Master's thesis

Master's Programme in Data Science

Designing an open-source cloud-native MLOps pipeline

Sasu Mäkinen

March 12, 2021

FACULTY OF SCIENCE
UNIVERSITY OF HELSINKI

Supervisor(s)

Prof. T. Mikkonen, Prof. J. K. Nurminen

Examiner(s)

Prof. T. Mikkonen, Prof. J. K. Nurminen

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Koulutusohjelma — Utbildningsprogram — Study programme Master's Programme in Data Science

Tekijä — Författare — Author

Sasu Mäkinen

Työn nimi — Arbetets titel — Title

Designing an open-source cloud-native MLOps pipeline

Ohjaajat — Handledare — Supervisors

Prof. T. Mikkonen, Prof. J. K. Nurminen

Työn laji — Arbetets art — Level Master's thesis	Aika — Datum — Month and year March 12, 2021	Sivumäärä — Sidoantal — Number of pages 66 pages, 4 appendix pages

Tiivistelmä — Referat — Abstract

Deploying machine learning models is found to be a massive issue in the field. DevOps and Continuous Integration and Continuous Delivery (CI/CD) has proven to streamline and accelerate deployments in the field of software development. Creating CI/CD pipelines in software that includes elements of Machine Learning (MLOps) has unique problems, and trail-blazers in the field solve them with the use of proprietary tooling, often offered by cloud providers.

In this thesis, we describe the elements of MLOps. We study what the requirements to automate the CI/CD of Machine Learning systems in the MLOps methodology. We study if it is feasible to create a state-of-the-art MLOps pipeline with existing open-source and cloud-native tooling in a cloud provider agnostic way.

We designed an extendable and cloud-native pipeline covering most of the CI/CD needs of Machine Learning system. We motivated why Machine Learning systems should be included in the DevOps methodology. We studied what unique challenges machine learning brings to CI/CD pipelines, production environments and monitoring. We analyzed the pipeline's design, architecture, and implementation details and its applicability and value to Machine Learning projects.

We evaluate our solution as a promising MLOps pipeline, that manages to solve many issues of automating a reproducible Machine Learning project and its delivery to production. We designed it as a fully open-source solution that is relatively cloud provider agnostic. Configuring the pipeline to fit the client needs uses easy-to-use declarative configuration languages (YAML, JSON) that require minimal learning overhead.

ACM Computing Classification System (CCS)

Software and its engineering → Software notations and tools → Development frameworks and environments → Application specific development environments

Software and its engineering → Software notations and tools → Software configuration management and version control systems

Avainsanat — Nyckelord — Keywords

MLOps, continuous delivery, continuous integration, machine learning, pipeline, cloud-native

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Contents

A Tables of tools requirements

1 Introduction

The process of model deployment is a difficult part of the machine learning project's lifecycle [65, 106]. There is proven success in rapid development cycles by incorporating DevOps methodology [97, 11, 32] and tooling for Continuous Integration and Continuous Delivery (CI/CD) into software development [34, 29].

Machine learning is often a small part of software systems, but a software that incorporates machine learning has fallen behind on the CI/CD trend. Machine learning systems introduce added complexity and unique problems to the CI/CD pipelines [91]. Extending DevOps methodology to include machine learning system features is widely referred to as MLOps [69].

Current popular products for MLOps are often proprietary and offered by different cloud providers, Amazon, Google, or Microsoft [2, 1, 10]. Proprietary solutions reduce extendibility and transparency on a pipeline while enforcing heavy vendor lock-in. Cloud providers services are often not a feasible solution for companies that work on regulatory software devices or software with user-privacy concerns. They require an MLOps solution that can be run cloud-agnostic and on-premises machines. Current end-to-end open-source and cloud-agnostic solutions for MLOps pipelines are often incomplete, immature, non-generalized, hard to learn and use, or not fit for production.

This thesis aims to design an MLOps pipeline for cloud-native applications that can not rely on proprietary software or cloud provider's terms and conditions. We propose an open-source easy-to-use MLOps pipeline that provides a Kubernetes based cloud-native and production-ready environment for CI/CD and monitoring for machine learning systems. Pipeline generated model's inferences are traceable to its data and parameters for governance purposes. It also provides an interface for model exploration with automatic tracking of data, parameters and metrics.

In Chapter 2, we introduce our research approach, research questions and design objectives for the MLOps pipeline. In Chapter 3, we define DevOps methodology, describe CI/CD pipelines, and their requirements, describe why DevOps and CI/CD it is beneficial in software development, and define its' implementation phases. We describe cloud-native technologies to understand and motivate their usage. In Chapter 4, we extend and compare the DevOps methodologies and CI/CD pipelines into machine learning systems, to find the problems and requirements of a MLOps pipeline. Chapter 5 goes over the design and implementation details of our proposal, including an overview of tools, methods and architecture of the pipeline. We describe how it implements the elements and phases of DevOps defined in earlier sections, and satisfies the requirements of an end-to-end machine learning systems CI/CD pipelines. Chapter 6 goes through a demonstration of implementing a simple machine learning project's MLOps requirements on this MLOps pipeline. Chapter 7 contains discussion on limitations, future, and related work. Chapter 8 ends the thesis with conclusions.

2 Research approach

The background information is gathered as a literature review. All design and implementation of an MLOps pipeline as the artefact is conducted as Design Science [80]. We conduct design science because of the empirical need and applicability of creating the artefact for evaluation as a solution to our identified problem in the domain of MLOps. The key phases of the applied Design Science research process are detailed in following sections.

2.1 Problem identification and motivation

In this thesis, we consider the following research questions:

RQ1: What is required from a modern MLOps pipeline?

RQ2: How feasible is designing and implementing a MLOps pipeline with existing open-source cloud-native tooling?

RQ3: How cloud provider agnostic the solution can be?

To find answers to **RQ1** we conduct a literature review of the background in the field of machine learning, and software engineering. We conduct Design Science research process to build an artefact to answer **RQ2** and **RQ3**. Further problem identification and motivation for the artefact design is conducted in Chapters 3 and 4.

2.2 Objectives for a solution

As objectives for the solution, we have three distinct sets of features that guide us towards a better artefact: must-have objectives, features that correlate with positive development performance, and features that solve problem definition specific issues. These objectives are reflected on every selected tool and the whole artefact.

As must-have objectives for the MLOps pipeline we list the following features:

Obj1: Must provide features of the MLOps methodologies CI/CD pipelines, on the basis of **RQ1**.

Obj2: Must be cloud-native.

Obj3: Must be open-source.

These three feature ensure that the solution is usable and extendable for its job in modern software infrastructure.

Accelerate report [34] lists as the features of a toolchain that correlate with positive development performance as: how easy it is to use the toolchain?, how useful the toolchain is in accomplishing job-related goals?, and open-source and customizability? From the report we specify our set of objectives and considerations for the MLOps pipeline design and implementation as:

Obj4: Ease-of-use for developers using the tools.

Obj5: Effectiveness in accomplishing its job.

Obj6: Extendability and customizability.

We also consider the following features to solve our problem definition:

Obj7: Maturity of the tool – in this rapidly changing landscape, we want something that is industry-ready.

Obj8: Level of integration with runtime – we consider this for the interoperability of tools monitoring, debugging and portability.

Obj9: Generality – we want every data scientist to be able to integrate into the MLOps pipeline without significant added skill or knowledge, enabling focus on business issues.

To summarize the above objectives, we want to leverage mature tools that integrate well into a cloud environment that use general-purpose configuration languages like JSON and YAML, or general Python classes, functions and scripts as its user-facing building blocks. We want an tool that requires minimal source code changes to any services or processes to integrate all the steps and pieces of the pipeline. We want the pipeline to be portable to any cloud platform, on-premises machines, and simulated cloud environments on local machines. This way, the pipeline can be used on many projects having requirements on its cloud environment.

2.3 Design and development

The design and development artefact are presented to show how the artefact works and how it is built (Chapter 5). The design follows the identified problems and objectives of the artefact. The design choices are rationalized in Appendix A.

2.4 Demonstration and evaluation

We demonstrate the artefact by conducting a small use-case demonstration to show how the artefact is used (Chapter 6) and conduct an evaluation based on the objectives, demonstration and the design of the artefact (Chapter 7).

3 DevOps

DevOps – a portmanteau of "Development" and "Operations" – is a methodology with principles and practices bringing developers and operators to work together or as DevOps engineers. More specifically, the artefacts of DevOps methodology adds into the software development process [12, 109] with rapid and automated software delivery and quality assurance from source code changes with CI/CD pipelines [52], feedback with production monitoring, and configuration consistency with infrastructure-as-code or data [7, 49].

3.1 DevOps life-cycle

In practice, DevOps aims to use tooling and workflows to automate one or more of the phases of the DevOps life-cycle: Coding, building, testing, releasing, configuring, and monitoring (Figure 3.1).

Coding phase includes development, code review and version control tools. For example, a team decides to use Git as a version control tool and Github as a remote repository. The team defines a set of automatically enforced code style guidelines and test coverage percentage, decides on using the branching strategy of Trunk-Based Development [79] and enforces all changes to the main branch to be reviewed and accepted by a senior developer.

Building phase consists of the automatic creation and storing of artefacts. For example, a team decides to create a runnable container image of their product.

Testing phase includes continuous testing tools, team setups an environment where a set of ever-increasing set of tests is automatically run against every code revision, giving crucial

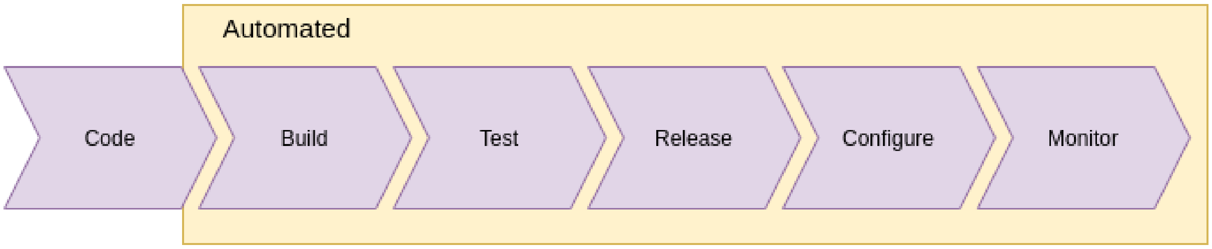

Figure 3.1: Phases of DevOps

information on its quality. Set of tests can range from unit, integration, configuration, end-to-end, and performance testing.

Releasing phase consists of release strategy and automation. The team has to decide on how to get their product released. For example, they decide if the product is released internally as a staging release first, and if they are using A/B testing in production. Also, they have to concern themselves with what to do when something is wrong with the deployment, create their rollback strategy.

Configuring phase consists of automatic infrastructure configuration and management. Best practices include declaring the production **infrastructure as code** [71, 47, 7], which means a set of scripts to reproduce the software's running environment and infrastructure from operating systems to databases and specific services and their networking configuration. The infrastructure can also be version controlled, tested, and deployed with the rest of the code.

Monitoring phase ranges from product performance to end-user experience monitoring. For example, it can cover how long database queries or website loading takes or how many of users is using specific features of the product or how many visits to a website ends up in registration or how many new users churn in a specific set of time. Monitoring phase covers automatic alerting of crashes or possible system metric thresholds, e.g. CPU utilization threshold. Production monitoring is essential to make sure the team is building the right product [23, 14].

3.2 Pipelines

To achieve CI/CD, developers create "pipelines", which are written manifestations of how to automatically build, test, release and configure software release [52]. The pipeline has an event that triggers the steps in a sequential fashion. If any step fails it will stop the pipeline from advancing further and gives feedback to the developers. In context Software Development the triggering event is often a new code revision. When each revision is continuously tested and proven to be releasable, the codebase stays in a working state, which means that **every** single code revision made can lead up to a new release of the software. CI/CD supports practice of regular, small commits, and even dozens of releases per day.

Without CI/CD there is a big risk of long-lived branches of broken code that requires integration into the software, blocking the entire development team from delivering other features, resulting in so called "integration hell", a situation where delivery is in halt for a long-period of time. With robust CI/CD there is no added cost of integration and delivery when implementing features, the Software Development team can stay agile and make high level decisions in short-term cycles, e.g. change their requirements, re-scope their product, and do experimental feature research.

Table 3.1: Examples of tools for specific phases of CI/CD pipeline automation.

Phase	Tools
Build	XCode [114], Docker [24], Gradle [39].
Test	pytest [85], Cypress [50], Mockito [70].
Release	Jenkins [51], Spinnaker [98], Flux [31].
Configure	Terraform [101], Ansible [45], CloudFormation [9].
Monitor	New Relic [73], Sentry [94], Prometheus [83].

In practice, the creating CI/CD pipelines consists of various tools, that are designed to solve a specific tasks or subset of tasks in a specific context of the DevOps life-cycle, e.g. building phase of a website. Examples of tools can be seen in Table 3.1. These tools are often interchangeable and a resulting set of tools is often called a toolchain. The CI/CD pipeline is an written artefact running selected toolchain on machines or cloud. The pipelines can be written to run on hosted services, e.g. **GitHub Actions**, **CircleCI**, or **Travis** [27, 20, 104]. There's also tools to run CI/CD pipelines on own machines, e.g. **Jenkins** or **ArgoCD** [51, 5]. These tools provide a configuration language to run execution steps and tools as a pipeline. It is not uncommon to see the pipeline split into multiple instructions running machines that each handles a subset of the process, e.g. building and testing happening in GitHub Actions and release on a local machine.

Key elements to consider in a CI/CD implementation are [52, 36]:

- Rollback capability,
- Observability and alerting,
- Security,
- Time to production.

For every single release, there needs to be a strategy to rollback the release into a previous version if something goes wrong. An easy solution for rollback could be running the older version through the same CI/CD pipeline. Rollbacks are not always straightforward, even if the running service can be rolled back, there is also possible data migrations and downtime of deploying a new version to consider.

Each release should be transparent on *what has changed*, *who authorized the change* and should be alerted or observable to the development team. The team needs to know if the deployment was successful and when. There should be clear alerts of broken code in the pipeline and possible failures in the release. If something goes wrong with the deployment, and there is no clear audit trail of changes, it can be tough to know where to rollback the system and what is causing issues in the deployment. Imagine a situation where a developer manually connects to a production machine and accidentally deletes a key file from the filesystem, which causes system failures after a few days. There is no auditable trace of changes, no one knows where to rollback, and even a rollback in the code might not help if it does not reproduce the missing file – a recipe for complete mayhem.

Security considerations for CI/CD pipelines are mostly regarding managing who can authorize deployments, and how to manage secrets in the pipelines. Limiting developers' access is critical for both security and observability – all production changes go through CI/CD pipelines. Giving third-party CI/CD tools access to a system and its secrets can leave the system vulnerable in case of third-party security breaches.

Time to production is essential for a company to stay agile. Long-lasting tests and builds can block other releases resulting in bigger commits, merge conflicts, and piling issues to be cleared [52].

In a machine learning context, the pipeline's steps are extended to handle the life-cycle of the model and data, but the elements, benefits, and objectives stay the same.

3.3 Benefits of the DevOps methodology

Elements of DevOps has a positive impact on **organization performance** [33, 32] as illustrated Figure 3.2, via multiple factors such as:

- **IT performance** – how long does it take for a system to restore from a defective version or bugs, how long does it take for a written feature to end in production and how often the organization deploy code.

