and they are ephemeral,

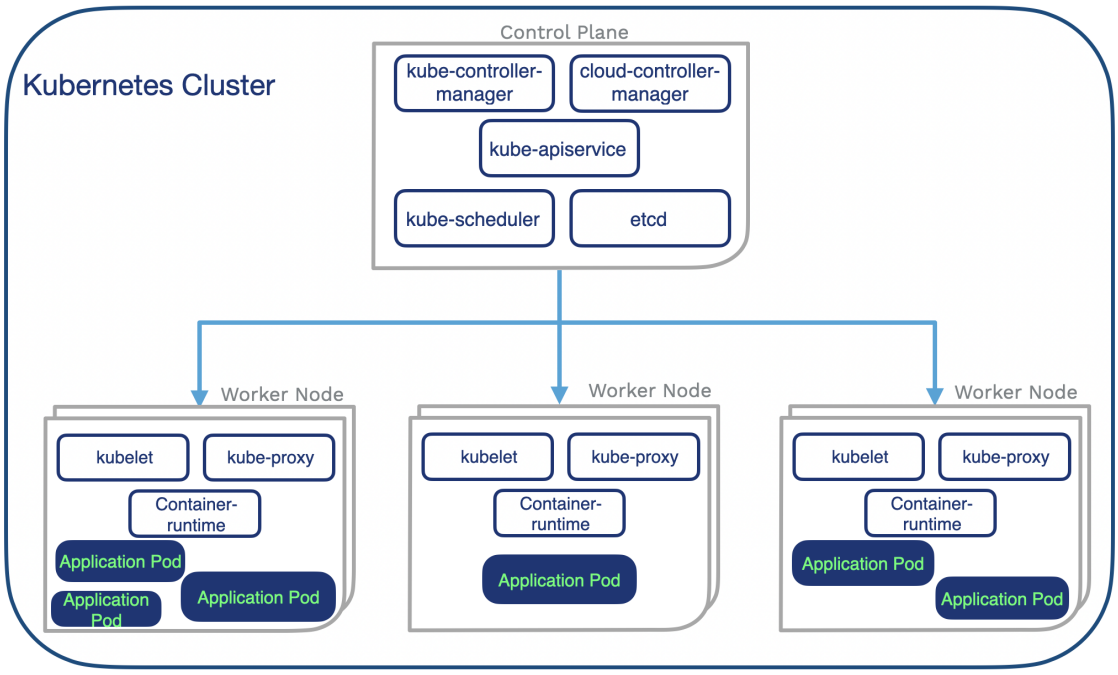

Figure 2.8: Figure illustrating a Kubernetes cluster with four nodes and their main components (One control plane and three workers). [44]

meaning that data should never be stored permanently in them since it will be lost if the pod goes down. In more advanced use cases one pod can contain a group of tightly coupled containers that share namespaces and filesystem volumes. One such example is a pod consisting of a main application container and a sidecar container. The Sidecar container is a secondary container that runs in the same pod as the main container and can for instance display or send logs from the main application container.[65]

Since every pod has its own unique internal IP address it is inconvenient to use it in applications that access pods. Every time a container is restarted it gets placed in a new pod with a new IP address and can thus no longer be accessed through the old address. Problems also arise when trying to access several pods in a group that comprise a network application or if a single pod application needs several copies of itself. For these reasons Kubernetes uses Services which are abstractions that can be thought of as labels (for groups of pods) with internal static IP addresses. Other applications are configured to access the service rather than the pod(s) thus making them independent of the number of copies or actual pods.

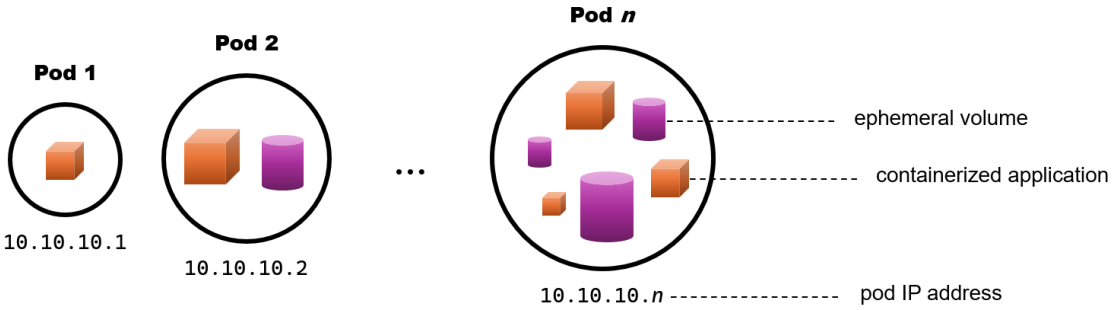

Figure 2.9: Figure illustrating Kubernetes pods. The single container pod (Pod 1) is the most common although pods can contain multiple containers and volumes. The illustration is inspired by [69].

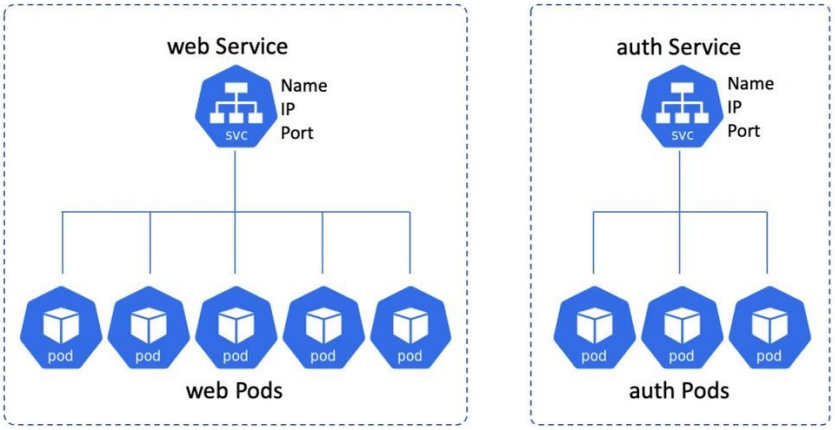

Figure 2.10: Figure illustrating two Kubernetes services "web service" and "auth service". Groups of pods are exposed on the network through services [21].

Kubernetes clusters are not meant to store any data permanently. Instead external storage spaces are mounted into the cluster. These could be anything from a cloud storage and storage cluster to a folder on a local computer. Kubernetes uses Persistent Volumes to assign storage that the cluster is allowed to use. Persistent Volumes are cluster resources just as nodes, according to the Kubernetes docs, however it is important to note that they are abstractions and not actual storage volumes so they need to be mapped to a real resource. After defining a storage in the cluster through a PersistentVolume, parts of the storages can be requested with PersistentVolumeClaims that define how much of the storage a pod can use as well as access modes. Applied PersistentVolumeClaims will automatically find suitable PersistentVolumes in the cluster and bind to them. According to the Kubernetes docs pods can be seen as consuming node resources while PersistentVolumeClaims

consume PersistentVolume resources [70].

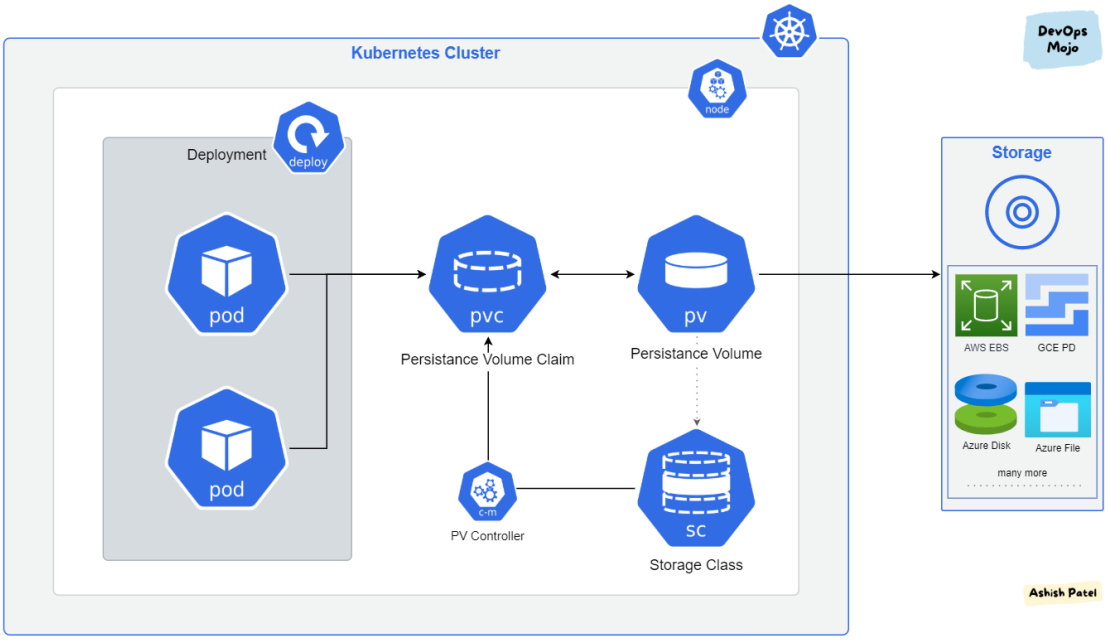

Figure 2.11: The external storage is outside of the cluster. It is defined in the cluster with a PersistentVolume (pv) and then a PVclaim (pvc) is bound to it. The PVclaim defines how much storage the pods can consume. [5]

Essential concepts and components related to a functional cluster

This subsection will go into some important components that play vital roles in a Kubernetes cluster.

- **Swap Memory**
  Swap is a preconfigured space on the hard disk, where a page of Random-access Memory (RAM) is copied [31]. However, according to the official documentation an important requirement for `kubeadm` is the deactivation of swap space [60], which has created debate for Kubernetes communities in the past. Even though a beta support for swap memory on Linux has been released since 2023, according to the documentation of the same release, activated swap can pose risk to environments subjected to performance constraints [22]. Therefore, this work follows the official protocol by deactivating swap every time `kubeadm` is used.

- **Nodes**
  A Kubernetes node is a physical machine (server/computer) or virtual machine that hosts pods. All nodes have a kubelet, container runtime and a kube proxy. Nodes can be categorized, for instance into cpu intensive or memory intensive node groups. Nodes are never supposed to save any type of data internally since it will be lost if the node fails.

- **Pod Network Add-ons**
  A Kubernetes pod network add-on can provide different functionalities however as a minimum they allow nodes in a Kubernetes cluster to communicate with each other. Pod network add-on's route traffic between hosts but can also facilitate load balancing and service discovery (automatically detecting devices on a network) [7]. This project uses Flannel [16] which is a simple pod network add-on that focuses on handling traffic between nodes and not between individual pods. It is a layer 3 IPv4 network that provides temporary IP addresses (subnet lease's) to all the nods in the cluster which eliminates the need to configure ports. Another example of a more advanced pod network add-on that also provides network policy is Calico [72].

- **Kubelet**
  This is an important cluster component that acts as a node agent with several key responsibilities. The kubelet runs on all the nodes and executes commands from the control plane, serving as the link between worker nodes and master node. It is fundamentally responsible for managing nodes, making sure they are healthy and running and executing pods [4]. The kubelet can loosely be imagined as the "brain" of each node that invokes the use of different "limbs" such as the container runtime for starting containers.

- **etcd**
  To be able to track different states of a cluster, Kubernetes implements the etcd key-value storage system. It stays consistent thanks to the Raft Algorithm [45] which, in short, elects a leader node that contains a value agreed upon by the rest of the cluster. The keys are usually resources while the values represent states. In the context of Kubernetes the etcd stores information such as current cluster state, desired cluster state, configuration resources and runtime data. This allows the control plane to make appropriate adjustments, based on the information in the etcd, to ensure that nodes are not overloaded and that tasks get scheduled on nodes with the required resources [3].

- **`kubeadm`**
  Kubeadm is a bootstrapping tool built to start a minimum viable Kubernetes cluster according to best practises. It starts the kubelet service and deploys the required pods on the control plane etc. automatically when running `kubeadm init`. For easy cluster resetting and joining worker nodes the `kubeadm reset` and `kubeadm join` commands are provided. Kubeadm does not install "nice to have" features such as the Kubernetes dashboard. [60]

- **`kubectl`**
  To interact with the Kubernetes control plane a command line tool called kubectl can be installed. Everything from querying number of nodes or pods to starting new jobs or deployments can easily be done with kubectl. The tool will by default look for a configuration file in the `~/.kube/` directory, that contains information on where the control plane is hosted and how it can be accessed. All cluster communication is conveniently done with kubectl. [57]

- **Kube Controller Manager**
  Kubernetes uses a collection of different controllers to monitor the state of the cluster. These are essentially infinite loops that watch for changes and attempt to bring the current state of an object to the desired state. One example is the Node controller which ensures that nodes are healthy and if not the workload is moved to another node. Another example is the replication controller which monitor the number of pod replicas and attempts to always keep the desired number of pod replicas running. There are many more controllers in Kubernetes and all of the core controllers are embedded in the Kube-controller-manager, which is a single binary file. [73]

- **Cloud Controller Manager**
  If you are using cloud services in your Kubernetes cluster the cloud provider will require its own logic, for instance when setting up a node. The cloud-controller-manager embeds cloud-specific control logic and essentially links your cluster to the cloud providers API [56].

- **Kube API Server**
  The kube-api server exposes the kubernetes Application Programming Interface (API) to end users and different parts of the cluster that use the API. Operations on the cluster are performed through the Kubernetes API. Tools like kubectl and kubeadm use the API [68] [63].

- **Kube Scheduler**
  This is a process running on the control plane that is responsible for matching pods with nodes. Newly created pods are initially unscheduled so the kube scheduler automatically places the pod on an appropriate node, taking into account resource requirements, node workload and more. The kube-scheduler continuously checks for unscheduled pods and can reschedule pods onto other nodes in the case of node failure. Kube-scheduler distinguishes itself from the kubelet in that it only schedules pods while the kubelet runs the pods [53] [61].

- **Kube Proxy**
  Kube-proxy runs on all the nodes in the cluster and maintains a network routing table that maps service IP addresses to pod IP addresses. When a service is created with a corresponding set of pods, the kube-proxy component will make sure that all traffic incoming to the service will be redirected to the pods in the service. The key responsibility of the kube-proxy is to watch for changes in Kubernetes cluster services and translate these changes into network rules [27].

- **Pods**
  Pods are the smallest deployable objects in a Kubernetes cluster. They can contain one or several closely related containers and each pod has its own internal IP address that is lost when the pod goes down.

- **Services**
  Services expose groups of pods on the network and provide a static IP address that can be used to access pods, and their replicas, connected to the service. It also facilitates pod load balancing.

- **Deployments**
  Deployments ensure that pods and their replicas keep running indefinitely. If a pod goes down a new will be created to keep the desired number of pods always available. An NGINX server is an example of an app that is usually run as a deployment.

- **Jobs**
  Jobs, contrary to deployments, run tasks that are not active indefinitely. A job will keep retrying to execute its pods until a specified number of them successfully terminate or the maximum amount of retries is reached. An example application that could be run as a job is ML model training. When

the model has been successfully trained the job finishes and its logs and results are saved.

- **Persistent Volumes**
  Persistent volumes are storage resources in the cluster. They need to be mapped to an actual storage such as a directory on a computer or a cloud storage.

- **Persistent Volume Claims**
  Persistent Volume Claims bind to Persistent Volumes and consume available storage in the cluster. They are used by pods.

- **Secrets**
  These are objects that contain sensitive data. For instance, when trying to access private registries the log-in credentials can be stored as secrets in the cluster to be used during container creation and image pulling. Secrets are not encrypted by default and anyone with access to the cluster API consequently has full access and control of all secrets.

- **Token**
  A bootstrap token or simply token is a means of authentication for a Kubernetes cluster. Unless configured or otherwise specified, a token is generated during the initialization of a new cluster and can be used when joining nodes to an existing cluster. A token is a specific type of secret and typically has an expiration date, after which a new token can be generated for relevant authentication processes. The token is a string that must satisfy the regular expression `[a-z0-9]{6}.[a-z0-9]{16}`, a string comprised of six alphanumeric characters separated by a dot from sixteen alphanumeric characters. Analytically, the first part is considered public information and, after the dot, the second part is secret.

Kubernetes Dashboard

The Kubernetes dashboard is a web-based Graphical User Interface (GUI) providing an overview of everything related to a Kubernetes cluster [59]. It offers a user-friendly visual representation of resources and applications in the cluster.