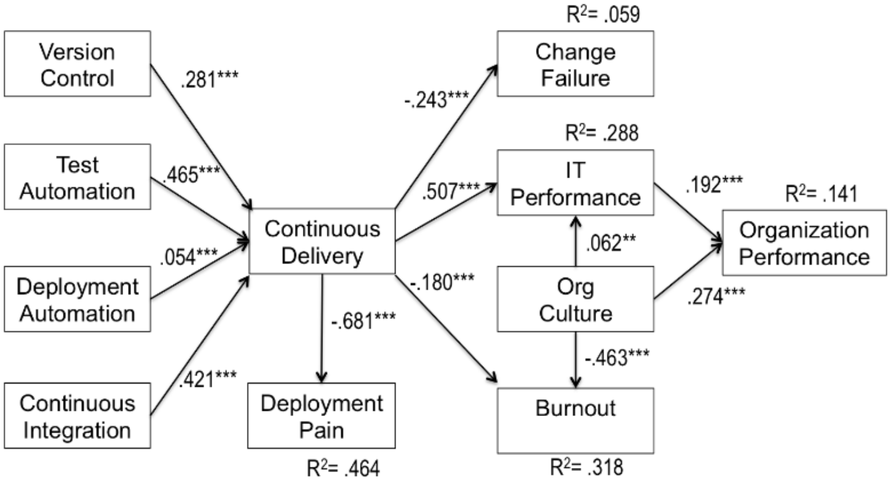

Figure 3.2: Impact of DevOps factors [33]

- **Change failure** – how often bad revisions are deployed to production.
- **Burnout** – how many or often developers feel burned out, exhausted or indifferent of their work.
- **Deployment Pain** – do developers fear code deployments or perceive them harmful.

There are several reasons why DevOps has been observed to be useful for developers and organizations in software projects:

While learning DevOps practices is hard and time consuming [56], DevOps improves organization performance factors and saves money and resources in the long run by trust and governance in software quality. Characteristics of quality software include understandability, completeness, conciseness, portability, consistency, maintainability, testability, usability, reliability, structuredness, and efficiency [13]. Software deployment pipeline can automatically test software development team's code revisions in version control to match these software quality characteristics and have it released to the production environment. Eliminating a series of operations, traditionally done manually, saves minutes, hours or even days of manual labour for each software release [34].

DevOps mitigates change failures and reduces deployment pain. Software deployment pipeline with automatic tests forces all code changes to run through the same set of tests with possibly new tests introduced for this and future revisions. Infrastructure as code, the practice of writing the infrastructure either as scripts or in a declarative fashion, increases the truck factor of the production environment as everything running on the machine or server is transparently defined and reproducible. The truck factor is a commonly used measurement of how much key personnel can be hit by a truck or otherwise disappear from a software development team until the team would be unable to deliver updates to the system because of lack of knowledge [8].

DevOps brings happiness. Listed as top reasons for unhappiness in developers work are: Time pressure, low-quality code, mundane or repetitive tasks, broken code, and underperforming colleagues [43]. These are all issues DevOps aims to solve. Developer happiness is a crucial factor in developer productivity [42].

DevOps practices creates a faster feedback loop. With robust monitoring systems development teams can respond to defects, alerts or other metrics rapidly and thus improve code quality or better target business goals.

Incorporating DevOps into the machine learning context, we hope to see similar benefits as in traditional software development.

3.4 Production environment

Managing a production environment is challenging, and even more-so with software that has special requirements with hardware, such as machine learning systems. The production environment is a set of hardware and software of a service that the end-users are directly or indirectly in contact with. The environment includes physical computer components, operating systems, **all** installed software in the machine, network configuration, and even the state of the software and hardware, e.g. used disk space or cached data.

The environment needs to be fail-safe and reproducible. To reduce the risk of new features of a system not working on production environment – even though it might work on developers own environment, it is common practice to run a staging environment, which is supposed to mimic the production environment as closely as possible. Some teams even use development environments that mimic the production environment to have hands-on tests and exploration on new features before committing with the features to the staging – and eventually the production environment. Creating a close replication of an environment is hard, but much success is found by abstracting layers of the environment by virtualization, and a declarative infrastructure as code definition of infrastructure and installed software.

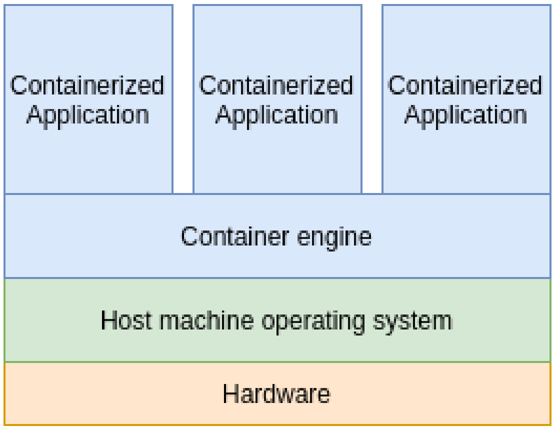
Containerized Application
Containerized Application
Containerized Application
Container engine
Host machine operating system
Hardware

Figure 3.3: OS-level virtualization.

3.4.1 Virtualization and Containers

Virtualization means creating a virtual layer of hardware, operating system or storage. It enables applications to run agnostic of its underlying hardware and systems. In modern software development, OS-level virtualization [116] is often done via containerization (Figure 3.3). In this scheme, each application or service of a full system is shipped with its container environment, running on top of the physical hardware's operating system's kernel.

A container is built from an image, which is a template of instructions on building the container. An image can include the operating system and any software needed to run a service, e.g. programming language compiler and step-by-step instructions for installing and running the service. The image can also define container configurations such as exposed network ports. Images can reference other images as a starting point and extend from there. A container is run in a container runtime, a process which manages the lifecycle of a container.

Implementing a software system from several independently deployable components is often called Microservices Architecture, where the components are referred to as microservices [67]. Containers works as great deployable units for microservices.

Implementations of container images and runtimes often comply to the **Open Container Initiative** (OCI) [75], a Linux Foundation project [103] designed to provide an open and general purpose format and interface for container images and runtimes [76, 77].

3.4.2 Container orchestration

Software applications and systems consisting of multiple containerized workloads or services requires orchestration. Orchestration concerns the management of the life-cycles of multiple dynamic and interconnected container workloads [15]. Orchestrating containers in a production setting can be a considerable amount of manual work, problems in container life-cycle to consider are, for example, how to start or replace a container? What to do when a container unexpectedly goes down? How to discover the network addresses of different containers? How to connect a container service to the storage it needs? Is a container available for connections? When to replicate a specific container? How to balance the service load between container replicas?

In dynamic and large scale environments where the running machines and services change frequently, it is virtually impossible to provide a highly-available service with just manual labour. Luckily there are several software projects which provide automatic orchestration of containerized workloads. Orchestrators do the heavy-lifting for the operators to provide highly-available and robust service systems.

As an example, Dockerfile [Listing 3.1] includes steps to install an operating system and all necessary prerequisites for installing golang:1.15.2 (used for tooling) [25] and the language itself, building application binary and then throwing all compiling necessary files away and in the end only including the final application binary and instruction to run it on container startup. When built to an image, it can be shipped to any machine with a container runtime as a self-contained and independent unit.

Listing 3.1: Dockerfile for a golang application

```
FROM golang:1.15.2 AS build
WORKDIR /usr/src/app
COPY . .
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build
FROM scratch
COPY --from=build /usr/src/app/client /usr/src/app/client
ENTRYPOINT [ "/usr/src/app/client" ]

```

3.4.3 Kubernetes

Kubernetes is the de-facto industry standard for container orchestration in the cloud-native space [18]. Kubernetes is an open-source platform to run containerized services as a cluster, providing several benefits and features to run reliable and portable applications.

Kubernetes provides service discovery and load balancing, exposing the service's domain name system (DNS) name and automatically distributes traffic to multiple replicas of the same service to keep the traffic stable. Kubernetes can automatically replicate all services based on CPU loads [112].

Kubernetes can automatically rollout or rollback new versions of services with zero downtime. It will launch up new versions of the service, wait for it to instantiate or respond positively to user-defined readiness check. When the new version is ready, it will automatically switch all traffic to the new version and take down the old one.

Kubernetes does bin packing. The developer defines Kubernetes how much CPU and memory each container requires. Kubernetes can automatically organize containers onto cluster nodes to make the best use of the cluster's resources. In the context of machine learning we can use *node selectors* and *node labels* to define specific workloads to be scheduled on labeled machines, e.g. a resource heavy training workload to run on a machine with a powerful graphics card.

Kubernetes is self-healing, meaning that if any container goes down unexpectedly or fails to respond to user-defined health checks, Kubernetes automatically replaces it.

All resources in a Kubernetes cluster are declarative containerized workloads running in pods. It relies on an OCI compliant container runtime to run the containers. a Pod is a set of running containers on a cluster, the smallest deployable unit a Kubernetes cluster can have. The standard workload resources in a Kubernetes cluster are Deployment, ReplicaSet, StatefulSet, DaemonSet and Jobs. The Kubernetes API is extended by writing custom controllers, which controls these workload resources or pods directly. For example, a custom controller could be a Cronjob which controls a set of jobs to work in defined intervals, and a Job controls that defined pods successfully run its workload and terminates.

4 MLOps

MLOps is the practice of applying DevOps to a software project including machine learning systems [69]. Machine learning brings in new roles and elements to the traditional software development process such as data, data scientists, data engineers, machine learning engineers, models and their regulation, model training pipelines, and model monitoring. These additions can increase the complexity of a Continuous Delivery system significantly.

In machine learning systems, the automated CI/CD pipelines are used to accelerate delivery and improve reproducibility [63, 44]. Valohai, a Finnish startup, surveyed 330 professionals in the machine learning domain, what they were working on in the last three months, and what obstacles they deem as relevant in their work. Most of the respondents said that they were working on topics that MLOps aims to automate, such as deploying to production, automating model re-training and model monitoring [106].

When categorizing answers between the people, who are in early stages of machine learning development, i.e. currently with issues regarding gathering or figuring out how to use data, learning machine learning technologies or proving their worth, we see that the perceived challenges shift more towards issues MLOps are trying to solve (Figure 4.1). With companies in later stages of development, we see respondents realize the challenges with deploying models to production and experiment tracking and comparison [65].

4.1 Machine Learning systems process

Delivering machine learning system to production from the ground up consists of several steps which can be manual labour or made by an automatic delivery pipeline. A machine learning process consists of [3]:

1. model requirements -> Not in pipeline,
2. data cleaning -> Extract-Transform-Load (ETL) (Section 4.1.1),
3. data labeling -> Extract-Transform-Load (ETL) (Section 4.1.1),
4. feature engineering -> training pipeline (Section 4.1.3),

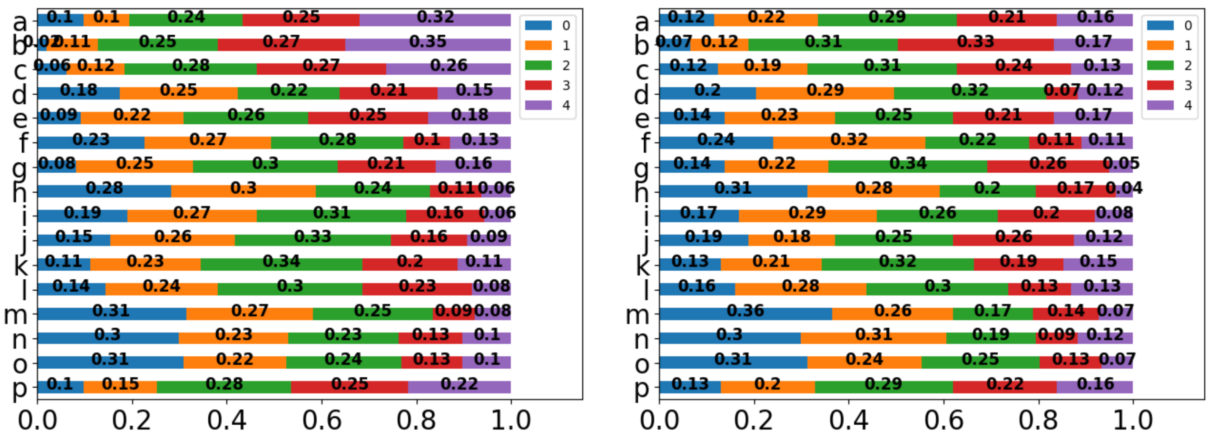

Figure 4.1: Perceived challenges with early-stage (left) machine learning organizations and mature-stage (right) organizations. Data from State of ML 2020, Valohai Survey. Legend: a) Lack of data; b) Messiness of data; c) Accessibility of data; d) Not enough data scientists; e) Not enough engineers/DevOps personnel; f) Not enough budget for purchases (solutions, computing); g) Difficulty of developing an effective model; h) Difficulty of using cloud resources for training; i) Difficulty of building training pipelines; j) Difficulty of deploying models; k) Difficulty of tracking and comparing experiments; l) Difficulty of collaborating on projects; m) Lack of version control; n) Lack of executive buy-in; o) Difficult regulatory environment; p) Unclear or unrealistic expectations. 0-4 scale of how relevant the challenge is.

5. model training -> training pipeline (Section 4.1.3),
6. model evaluation -> training pipeline (Section 4.1.3),
7. model deployment -> serving pipeline (Section 4.1.4),
8. model monitoring -> serving pipeline (Section 4.1.4).

Model requirements step consists of gathering understanding about what data or model algorithms to use to solve a problem. The value is the initial findings, and as such, it is considered something that is not to be automized or reproduced.

A machine learning Continuous Delivery pipeline consists of **Extract-Transform-Load (ETL)**, training and serving pipelines [88] (Figure 4.2). ETL handles process steps of data cleaning, labeling, and makes sure the processed data is accessible for further steps. Training pipeline handles that a model is created, tested and evaluated. Training pipeline handles machine learning process steps of feature engineering, model training and model evaluation. Serving pipeline handles model deployment and monitoring, transfers the trained model for the end-users and setups monitoring systems for it.

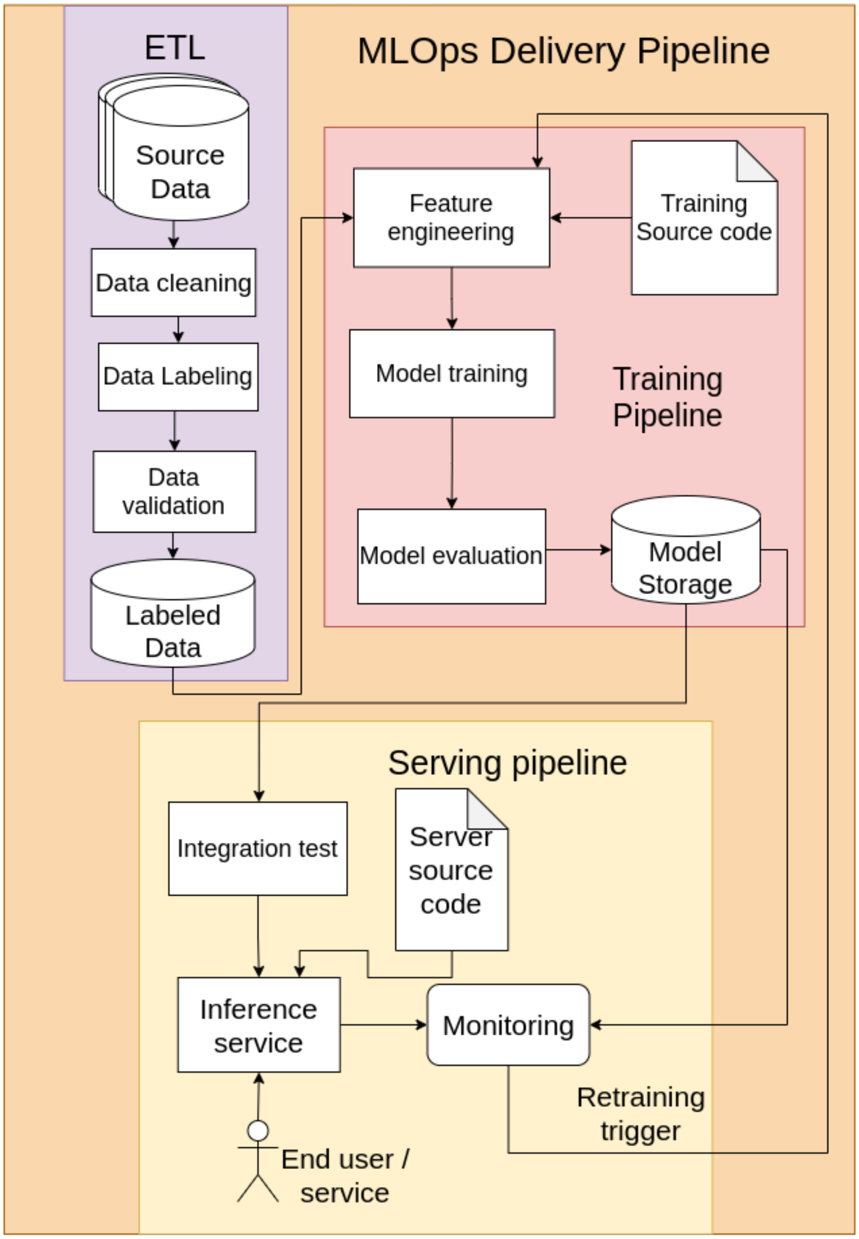

Figure 4.2: MLOps CI/CD pipeline.

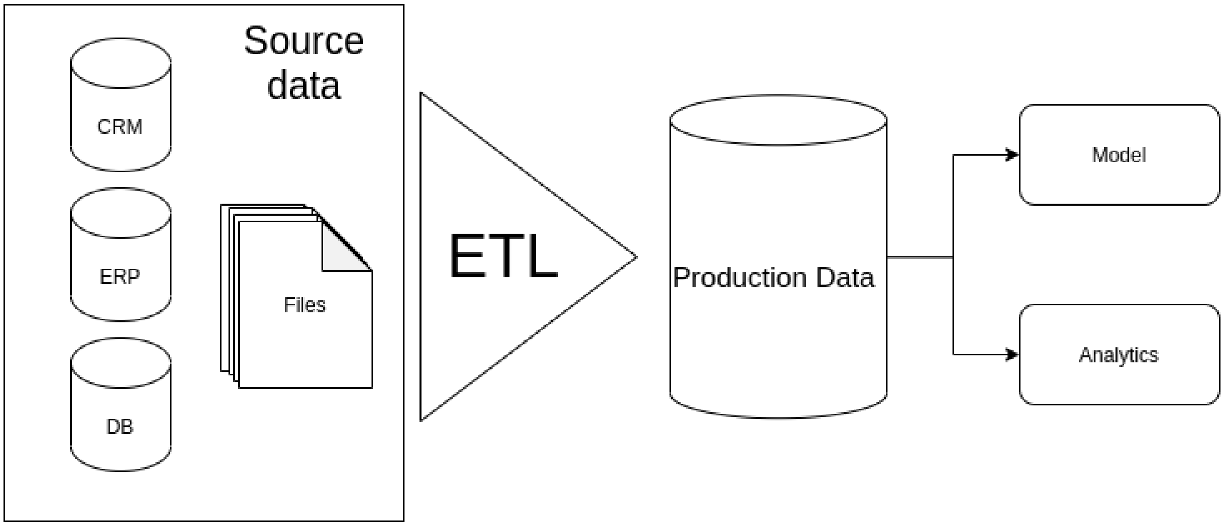

Figure 4.3: Function of ETL

4.1.1 Extract-Transform-Load (ETL)

The first thing building machine learning system requires is an ETL procedure [108, 112] (Figure 4.3). The extract phase is extracting data from multiple data sources onto a production database, data store, or data lake. Original data can be of any size, in any format and any geographical location.

In the transform phase, the data is cleaned and labeled, removing misspellings, inconsistencies, duplicate, conflicting or missing information, transforming it into a uniform format, and aggregated with other sources into its final business ready format. For some models, e.g. supervised learning models, the data requires labeled true values. Depending on the problem and models used, the labeling step might not be needed, and it might not be possible to automate. If labeling can not be automated, it requires a manual step, where an engineer receives the cleaned data, labels it and then triggers the pipeline onwards.

The transformed data is then loaded into production databases or data stores for training pipelines or other clients to use for analysis or other business requirements.

The ETL procedure can be as shallow as extracting rows from a single CSV onto a database table, or consist of hundreds of transformation operations on globally extracted data, and loading terabytes of data onto a data lake.

4.1.2 Exploration

Exploration steps are not considered part of the deployment or model training pipelines, but give the team invaluable information on what to build and how, as such this step does not concern MLOps, which is why it is not shown on figures.

Data exploration is done to understand the data better and provide insight into its form and business applicability. Exploration is done with just reading the data, visualizing or with statistical analysis. A better understanding of the data provides information on which models or algorithms would be the best for achieving business goals.

In model exploration, several different models are trained with the data to see which one seems most applicable.

4.1.3 Training pipeline

The model training and hyper-parameter optimization starts phase after choosing a model. The model is evaluated against set metrics and its trained several times over different hyper-parameters, e.g. learning rates, and the best performing model is elevated to testing stage, and others discarded. There are multiple strategies for hyper-parameter optimization and it is up to implementation what to use.

In evaluation stage, trained models are tested that they have good results in selected metrics with previously unseen data. The data is completely split from any evaluation or training data used earlier in the model training. It is recommended to test the model performance with known edge cases, adversarial inputs and other cherry-picked inputs that might surface unwanted bias or discrimination in the model. These tests should work as a benchmark for all possible iterations of trained models – so they're comparable with each other. At this stage the model performance should be compared to the current production model. A model with acceptable or better test score is picked as the final production-ready candidate.

The final model is stored in a location, services using it can access it. The stored model can be on a server disk, remote data storage or inside client applications.

Computationally expensive model training processes requires access to GPU hardware resources. Figure 4.4 presents performance differences of different models training processes with varying hardware, showing that using GPUs dramatically increases performance, regardless of model and framework [38].

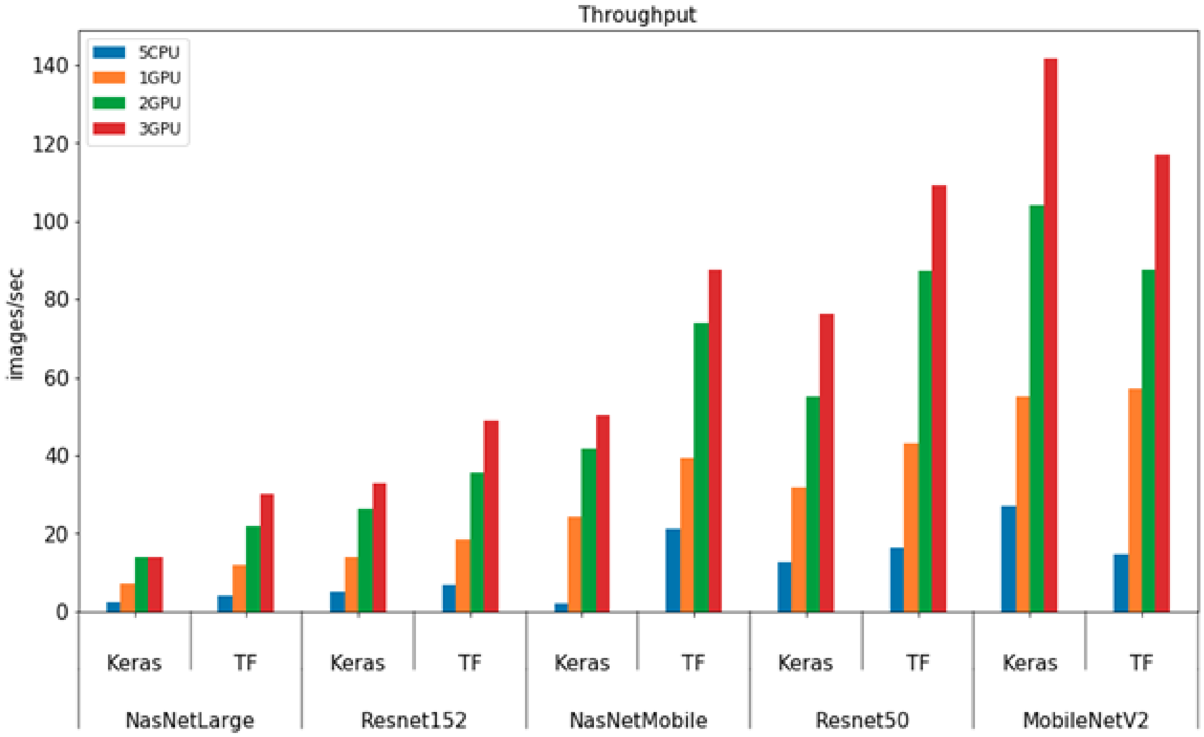

Figure 4.4: Processing speed increases with GPU-units, with all machine learning libraries and tasks [38].

4.1.4 Serving pipeline

The served model provides an interface for end users or services to input data into the model. The model output is redirected to an appropriate place, usually back to the entity giving the input, or to another service or model for further processing. The model interface is called an inference server or service.

The inference server can be served as a web server or a client application, such as a mobile app, or otherwise. Inference serving requires written application code and a robust infrastructure to succeed. The infrastructure has to be scalable and self-healing for the inference service to stay online. The model itself requires to be monitored to detect bias, drifting or adversarial attack. When detecting faulty models the actions to fix it must be swift to avoid further damage. Re-training is a common fix for biases and drifting, which requires training the model with new data, starting the whole process all over again, and without automation and version control it is hard to deliver fast.

4.2 Elements of MLOps

MLOps is a superset of DevOps. Introducing machine learning to a software system increases the complexity of the system, as well as its CI/CD pipeline, as shown in Figure 4.2. More elements need to be in version control, and a single training processes can take even weeks to complete, with several orchestrated steps that may require special hardware.

The Continuous Delivery pipeline always needs to know the latest address where the production data is stored, and have authorized access to it. Data can be vast and updating even daily or in real-time. The data is often transferred through the internet, from the storage to the pipeline, making processes longer and bringing weight to even geographical distances between the data and the pipeline system. For traceability and governance of the trained model, the dataset needs version control. This is done by storing snapshots or data differences of the dataset.

4.2.1 Artefacts

In traditional software systems, the product artefact is built from a specific snapshot of code. However, in machine learning systems, the artefact is a product of the source code and the data used to train the involved model(s), visualized in Figure 4.5. Often the training source code also takes model hyperparameters as parameters. In machine learning system, to reproduce an artefact, its source code, hyperparameters and training data needs to be version controlled.

While traditional software requires unit, integration, security and end-to-end system testing, an machine learning system requires all of this plus data and model testing. Testing phases are defined in the pipeline as declarative steps. If any tests fail the defined criteria, the pipeline execution is terminated, and thus the model is never released. The pipeline also can consist of steps where model selection is done based testing and evaluation metrics.

As the training can take hours to several days to complete, the training pipeline must be capable of restoring progress from a checkpoint in case of fault or suspension of the system. Because of this same time constraint, it is also essential that the pipeline can run multiple training sessions in parallel, independently from each other.

After a specific version of the model is created, it requires evaluation and testing in the

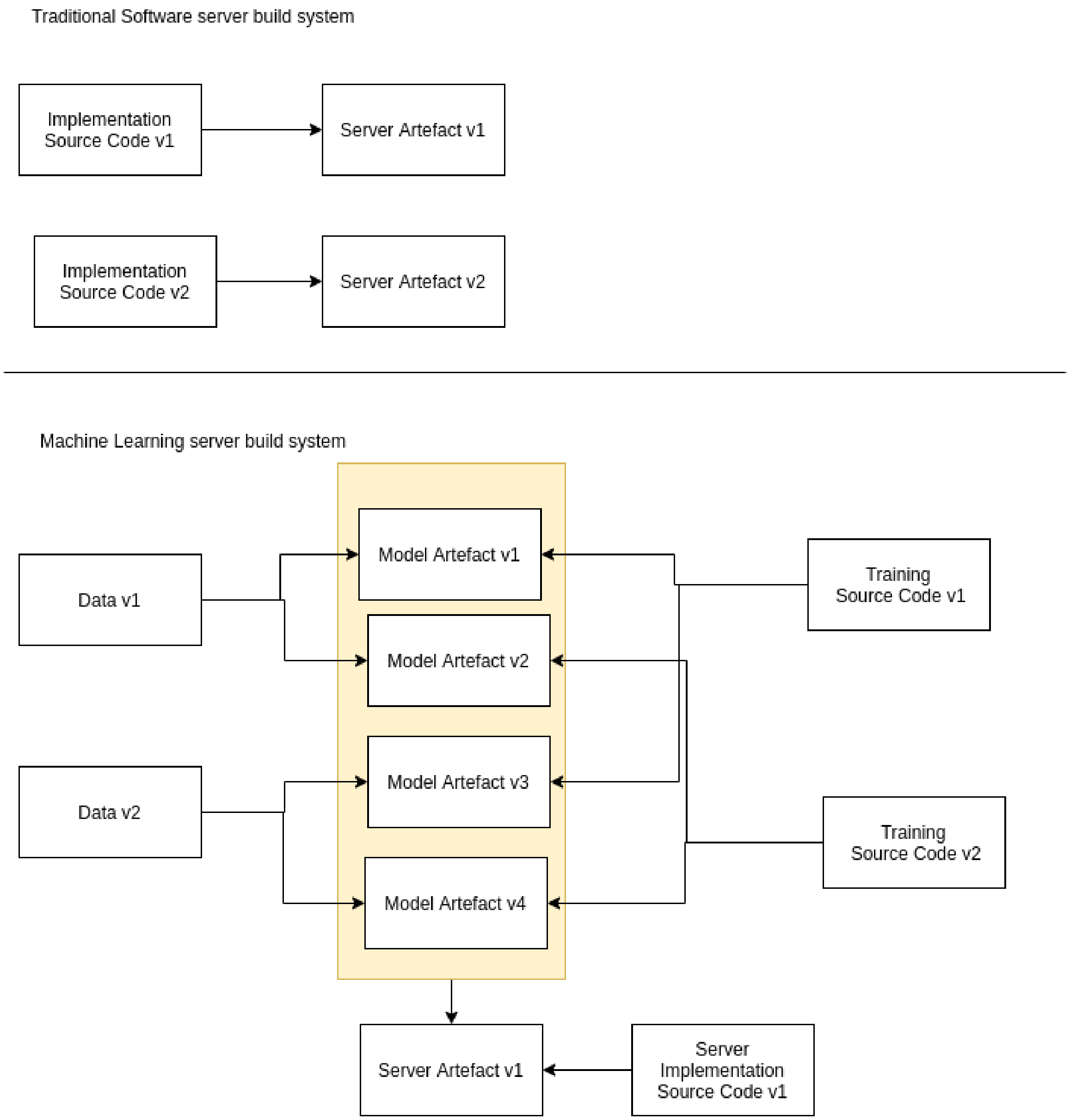

Figure 4.5: Factors in building a server artefact with machine learning are often more complex than in traditional software

pipeline before green-lighting it as production-ready. A model is tagged with a version number and metadata considering its hyperparameters, as well as its dataset, training, and evaluation code versions [89, 44]. With the necessary metadata, the model can be reproduced in the future, and its performance traced back to its data, training and evaluation. This reproducible environment is paramount for the model development. For example, to investigate why faults such as biased inference occur and how to eliminate it.

4.2.2 Deployment and production monitoring

The serving pipeline takes care of the deployment strategy of the model. When a new model is trained – or in traditional software, a new version of the code is created, it is common to deploy it into a staging environment before going into production. After staging is green-lighted, the new artefacts might be fully deployed with various deployment strategies. For example as a rolling A/B release, where a steadily increasing portion of the production traffic is split to the new version ultimately replacing the older. This strategy is used to monitor and benchmark the new version's performance with a smaller portion of end-users to detect faults and rollback before any more damage done.