Volcano and TorchX

Volcano is a high performance system built on Kubernetes. It provides batch scheduling, that is often useful when running DDP workloads on Kubernetes clus-

ters, and is integrated with popular libraries and frameworks [77] as illustrated in Figure 2.12. Combining PyTorch Lightning, Volcano and TorchX allows for seamless DDP training on Kubernetes. Volcano has the ability to schedule groups of pods (PodGroups), which is required for the job launcher, TorchX [42], to work with Kubernetes. PyTorch Lightning is a lightweight deep learning framework assisting with ML development and scaling [15]. Among other functionalities, it provides the ability to handle device placement and run an ML model on different servers and, when combined with Volcano, it enables distributed training across multiple nodes within a Kubernetes cluster.

Figure 2.12: Volcano works on top of Kubernetes [76].

2.6 Miscellaneous

This section provides a descriptive background of the tools and methods pertinent to this thesis, which contribute to the overall research. Although these topics are either already widely discussed or do not fit neatly into the Sections above, they are deemed essential for the effective outcome of this thesis work. These include version control, virtualization and file sharing.

2.6.1 Version Control (Git)

Git is a significantly popular and powerful tool for storing and processing data, making it unique in comparison to other version control systems [9]. Specifically, Source Code Management (SCM) ensures code storing and data versioning for any framework, including ML projects. Thus, each version of the code and the data is documented and shared among collaborators [26]. In the context of this thesis,

the relevant source code repositories created with `git` and hosted on GitHub are accessible in Appendix A.

2.6.2 Virtualization (VirtualBox)

While having many physical servers can prove computationally powerful, it is also resource-demanding. A popular approach to this issue is virtualization. Virtualization is achieved with a hypervisor, a software layer enabling the deployment of multiple virtual machines, each with its own operating system, on one physical server [33]. A VM is an environment that can host an operating system with libraries and applications managed by the hypervisor. A simple type of this architecture is illustrated in Figure 2.13. In the scope of this thesis VirtualBox, a popular open source and professional virtualization software solution, was chosen [37].

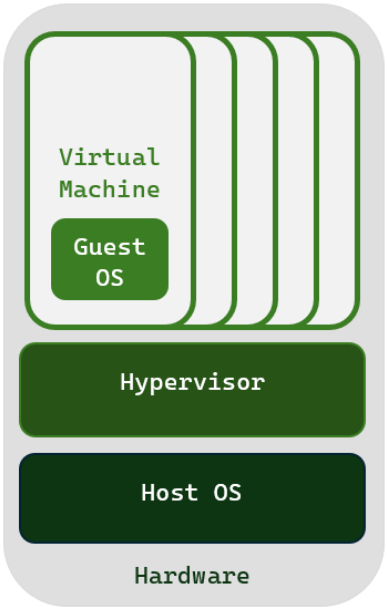

Figure 2.13: A type of virtualization.

2.6.3 File Sharing (Samba)

Samba is the standard interoperability suite for Linux and Unix systems, facilitating seamless integration with Windows environments [8] [71]. It facilitates communication between different operating systems by establishing a uniform network protocol. Due to its availability and compatibility with many popular open source

software systems, Samba is used by numerous corporations and organizations of all kinds, as well as private users [71]. In the scope of this thesis, Samba is used for sharing directories between different servers that coexist in the same network by utilizing standard Transmission Control Protocol/Internet Protocol (TCP/IP) networking.

Chapter 3

Methods

This chapter discusses a comprehensive methodology of the pursued DevOps practices. It includes a thorough explanation of global evaluation criteria and the work built around them, aiming to build a pipeline on physical hardware and virtual machines with containerized features and ML applications.

3.1 High Level Requirements

To integrate ML with DevOps, one must regard numerous available options and form general selection criteria. To approach this matter, four high level requirements should be considered.

Automated and Reproducible Work

All processes should involve as less manual interaction as possible, while being reproducible from the ground up. This involves detailed documentation of all practices, including errors and solutions, to increase accuracy and efficiency. Three primary methods utilized for automation during this thesis are docker containers, shell scripts and configuration files.

Open Source Software

Software that is compatible with this thesis should be selected to offer accessibility and community support. Therefore, only open source material is considered, preferably under MIT [34] or Apache [2] licences.

Frameworks for Distributed Training and Load Balancing

Even though it is beneficial to consider single node training as an initial task, the main goal of this thesis should incorporate distributed training of ML models, since many models are complex or require large datasets. Consequently, it is also deemed necessary to consider software for load balancing computational resources, optimising performance and scalability.

Version Controlling

To enable progress tracking, usage of Git as a version controlling system is deemed necessary. As stated in Section 2.6.1, this assists with collaboration and tracks the progress of all ML related tasks. The repository containing all material relevant to this thesis is located in Appendix A.

3.2 Simple CI Pipeline on GitLab using CML

To gain familiarity with the concept of CI/CD in the context of DevOps , a simple pipeline can be built on GitLab using an introductory example of CML [24] which is an open source ML tool presented in Section 2.1. This model is a random forest classifier using the `scikit-learn` Python library [40], for which the training and testing features and labels are provided in the same repository. Since GitLab has all of the required infrastructure only a single CI configuration `.gitlab-ci.yml` file is needed. The contents of `.gitlab-ci.yml` can be found in Appendix B.

The `train-and-report` job involves only one step, for which the Docker `image` and the `script` are provided by iterative.ai [14]. Inside the CML container all the required Python packages are installed via `pip` before the ML script `train.py` is run. With the completion of a pull or merge request in the repository the pipeline is triggered automatically, presenting logs along the process. After successful completion two artifacts are generated and saved for future reference: one simple text file containing the model accuracy and a plot representing the confusion matrix.

3.3 Kubernetes pipeline

With an initial comprehension of a practical CI/CD implementation a reasonable next objective is setting up a Kubernetes cluster to support the DevOps pipeline. This alleviates the constraint of the pipeline being solely on GitLab.

3.3.1 Infrastructure Overview: Servers and Network

Although Kubernetes is a very useful tool, it requires experience with configuration and setup. A primary infrastructure overview is therefore crucial in understanding a Kubernetes cluster. Setting a network layout can play a major role in resource allocation, load balancing as well as debugging. Figure 3.1 illustrates a network diagram of the cluster, which was set up during this thesis. The central router is the primary networking device and is connected to an external network.

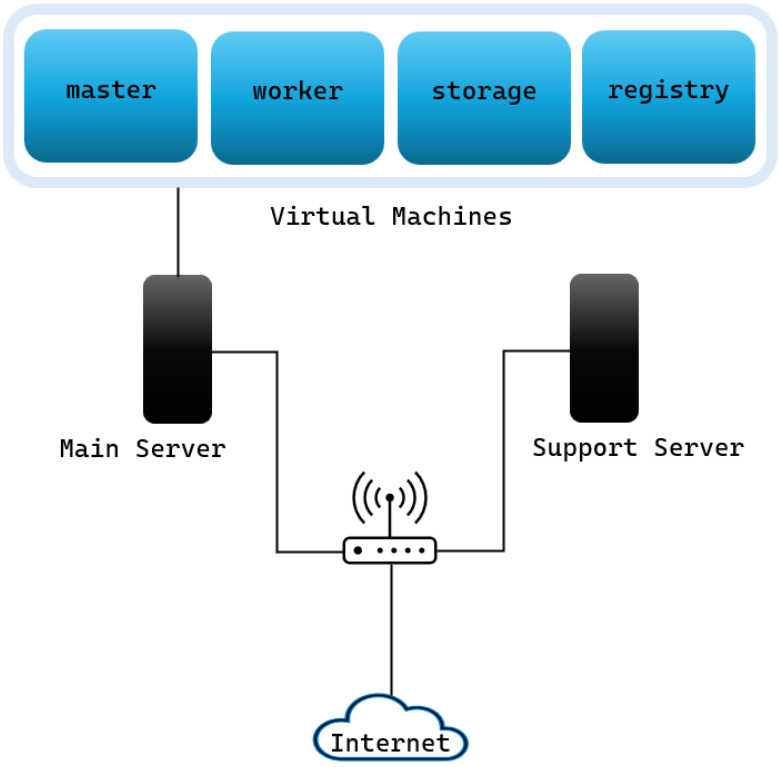

Figure 3.1: Network topology diagram.

Physical Servers

Two available servers are connected to the router via Ethernet cables, named main and support server in Figure 3.1. Each server includes one Graphics Processing Unit (GPU), which can be utilized for faster performance in ML projects. The main server is used for hosting virtual machines and the support server acts as a Kubernetes worker node. The chosen OS was Ubuntu Desktop 22.04 LTS on both servers.