The serving pipeline automatically integrates the served models into various monitoring systems to track performance and detect faults like model drifting, where the model's domain has changed radically since its training. A common way to fix models concept drift is to re-train the model with an updated dataset. A monitoring system should evaluate the model performance with live data detect the drift and trigger a re-training workflow, automatically training a new model and serving it once its ready [64].

Various monitoring measures are encouraged to be used to ensure a fast feedback-loop on system warnings, bugs, and downtime. A monitoring system is a software that actively reads or receives data from the production service. For example, a service may send all software errors to a monitoring system or a monitoring system could read metrics like concurrent users. A monitoring system often reacts to user-defined thresholds either by alarming developers or automatically changes the state of the production environment, e.g., by releasing an older version of the production service.

Choosing an action policy for faulty models is not trivial. The policy could be different for different alerts. For example, when drift is detected, it is not clear if the current model should be allowed to stay online. It is domain-specific. In some cases, the harm of wrong inference can be such that the model should be taken down immediately on detection or a new model is deployed as a hot-fix as soon as it finished training. In contrast, in other cases, the drift is not that critical, and the model should stay online until the re-trained model is tested, evaluated and safely elevated to the production.

The action policy should be changeable for each release, for example by stating that a specific release cannot be rolled back and any detected faults should not be acted upon at all or even as a nuclear option, take down the whole system. For example, when detecting a high error rate, the action could be to roll back to the previous version, but what if the older version had a significant issue with making discriminatory and illegal decisions? A naive solution would be to alert developers on fault detections and rely on the developers to solve all issues, but in most cases, this is not feasible or effective.

4.3 MLOps platforms

We introduce few mature MLOps platforms, which are selected based on different angles on open-source and level of vendor lock-in and discuss their shortcomings.

AWS Sagemaker [2] is an MLOps platform offered by AWS, which provides a rich industry-proven feature-complete set to the whole MLOps lifecycle – from data processing to deployments. However, the AWS Sagemaker is completely proprietary tooling, with heavy integration to the AWS other products and offerings. Using Sagemaker makes much sense if a company uses or intends to use AWS tooling on other aspects of their software development, and has no interest in migrating elsewhere – effectively using Sagemaker can lead to significant vendor lock-in. There is also no transparency on the inner workings of Sagemaker as it is proprietary software, which can be an issue for some developers.

Kubeflow [57] is an open-source cloud-native machine learning toolkit, designed to be run in Kubernetes. It has great integrations to many major cloud providers. It is also a feature-rich industry-proven tool with big company users [44]. The caveats of Kubeflow are that it requires quite a bit of work to integrate into other systems as it does not implement Continuous Delivery. Another caveat with Kubeflow is that it relies on writing in a Python library or DSL, which is an overhead for developing automated training pipelines.

Valohai [107] is an feature-rich end-to-end MLOps platform. It is proprietary but cloud-agnostic, running on most major cloud providers or on-premises. Valohai configuration is done with general-purpose YAML files, and the need for configuration is minimal. The main caveat of Valohai is that it is proprietary tooling. When using it, you rely on its features and quality – if some feature is needed for your solution or something is broken, there is nothing you can do to extend or fix the platform.

5 Design & implementation

To provide an easily configurable and extendable MLOps infrastructure, methodology, and toolchain for developers and data scientists. The design and implementation takes into account all stages of the machine learning systems process lifecycle – from data to training, to production and monitoring [†].

To simplify the presentation, here we only focus on the design and implementation, and overlook the rationale for tool selection.

Reasoning behind specific tool choices and mapping to objectives **Appendix A tables**.

5.1 GitOps

The GitOps, created by Weaveworks in 2017 [**noauthor_gitops_nodate-1**, 36, 62], is a way to implement cloud-native Continuous Delivery. Widely used Continuous Delivery tools use a Push-based model, whereas GitOps uses a Pull-based model. GitOps works especially well together with containers and with declarative configuration, which is why GitOps is seen trending in the cloud-native space. There are several tools developed to accomplish the GitOps workflow, notably **ArgoCD**, **Flux** and **Fleet** [5, 31, 30].

As visualized in Figure 5.1, at the core of the GitOps methodology is to define a declarative Infrastructure as Code representation of the system, which is stored in a Version Control System, e.g. Git [35]. The Git repository, as the single source of truth, declares the desired state of the system. Developers install software agents to the system that track the Git repository's desired state and the actual systems state and automatically steers the state to match the desired state.

In a Push-based model, when deploying to production, CI/CD pipeline would commonly have scripts that trigger on new Git revisions, which usually has steps to build and test the new version and in the end pushing the new configuration or applications into production using command-line tools with provided credentials.

In contrast, in a Pull-based model, developers state the desired state in the Git repository and software agents in production, automatically pulls the changes and applies them into

---

[†] The MLOps pipeline artefact can be seen at https://github.com/sasumaki/aiga-pipe.

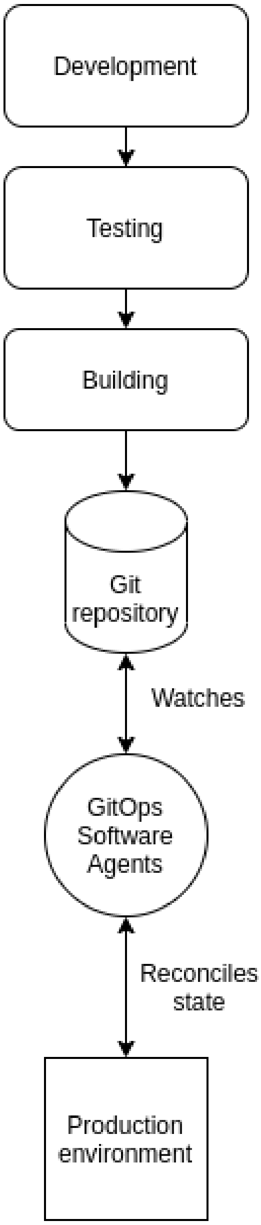

Figure 5.1: The GitOps workflow.

effect.

When using Pull-based model for Continuous Delivery, the system admins do not need to provide any developer or tool, credentials to access and modify their system. Restricting access credentials is a significant security improvement, eliminating the risk of credentials leaking to malicious actors via third party system breaches or human errors. Access to make changes to the system is handled by managing write-access to the tracked repositories.

Another improvement GitOps provides is risk reduction as rolling back the system is simple and requires no additional scripting or manual work from the system admins. As the whole system is declaratively defined and in version control, reverting the repository to a working revision is all that is needed to roll back the system entirely.

Using Git repositories as the single source of truth provides transparency on what is running on the cluster. New developers do not need to spend time trying to figure out, or documenting what is running, increasing the truck factor of the company.

With proper Git practices: small commits, descriptive commit messages, and pull request reviews – the changes made to the system are transparent, described and reviewed. A descriptive Git log is an excellent way to document information and background on why something is configured as it is, and by whom, further increasing the truck factor.

GitOps provides fast portability to the system. To move or clone the system, all that is needed is to spin up machines elsewhere and point it to track the same Git repository. GitOps makes developing easier as developers can have their development environments easily reproducible and matching each other's environments perfectly, reducing "*works on my environment*" issues.

On our Continuous Delivery implementation, we are using **Flux version 2** [31] to track and reconcile our production environment to the desired state, following the GitOps workflow. The GitOps is considered to be a part of the Serving pipeline of the MLOps delivery pipeline.

5.2 Model training and serving integration

Integrating the model to function in the training and serving pipelines requires a bit of work.

For model training process we have written a Python [111] wrapper that enables parameter passing to a user-written training function from a configuration file. The wrapper does automatic logging of parameters and outputs to a metadata file. The wrapper saves all JSON-formatted [22] logging in the user training function as runtime logs in the metadata file. Using the wrapper enables a data scientist to focus on writing a general training function. It makes integration of training workload with hyperparameter search and automatic result ranking to a pipeline easy. Automatic runtime logs enable providing transparent summaries of the training processes.

For now, with only python wrapper available, the training process must be written in Python to utilize these features.

Building a model to be able to receive inference requests, traditionally, includes much coding outside the scope of model development. For example, implementing an HTTP server [60] that runs inputs through the model requires the developer to write code for fetching the model, starting up and configuring a web server, configure routing, and write adequate logging and metrics. This work is rarely domain-specific.

To enable focus on model-specific tasks and run the model, we use a model wrapping technique provided by **seldon-core** [92], an open-source library for machine learning deployments. The model wrapper takes a class describing the model as a parameter, including functions for initialization and prediction and a file of required third-party dependencies and injects this model code to a functional web server. The server includes all necessary configurations to start the server, routing, logging, metrics, and eventing. The result is an OCI-compliant image running a inference server of the model. Seldon-core Kubernetes controllers handles the lifecycle of the model deployment inside the Kubernetes cluster (SeldonDeployment).

Seldon-core also provides pre-packaged inference servers, which we will cover later in Section 5.3.2.

5.3 Cluster architechture

Our solution is designed to be run in a Kubernetes cluster, to leverage its state-of-the-art container orchestration capabilities and existing tooling. Components of the cluster are visualized in Figure 5.2. The image includes all the most fundamental components that create the MLOps environment and pipeline. The following sections go through the architecture in order of concerns.

5.3.1 Artefact storage and version tracking

All code and infrastructure as a code declaration are in Git repository. Large model binaries and datasets are in cloud storage, as shipping the data with code images is unfeasible and redundant. A production workload can automatically fetch the model or data to local disk before using it, leveraging initialization containers in Kubernetes. A pod with a initialization container is kept at unready state until the initialization container has exited its process successfully.

Training pipelines launch with a unique tag, which is placed on all artefacts generated in the pipeline. With models it is tagged in the metadata file and also best practice would be to tag the cloud storage bucket access with the same tag. This way it is easy to trace and verify which artefacts are a result of which training run. All deployed models are present as a declaration in the cluster Git repository and its history. Data version control is assumed to be handled by third-party storage providers.

5.3.2 Inference Server

When the model deployed as an Inference server, we can leverage multiple pre-packaged inference servers provided by **Seldon-Core**, or write our own as a runnable image. Using pre-packaged servers eliminate a lot of the work required to get a model deployed. Pre-packaged servers offered are:

- **MLFlow Server** [68]
- **SKLearn Server** [96]
- **Triton Inference Server** [105]
- **Tensorflow Serving** [95]
- **XGBoost Server** [115].

Of these servers, **XGBoost** and **SKLearn** are offered through **Seldon-Core MLServer** [93]. These pre-packaged servers provide a gRPC or HTTP/REST server for specific model execution backends, e.g. **SKLearn Server** can be used to deploy a model created

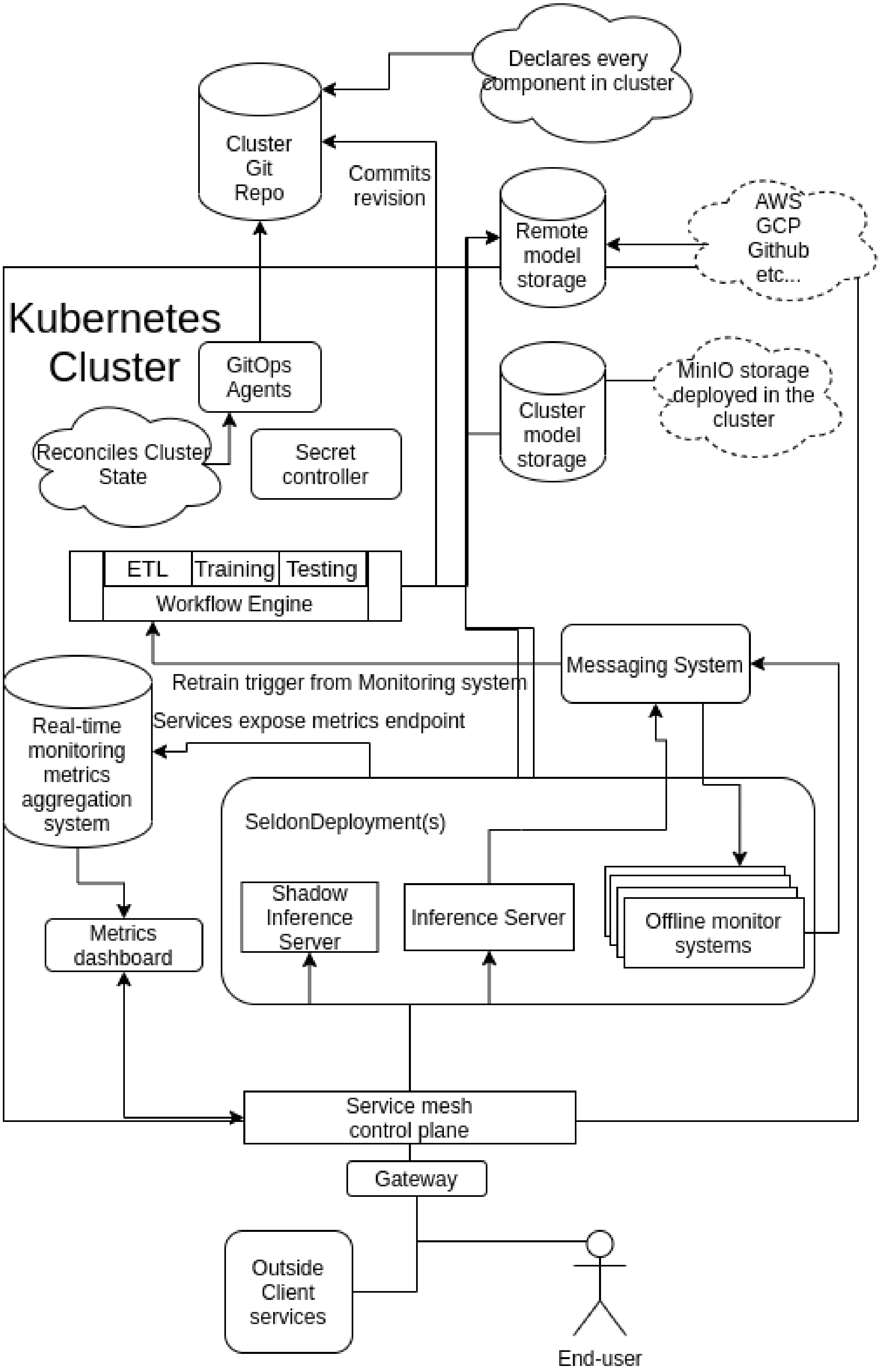

Figure 5.2: Cluster architecture – everything for ETL and training pipelines are running in the Workflow Engine, rest is for serving pipeline (Figure 4.2).