Virtual Machines

The main server hosts four virtual machines, as shown in Figure 3.1. As described in Section 2.6.2, they are all deployed on the main server hardware, but each with its own OS. The VMs are managed by VirtualBox 7.0.14 and for each VM the guest OS was chosen to be Ubuntu Server 22.04 LTS. Each VM plays a separate role, presented in Table 3.1 which also shows the chosen specifications. A more detailed description of their usage is provided later in this chapter. Lastly, the selected network attachment is "Bridged Adapter" which makes every VM visible in the same network as the physical servers.

Name	Role	RAM (MB)	Disk Space (GB)	CPUs
master	Kubernetes control plane	8192	40.00	4
worker	Kubernetes worker node	8192	40.00	2
storage	SAMBA storage	8192	50.00	2
registry	container registry	11264	100.00	4

Table 3.1: VM specifications.

3.3.2 Requirements for Setting up a Cluster with `kubeadm`

Static IP Addresses

Resolving static IP addresses for each server is not a necessary prerequisite. In fact, during the longest part of this thesis work, IP addresses were dynamic. However, to ensure stability, automation and reproducibility, configuring static IP is a key practice. Each server on the Local Area Network (LAN) in Figure 3.1 is manually assigned a specific IP address that remains constant. The configuration involved Dynamic Host Configuration Protocol (DHCP) to automatically obtain the Media Access Control (MAC) address and then set a static IP address for each server. As mentioned above, since every VM network attachment is "Bridged Adapter", every IP address is received from the same DHCP as the physical servers. Thus, all the physical and the virtual servers are available on the same LAN.

Resource Units

Some errors, when initializing a Kubernetes cluster especially in VMs, can occur due to resource shortages. For example, a common erratic practice is to set the number of Central Processing Units (CPUs) to 1 instead of 2, which produces the error below. This can be easily avoided by setting up VMs that satisfy the lower resource limits set by the official documentation [67].

```
[ERROR NumCPU]:the number of available CPUs 1 is less than the required 2
```

CRI Configuration

Before starting the control plane, a CRI needs to be selected, downloaded, installed and configured. After experimenting with CRI-dockerd and Containerd, the latter was chosen as the main CRI in the pipeline, as it seemed to be preferred in the official Kubernetes documentation [64]. During this process, two bugs were encountered. In the beginning, the control plane failed to initialize and instead returned a "`Kubelet is not running`" error. In an attempt to resolve this, system logs were studied and it was eventually concluded that the swap memory was not disabled, which is an important requirement of Kubelet.

After disabling swap the control plane initialized successfully for a few minutes before throwing a "`connection refused`" error when using any `kubectl` command. This time, the actual error could not be located through the system logs and after spending a considerable amount of time on troubleshooting, the problem was solved by modifying the Containerd configuration file.

Pod Network Add-on

In order for the control plane to be able to communicate with pods on other nodes a pod network add-on is required [58]. This routes the traffic between pods and allows them to send and receive traffic. Initially, Calico was tried, as it is one of the most popular pod network add-ons. However it was abandoned due to unnecessary functionality in favor of a much simpler pod network add-on called Flannel [28], which could be easily applied to the cluster with a single command in the control plane. Specifically, one can successfully deploy Flannel by adding the flag `--pod-network-cidr` with the value `10.244.0.0/16` to the `kubeadm init` command and, after setting up the Kubernetes admin configuration file, running the following command according to the official Flannel documentation [16]:

```
kubectl apply -f \
https://github.com/flannel-io/flannel/releases/latest/download/kube-flannel.yml
```

Static Bootstrap Token

A growing concern during the time span of this thesis was: "What happens to the cluster if a server fails?". The answer is that the whole cluster has to be reset again after the servers have been restored, which is a time-consuming process. One step in automating both the `kubeadm init` and `kubeadm join` processes was to specify a static bootstrap token, as a part of the authentication between the control plane and the workers. As described in Section 2.5.1, the token is typically a random string and a part of the `kubeadm join` command, which is generated dur-

ing `kubeadm init`. However, by adding the flags `--token` followed by the desired value and `--token-ttl` followed by 0 to the `kubeadm init` command, the token is set to the chosen value without expiration. On the worker node the `kubeadm join` command must also include the `--token` flag followed by the specified token value along with the `--discovery-token-unsafe-skip-ca-verification` flag. The latter allows a server to join the control plane without the need of the `--discovery-token-ca-cert-hash` flag. The full `kubeadm init` and `kubeadm join` commands are presented below in Section 3.3.3.

3.3.3 Setting up a Cluster with `kubeadm`

To create an initial cluster all nodes had to be virtual machines, as described in Figure 3.1 and Table 3.1, due to lack of resources.

Control Plane

The VM that hosts the control plane is named ”master”. To facilitate the first initialization of the control plane a custom shell script was created, which automatically downloads all required dependencies for Kubernetes, configures the Containerd CRI, initializes the control plane and applies the Flannel pod network add-on according to the requirements in Section 3.3.2. The full `kubeadm init` command is:

```
sudo kubeadm init \
--pod-network-cidr 10.244.0.0/16 \
--token [a-z0-9]{6}.[a-z0-9]{16} \
--token-ttl 0
```

Worker Node

A second VM named ”worker” was created as a Kubernetes worker node. The worker node was easily set up by using a modification of the control plane setup script that only installs the required dependencies and joins the master node without the need of adding any pod network add-on. The complete `kubeadm join` command is:

```
sudo kubeadm join master:6443  \
--token [a-z0-9]{6}.[a-z0-9]{16} \
--discovery-token-unsafe-skip-ca-verification
```

At this point a minimal K8s cluster with one master node (control plane) and one worker node had rapidly been created and was successfully running. Afterwards,

work on actual pipeline components could start.

Automating Cluster Restart

An important component of this thesis has been the Bash shell due to its ability to reduce manual interaction with the terminal. What is more, if a VM already has all dependencies installed and is ready to be a Kubernetes node then a good practice after reboot is to run a modified script. Taking into account the requirements in Section 3.3.2, two shell scripts were created which revert any previous changes made by `kubeadm init` or `kubeadm join` and then proceed to reset the cluster. These scripts were configured to run at user login by adding their complete path as the last line in the file `.bashrc`.

3.3.4 Crafting the First Pipeline Component

The next step was to create ML training and testing micro-services that would be solely responsible for training an ML model and testing it on a specific dataset utilizing the Kubernetes cluster. However, before this could be achieved, a suitable model with corresponding training and testing scripts had to be found, successfully run on local machines, containerized with Docker and finally integrated with the K8s cluster.

After some research a GitHub repository containing YOLOv5 was cloned and modified to run locally [52]. With this repository used as a base, a custom dataset of forklifts and people served as training data and then the resulting trained model was used to perform testing (detection) on a single `mp4` video. Due to limited resources at the time the number of epochs and batch size had to be significantly decreased. Although this simplified the debugging process significantly, it yielded results of lower quality. However, since the prediction quality was considered out of the project scope, no further effort was put into improving it.

To containerize the training and testing applications and turn them into micro-services two Dockerfiles were made using the official Python:3 image from Docker Hub as base image. For each of the applications a shell script, only for use outside of K8s, was made that builds and runs the respective Docker image with the right commands. During the `build` stage the Ultralytics' YOLOv5 GitHub repository is cloned and all the required Python packages are installed [18]. A local directory is then mounted into the container, where the retrained weights will be saved after training completion.

For both training and detection the `build` stage involves copying the configuration

file `data.yaml` into the container. This file specifies the location of the training and testing data as well as the object labels "forklift" and "person". When the training application runs the YOLOv5 is trained on the custom dataset. Initially, when attempting to `run` the docker container, using the default `run` command outputs an error stating that the container could not run successfully due to limited shared memory. This error can be solved by modifying the `run` command that starts the container. This is done by adding the flag `--shm-size=10g` which sets the shared memory space inside the container to 10 gigabytes thus successfully retraining the YOLOv5 model as a micro-service.

After training the resulting weights file `best.pt` is saved, copied into the mounted directory and later used to detect forklifts and people during testing. The testing is performed on each frame of a single video file, the output of which is eventually stored in the mounted directory, as well. The Dockerfile and the shell script for testing are quite similar to the respective files for training with only very few changes. This proves time efficiency and cohesiveness across this ML project.

Containerizing these ML applications successfully was an important step towards automation. Before integration into the Kubernetes cluster however a data storage directory needed to be created along with a container registry from which the cluster pulls the custom training image for pod creation.

3.3.5 Creating External Storage

To create an external file storage for the K8s cluster a VM named "storage" was created according to the description in Section 3.3.1. After installing a Samba server and creating a small configuration file specifying access privileges the VM was able to share a directory locally on the network. This shared directory had to be mounted on the worker node which posed an issue since the IP address of the storage VM was dynamically allocated during the beginning of this thesis. To solve this issue a custom mounting script was made which used `nbtscan` to resolve the IP address from the Network Basic Input/Output System (NetBIOS) name of the storage VM. With the shared directory mounted on the worker the external file storage could be abstracted into the K8s cluster. This was done by applying a `PersistentVolume` type configuration file to the control plane that made 10 gigabytes of storage from the shared Samba directory accessible by the K8s cluster.

3.3.6 Configuring Container Registry

The registry VM hosts the private container registry locally. The configuration installs `docker-registry` from docker.io on the VM and authenticates with specified username and password. What is more, the registry should be set as insecure in the docker daemon on each relevant node, including the registry, and the registry port enabled on the firewall.

A major obstacle that slowed down the progress of this thesis was dynamic IP addresses. It became a problem since the authors did not have full network privileges. Both the docker daemon configuration and the Containerd configuration need to include the IP address of the registry which in turn introduces severe inconveniences when the IP changes. Due to this a decision was made to set up a separate router for this project and configure static IP addresses for all the VMs, as described in Section 3.3.1. Afterwards, building and pushing the YOLOv5 training and testing images to the local Docker registry is possible from any node connected to the local network and logged in to the registry.

For the K8s cluster to access the image through the Containerd CRI a cluster secret containing the local registry credentials has to be added to the cluster. The secret is added by applying a `yaml` configuration file on the control plane. From the secret both the username and password can be extracted as a base 64 string and added to the Containerd configuration file.

3.3.7 Running the First Pipeline Components in a Kubernetes Cluster

With the cluster being able to access an external storage and external container registry the YOLOv5 training and testing micro-services can be integrated into the cluster. This can be achieved by firstly creating a `PersitentVolumeClaim`, which defines a portion (3 gigabytes) of the `PersistentVolume` described in Section 3.3.5, and a `Job`. These are separate configuration files that are applied to the cluster through the control plane.

In the training `Job` file it is necessary to specify a volume mount to a shared memory directory inside the container. This ensures that the container is running with more shared memory and thus without getting killed. The detection `Job` is almost identical to the training `Job`. The main difference is calling another function from the same YOLOv5 repository during execution. This function uses the `mp4` video located in the `PersistentVolume` and accessed with the same `PersistentVolumeClaim` as the training `Job`. When this `Job` is run it will attempt

to detect forklifts and output the result in the pod logs.

3.3.8 Kubernetes Dashboard

To facilitate monitoring the cluster the official Kubernetes dashboard can be installed by applying configuration files provided by the Kubernetes team. This creates a new cluster namespace with two pods running in it. At this point an error can result in the pods never becoming ready. Troubleshooting for this error involves checking the pod logs with `kubectl`. To resolve this issue try removing a container network interface of type "bridge" on the hosting node. The K8s service named `dashboard-service` has to be edited to be of type `NodePort` rather than `Cluster IP`. This enables access to the dashboard from a browser on the local network by typing in the IP address of the master node. To be able to login via the dashboard a service account, cluster role binding and a secret have to be created by applying three configuration files on the cluster. After this a token is generated with a single command which is used to login on the dashboard. [50]

3.3.9 Distributed Training in a Kubernetes Cluster

To introduce further functionality to the K8s cluster a second worker node can be created to enable DDP training. To be effective both the physical server, where the training is initiated (client), and the cluster need to be configured accordingly. By following the method below the client does not need to be a node in the K8s cluster rather it could be any physical server.

On the cluster side Volcano [77] is installed by applying configuration files provided by the Volcano team. On the client side a regular Python virtual environment is set up with a small project containing some example code. To be able to use TorchX to schedule jobs on Kubernetes the Kubernetes Python client package and TorchX must be installed with `pip` beforehand. In order for TorchX to find the K8s cluster and interact with it, it is necessary to copy the kube configuration file from the master node, generated during `kubeadm init`, and place it in a new directory ∼/.kube/ on the client. This is the default directory, where TorchX expects to find the configuration file.

The TorchX launcher is a tool that can be used to schedule DDP training on the cluster. It uses a default docker image and pulls it in the current client workspace. The image is updated with the existing components of the workspace, such as `pip` packages and Python code, and then pushed to a specified registry. Before running TorchX, a configuration file has to be made that specifies Volcano queue and image repository to upload the modified image to. Then it is built and pushed to

the specified container registry, in this case Docker Hub.

After logging in to the registry with the option ”docker login”, example scripts provided by TorchX [42] and PyTorch Lightning [43] can be executed on the K8s cluster using a single TorchX run command with Kubernetes specified as the scheduler argument and the Python script as a regular argument. Since the job is launched by TorchX and scheduled on Kubernetes via Volcano the pods in the cluster pull the edited image, containing the workspace, and execute the distributed training. PyTorch Lightning manages the way training is distributed among processes.

Chapter 4

Results

This chapter articulates the result obtained by following everything that was discussed in Methods, Chapter 3. This mainly includes the artifacts obtained by the CML pipeline and the architecture of the final pipeline using Kubernetes.

4.1 Simple CI Pipeline on GitLab using CML

In this section, the results of the simple CML model are presented. As described in Section 3.2, upon triggering, the Continuous Integration (CI) pipeline generates job logs alongside two artifacts. Figure 4.1 represents a successful output of the pipeline logs on GitLab, completed in 36 seconds.

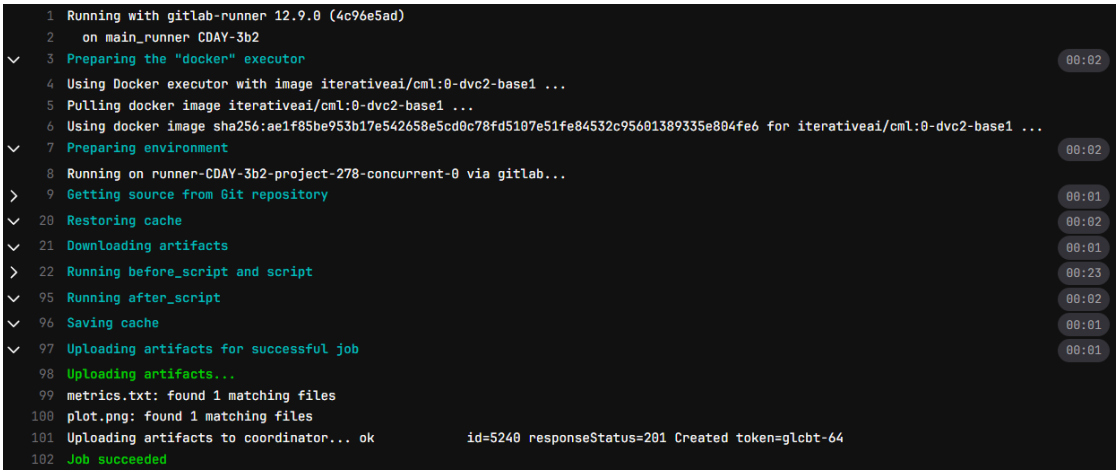

Figure 4.1: CI pipeline logs on GitLab using the simple CML model.

The first artifact is a `.txt` file comprising of the model accuracy, the contents of which are shown below, and the second artifact is the confusion matrix of

the model, depicted in Figure 4.2. As expected, the short CI pipeline is rapidly executed and the artifacts prove the model's high accuracy.

```
Accuracy: 0.864
```

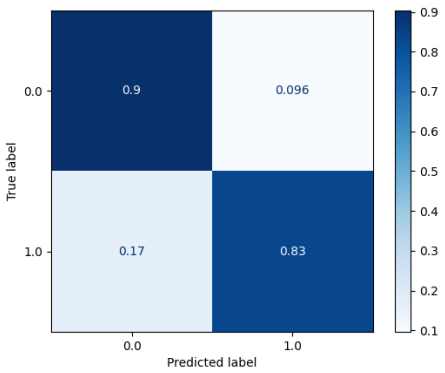

Figure 4.2: Confusion matrix depicting the performance of the simple CML model, saved as a CI artifact.