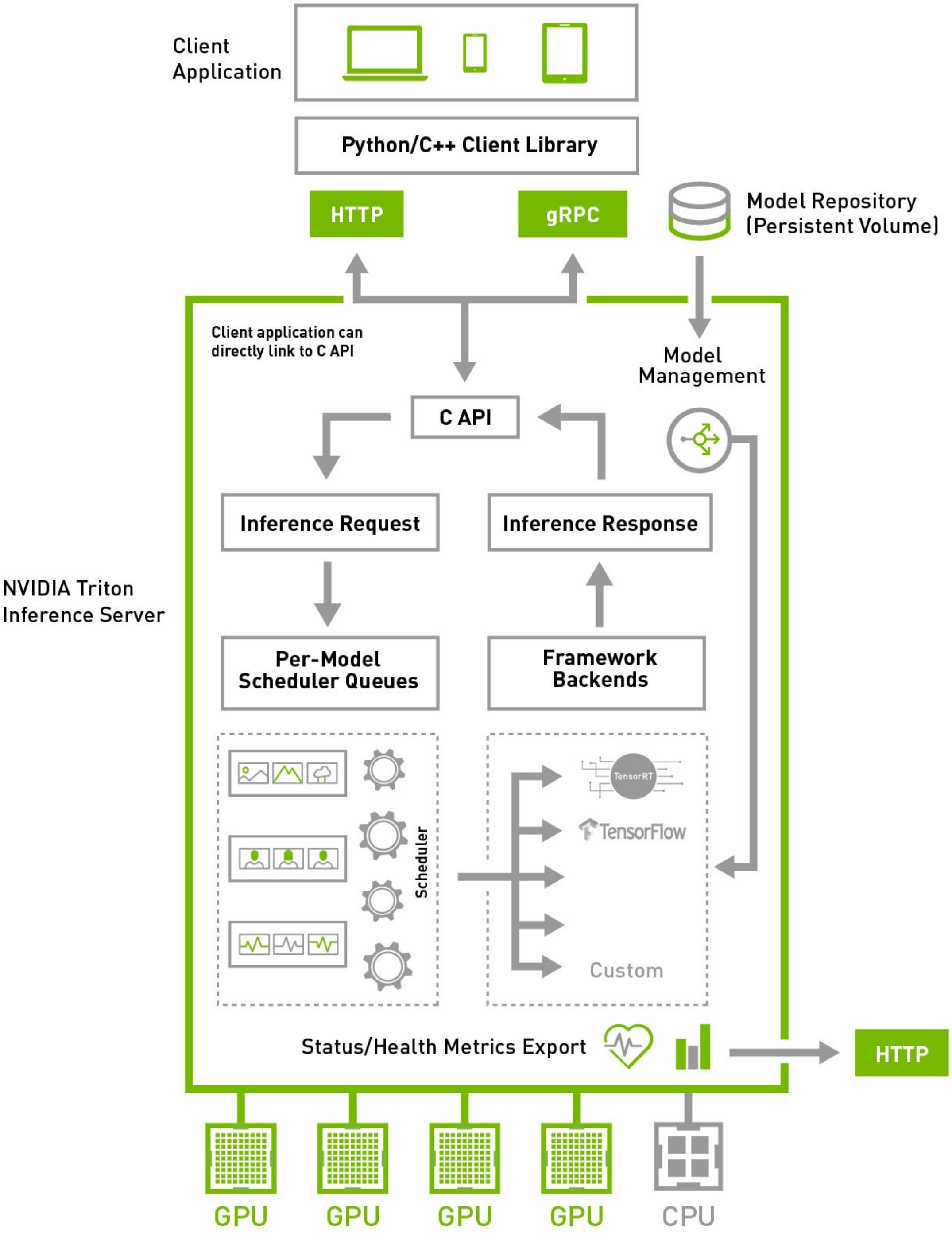

Figure 5.3: Triton Inference Server diagram [105]

and serialized with **Scikit-learn** machine learning library [90]. The servers implement request scheduling and batching to fit use-cases where real-time inference is not feasible. The **Triton Inference Server** can run multiple models inside the server, schedule and batch each inference request on a per-model basis (Figure 5.3). The **Triton Inference Service** is designed to be used in Kubernetes environment, as such it shipped as a Docker container, exposes status, health and metrics in **Prometheus** usable [84, 83] format for Kubernetes liveness probes and metric monitors [74]. Prometheus is a common product for aggregating and querying real-time metrics in Kubernetes ecosystems. The provided servers cover the deployment of many of the used machine learning libraries and use-cases in the industry.

Most of the pre-packaged servers offer three protocols:

- **REST and gRPC Seldon protocol** [81]
- **REST and gRPC Tensorflow Serving Protocol** [100]
- **REST and gRPC KFServing Protocol V2** [58]

The protocols define the API endpoints and payload format used to communicate with the server and how the data is chained between multiple models. The KFServing Protocol V2 is a joint project between **KFServing** and **Nvidia Triton Inference Server** to design and offer a standardized machine learning inference Protocol.

The KFServing Protocol V2 REST/HTTP defines server endpoints such as

- Health:
  GET v2/health/live
  GET v2/health/ready
  GET v2/models/${MODEL_NAME}[/versions/${MODEL_VERSION}]/ready
- Server Metadata:
  GET v2
- Model Metadata:
  GET v2/models/${MODEL_NAME}[/versions/${MODEL_VERSION}]
- Inference:
  POST v2/models/${MODEL_NAME}[/versions/${MODEL_VERSION}]/infer

The inference end point POST endpoint defines a specific JSON-schema for request and response, and a valid example request could look like Listing 5.1.

Listing 5.1: Example of KFServing Protocol V2 REST request

```
POST /v2/models/model_name/infer HTTP/1.1
Host: localhost:3000
Content-Type: application/json
Content-Length: <xxx>
{
"inputs" : [
        {
            "name": "INPUT0",
            "data": [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],
            "datatype": "INT32",
            "shape": [1,16]
        }
    ]
  ],
}

```

5.3.3 Workflow engine

The order of work is often relevant; for example, a model deployment process requires the model training to have finished before execution. For handling the sequence of processes automatically, we need workload orchestration.

In our implementation we use **Argo workflows** – container-native workflow engine for orchestrating parallel jobs on Kubernetes [113]. In the context of our design, we use the workflow engine to run ETL and training pipeline workloads.

Argo workflows provide a declarative interface to define workloads and their order of execution, either as a sequence of workloads or as a directed acyclic graph. Argo workflows allow parallel workloads that enable multiple instances of the training pipeline to be launched on-demand, utilized in model exploration and hyperparameter tuning. Workflow

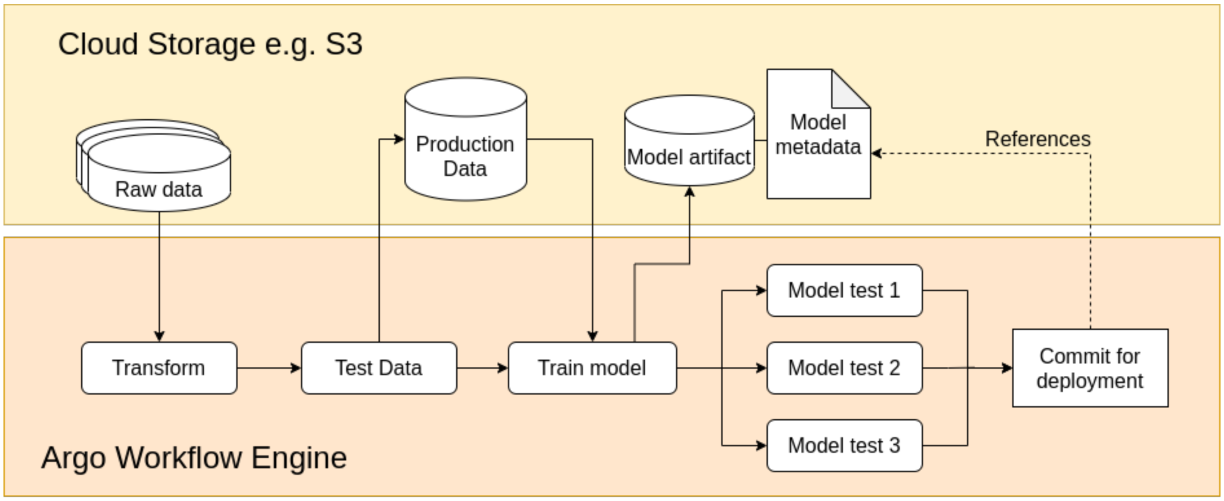

Figure 5.4: Argo workflow example

configuration also enables an easy way to implement artefact saving and passing between workloads.

A YAML Kubernetes manifest of a model training workload declares the image and command to run the training, with input data fetched from a cloud storage bucket and output model artefact saved into another bucket Listing 5.2. The training image does not need to implement downloading or uploading artefacts, other than having a reference of them on the local disk. Argo workflows launch an initialization container to download the files and another container to upload after workload execution. The output parameters are used as a parameter input in following workloads, such as deployment processes. Figure 5.4 demonstrates an example of an Argo workflow model training process.

Listing 5.2: Snippet of Argo Workflow template

```
− name: training
  container:
    image: sasumaki/ lol −train
    args: [ aiga_train , Train , −−parameters, "{{inputs.parameters}}"]
  inputs:
    parameters:
      - name: learning_rate
     artifacts :
      - name: data
      path: /app/data
      s3:
        endpoint: s3.amazonaws.com
        bucket: aiga −data
        key: MNIST_data
    outputs:
      parameters:
        - name: S3_MODEL_URI
          value: s3:// aiga −models/{{pod.name}}
       artifacts :
      - name: model
        path: /app/ outputs /model.onnx
        s3:
          endpoint: s3.amazonaws.com
          bucket: aiga −models
          key: "{{pod.name}}/model.onnx"
...
steps:
 - −name: train
    template: training
    arguments:
      parameters:
        - name: learning_rate
          value: "{{item.rate}}"
    withItems:
      - { rate: 0.0002 }
      - { rate: 0.0003 }

```

With **Argo events** [6] we create a webhook to allow triggering workflows from remote services, such as monitoring systems. With these Argo tools, developers can configure a complete end-to-end model training, evaluation and deployment process in an automated, reproducible and scalable way without any human interaction needed.

5.3.4 Networking and messaging

Networking defines how the different services in the cluster share data between each other. We use mainly two ways of networking in our implementation: via a service mesh and cloud messaging. A service mesh is a way to network a series of services on the application layer, typically enabling automatic service discovery, traffic control, and complex load balancing. As a service mesh in our cluster, we use **Istio** [48], because of its maturity, robust integration with Seldon-core, and rich feature set:

- Dynamic service discovery
- Routing rules
- Load balancing with round-robin, random, weighted or least requests strategies
- Rich security by many encryption, authentication and authorization features
- Staged rollouts and traffic splitting
- Network resiliency features by retries, failovers, circuit breakers, and fault injection.
- Observability by metrics, tracing and logging.

Istio enables a plugin gateway configuration to publish endpoints to deployed models Listing 5.3, automatic load balancing and metrics to import in Prometheus.

Listing 5.3: Minimal Istio gateway configuration to publish HTTP model endpoints

```
apiVersion: networking . istio . io/v1alpha3
kind: Gateway
metadata:
  name: seldon−gateway
  namespace: istio −system
spec:
  selector :
    istio : ingressgateway
  servers:
  - hosts:
    - '*'
    port:
      name: http
      number: 80
      protocol: HTTP
```

A messaging system is used to pass and filter inference inputs and outputs from models to monitoring systems. The system is based on **Knative eventing** [55] brokers and triggers and **NATS Streaming** [72] channels. A Seldon-core deployment automatically publishes all model inputs and outputs as **CloudEvents** [17] to a Knative eventing broker. CloudEvents is a standardized specification of message data. The message is filtered and routed to specific channels by a user-defined trigger Listing 5.4 Figure 5.5. Filtering is done based on CloudEvents attribute headers. Referred monitoring services receive these messages as HTTP Post requests, without an explicit subscription.

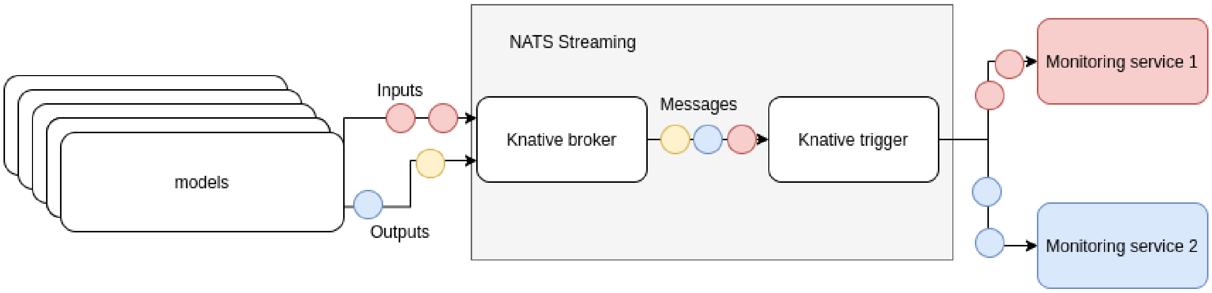

Figure 5.5: Model to monitoring service messaging. Different colored balls represent messages with a different type attribute.

Listing 5.4: Knative eventing trigger configuration to filter all model requests in nats-broker to a specific service

```
apiVersion: eventing . knative .dev/v1beta1
kind: Trigger
metadata:
  name: outlier − detector
  namespace: seldon−system

spec:
  broker: nats−broker
   filter :
     attributes:
       type: io . seldon . serving . inference . request
  subscriber:
     ref:
       apiVersion: machinelearning . seldon . io /v1
       kind: SeldonDeployment
       name: detector


```

These networking methods enable us to configure complex many-to-many messaging schemes, automatic routing and deployment strategies between dynamic fleets of services on the infrastructure level without explicit network coding in the services.

5.3.5 Deployment Strategies

For deployments, the system provides several stages of verification and validation. First, in every code commit we can provide an automatic test that the new infrastructure declaration can be automatically built inside a docker environment and test end-to-end that requests are going through, monitoring is working and other system-wide tests. After successful tests, the infrastructure is accepted into staging or production branch.

Another cluster with GitOps agents can be configured to watch and reconcile a specific staging folder or branch in the infrastructure repository. With the staging cluster, developers can validate their system before a production release.

Istio service-mesh and Seldon-core enable ways to test a new production environment model without much or any risk of catastrophes. Shadow deployment, also known as shadow testing [4], is a deployment strategy where a model service is created inside the cluster as a "shadow model" with all production traffic coming into a production model is mirrored into the shadow model. However, the shadow does not propagate the output to the end-user. Using a shadow model we can monitor, validate and compare how a new model would behave in a production setting without it interfering with the rest of the system as long as the shadow is a stateless application.

Shadow can be gradually deployed as the production version as a canary deployment. In canary deployment, the traffic is not mirrored like in shadow but split between the new and old version. The percentage of split traffic can be increased gradually, eventually hitting 100% of the traffic. These deployment strategies decrease the risk of releasing a faulty version of the model as the model runs in a real production environment, no test cases or simulation involved.

5.3.6 Real-time monitoring and alerting

We use **Prometheus** for real-time metrics of Inference server requests and inference time, each metric is also tagged with its service name, deployment name, predictor name, predictor version, model name, model image, model version, and used training data. The tags can be used as filters to draw more granular data from the metrics. Seldon-core enables users to create more custom metrics for prometheus in their model image. Prometheus also enables creating system alerts on user defined metrics thresholds, e.g. if inference time gets too long.

We use **Grafana** [40, 41] to create dashboard view of metrics in Prometheus. Grafana provides observability by filterable time-series data visualizations of the Prometheus metrics. The dashboards can be exposed as a public web-service with authentication.

5.3.7 Offline monitoring systems

this context, offline monitoring systems refer to services that receive inputs and outputs of inference servers deployed in the cluster. They monitor that there are no issues with them, e.g. Concept drifting of models. Offline monitors do not respond in real-time, and they do not interfere with responses of the models. Instead, all inputs and outputs are aggregated and distributed in asynchronous fashion using NATS Streaming and a message broker. Concept drift [110], adversarial attack [28], outlier detection [46] and other typical monitoring applications in machine learning are often machine learning models. The models can have long inference times or do batch processing, making them unfeasible in real-time inference services. These systems are used to alert developers or trigger different actions in the cluster, e.g. re-training of a model.

6 Demonstration

To demonstrate the pipeline, we integrate a simple end-to-end machine learning process of a neural network that recognizes handwritten digits [†].

6.1 ETL

We use MNIST [61], a dataset of handwritten digits. The dataset consists of 70 000 labeled digits (Figure 6.1). The digits are 28 x 28 pixels of size and each pixel is encoded as grayscale integer values between 0-255. We have saved the digits' source data in S3, an object datastore offered by Amazon [16]. The S3 files themselves are version controlled by Amazon. The data could be saved in any cloud providers data storage or inside the hosted cluster's Minio server. From the data, we have set up a Argo workflow pipeline that first does an arbitrary check on the data and saves it into another file in S3. The check is a data function, which has a chance of failure, demonstrating a validation step in a production use-case. The data is clean so we do not need to do any data processing or labeling for the model training, but if we did it would be done in similar fashion as the validation.

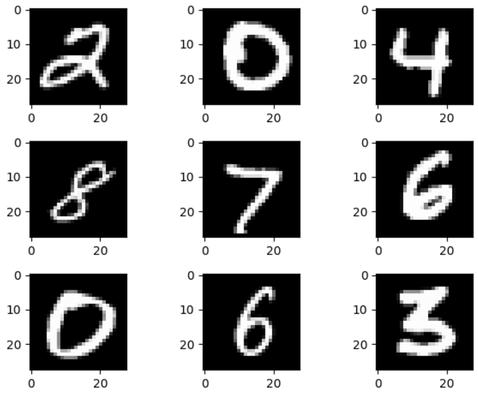

Figure 6.1: Samples of mnist data visualized.

[†] The demonstration using the pipeline can be found at https://github.com/sasumaki/asd.

6.2 Model training and evaluation

Using the data, we run a training step with declarative parameters. Training implementation can be found at https://github.com/sasumaki/mnist. The model is a convolutional neural network built with Tensorflow [99] and Keras [54] as a programming interface Listing 6.1.

Listing 6.1: Keras model definition for mnist digit recognition

```
 class MyModel(Model):
    def __init__(self):
      super(MyModel, self).__init__()
      self.conv1 = Conv2D(32, 3, activation='relu')
      self.flatten = Flatten()
      self.d1 = Dense(128, activation='relu')
      self.d2 = Dense(10)

model = MyModel()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=
   True)
optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
train_loss = tf.keras.metrics.Mean(name='train_loss')
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='
   test_accuracy')

model.compile(optimizer=optimizer,
            loss=train_loss,
            metrics=[train_accuracy])

```

The model training image creates a file for the model, which is defined as an output artefact in the Argo workflow declaration Listing 5.2, which automatically saves it into Amazon S3. The S3 address is pushed to staging branch of the model repositories deployment declaration, where it is automatically tested by Github Actions CI tool [27] that checks it can be run in Kubernetes as own wrapped implementation or as Triton Inference Server, and has good enough accuracy. The staging model is also deployed into the production cluster as a shadow deployment. When the model is approved by the repositories CI test,

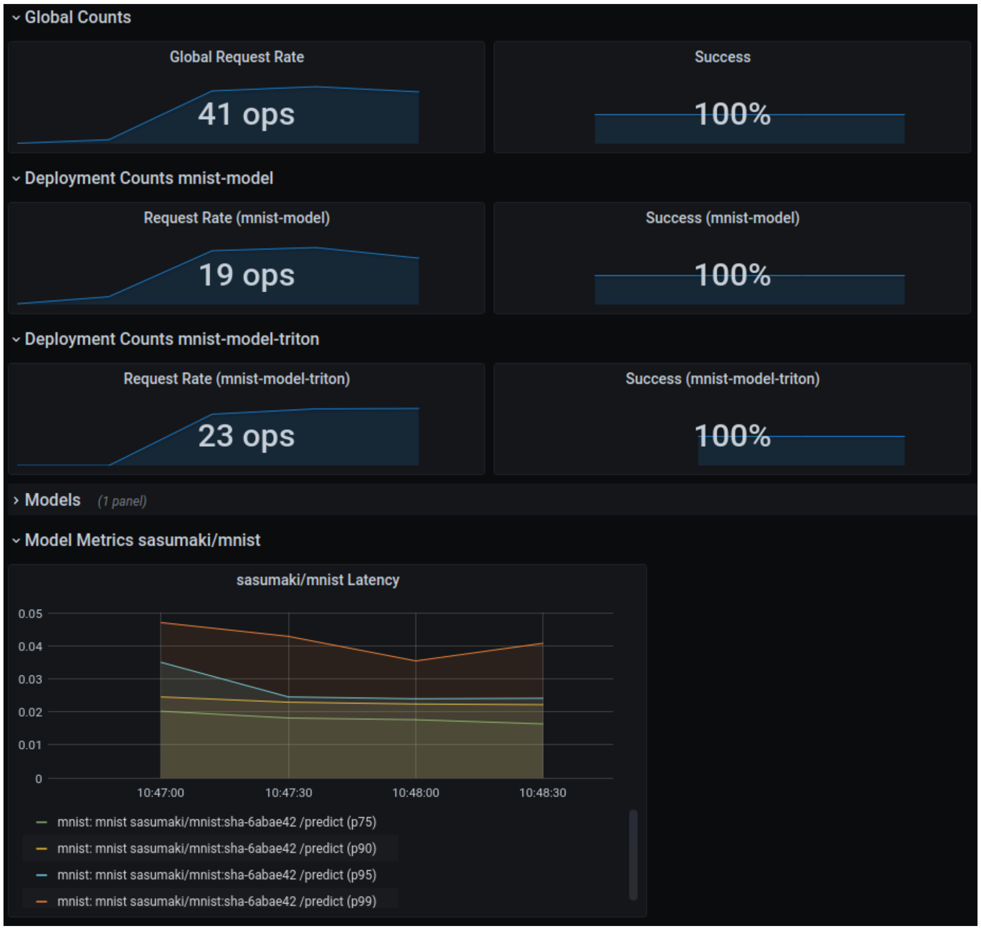
Global Counts
Global Request Rate
41 ops
Success
100%
Deployment Counts mnist-model
Request Rate (mnist-model)
19 ops
Success (mnist-model)
100%
Deployment Counts mnist-model-triton
Request Rate (mnist-model-triton)
23 ops
Success (mnist-model-triton)
100%
Models (1 panel)
Model Metrics sasumaki/mnist
sasumaki/mnist Latency
0.05
0.04
0.03
0.02
0.01
0
10:47:00
10:47:30
10:48:00
10:48:30
mnist: mnist sasumaki/mnist:sha-6abae42 /predict (p75)
mnist: mnist sasumaki/mnist:sha-6abae42 /predict (p90)
mnist: mnist sasumaki/mnist:sha-6abae42 /predict (p95)
mnist: mnist sasumaki/mnist:sha-6abae42 /predict (p99)

Figure 6.2: Screenshot of Grafana dashboard showing model metrics.

it automatically creates a pull request to the main branch.

6.3 Model serving and monitoring

When merged to the main branch, the GitOps agents in the cluster will trigger a rolling update of a new version of the inference server serving the new model. We have an implementation where we use a Seldon model wrapper to create the inference server and we a version where we use Triton as the inference server (Listing 6.2). These require the model to have specific folder structures and file naming conventions in S3 which means the same S3 address can not be used in both implementations. A deployed model can be then accessed as demostrated in Listing 6.3. The models' metrics are automatically connected to the clusters monitoring metrics aggregation system and immediately seen in Figure 6.2 dashboard. Inferences are also sent to the messaging broker, where they are propagated to a service acting as a real-time monitoring system such as a drift detector. This service will call a webhook triggering the training pipeline all over again.

We've run this demonstration on a local machine in K3d, a lightweight Kubernetes distribution in docker environment [86], and in the cloud at Civo managed Kubernetes service [66].

Listing 6.2: An example of mnist deployment with Triton Inference and KFServing V2 protocol.

```yaml
apiVersion:  machinelearning . seldon . io / v1alpha2
kind: SeldonDeployment
metadata:
  name: mnist-model-triton
  namespace: seldon-system
spec:
  protocol:  kfserving
  predictors:
  - graph:
      implementation: TRITON_SERVER
      modelUri: s3://aiga-models/mnist-4m59n-4092860028
      envSecretRefName: seldon-init-container- secret
      name: mnist
    name: mnist
    replicas :  1
```

Listing 6.3: Python script for getting a prediction out of a model Listing 6.2.

```python
url = "http://localhost:8081/seldon/seldon-system/mnist-model-triton/v2/models/mnist/infer"
data = {"inputs":[{"name":"input_1","data": x.tolist() ,"datatype":"FP32","shape":[1,28,28,1]}]}
headers = {"Content-Type": "application/json"}
r = requests.post(url, data = json.dumps(data), headers = headers)
res = r.json()["outputs"][0]["data"]
prediction = int(np.argmax(np.array(res).squeeze(), axis=0))
```

7 Discussion

We discuss the results of the research questions, evaluation and limitations of the MLOps pipeline as the Design Science artefact, and related work that has been done during this thesis.

7.1 Results

We determine the requirements of the MLOps pipeline as a system that is able to handle all processes of a machine learning process in an automated and reproducible fashion. That is, data cleaning, data labeling, feature engineering, model training, model evaluation, model deployment, and model monitoring **RQ1**.

Our MLOps pipeline is able to handle all these steps in the process with open-source and cloud-native tooling **RQ2**. Data labeling is not handled in any special fashion. If the labeling is an automatic process that is done with, for example unsupervised methods, it is possible to plugin the workflow. If labeling is done manually by domain experts, it cannot be automatically plugged into the pipeline. A way to incorporate labeling is to have automatic cleaning and alert of a finished clean dataset that a domain expert can label and then manually trigger the pipeline to continue onwards on the process. Deployment and monitoring is automatic with GitOps agents and a deployed model will automatically attach on monitoring systems in place.

All components are open-source and is cloud provider agnostic to a degree **RQ3**. We do not use any tooling that is, or depends on, other tooling that is proprietary to any cloud provider. Managed Kubernetes platforms can have unique elements in them that *can* have an effect on migrating the pipeline between different providers. We can also manage our own Kubernetes cluster on different cloud providers, which would eliminate this issue. Also we can move the pipeline on on-premises Kubernetes clusters. We can run the pipeline in simulated Kuberenetes environments.

The naive demonstration (Chapter 6) shows that most of the MLOps pipeline requirements can be met by the solution either by design or by the underlying technologies (Table 7.1). Using the pipeline should increase development performance in automizing CI/CD of machine learning projects. It is relatively straight forward to use but requires some knowledge of working with Kubernetes (Table A.1). The pipeline relies on the Kubernetes cluster heavily with chosen tooling that integrates well with it. The chosen tools are mature, at least in the context of a new field such as MLOps. As shown in the demonstration, there are little changes done to any source code of the mnist business-relevant pieces, and there is no additional code for the pipeline. Every piece is integrated together using YAML configuration (Table A.2).

Tool specific objectives are discussed in Appendix A, to provide rationale for tool decisions and granular view of the design objectives.

Table 7.1: MLOps pipeline must-have objectives

Obj1	The MLOps pipeline provides the MLOps pipeline features discussed in this thesis as demonstrated in Chapter 6.
Obj2	Everything running in the pipeline is containerized workloads, and it is seamlessly running in Civo Cloud – managed Kubernetes environment, so the pipeline is cloud-native.
Obj3	All components are open-source.

Table 7.2: MLOps pipeline objectives that correlate with development performance

Obj4	The whole pipeline can be installed in any Kubernetes cluster with creating a GitHub repository with flux bootstrap command and a simple configuration YAML Listing 7.1. Objectives for model serving with the pipeline is writing deployment YAML declarations Listing 6.2 of running containerized workloads, and for training workflow declarations, which are relatively simple and client-friendly. The easiness of this is yet to be quantified, but we think it should be easy enough for a developer or a data scientist.
Obj5	The pipeline manages to handle a use-case as demonstrated in Chapter 6, more complex and business-critical experiments needs to be quantified.
Obj6	The pipeline is fully open-source and as such customizable. There is customization lined up for future work.

Table 7.3: MLOps pipeline Additional considered design objectives

Obj7	We considered several factors: a CNCF Technical Oversight Committee (TOC), the financial company backing, age of the product, number of contributors, rate of commits over lifetime and downloads. The TOC evaluates maturity of cloud-native products. The levels of maturity are "sandbox", "incubating", and "graduated", where graduated signifies the most mature level [19]. A graduated product should be safe to adopt on the highest enterprise levels. Most of the tools used are very mature in the CNCF landscape, but some are new or new versions of previously successful tools. For example, we use **Prometheus** and **Kubernetes** are considered graduated CNCF products. Incubating stage refers to products that have achieved requirements on commiters, versioning, documentation and have a "document that it is being used successfully in production by at least three independent end users which, in the TOC's judgement, are of adequate quality and scope" [19]. Overall the tools used are mature and production ready solutions. See Table A.3 for tool-by-tool evaluation.
Obj8	All tools used are heavily integrated with Kubernetes, often providing custom resource definitions and controllers, health check endpoints and metrics endpoints for Prometheus usable data.
Obj9	Something we have not yet quantified. However, the demonstration Chapter 6 shows that it should be relatively simple to install and add your model deployments and workflows in the pipeline, using only general level languages, such as YAML configuration. There is little overhead making business oriented Python scripts or services compatible with the pipeline.

Listing 7.1: example of installing MLOps pipeline in cluster

```
 GitOps bootstrap = $ flux bootstrap github --owner=sasumaki --repository=asd --path=cluster

 cluster/cluster-infrastructure.yaml:
 ---
apiVersion: source.toolkit.fluxcd.io/v1beta1
kind: GitRepository
metadata:
  name: cluster-system
  namespace: flux
spec:
  interval: 30m0s
  ref:
    branch: main
  url: https://github.com/sasumaki/aiga-pipe
---
apiVersion: kustomize.toolkit.fluxcd.io/v1beta1
kind: Kustomization
metadata:
  name: cluster-system
  namespace: flux
spec:
  interval: 30m0s
  path: ./cluster
  prune: true
  sourceRef:
    kind: GitRepository
    name: cluster-system
  validation: client

```

7.2 Limitations

The design's main limitations are running in a Kubernetes cluster and the need for building containerized workloads. Containerization requires knowledge and effort out of the developers using the pipeline. The pipeline itself is complex and requires in-depth knowledge of Kubernetes and its tooling to manage and develop. In production, this could mean a full-time employee who is in charge of managing the pipeline. The pipeline aims to save the time of data scientists and developers working with business issues in a mature – production-ready and scaling teams. It is not recommended to incorporate such an environment in the early stages of machine learning services or business.

The pipeline solution does not consider data or model version control but relies on third-party data storage providers version control, such as Amazon S3. We do not think it is necessary for the pipeline to take care of data version control. The pipeline is given addresses of data and models in its configuration. Any processes modifying them generate new versions of the data they pass onwards to other processes or save them into the cloud with a new address.

There are differences between Kubernetes distributions. Because of this, we cannot promise a seamless transition between various cloud platforms. The differences and required changes are usually small; for example, there were no changes needed between local K3d and Civo cloud. For example, in Google's managed Kubernetes service GKE [59], you may need to grant your account the ability to create new cluster roles for Argo Workflows to install correctly. These Kubernetes distribution relevant issues would need to be manually handled in the pipeline configuration and infrastructure, which should be possible for at least the major Kubernetes cloud providers.

7.3 Related work

There was related work being released during the time working on this thesis, answering some of the research questions and problems. K3Ai [53] is an "infrastructure in a box" – a command-line tool that offers integrated installation many of the same tools in a Kubernetes cluster. They aim to solve "combining a multitude of tools and frameworks that help Data Scientist and Data Engineers to solve the problem of building end-to-end pipelines" and state as their requirements as:

- Must be **FAST** to be built and EASY to be destroyed.
- Must be **AVAILABLE** everywhere no matter if it's on-prem, on-cloud, or in the remote universe.
- must be **REPRODUCIBLE** you want to be able to replicate the scenario again and again without having every time to re-configure things from scratch.

These goals and requirements match many of our own. The same components of K3Ai as in our pipeline are Argo and Triton Inference Server. As for differences, it offers Kubeflow and MLflow, which are commonly used model training and exploration tools we specifically wanted to avoid because they relied on DSL libraries.

Also released during our work is "**ZenML** [37], an extensible MLOps framework to create reproducible pipelines". ZenML lists as key features:

- Guaranteed reproducibility of your training experiments. Your pipelines are versioned from data to model, experiments automatically tracked and all pipeline configs are declarative by default.
- Guaranteed comparability between experiments.
- Ability to quickly switch between local and cloud environments (e.g. Kubernetes, Apache Beam).
- Built-in and extensible abstractions for all MLOps needs – from distributed processing on large datasets to Cloud-integrations and model serving backends
- Pre-built helpers to compare and visualize input parameters as well as pipeline results (e.g. Tensorboard, TFMA, TFDV).
- Cached pipeline states for faster experiment iterations.

The ZenML feature set is promising, and it offers most of our requirements and objectives. It even offers to switch outside the Kubernetes environment. It mostly leverages different cloud platforms proprietary solutions for training like AWS Sagemaker [2], Google AI Platform [1], and Azure machine learning [10], which is something our solution wants to avoid. ZenML is also configured with a custom DSL library which is something we wanted to avoid.

8 Conclusions

We reviewed the requirements of an modern machine learning pipeline that delivers automation and reproducibility in most steps of the machine learning process. We designed an open-source cloud-native MLOps pipeline, that should fit into most machine learning projects and teams that are aiming to automize and scale their machine learning process. The pipeline can be run on multiple cloud providers Kubernetes environments as well as on-premises Kubernetes and simulated Kubernetes on local machines. We evaluated the pipeline based on multiple features we recognized as must-have requirements, features that correlate with positive development performance, and features that solve problem definition specific issues.

As future work we would want to create a custom operator to orchestrate retraining lifecycles, as of now the solution is a simple webhook triggering. The operator could take into consideration, for example not to re-trigger the training process if one is already in progress or otherwise have more complex scenarios when to trigger. The pipeline should be made more lightweight. Currently a user would need a high-end computer to run the pipeline locally. If the pipeline were more lightweight it would be easier to develop and experiment on local machines. Also cluster costs increases as better hardware is required. Istio could be changed to another lighter service mesh implementation, but is not because of other tooling has integration with it. Knative-serving installs many controllers and resources that should not be needed, these should be trimmed down and only use what is required. We should consider the cybersecurity of this pipeline. We would want to conduct a case study to quantify the results, performance and feasibility of our MLOps pipeline. Implementing a real, complex, and resource heavy machine learning system into our pipeline would be needed. Implementing a managed phase for data labeling could be useful. User interface forms for adding deployments, workflows, and triggers could be an interesting addition to make the pipeline easier to use. Also a UI for training experiment tracking would be great addition.

We believe that with further validation and development discussed in future work, this solution can be recommended as a selected MLOps pipeline on most production scale machine learning projects.

Bibliography

[1] *AI Platform | Google Cloud*. URL: https://cloud.google.com/ai-platform (Accessed: 4 February 2021).

[2] *Amazon SageMaker – Machine Learning – Amazon Web Services*. URL: https://aws.amazon.com/sagemaker/ (Accessed: 4 February 2021).

[3] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann. "Software Engineering for Machine Learning: A Case Study". In: (March 2019). URL: https://www.microsoft.com/en-us/research/publication/software-engineering-for-machine-learning-a-case-study/ (Accessed: 9 February 2021).

[4] *Application deployment and testing strategies | Solutions*. URL: https://cloud.google.com/solutions/application-deployment-and-testing-strategies (Accessed: 23 December 2020).

[5] *argoproj/argo*. original-date: 2017-08-21T18:50:44Z. December 2020. URL: https://github.com/argoproj/argo (Accessed: 22 December 2020).

[6] *argoproj/argo-events*. original-date: 2018-05-17T15:06:49Z. December 2020. URL: https://github.com/argoproj/argo-events (Accessed: 22 December 2020).

[7] M. Artac, T. Borovssak, E. D. Nitto, M. Guerriero, and D. A. Tamburri. "DevOps: Introducing Infrastructure-as-Code". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. May 2017, pp. 497–498. DOI: 10.1109/ICSE-C.2017.162.

[8] G. Avelino, L. Passos, A. Hora, and M. T. Valente. "A Novel Approach for Estimating Truck Factors". In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)* (May 2016). arXiv: 1604.06766 version: 1, pp. 1–10. DOI: 10.1109/ICPC.2016.7503718.

[9] *AWS CloudFormation - Infrastructure as Code & AWS Resource Provisioning*. URL: https://aws.amazon.com/cloudformation/ (Accessed: 15 February 2021).

[10] *Azure Machine Learning - ML as a Service | Microsoft Azure*. URL: https://azure.microsoft.com/en-us/services/machine-learning/ (Accessed: 4 February 2021).

[11] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[12] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. "Manifesto for agile software development". In: *Available at https://agilemanifesto.org/ accessed Dec 21, 2020* (2001).

[13] B. W. Boehm, J. R. Brown, and M. Lipow. "QUANTITATIVE EVALUATION OF SOFTWARE QUALITY". In: (), p. 14.

[14] B. W. Boehm. "Verifying and Validating Software Requirements and Design Specifications". In: *IEEE Software* 1.1 (February 1984), pp. 75–88. ISSN: 07407459. DOI: http://dx.doi.org/10.1109/MS.1984.233702. URL: https://search.proquest.com/docview/215842250/abstract/5ED70725FEB84F17PQ/1 (Accessed: 20 January 2021).

[15] E. Casalicchio and S. Iannucci. "The state of the art in container technologies: Application, orchestration and security". In: *Concurrency and Computation: Practice and Experience* 32.17 (2020), e5668. ISSN: 1532-0634. DOI: https://doi.org/10.1002/cpe.5668. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5668 (Accessed: 5 February 2021).

[16] *Cloud Object Storage | Store & Retrieve Data Anywhere | Amazon Simple Storage Service (S3)*. URL: https://aws.amazon.com/s3/ (Accessed: 3 February 2021).

[17] *cloudevents/spec*. original-date: 2017-12-09T21:18:13Z. January 2021. URL: https://github.com/cloudevents/spec (Accessed: 21 January 2021).

[18] *CNCF Kubernetes Project Journey Report*. URL: https://www.cncf.io/cncf-kubernetes-project-journey/ (Accessed: 18 November 2020).

[19] *cncf/toc*. URL: https://github.com/cncf/toc (Accessed: 4 February 2021).

[20] *Continuous Integration and Delivery*. URL: https://circleci.com/ (Accessed: 15 February 2021).

[21] *cortexproject/cortex*. original-date: 2016-09-09T11:23:12Z. February 2021. URL: https://github.com/cortexproject/cortex (Accessed: 18 February 2021).

[22] D. Crockford <douglas@crockford.com>. *The application/json Media Type for JavaScript Object Notation (JSON)*. URL: https://tools.ietf.org/html/rfc4627 (Accessed: 7 January 2021).

[23] N. Delgado, A. Q. Gates, and S. Roach. “A taxonomy and catalog of runtime software-fault monitoring tools”. In: *IEEE Transactions on Software Engineering* 30.12 (December 2004), pp. 859–872. ISSN: 1939-3520. DOI: 10.1109/TSE.2004.91.

[24] *Docker overview.* November 2020. URL: https://docs.docker.com/get-started/overview/ (Accessed: 18 November 2020).

[25] *docker-library/golang.* URL: https://github.com/docker-library/golang (Accessed: 18 November 2020).

[26] *Exporterhub.* URL: https://exporterhub.io/ (Accessed: 18 February 2021).

[27] *Features • GitHub Actions.* URL: https://github.com/features/actions (Accessed: 3 February 2021).

[28] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner. “Detecting Adversarial Samples from Artifacts”. In: *arXiv:1703.00410 [cs, stat]* (November 2017). arXiv: 1703.00410. URL: http://arxiv.org/abs/1703.00410 (Accessed: 7 January 2021).

[29] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. “Development and Deployment at Facebook”. In: *IEEE Internet Computing* 17.4 (2013), pp. 8–17. DOI: 10.1109/MIC.2013.25.

[30] *Fleet Management for Kubernetes is Here.* 0. URL: https://rancher.com/blog/2020/fleet-management-kubernetes/ (Accessed: 10 February 2021).

[31] *fluxcd/flux2.* original-date: 2020-04-24T09:38:21Z. December 2020. URL: https://github.com/fluxcd/flux2 (Accessed: 8 December 2020).

[32] N. Forsgren. *Accelerate : The Science of Lean Software and Devops: Building and Scaling High Performing Technology Organizations.* It Revolution Press. ISBN: 1-942788-33-9. URL: https://www.bookdepository.com/Accelerate/9781942788331 (Accessed: 18 November 2020).

[33] N. Forsgren and J. Humble. *The Role of Continuous Delivery in IT and Organizational Performance.* SSRN Scholarly Paper ID 2681909. Rochester, NY: Social Science Research Network, October 2015. DOI: 10.2139/ssrn.2681909.

[34] N. Forsgren, D. Smith, J. Humble, and J. Frazelle. *2019 Accelerate State of DevOps Report.* Tech. rep. 2019. URL: http://cloud.google.com/devops/state-of-devops/.

[35] *git/git*. original-date: 2008-07-23T14:21:26Z. December 2020. URL: https://github.com/git/git (Accessed: 2 December 2020).

[36] *GitOps*. URL: https://www.gitops.tech/ (Accessed: 2 December 2020).

[37] m. GmbH. *ZenML - Reproducible Open-Source MLOps*. URL: https://zenml.io/ (Accessed: 4 February 2021).

[38] *GPUs vs CPUs for deployment of deep learning models*. URL: https://azure.microsoft.com/en-us/blog/gpus-vs-cpus-for-deployment-of-deep-learning-models/ (Accessed: 27 November 2020).

[39] *Gradle Build Tool*. URL: https://gradle.org/ (Accessed: 15 February 2021).

[40] *Grafana Plugins - extend and customize your Grafana*. URL: https://grafana.com/grafana/plugins (Accessed: 18 February 2021).

[41] *grafana/grafana*. original-date: 2013-12-11T15:59:56Z. January 2021. URL: https://github.com/grafana/grafana (Accessed: 7 January 2021).

[42] D. Graziotin and F. Fagerholm. “Happiness and the Productivity of Software Engineers”. In: *Rethinking Productivity in Software Engineering*. Ed. by C. Sadowski and T. Zimmermann. Berkeley, CA: Apress, 2019, pp. 109–124. ISBN: 978-1-4842-4221-6. DOI: 10.1007/978-1-4842-4221-6_10.

[43] D. Graziotin, F. Fagerholm, X. Wang, and P. Abrahamsson. “On the Unhappiness of Software Developers”. In: *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. EASE’17. New York, NY, USA: Association for Computing Machinery, June 2017, pp. 324–333. ISBN: 978-1-4503-4804-1. DOI: 10.1145/3084226.3084242.

[44] Y. Guo. *Uber ATG’s Machine Learning Infrastructure*. March 2020. URL: https://eng.uber.com/machine-learning-model-life-cycle-version-control/ (Accessed: 8 February 2021).

[45] A. Hat Red. *Ansible is Simple IT Automation*. URL: https://www.ansible.com (Accessed: 15 February 2021).

[46] V. Hodge and J. Austin. “A Survey of Outlier Detection Methodologies”. In: *Artificial Intelligence Review* 22.2 (October 2004), pp. 85–126. ISSN: 1573-7462. DOI: 10.1023/B:AIRE.0000045502.10941.a9.

[47] M. Hüttermann. “Infrastructure as Code”. In: *DevOps for Developers*. Ed. by M. Hüttermann. Berkeley, CA: Apress, 2012, pp. 135–156. ISBN: 978-1-4302-4570-4. DOI: 10.1007/978-1-4302-4570-4_9.

[48] *Istio*. URL: /latest/ (Accessed: 21 January 2021).

[49] R. Jabbari, N. Ali, K. Petersen, and B. Tanveer. “What is DevOps?: A Systematic Mapping Study on Definitions and Practices”. In: May 2016, pp. 1–11. DOI: 10.1145/2962695.2962707.

[50] *JavaScript End to End Testing Framework*. URL: https://www.cypress.io/ (Accessed: 15 February 2021).

[51] *Jenkins*. URL: https://www.jenkins.io/ (Accessed: 15 February 2021).

[52] Jez Humble and David Farley. *Continuous Delivery : Jez Humble : 9780321601919*. Pearson Education (US). ISBN: 0-321-60191-2.

[53] *K3ai (ke3ai)*. URL: https://docs.k3ai.in/ (Accessed: 4 February 2021).

[54] *Keras: the Python deep learning API*. URL: https://keras.io/ (Accessed: 9 February 2021).

[55] *Knative Eventing*. URL: https://knative.dev/docs/eventing/ (Accessed: 21 January 2021).

[56] J. Kousa, P. Ihantola, A. Hellas, and M. Luukkainen. “Teaching Container-Based DevOps Practices”. In: *Web Engineering*. Ed. by M. Bielikova, T. Mikkonen, and C. Pautasso. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 494–502. ISBN: 978-3-030-50578-3. DOI: 10.1007/978-3-030-50578-3_34.

[57] *Kubeflow*. URL: https://www.kubeflow.org/ (Accessed: 11 February 2021).

[58] *kubeflow/kfserving*. URL: https://github.com/kubeflow/kfserving (Accessed: 19 January 2021).

[59] *Kubernetes - Google Kubernetes Engine (GKE)*. URL: https://cloud.google.com/kubernetes-engine (Accessed: 3 February 2021).

[60] P. J. Leach, T. Berners-Lee, J. C. Mogul, L. Masinter, R. T. Fielding, and J. Gettys. *Hypertext Transfer Protocol – HTTP/1.1*. URL: https://tools.ietf.org/html/rfc2616 (Accessed: 23 December 2020).

[61] Y. LeCun and C. Cortes. “MNIST handwritten digit database”. In: (2010). URL: http://yann.lecun.com/exdb/mnist/ (Accessed: 14 January 2016).

[62] T. A. Limoncelli. “GitOps: a path to more self-service IT”. In: *Commun. ACM* 61.9 (2018), pp. 38–42. DOI: 10.1145/3233241.

[63] L. E. Lwakatare, I. Crnkovic, E. Rånge, and J. Bosch. “From a Data Science Driven Process to a Continuous Delivery Process for Machine Learning Systems”. In: *Product-Focused Software Process Improvement*. Ed. by M. Morisio, M. Torchiano, and A. Jedlitschka. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 185–201. ISBN: 978-3-030-64148-1. DOI: 10.1007/978-3-030-64148-1_12.

[64] L. E. Lwakatare, A. Raj, I. Crnkovic, J. Bosch, and H. Olsson. “Large-Scale Machine Learning Systems in Real-World Industrial Settings A Review of Challenges and Solutions”. In: *Information and Software Technology* 127 (July 2020), p. 106368. DOI: 10.1016/j.infsof.2020.106368.

[65] S. Mäkinen, H. Skogström, E. Laaksonen, and T. Mikkonen. “Who Needs MLOps: What Data Scientists Seek to Accomplish and How Can MLOps Help?” In: May 2021.

[66] *Managed Kubernetes service, powered by K3s*. URL: https://www.civo.com (Accessed: 3 February 2021).

[67] *Microservices*. URL: https://martinfowler.com/articles/microservices.html (Accessed: 5 February 2021).

[68] *MLflow Documentation — MLflow 1.13.1 documentation*. URL: https://www.mlflow.org/docs/latest/index.html (Accessed: 19 January 2021).

[69] *MLOps: Continuous delivery and automation pipelines in machine learning*. URL: https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning (Accessed: 19 November 2020).

[70] *Mockito framework site*. URL: https://site.mockito.org/ (Accessed: 15 February 2021).

[71] K. Morris. *Infrastructure as Code*. Google-Books-ID: UW4NEAAAQBAJ. "O’Reilly Media, Inc.", December 2020. ISBN: 978-1-09-811464-0.

[72] *nats-io/stan.go*. original-date: 2016-01-20T15:49:02Z. January 2021. URL: https://github.com/nats-io/stan.go (Accessed: 7 January 2021).

[73] *New Relic | Deliver more perfect software*. URL: https://newrelic.com/ (Accessed: 15 February 2021).

[74] *NVIDIA Triton Inference Server*. March 2020. URL: https://developer.nvidia.com/nvidia-triton-inference-server (Accessed: 20 January 2021).

[75] *Open Container Initiative - Open Container Initiative*. URL: https://opencontainers.org/ (Accessed: 7 December 2020).

[76] *opencontainers/image-spec*. original-date: 2016-03-22T19:34:41Z. December 2020. URL: https://github.com/opencontainers/image-spec (Accessed: 7 December 2020).

[77] *opencontainers/runtime-spec*. original-date: 2015-06-05T23:30:10Z. December 2020. URL: https://github.com/opencontainers/runtime-spec (Accessed: 7 December 2020).

[78] *Overview of Kubeflow Pipelines*. URL: /docs/pipelines/overview/pipelines-overview/ (Accessed: 17 February 2021).

[79] paul-hammant. *Trunk Based Development*. URL: https://trunkbaseddevelopment.com/ (Accessed: 20 November 2020).

[80] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. "A Design Science Research Methodology for Information Systems Research". In: *Journal of Management Information Systems* 24.3 (December 2007), pp. 45–77. ISSN: 0742-1222. DOI: 10.2753/MIS0742-1222240302.

[81] *Prediction APIs — seldon-core documentation*. URL: https://docs.seldon.io/projects/seldon-core/en/latest/reference/apis/index.html (Accessed: 19 January 2021).

[82] T. F. project. *Overview - Flux | GitOps Toolkit*. URL: https://toolkit.fluxcd.io/components/ (Accessed: 18 February 2021).

[83] *Prometheus - Monitoring system & time series database*. URL: https://prometheus.io/ (Accessed: 7 January 2021).

[84] *prometheus/prometheus*. original-date: 2012-11-24T11:14:12Z. January 2021. URL: https://github.com/prometheus/prometheus (Accessed: 7 January 2021).

[85] *pytest: helps you write better programs — pytest documentation*. URL: https://docs.pytest.org/en/stable/ (Accessed: 15 February 2021).

[86] *rancher/k3d*. original-date: 2019-04-02T11:30:11Z. February 2021. URL: https://github.com/rancher/k3d (Accessed: 3 February 2021).

[87] 2. M. Read. *Extensibility*. URL: /latest/docs/concepts/wasm/ (Accessed: 18 February 2021).

[88] D. Sato, A. Wilder, and C. Windheuser. *Continuous Delivery for Machine Learning*. September 2019. URL: https://martinfowler.com/articles/cd4ml.html (Accessed: 21 December 2020).

[89] S. Schelter, J.-H. Böse, J. Kirschnick, T. Klein, and S. Seufert. “Automatically Tracking Metadata and Provenance of Machine Learning Experiments”. In: p. 8.

[90] *scikit-learn/scikit-learn*. original-date: 2010-08-17T09:43:38Z. January 2021. URL: https://github.com/scikit-learn/scikit-learn (Accessed: 19 January 2021).

[91] 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*. 2015, pp. 2503–2511.

[92] *Seldon Core - Open Source Machine Learning Deployment for Kubernetes*. URL: https://www.seldon.io/tech/products/core/ (Accessed: 8 December 2020).

[93] *SeldonIO/MLServer*. original-date: 2020-06-16T18:01:37Z. January 2021. URL: https://github.com/SeldonIO/MLServer (Accessed: 19 January 2021).

[94] *Sentry | Application Monitoring and Error Tracking Software*. URL: https://sentry.io/welcome/?utm_source=google&utm_medium=cpc&utm_campaign=9575834316&content=423899439833&utm_term=sentry&gclid=Cj0KCQiA1KiBBhCcARIsAPWqoSpe wcB (Accessed: 15 February 2021).

[95] *Serving Models | TFX*. URL: https://www.tensorflow.org/tfx/guide/serving (Accessed: 19 January 2021).

[96] *SKLearn Server — seldon-core documentation*. URL: https://docs.seldon.io/projects/seldon-core/en/stable/servers/sklearn.html (Accessed: 19 January 2021).

[97] J. Smeds, K. Nybom, and I. Porres. “DevOps: A Definition and Perceived Adoption Impediments”. In: *Agile Processes in Software Engineering and Extreme Programming*. Ed. by C. Lassenius, T. Dingsøyr, and M. Paasivaara. Cham: Springer International Publishing, 2015, pp. 166–177. ISBN: 978-3-319-18612-2.

[98] *Spinnaker*. URL: https://www.spinnaker.io/ (Accessed: 15 February 2021).

[99] *TensorFlow*. URL: https://www.tensorflow.org/ (Accessed: 9 February 2021).

[100] *tensorflow/serving*. URL: https://github.com/tensorflow/serving (Accessed: 19 January 2021).

[101] *Terraform by HashiCorp*. URL: https://www.terraform.io/ (Accessed: 15 February 2021).

[102] *Thanos*. URL: https://thanos.io/ (Accessed: 18 February 2021).

[103] *The Linux Foundation – Supporting Open Source Ecosystems*. URL: https://www.linuxfoundation.org/ (Accessed: 7 December 2020).

[104] *Travis CI - Test and Deploy Your Code with Confidence*. URL: https://travis-ci.org/ (Accessed: 15 February 2021).

[105] *triton-inference-server/server*. original-date: 2018-10-04T21:10:30Z. January 2021. URL: https://github.com/triton-inference-server/server (Accessed: 19 January 2021).

[106] Valohai. *Practical MLOps: How to Get Ready for Production Models*. URL: https://valohai.com/mlops-ebook/ (Accessed: 19 November 2020).

[107] *Valohai MLOps Platform*. URL: https://valohai.com/product/ (Accessed: 11 February 2021).

[108] P. Vassiliadis. *A Survey of Extract–Transform–Load Technology*. article. ISSN: 1548-3924 Issue: 3 Journal Abbreviation: IJDWM Pages: 1-27 Publisher: IGI Global Volume: 5. July 2009. DOI: 10.4018/jdwm.2009070101. URL: www.igi-global.com/article/survey-extract-transform-load-technology/3894 (Accessed: 19 November 2020).

[109] M. Virmani. “Understanding DevOps bridging the gap from continuous integration to continuous delivery”. In: *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*. May 2015, pp. 78–82. DOI: 10.1109/INTECH.2015.7173368.

[110] G. I. Webb, R. Hyde, H. Cao, H. L. Nguyen, and F. Petitjean. “Characterizing concept drift”. In: *Data Mining and Knowledge Discovery* 30.4 (July 2016), pp. 964–994. ISSN: 1573-756X. DOI: 10.1007/s10618-015-0448-4. URL: https://doi.org/10.1007/s10618-015-0448-4 (Accessed: 7 January 2021).

[111] *Welcome to Python.org*. URL: https://www.python.org/ (Accessed: 7 January 2021).

[112] *What Is MLOps? | SpringerLink*. URL: https://link.springer.com/chapter/10.1007/978-1-4842-6549-9_3 (Accessed: 13 January 2021).

[113] *Workflows & Pipelines | Argo*. URL: https://argoproj.github.io/projects/argo/ (Accessed: 8 December 2020).

[114] *Xcode*. URL: https://developer.apple.com/xcode/ (Accessed: 15 February 2021).

[115] *XGBoost Server — seldon-core documentation*. URL: https://docs.seldon.io/projects/seldon-core/en/latest/servers/xgboost.html (Accessed: 19 January 2021).

[116] Y. Yu. “Os-level virtualization and its applications”. PhD Thesis. The Graduate School, Stony Brook University: Stony Brook, NY., 2007.

Appendix A Tables of tools requirements

Table A.1: Tool specific must-have objectives.

Tool	**Obj1**: Must provide features of the MLOps methodologies CI/CD pipelines	**Obj2**: Must be cloud-native	**Obj3**: Must be open-source.
Argo Workflows	Used for orchestrating order of tasks in the ETL and Training pipelines.	Designed to run in Kubernetes and leverage multiple parallel machines and jobs.	Open-source
Seldon Core	Handles fetching created models, creating an inference server and API's for inference and metrics.	Designed to be run in Kubernetes. Enables complex deployment strategies, e.g. A/B out of the box.	Open-source
Knative Eventing	Used as a tool to specify messaging routes between different services, e.g. post specific model's inference requests and responses to specific set of monitoring services.	Designed for messaging between Kubernetes services.	Open-source
NATS Streaming	Acts as a back-channel for Knative Eventing messages.	Designed for cloud-native applications and microservices. A CNCF Project.	Open-source
Prometheus	A real-time metrics and alerting system.	A CNCF Graduated project.	Open-source
Grafana	Analytics platform for Prometheus metrics	Designed for cloud-native applications, can import data from mixed data sources	Open-source
Istio	Offers easier networking and service discovery for all the models and monitoring services, enables a lot of Seldon-core features.	Built for Kubernetes.	Open-source
Flux v2	Offers Continuous Delivery and declarative Kubernetes cluster configurations following the GitOps methodology.	Designed for Kubernetes.	Open-source
MinIO	Works as a temporary in-cluster storage for running workloads artefacts. Can be configured to be a persistant main storage.	Comes with container images and custom Kubernetes Operator	Open-source
SealedSecrets	Handles secret management in our Continuous Delivery scheme.	Designed for Kubernetes	Open-source

Table A.2: Tool specific objectives that correlate with development performance.

Tool	**Obj4**: Ease-of-use	**Obj5**: Effectiveness in its job	**Obj6**: Customizability
Argo Workflows	Writing workflow manifests is rather straightforward (Listing 5.2). All actual code are usually just images to run and other things to consider are sequence of running them, input/output passing and giving the workflow secrets. Passing hyperparameters to training jobs is easy when using the model wrapping method discussed in Chapter 5.	Is able to configure complex workflows, with great interface to parameter and artefact passing and storing.	As it is open-source it can be customized. There are a lot of solutions that use argo as a workflow engine on the background, e.g. Kubeflow pipelines [78].
Seldon Core	Creating a seldon core deployment is very similar to a basic Kubernetes API deployment, with the added complexity of configuring a graph for your inference, and possible A/B or shadow deployment schemes. So if data needs to go through multiple models or various pre-processing steps, it adds complexity. Otherwise it is straightforward (Listing 6.2).	Seldon-core delivers a lot of great features, complex deployment strategies, model fetch and initialization, inference servers, monitoring, and health endpoints, and outputs CloudEvents.	The seldon-core is being extended to explainable AI projects and integrated with various services, e.g. Kubeflow. As such it seems like a very extendable project.
Knative Eventing	Configuring messages from models to monitoring system is easy as long as you know the attributes of the Seldon-Deployment messages to filter with.	Knative Eventing provides a way to configure messages between services on the configuration level without modifying the code of individual services	It is open-source but could not find out if it is being customized or extended elsewhere.
NATS Streaming	NATS Streaming is not something a user of this pipeline sees or has to worry about.	It provides secure, scalable, and lightweight messaging backend.	It is open-source
Prometheus	Prometheus has its own structure for metrics and a query language that needs to be learned.	It is a great tool for real-time metrics and has interoperability with vast amount of systems.	It is open-source, highly customizable and modular. There's dozens of solutions extending Prometheus, for example, in fields of deploying, availability, analytics, dashboards, and exporters [26, 102, 21].
Grafana	Using with Prometheus requires developers to know Prometheus query language, as well as Grafana dashboard JSON-configuration. No custom coding of graphs or web services necessary.	It provides analytics and visualizations of the metrics exported to Prometheus, in a easy way.	There is a library of plugins created to customize and extend Grafana [40].

Istio	Configuring routing, TLS connections, domain names, and larger networks of services can be extremely difficult.	It is used mostly because of high integration with other tooling. For example, Istio is the driving technology behind eventing configuration and complex deployment strategies.	There is a plugin system [87], and various tools are integrating with Istio capabilities e.g. Seldon-core.
Flux v2	Mostly automatic system, that does not concern developers. If something goes wrong with deployments, it might need manual debugging.	It works great for automatic deploying of microservices in the cluster, following the GitOps methodology.	The tool is new and there's is no popular tools extending on this tool. The tool itself is using **GitOps Toolkit** as its runtime, which is designed to be extendable for other tooling [82].
MinIO	Unseen for developers by default. Can be configured to be the main data storage. It has an API that implements the AWS S3 API, so changing between the two does not require any changes to code.	It works greatly in passing artefacts between workloads in Argo. It also works as a substitute for data storages, e.g. S3. Using Minio as the main storage requires operators to have disk space and node storage management in the Kubernetes cluster, which can introduce a lot of work and complexity.	Minio is being integrated in some cloud projects. No known customized or extended projects.
SealedSecrets	Objuires fetching public key from the SealedSecrets operator upon creating a new cluster. In the example demonstration we have a easy-to-use bash script to handle heavy-lifting of using SealedSecrets.	Provides a way to store secrets in a public repository, without anyone having private key to decrypt them.	It is open-source but no known examples of extension or customization.

Table A.3: Tool specific additional considered design objectives.

Tool	**Obj7**: Maturity of the tool	**Obj8**: Level of integration with runtime	**Obj9**: Generality
Argo Workflows	CNCF Incubating project.	Designed to run containerized workloads in Kubernetes. Provides Prometheus metrics endpoint.	All workflows are configured in YAML configuration files. Everyone is able to configure even complex workflows.
Seldon Core	**Seldon-core** is not featured in the CNCF space, but its several years of development, over hundred contributors, over fifty releases, rich feature-set and over 2 million downloads suggests that it is mature for our MLOps pipeline.	Designed for Kubernetes deployments. Provides Prometheus metrics endpoints, API health and readiness endpoints. Sends CloudEvents by default and integrates with Istio networking for complex deployment schemes and API routing.	Developers don't need to concern themselves with anything else than writing a SeldonDeployment YAML manifest.

Knative Eventing	Considered to be in the CNCF sandbox stage but is backed and used by Google, which gives it credibility on our evaluation.	Designed for Kubernetes messaging.	YAML configurations for attaching services. Monitoring systems has to be programmed to read and send CloudEvents, which requires learning a SDK.
NATS Streaming	CNCF Incubating project.	It is running in containers and designed for cloud-native microservice architectures.	Developers and Data Scientist does not interact or see it.
Prometheus	CNCF Graduated project.	De-facto monitoring solution in modern Kubernetes projects. Almost every tool in this pipeline exports its own Prometheus metrics.	Creating new metric queries or exporting own custom metrics will require learning of syntax.
Grafana	CNCF Sandbox project. Started 8 Years ago, with hundreds of commits per month and over thousand contributors. Over 75 million dollar funding according to CNCF. Used commonly together with Prometheus.	Heavy integration with Prometheus.	Building new graphs requires added knowledge of Prometheus syntax and Grafana JSON configuration.
Istio	Considered to be in the CNCF sandbox stage but is backed and used by Google, which gives it credibility on our evaluation.	Designed for Kubernetes networking. Other tools like Seldon-core and Knative depends partly on Istio.	Configuring networking requires added knowledge in any production environment. Istio is not the easiest or the most general in anyway.
Flux v2	In CNCF sandbox stage, and we consider it the least mature tool in our pipeline, as it was released just weeks before we started working on this following success of Flux V1.	A great tool for GitOps continuous delivery in Kubernetes clusters.	Data Scientist needs to run a bootstrap command when starting a project to start the GitOps syncing – otherwise it does not concern developers.
MinIO	CNCF sandbox project. Started 6 years ago, dozens of commits per month and 297 contributors. 23.3 million dollar funding according to CNCF.	Runs as a container. No other special integration.	A general purpose data storage. Works like AWS S3.
SealedSecrets	Not a very mature project.	Works great for managing Kubernetes environment secrets in Git. A key feature to have with the GitOps methodology.	A Data Scientist needs to know to run a CLI program to crypt all secrets they introduce to the project.