4.2 The Finalized Pipeline

The pipeline is supported by a K8s cluster with a storage mounted to the cluster and accessing a container registry. Each function in the pipeline is seen as a microservice and realized in the cluster as an application running inside a container.

The final k8s cluster consists of a master node (control panel) and two worker nodes that all run on separate virtual machines hosted by VirtualBox. All the pods that comprise the pipeline are exclusively run on the worker nodes. Each pod uses only cluster memory, without storing any data locally, unless specified to store an artifact in the mounted storage. Specifically, the cluster memory is a shared folder on another virtual machine. This external memory is mounted into the cluster and internally accessed through a Persistent Volume configuration. Each pod is then accessing parts of the mounted storage through Persistent Volume Claims.

The Docker images running in the pods are stored in a container registry. Unlike the storage, the container registry is not mounted to the cluster. Instead, docker images are directly pushed and pulled from it as long as the pods have access to its IP address. Pods authenticate to the container registry with a K8s

`Secret`, applied through a configuration file. A problem with the local registry is that it is crashing so to ensure a seamless process Docker Hub is used instead.

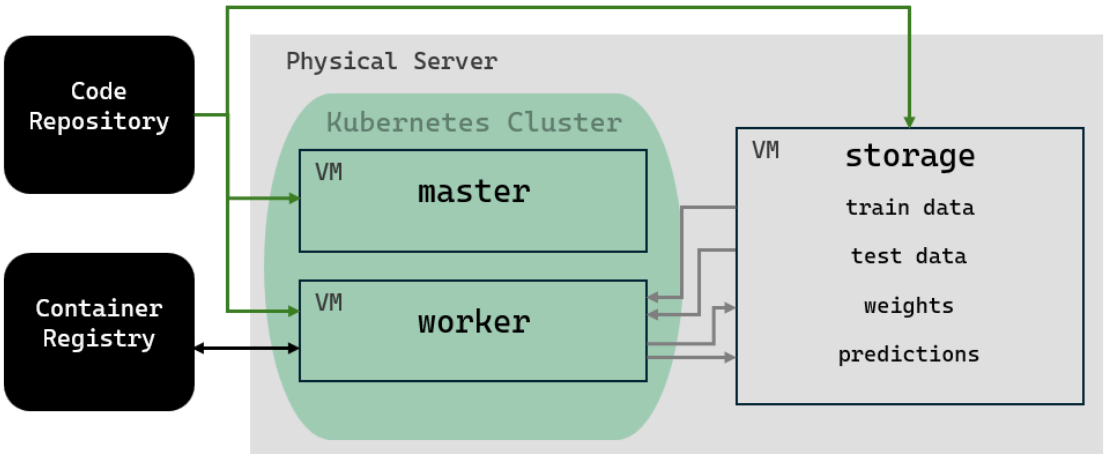

Figure 4.3: Parts of the pipeline used for object detection.

4.2.1 Object Detection Training and Testing

There are two microservices running in the cluster, one that trains the YOLOv5 model on a custom dataset and one microservice that performs detection on a video using the result from the training microservice, as described in Section 3.3.4. Both are based on the same code repository which is downloaded into the Docker container at startup. All training and testing scripts are included in the repository. Once the model has been trained on a custom dataset the resulting weights are stored in the shared folder and later accessed by the detection container. The full structure of the active parts of the K8s cluster is depicted in Figure 4.3.

4.2.2 Distributed Data Parallel Training

As described in Section 3.3.9, distributed training is achieved using PyTorch Lightning and then launched with TorchX. By storing the kube configuration file locally and downloading the K8s Python package, TorchX can automatically schedule the training on the K8s cluster after Volcano has been installed. TorchX handles the scheduling while K8s ensures that processes are distributed across the resources in the cluster. This combination has proven very time and resource efficient without demanding the complex implementation characteristic to DDP training. The structure of the pipeline used for DDP training is illustrated in Figure 4.4. Figures 4.5 and 4.6 show the difference in training time when training on one compared to two nodes.

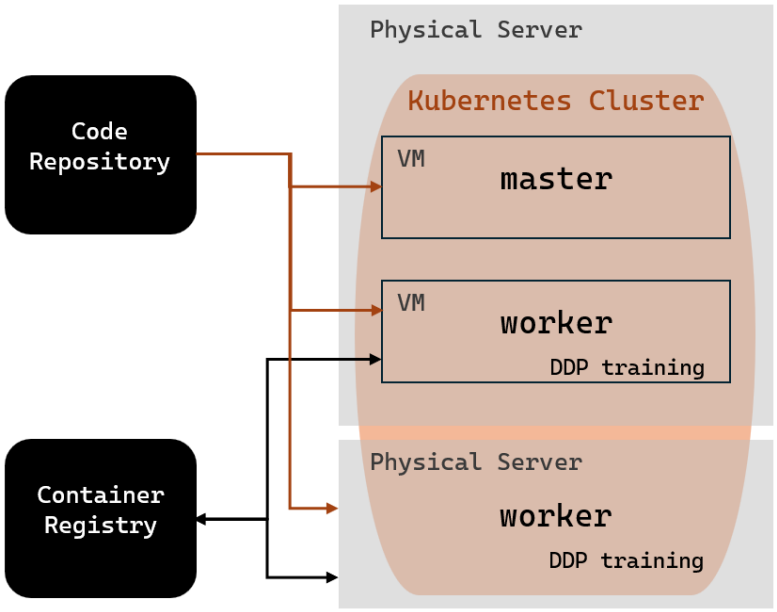

Figure 4.4: Parts of the pipeline used for DDP training.

```
State:          Terminated
  Reason:       Completed
  Exit Code:    0
  Started:      Thu, 23 May 2024 09:28:41 +0000
  Finished:     Thu, 23 May 2024 10:22:16 +0000
Ready:          False
Restart Count:  0
Limits:
  cpu:     2
  memory:  1024M
```

Figure 4.5: It almost took an hour to train on the MNIST data set with one worker.

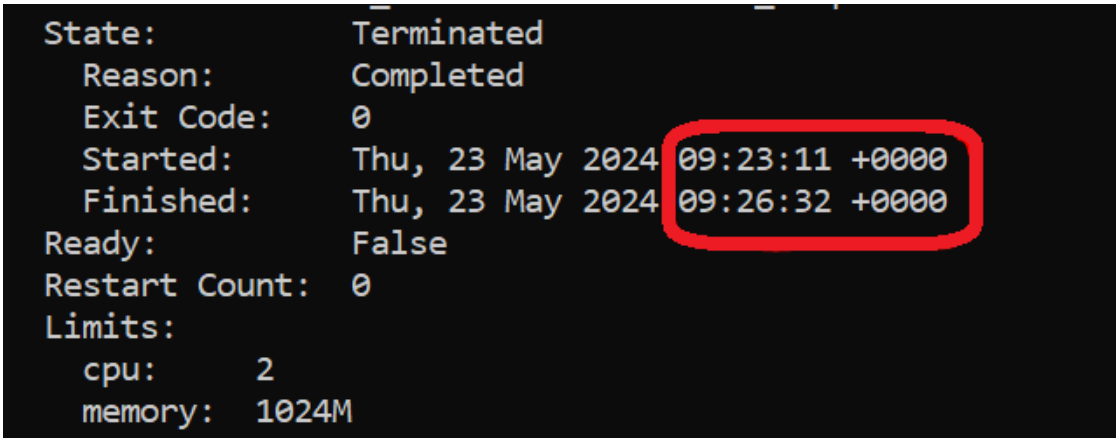

Figure 4.6: Training on two workers significantly reduced the training time to approximately 3 minutes.

Chapter 5

Discussion

This chapter provides a critical overview of this thesis work, including inhibitory factors to its progress, as well as ideas for possible improvements. It compares the perceived differences between DevOps and MLOps and discusses how certain tactics affected the overall outcome of this thesis. Moreover, this chapter proposes software alternatives and solutions to issues regarding the advancement of the final pipeline.

Initially, the focus of this thesis was on creating CI/CD pipeline components for MLOps. However, the interest shifted to building the pipeline infrastructure using DevOps. As the authors of this thesis had no previous experience with neither DevOps, MLOps nor container orchestration, creating and configuring a functioning on-premises Kubernetes cluster from scratch proved to be quite a challenge. In addition, the time constraints and lack of Kubernetes expert advice certainly limited the outcome. Despite this, eventually a fully functional K8s cluster with some sample features was created with relatively few and inexpensive resources. The authors believe that this project can serve as an introduction to basic container orchestration with Kubernetes. It also has a lot of potential for improvement and further work could include adding more nodes and replacing the virtual machines with physical machines to achieve the full intended potential, as well as adding more features to the cluster. Using a better storage such as a cluster instead of a local directory is also recommended. With a real on-premises K8s cluster the owner has full control and can use it with sensitive data that is not suitable for third party providers.

Having argued that Kubernetes is so difficult to set up, the reader might wonder: *Why should one not just use a tool with much simpler built-in CI/CD, such as GitLab?* While this is certainly sufficient for many use cases, it has limited functionality when working with a lot of data and traffic. What is more, creating an

internally licensed project, that involves high-level customisation with open source software, is indeed demanding. The authors realized early on that with Kubernetes, there are almost no limits to scalability. This is especially true when Kubernetes is built on bare metal. Many DevOps developers and engineers argue that having a Kubernetes cluster on physical machines can prove to be a powerful tool and that frameworks offering automated cluster configuration lack customisation. Kubernetes can also be used beyond building CI/CD pipelines since container orchestration is inherently a broad practice. On the other hand, they also warn that usually experience is a prerequisite in order to achieve potent customisation. Most enterprises prefer cloud providers because they offer accessibility, easier configuration and abundant resources. The most popular cloud providers for Kubernetes are Amazon Web Services (AWS) and Googles Kubernetes Engine (GKE), which require paid subscription. Thus, they are deemed unsuitable for this thesis.

Another interesting point of discussion is the choice of Kubernetes in the context of ML pipelines instead of new and established open source MLOps tools such as Kubeflow, MLflow or Airflow. These tools were investigated to determine their value for this thesis work. Although they abstract away the complexity of managing ML workflows with a plethora of pre-integrated ML components and pipeline management, they lack in flexibility and maturity. They are optimized with sole focus on developing and engineering ML, which in turn restricts users to code notebooks. And, although they are efficient for their purpose, they are not deemed sufficient thesis material. These are the main reasons why the simple CML pipeline has not been further developed and adapted. On the other hand, Kubernetes is a well-documented and long-established software tool adopted by several industries worldwide, which also allows precise control over any infrastructure. Therefore, knowledge and experience with Kubernetes is considered a worthwhile investment, despite the difficulties.

To make this result resemble an end-to-end MLOps pipeline more steps such as data collection, data cleaning, feature engineering and model deployment should be added. Implementing web API was considered and would have been used to interact with a deployed model through a web browser. The idea was to turn the containerized micro-services into applications that could be deployed with a web API. As a candidate, FastAPI [46] was mainly considered among other software solutions. This part was not implemented due to lack of time. Instead, the focus was shifted to testing and refining already existing work.

As already mentioned above, an important improvement is considering a more current and suitable storage option for the cluster. Using a local folder as exter-

nal storage is not considered appropriate for a production cluster. Instead, the authors recommend exploring CEPH [78]. CEPH is an open source distributed storage system with scaling capability. It requires more hardware to maintain the storage cluster, as well as more effort to set up and maintain, but provides reliable storage as a result. If one does not need to store any sensitive data and wants to avoid setting up and maintain their own storage cluster, the authors of this thesis recommend using cloud storage providers. Although these services can sometimes be expensive and are provided by a third party, they are very convenient because the user never has to worry about administration or maintenance.

In this project, the Flannel pod network add-on was chosen, because it is relatively small and simple. However in more advanced clusters that require network policies, other add-ons should be used, for instance Calico.

SMILE IV Project Contribution

In collaboration with Infotiv Technology Development, this thesis project is currently integrated into the SMILE IV project. With the authors' assistance, Infotiv Technology Development launched a simulation on the Kubernetes cluster as a `Deployment` and created pods that can collect data from any specific camera mounted in a virtual warehouse with forklifts, boxes and people. Appendix A provides access to the code repository for the SMILE-IV project.

For the SMILE IV project, a Kubernetes cluster can be a valuable solution. In particular, the simulation is CPU intensive because it is based on a model that processes many frames per second for multiple cameras. In the next phases of the project, the goal is to collect information from processed frames, perceive the environment, and assist agents in navigating to specified locations. Therefore, this process can be distributed across many physical machines hosting a Kubernetes cluster.

Chapter 6

Conclusion

We began this thesis with the hope of exploring and researching different implementations of MLOps. Nevertheless, our interest quickly shifted towards the more established DevOps, in particular Docker and Kubernetes, and utilizing them to build a pipeline with ML code. Thus, we discovered the benefits and difficulties that follow a more sophisticated setup, without losing the ML element.

Setting up a K8s cluster from scratch requires time and effort, especially for young developers and engineers. It is not analogous to downloading and installing a single software application for immediate use. A K8s cluster should ideally have one or several administrators who are responsible for setting up the cluster, maintaining it and adding resources to it such as nodes and persistent volumes. Then the developers who create the applications are responsible for integrating their applications into the cluster by writing configuration files. As an alternative to setting up an on premise cluster, there are cloud providers that offer ready to use solutions and handle all the administration. These can however be expensive and do not come with full control over the cluster.

To conclude, DevOps are immensely powerful and popular tools, which are not limited to ML, although they support it impeccably. On the other hand, MLOps should be used for smaller ML projects and should most definitely be combined with established DevOps. In this sense, both DevOps and MLOps make the future of ML look structured and cost-effective.

Bibliography

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

[2] Apache License, Version 2.0. `https://www.apache.org/licenses/LICENSE-2.0.txt`. Accessed: Tuesday 11th June, 2024.

[3] armosec. What is etcd? `https://www.armosec.io/glossary/etcd-kubernetes/`. Accessed: Tuesday 11th June, 2024.

[4] armosec. What is Kubelet? `https://www.armosec.io/glossary/kubelet/`. Accessed: Tuesday 11th June, 2024.

[5] Ashish Patel. Kubernetes — Storage Overview — PV, PVC and Storage Class. `https://medium.com/devops-mojo/kubernetes-storage-options-overview-persistent-volumes-pv-claims-pvc-and-storageo` Accessed: Tuesday 11th June, 2024.

[6] atlassian. What Is DevOps? `https://www.atlassian.com/devops`.

[7] avinetworks. Service Discovery Definition. `https://avinetworks.com/glossary/service-discovery/`.

[8] Carter, G., Ts, J., and Eckstein, R. *Using Samba, 3rd Edition*. O'Reilly Media, Inc., 2007.

[9] Chacon, S., and Straub, B. *Pro git*. Apress, 2014.

[10] Conscia. DevOps. `https://conscia.com/se/losningar/digital-transformation/devops/`.

[11] Containerd Authors. Containerd. `https://github.com/containerd/containerd/tree/main`. Accessed: Tuesday 11th June, 2024.

[12] Docker, Inc. *Docker Hub*. `https://hub.docker.com/`.

[13] Docker, Inc. *Docker*, March 20 2013. https://www.docker.com/.

[14] DVC.ai. CML - Continuous Machine Learning. https://cml.dev/. Accessed: Tuesday 11th June, 2024.

[15] Falcon, William and The PyTorch Lightning team. PyTorch Lightning. https://www.pytorchlightning.ai, Repository Code: https://github.com/Lightning-AI/lightning.

[16] flannel-io. Flannel. https://github.com/flannel-io/flannel. Accessed: Tuesday 11th June, 2024.

[17] Gift, N., and Deza, A. *Practical MLOps*. O'Reilly Media, Inc., 2021.

[18] Glenn Jocher. Ultralytics YOLOv5, 2020. https://github.com/ultralytics/yolov5.

[19] Google. What is container orchestration? https://cloud.google.com/discover/what-is-container-orchestration.

[20] Google Cloud. MLOps: Continuous Delivery and Automation Pipelines in Machine Learning, 2023. https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning.

[21] Harinderjit Singh. Inspecting and Understanding Kubernetes (k8s) Service Network. https://itnext.io/inspecting-and-understanding-service-network-dfd8c16ff2c5. Accessed: Tuesday 11th June, 2024.

[22] Holder, I. Kubernetes 1.28: Beta support for using swap on linux. https://kubernetes.io/blog/2023/08/24/swap-linux-beta/#:~:text=Since%20enabling%20swap%20permits%20greater,account%20for%20swap%20memory%20usage, August 2023.

[23] Hugging Face. https://huggingface.co/. Accessed: Tuesday 11th June, 2024.

[24] Iterative. example_cml. https://github.com/iterative/example_cml. Commit: 56de3ef.

[25] Kaggle. https://www.kaggle.com/. Accessed: Tuesday 11th June, 2024.

[26] Karamitsos, I., Albarhami, S., and Apostolopoulos, C. Applying devops practices of continuous automation for machine learning. *Information 11*, 7 (2020).

[27] KodeKloud. Kube-Proxy: What Is It and How It Works. https://kodekloud.com/blog/kube-proxy/.

[28] Koukis, G., Skaperas, S., Kapetanidou, I. A., Mamatas, L., and Tsaoussidis, V. Performance evaluation of kubernetes networking approaches across constraint edge environments, 2024.

[29] Kreuzberger, D., Kühl, N., and Hirschl, S. Machine learning operations (mlops): Overview, definition, and architecture. *IEEE Access 11* (2023), 31866–31879.

[30] Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., and Chintala, S. Pytorch distributed: Experiences on accelerating data parallel training, 2020.

[31] Linux.com Editorial Staff, and Gary Sims. All about linux swap space. *Linux.com* (September 2007). Accessed: Tuesday 11th June, 2024.

[32] Merkel, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal 2014* (03 2014).

[33] Miona Aleksic. Containerization vs. Virtualization : understand the differences. `https://ubuntu.com/blog/containerization-vs-virtualization`. Accessed: Tuesday 11th June, 2024.

[34] The MIT License (MIT). `https://spdx.org/licenses/MIT.html`. Accessed: Tuesday 11th June, 2024.

[35] Neal Analytics. MLOps. `https://nealanalytics.com/expertise/mlops/`, Accessed: Tuesday 11th June, 2024.

[36] Nicolas Ehrman. Container Runtimes Explained, 2024. `https://www.wiz.io/academy/container-runtimes`.

[37] Oracle Corporation. *VirtualBox*, January 17 2007. `https://www.virtualbox.org/`.

[38] Overflow, S. Developer Sentiment around AI/ML. `https://stackoverflow.co/labs/developer-sentiment-ai-ml/`. Accessed: Tuesday 11th June, 2024.

[39] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. *CoRR abs/1912.01703* (2019).

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

[41] Pierre Leveau. How to Perform Distributed Training? `https://kili-technology.com/data-labeling/machine-learning/how-to-perform-distributed-training`.

[42] PyTorch. TorchX. `https://github.com/pytorch/torchx`. Accessed: Tuesday 11th June, 2024.

[43] PyTorch. TorchX. https://github.com/pytorch/torchx. Accessed: Tuesday 11th June, 2024.

[44] Quobyte. What is Kubernetes and How It Works. https://www.quobyte.com/storage-explained/what-is-kubernetes/. Accessed: Tuesday 11th June, 2024.

[45] Raft Team. The Raft Consensus Algorithml. https://raft.github.io/.

[46] Ramírez, S. Fastapi, 2020. https://fastapi.tiangolo.com, code repository: https://github.com/tiangolo/fastapi.

[47] Recupito, G., Pecorelli, F., Catolino, G., Moreschini, S., Nucci, D. D., Palomba, F., and Tamburri, D. A. A multivocal literature review of mlops tools and features. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (2022), pp. 84–91.

[48] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. You only look once: Unified, real-time object detection, 2016.

[49] Regehr, J. Race condition vs. data race. https://blog.regehr.org/archives/490, March 13 2011. Embedded in Academia.

[50] Sahid. Kubernetes dashboard: An overview, installation, and accessing. https://k21academy.com/docker-kubernetes/kubernetes-dashboard/, December 8 2023.

[51] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems* (2015), C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28, Curran Associates, Inc.

[52] SelimSavas. forklift-and-people-detection-with-YOLOv5. https://github.com/SelimSavas/forklift-and-people-detection-with-YOLOv5. Accessed: Tuesday 11th June, 2024.

[53] Shingai Zivuku. In-Depth Analysis of Kubernetes Scheduling. https://zivukushingai.medium.com/in-depth-analysis-of-kubernetes-scheduling-abf08f949924.

[54] Stack Overflow. Stack Overflow Annual Developer Survey (2023). https://survey.stackoverflow.co/2023/. Accessed: Tuesday 11th June, 2024.

[55] Szeliski, R. *Computer Vision: Algorithms and Applications*, 2nd ed. Springer, 2022. Final draft, September 30, 2021.

[56] The Kubernetes Authors. Cloud Controller Manager. https://kubernetes.io/docs/concepts/architecture/cloud-controller/. Accessed: Tuesday 11th June, 2024.

[57] The Kubernetes Authors. Command line tool (kubectl). https://kubernetes.io/docs/reference/kubectl/. Accessed: Tuesday 11th June, 2024.

[58] The Kubernetes Authors. Creating a cluster with kubeadm. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/. Accessed: Tuesday 11th June, 2024.

[59] The Kubernetes Authors. Deploy and Access the Kubernetes Dashboard. https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/. Accessed: Tuesday 11th June, 2024.

[60] The Kubernetes Authors. Installing kubeadm. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/. Accessed: Tuesday 11th June, 2024.

[61] The Kubernetes Authors. kube-scheduler. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/. Accessed: Tuesday 11th June, 2024.

[62] The Kubernetes Authors. Kubernetes. https://kubernetes.io/. Accessed: Tuesday 11th June, 2024.

[63] The Kubernetes Authors. Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/. Accessed: Tuesday 11th June, 2024.

[64] The Kubernetes Authors. Migrating from dockershim. https://kubernetes.io/docs/tasks/administer-cluster/migrating-from-dockershim/. Accessed: Tuesday 11th June, 2024.

[65] The Kubernetes Authors. Pods. https://kubernetes.io/docs/concepts/workloads/pods/. Accessed: Tuesday 11th June, 2024.

[66] The Kubernetes Authors. Pull an Image from a Private Registry. https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/. Accessed: Tuesday 11th June, 2024.

[67] The Kubernetes Authors. Resource Management for Pods and Containers. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#resource-units-in-kubernetes. Accessed: Tuesday 11th June, 2024.

[68] The Kubernetes Authors. The Kubernetes API. https://kubernetes.io/docs/concepts/overview/kubernetes-api/. Accessed: Tuesday 11th June, 2024.

[69] The Kubernetes Authors. Viewing Pods and Nodes. https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/. Accessed: Tuesday 11th June, 2024.

[70] The Kubernetes Authors. Viewing Pods and Nodes. https://kubernetes.io/docs/concepts/storage/persistent-volumes/. Accessed: Tuesday 11th June, 2024.

[71] The Samba Team. *SAMBA*. https://www.samba.org/.

[72] Tigera. Calico. https://github.com/projectcalico/calico. Accessed: Tuesday 11th June, 2024.

[73] VICTOR HERNANDO. How to Monitor kube-controller-manager. https://sysdig.com/blog/how-to-monitor-kube-controller-manager/. Accessed: Tuesday 11th June, 2024.

[74] Vijayakumar, A., and Vairavasundaram, S. Yolo-based object detection models: A review and its applications. *Multimedia Tools and Applications 74*, 13 (Mar 2024), 4567–4590. https://doi.org/10.1007/s11042-024-18872-y.

[75] Vinnova, FFI, Fordonsstrategisk forskning och innovation. SMILE IV project. https://www.vinnova.se/p/smile-iv/, 2023. Financed by Vinnova, FFI, Fordonsstrategisk forskning och innovation under the grant number 2023-00789.

[76] Volcano Community Maintainer. Volcano: Collision between containers and batch computing. https://www.cncf.io/blog/2021/02/26/volcano-collision-between-containers-and-batch-computing/. Accessed: Tuesday 11th June, 2024.

[77] Volcano Contributors. Volcano: Kubernetes Native Batch System. https://volcano.sh/en/, 2021. Incubating project of the Cloud Native Computing Foundation (CNCF).

[78] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system, 2006. https://dl.acm.org/doi/10.5555/1298455.1298485, official website: https://ceph.io/en/, code repository: https://github.com/ceph/ceph.

Appendix A

Source Code Repositories

A.1 Thesis Code Repository

https://github.com/infotiv-research/Kubernetes-MLOps/

A.2 Code Repository for SMILE-IV

https://github.com/infotiv-research/infobot/

Appendix B

CML Pipeline

B.1 .gitlab-ci.yml

```yaml
train-and-report:
  stage: test
  image: iterativeai/cml:0-dvc2-base1
  script:
    - pip install -r requirements.txt
    - python train.py

  artifacts:
    paths:
      - metrics.txt
      - plot.png
```

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY