FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Design and Implementation of Machine Learning Operations
Student:	Bc. Michal Bacigál
Supervisor:	Ing. Miroslav Čepek, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

Reproducibility is one of the critical issues of every machine learning project. The final model depends on various input factors, such as the acquisition and preprocessing of train/test data, proper setting of model hyperparameters and the overall configuration of the learning process, to name a few. Machine Learning Operations (or MLOps for short) aims to bring order into machine learning projects. Over the past few years, numerous open-source and closed-source MLOps frameworks were released. Each project has its strengths/advantages and weaknesses/disadvantages. In their master thesis, the student will provide an introduction to the field of MLOps, analyze the existing MLOps frameworks and design an example stack using a subset of the reviewed frameworks based on the requirements discussed with the supervisor.

The requirements for the thesis include the following:
1. Summarize the main principles, advantages and phases of MLOps and its life cycle. Discuss the state of MLOps globally and justify its growing importance for businesses.
2. Perform an overview/a survey of existing open-source machine learning operations frameworks, and summarise their main features, key advantages, and disadvantages for use in personal and commercial settings.
3. Summarize the most important requirements for the design of the MLOps stack. Describe the potential setup. Based on the discussion with the supervisor, suggest the appropriate tools, justify your choice over the available alternatives, and describe the final design of the stack.
4. Design and implement a way to use and interact with the stack and document its

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 15 February 2023 in Prague.

usage. Provide a deeper review of the components used and describe how they interact. If the stack depends on any supplementary frameworks not previously described (e.g. Docker, Kubernetes), provide their overview.

5. Demonstrate the usage of the stack on a practical example (e.g. a simple classification task on publicly available data). Describe the steps, such as data acquisition and versioning, hyperparameters setting, experiment tracking, model evaluation and deployment using CI/CD.

6. Discuss the possible ways the implementation will be expanded and maintained in the future and justify its suitability for the task.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 15 February 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

Design and Implementation of Machine Learning Operations

Bc. Michal Bacigál

Department of Applied Mathematics
Supervisor: Ing. Miroslav Čepek, Ph.D.

May 4, 2023

Acknowledgements

I hereby express my sincere gratitude to my advisor, Ing. Miroslav Čepek, Ph.D., for his invaluable guidance throughout the process of writing this thesis.

I would also like to thank my family and friends for their continuous support.

Declaration

In Prague on May 4, 2023 ....

Citation of this thesis

Bacigál, Michal. *Design and Implementation of Machine Learning Operations.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Rast popularity a významu aplikácie strojového učenia v rôznych odvetviach viedol k postupnému rozšíreniu princípov DevOps o koncepty súvisiace s dátami a modelmi, výsledkom čoho došlo k vzniku paradigmy známej ako MLOps. Táto diplomová práca skúma jej dôležitosť, hlavné princípy a fázy, ktoré s ňou súvisia. V práci poskytujeme prehľad MLOps nástrojov a ich hlavných funkcionalít, na základe ktorého vyberáme tie najvhodnejšie pre účely využitia v procese výuky. Výsledkom tejto analýzy je "proof of concept" riešenie, ktoré môže slúžiť ako základ pre ďalší výskum možností začlenenia operácii strojového učenia za účelmi zjednodušenia procesu vývoja modelov pre študentov a výskumníkov na našej univerzite.

Klíčová slova MLOps, strojové učenie, reprodukovateľnosť, umelá inteligencia, DevOps

Abstract

The growing popularity and importance of machine learning adoption across industries have led to a gradual enrichment of DevOps principles with data- and model-related concepts, forming a paradigm known as MLOps. This diploma thesis explores its importance and describes the main principles and phases involved. We perform a summary of MLOps tools and their features, which we use to select the appropriate tools for use in our academic setting, and design a proof of concept solution that can be used as a basis for further research and incorporation of machine learning operations to simplify the model development process for students and researchers at our university.

Keywords MLOps, machine learning, reproducibility, artificial intelligence, DevOps

Contents

List of Figures

List of Listings

List of Tables

Introduction

"MLOps is the natural progression of DevOps in the context of AI. While it leverages DevOps' focus on security, compliance, and management of IT resources, MLOps' real emphasis is on the consistent and smooth development of models and their scalability."
—Samir Tout, *Professor of Cybersecurity, Eastern Michigan University's SISAC [1]*

Motivation

The Journey of Machine Learning

Eighty years ago, amidst global war, a research paper written by American neurophysiologist Warren McCulloch and logician Walter Pitts introduced the first mathematical model designed to mimic the basic building block of the nervous system — a biological neuron [2]. In the decade that followed, Alan Turing challenged the world to a round of 'the imitation game' [3], Frank Rosenblatt designed the first artificial neural network (also known as the perceptron) [4], and Arthur Samuel, the coiner of the term machine learning [5], entertained the viewers of a morning news program with a computer program that managed to defeat a human player at a game of checkers [6].

It is safe to say that the journey of machine learning as a field of computer science has been all but uneventful (Figure 0.1). The initial excitement of the 1950s and early 1960s led to high expectations, which remained unfulfilled, resulting in a series of alternating periods of disappointment, scepticism and a decrease in research funding (also known as AI winters), followed by stretches of renewed hope [7] — and it was not until the early 2010s that it started to gain significant traction again, owing much of its success to the rapid evolution in computing power that paved the way for the expansion of deep learning [8].

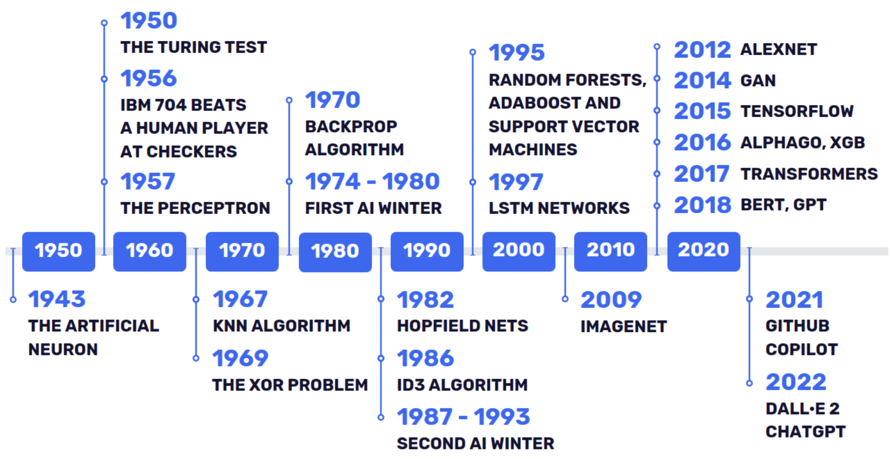

Figure 0.1: A timeline of a few selected milestones in machine learning history.

The Hidden Value of Machine Learning (Operations)

To say that machine learning, or artificial intelligence in general, is a rapidly evolving field is a vast understatement. Between 2017 and 2022, the volume of global investments quadrupled to more than $176 billion [9, p. 151], and the field is projected to contribute as much as $15.7 trillion to the global economy in 2030 [10, p. 3]. Furthermore, it has become apparent that machine learning is no longer just a means of entertainment but is becoming increasingly important in business operations, with the adoption rate doubling over the past five years, both in terms percentage of respondents and the number of utilized AI capabilities [11, p. 3]. The most prominent use case of AI remains to be service operations optimization, utilizing tools such as robotic automation, computer vision, NLP and virtual assistants [11, p. 4]. Regardless of the application, these surveys have shown that companies which were able to adopt AI in their processes have seen both a decrease in costs and an increase in revenue.

In 2022, Verta Insights — a research group at Verta.ai — surveyed over 200 stakeholders on the state of MLOps and its adoption with the aim of identifying the business processes that differentiate the industry leaders from their competition [12]. The findings indicate that companies which are actively using an MLOps platform and have designated a team to manage it have a higher tendency to launch new features and models into production successfully and meet their overall targets. Furthermore, leading organizations have a reportedly more systematic approach to model monitoring and tend to re-train their models less frequently than their worse-performing counterparts.

The Challenges of Machine Learning Operationalization

The process of operationalizing machine learning, however, remains to be a daunting task for a lot of businesses worldwide [13], many of which overestimate their preparedness for the incorporation of AI into their processes [14]. This often results in a substantial waste of valuable resources. The exact proportion of projects that fail to deliver varies significantly by source, ranging from 34 % [15] to 85 % [16]. According to some surveys, this may be attributed to the underutilization of proper operationalization practices, such as [12] which reports that:

- 55 % of companies do not have a team designated for helping models get into production and/or have not adopted the use of an MLOps platform, or that they are not aware of it,
- companies are struggling to establish or scale MLOps teams, finding it difficult to transform data scientists into data engineers,
- 50 % of companies only manage to successfully meet their targets ”sometimes” or less frequently,
- in 56 % of cases, less than half of the organization’s projects make it into production.

There have been attempts to identify the greatest challenges in machine learning operationalization [12][17][18], with the following being the most commonly mentioned ones:

- sub-optimal quantity and/or quality of available relevant data, and other data-related problems,
- difficulties proving the business value of ML projects,
- the lack of talented engineers who could help companies implement the required technologies,
- difficulties choosing and implementing the right MLOps tools and technologies.

Aim of the Thesis

The main objective of this diploma thesis comprises three main goals:

- to provide the reader with, or expand their knowledge of MLOps,
- to conduct a semi-comprehensive review of the current open-source MLOps tools and frameworks on the market and illustrate the advantages their adoption offers to the ML development process,
- to demonstrate the process of designing an MLOps stack for a specific use case, justify the rationale behind certain design decisions, and show the configuration and use of the stack in practice.

Thesis Structure

The thesis is partitioned into six primary chapters:

Chapter 1 introduces the paradigm and its historical and theoretical foundations, and discusses its main principles and implications for different stages of the machine learning project life cycle.

Chapter 2 surveys the various types of tools available in the market, highlights their key characteristics, and provides a detailed analysis of a selected subset of tools.

Chapter 3 clarifies the aim and scope of the implementation part, outlines the typical workflow of machine learning development in courses at our university, and specifies the functional and non-functional criteria for the stack.

Chapter 4 presents a comprehensive overview of the proposed architecture, its components, and their interactions.

Chapter 5 explains the tools chosen to implement the stack and details the implementation aspects of the stack and its components.

Chapter 6 reflects on the achievements of this work, assesses the fulfillment of its objectives, and offers recommendations for future work.

CHAPTER 1

Theoretical Foundations

In this chapter, we provide a brief introduction to the topic of MLOps. First, we explain what MLOps is, and identify the important milestones of its existence. Second, we explore its relationship to DevOps and summarize some of its main principles. Finally, we provide an overview of the phases involved in the machine learning development process, and highlight the ways the adoption of MLOps can help in their application.

1.1 Introduction to MLOps

The term machine learning operations, commonly abbreviated to MLOps, has been loosely defined and used by the machine learning community to describe the paradigm of deploying ML models into production in a rapid, continuous and organized manner [19]. In similar fashion to other technologies in their relative infancy, there is no consensus about the term's exact definition, however, it is often colloquially dubbed 'DevOps for ML' [19, 20], as it follows the same core principles as DevOps, while also incorporating the additional challenges resulting from the added complexity of working with convoluted, diverse and oftentimes constantly changing data and models [19, 21, 22].

The intricacies of machine learning operationalization were first described in a 2015 paper by *Sculley et al.* [23], which has been referred to as a milestone which put the trend of interest in MLOps in motion [21]. In their work, the authors argue that technical debt in ML tends to be more complex, and often harder to pinpoint than in software engineering due to its more systematic nature. Additionally, they identify and enumerate a subset of common issues and anti-patters and provide suggestions on how to eliminate them, or at the very least mitigate their negative effects. As time progressed, MLOps began to slowly gain traction and recognition by the industry, until a more gradual increase started in 2019, during which numerous tools and frameworks were initially released, and to this day continues to grow in popularity (Figure 1.1).

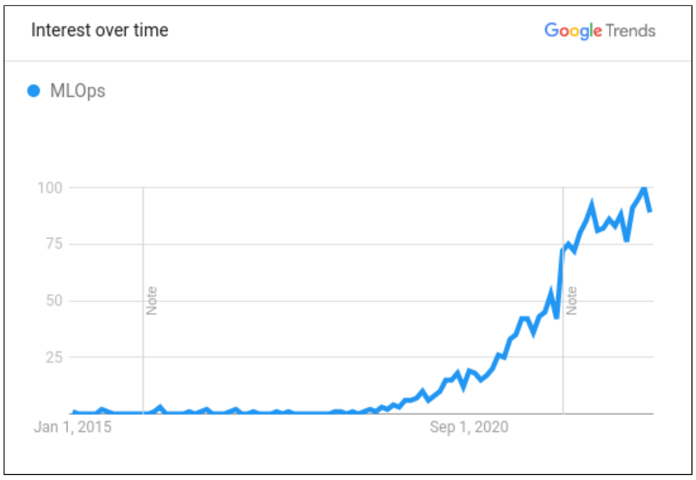

Figure 1.1: The growth of interest in MLOps over time (Google Trends).

To put emphasis on the continuity of the development process, MLOps diagrams are often pictured as circles or infinity symbols (Figure 1.2). The labels and stages vary by source, but can be generally grouped into four major areas: business, data, development and operations.

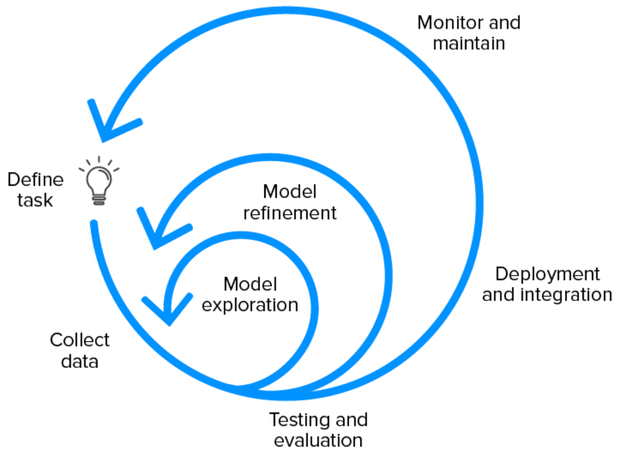

Figure 1.2: A simplified diagram of a machine learning project life cycle [24].

A knowledgeable reader may recognize that the latter two have been inherited from the DevOps paradigm, which formed in the late 2000s as a way of bridging the gap between the two aspects of software engineering [25]. In the pre-DevOps era, development and operations functioned separately, resulting in delayed deployment and misalignment between the two teams [26]. Today, DevOps has become one of the supporting pillars of a successful software engineering practice [19], and its principles have been adopted by its various derivatives, such as DevSecOps [27] and AIOps [28], among many others. These newly-formed paradigms are the consequence of its shortcomings in specific use cases and the natural progression in the industry. In the context of MLOps, the limitations have been attributed to the unpredictable nature of data and the complexity of models and interactions between them [23]. As a result, the basic principles of DevOps were enriched with the requirements of machine learning projects, paving the way for the increasing adoption of MLOps in practice.

There is no definitive guidebook for the successful application of MLOps, but there have been attempts to formalize the best practices by establishing the paradigm's main principles [20], which include:

- **The continuity of the model training and monitoring process**. In many production applications, data rarely remains stationary. As it changes over time, the model must be retrained to include this new information and adapt accordingly to avoid issues such as data and concept drift [29, 30, Section 1.2.3].
- **Reproducibility through data versioning and experiment tracking**. The performance of a machine learning model depends on a myriad of parameters. Keeping track of how a model was set up and which data was used to train it facilitates the reproducibility of past experiments [31] and increases the credibility of the model [32].
- **The incorporation of feedback loops**. As depicted in Figure 1.2, the development process includes multiple intermediate feedback loops which the model must undergo before it can move to the next stage. One such loop involves repetitive model exploration through hyperparameter optimization and validation to find its optimal configuration before it is subjected to training [33].

1.2 Life Cycle of Machine Learning Projects

In the previous section, we mentioned that the tasks involved in a machine learning project can be categorized into four groups. In our work, we will focus on the three technical stages and will discuss them in the following order:

- **data-related phases**: all phases involved in the process of transforming raw data into features,
- **model-related phases**: all phases involved in the design, training and validation of the model,
- **operations-related phases**: all phases involved in deploying the model and its subsequent monitoring and adaptation to changes in data.

1.2.1 Data-Related Phases

It could be argued that the most prominent indicator of how successful a machine learning project can become is the data it is built upon. Therefore, it is crucial that data scientists and engineers cooperate to identify suitable sources of data that is appropriate for the given business task, and put systems in place so that it can be continuously updated for further processing. This data, depending on the project requirements, can come in different forms, for instance:

- **internal**, i.e. data collected internally within an organization; such as sales and customer data or business performance metrics,
- **external**, i.e. data collected from outside sources; such as social media metrics, customer feedback, surveys or various paid data sets,
- **structured**, i.e. data with an easily searchable, pre-defined structure; such as bookings, e-mail addresses or data stored in relational databases,
- **unstructured**, i.e. data without a pre-defined structure; such as media content, medical records or human language (speech or written).

Once the proper data sources have been identified, a decision on the processing approach must be made. There are two main approaches to big data processing: **batch** and **streaming**, each with its benefits and drawbacks that need to be considered for individual use cases [34]. While the former approach is more periodical (or potentially on an ad-hoc basis) and takes in larger amounts of data at once (e.g. hourly or daily sales reports), the latter is based on real-time processing of continuously arriving data (e.g. stock prices in high-frequency trading systems). To accommodate for more specific use cases, hybrid methods such as **micro-batching** have emerged [35], which work on the basis of fixed small-sized batches and short intervals and execute when either of the two limits has been reached.

Depending on the nature of the ingested data and the design of the data pipeline, it is usually stored in storage solutions such as **data lakes** (for raw data), **data warehouses** (for organized, filtered data; used for analytics) or **databases** [36].

In real-life scenarios, data rarely comes in a clean form and must therefore undergo transformations based on rules set up by the data engineering team [20], utilizing processes such as **ETL** (extract-transform-load) or **ELT** (extract-load-transform). The transformations the data undergoes from its raw to processed form can be standardized and defined, using various orchestration tools, in the form of data pipelines, allowing for easier reproducibility and identification of errors in the transformation process.

Some of the processes involved in data preprocessing might comprise the following [37]:

- **Data cleaning.** Raw data sets are often dirty, mislabeled, and contain data that is wrongly formatted, missing or incorrect. The process of data cleaning increases the consistency and usability of the data by removing its imperfections.
- **Data augmentation.** If the volume or variance of raw data is not sufficient, additional data points can be generated by applying transformation techniques to the existing ones, which can help reduce overfitting.
- **Data annotation.** In supervised learning scenarios, all learning samples must be properly labelled. If the data does not contain these labels, the annotation process must be done manually. Data labelling tools use machine learning features to simplify this process and automate daunting tasks.
- **Data analysis.** In order to work with data, we must first understand it properly. Taking the extra time to analyze the relationships and correlations often results in a higher-quality model.
- **Feature engineering.** This process involves the design of features which better represent the underlying relationship between data points by combining properties of the data which by themselves may not provide as much value.
- **Data validation.** To ensure that we performed all the transformation steps correctly, we can use existing tools that verify the proper formatting, uniqueness, range, type and existence of data points; as well as statistical properties of the data in question.

1.2.2 Model-Related Phases

Once the data has been transformed into its desired form, the project enters the stage of **model development** (or **model selection**). In academia, this often entails the design of the model, and comparison of various architectures to find the one that gives the best results. In practice, it is more common to use a suitable pre-trained model and fine-tune in on the given task. Many machine learning tools come pre-packaged with such models that can be freely used as a basis of model training, such as TensorFlow [38] or Hugging Face [39].

Every model needs to be optimized for the given task, which often involves **model tuning**, or more specifically, **hyper-parameter optimization**. This process involves testing the performance of the chosen model depending on different combinations of parameters, and helps in choosing the optimal values [33]. To make this process easier and allow for subsequent visual comparison of HPO runs, many individual tools and end-to-end platforms provide built-in implementations of algorithms for searching the parameter space. After selecting the best parameters, the **model** is **trained** on the train data until desirable values of performance metrics are achieved, or until a termination condition is met [20]. Depending on the complexity of the task at hand, the model architecture and the amount of underlying data, the training process might require extensive computing power. The training process is usually run in one of three places:

- locally on a development machine (for simpler, non-production tasks),
- using on-premise architecture,
- in the cloud, which is becoming increasingly popular due to its availability and scalability [40].

Depending on the choice, processes must be set in place that can transfer the code to the execution environment, run it, and extract the results afterwards. Production-grade workflow orchestrators can be used to make this process easier, and usually even support automatic provisioning of cloud resources, if enough computing power is not available on-premise. Finally, in **model validation**, the model's ability to accurately predict data is compared to the requirements placed by the business objectives, and is either sent back to the beginning of the model engineering phase, or pushed forward into staging and production [20].

1.2.3 Operations-Related Phases

Once a model that has been validated and approved for deployment into production, it needs to be packaged into a format suitable for its deployment on the target infrastructure. This phase is better known as **model deployment** (or **model serving**), and typically comprises the following steps [20]:

1 The model is registered to the model registry (or model store). As part of this process, it is packaged, along with associated metadata, into a special format dictated by the implementation of the model registry.

2 The model serving component pulls the model from the model registry and deploys it on the target infrastructure. By the latter, we usually refer to one of the following:

   - Docker containers with pre-defined REST API endpoints,
   - Kubernetes clusters,
   - cloud-based services (e.g. Heroku, AWS EC2, Google Cloud Run),
   - ML platforms (e.g. Azure ML, Amazon SageMaker),

   If the infrastructure does not exist, some tools have built-in features that can automatically provision cloud-based resources with specified properties.

3 The deployed model is served to its consumers, usually via REST API. Individual tools may provide additional ways of running inference on the deployed models.

In a successful practice of MLOps, the development process does not end with deployment. Rather than that, models in production need **continuous monitoring** to ensure that their performance continues to meet the established requirements and does not decay over time as a result of changes in the structure of ingested data, a shift in the paradigm the model is to represent or as a consequence of unnoticed data engineering errors. This phenomenon is generally referred to as model drift and can be categorized into two main groups: concept drift and data drift.

Concept Drift A concept drift occurs when there is a change in the statistical distribution in one or more target variables of the model, or in other words, when its assumptions of stationarity about the relationships between input and output variables no longer apply due to a change in ground truths. One example of such occurrence could be the effect of a sudden change in customers' purchasing behaviour (e.g. a tendency to purchase different types of products) on a recommendation system, as was the case during the COVID-19 pandemic [29, 41]. The changes, however, do not necessarily have to be abrupt [42], but can also happen gradually, periodically or even sporadically for a short period of time (so-called blips), making them more difficult to detect and address should the proper monitoring tools not be in place.

Data Drift A data drift, also known as covariate shift, results from a change in the statistical distributions of one or more of the models' independent variables, leading to inaccurate and biased results [29]. An example of such drift in the previously mentioned context could be a change in the user demographics, such as age, sex or income levels.

To make the identification of model drifts easier and avoid the overhead of implementing their own drift detection system, teams can incorporate one of many open-source monitoring tools available on the market. These tools usually come with built-in implementations of commonly used methods for drift detection [43] and provide their users with convenient features such as dashboards with visualizations of the model's performance and automatic drift alerts (Figure 1.3). Consequently, developers can detect model drift in its early stages and adjust the model's performance through the application of **continuous training** [21].

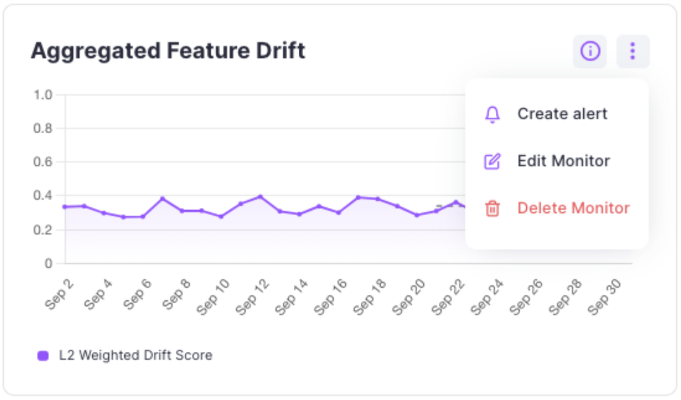

Figure 1.3: A feature drift monitoring feature in Deepchecks [44].

1.3 Chapter Summary

In this chapter, we:

- delineated the evolution of MLOps, summarized its main principles, and explored its relationship to DevOps,
- summarized the life cycle of a machine learning project and described how individual MLOps tools fall into this process.

Chapter 2

Tools and Frameworks

This chapter provides an overview of existing tools and frameworks for MLOps. First, we provide a deeper overview of workflow orchestrators and experiment trackers, and a description of a few selected individual tools. Second, we summarize additional categories of MLOps tools and enumerate notable solutions.

Methodology

The selection of tools in Subsections 2.1.2 and 2.2.2 was done mainly based on the tool's uniqueness, popularity and maturity. In many cases, the differences between individual tools are very marginal, and given the limited scope of the thesis, describing each one in detail would result in information redundancy. Additionally, we tried to minimize the selection of tools bound to a hosted platform, which partially contradicts the open-source nature of the tools. This condition was satisfied in all cases except for the selection of Weights & Biases, which we chose as an alternative to DVC due to the latter's experiment tracking features being too closely bound to the usage of other tools from the parent's company offering of MLOps solutions.

2.1 Workflow Orchestrators

Arguably, one of the most important steps of operationalizing machine learning processes is the proper standardization of pipelines. In every project, numerous steps are required to turn raw data into a deployable model — each with its own business logic, dependencies and resource requirements. The tools known as workflow orchestrators help establish a pre-defined structure for pipelines and ensure that all their steps are executed correctly, with all their dependencies met and properties set.

2.1.1 Key Features

The most common features of workflow orchestrators include the following:

- **dependency management**: The vast majority of orchestration tools are based around the concept of pipelines consisting of one or more steps, where each of the latter represents a well-defined task such as downloading a data set, performing pre-processing tasks on it, training a machine learning model, and deploying it. To ensure that steps are executed in the proper order and that all of their dependencies are met, pipelines are usually implemented in the form of directed acyclic graphs (Figure 2.1).

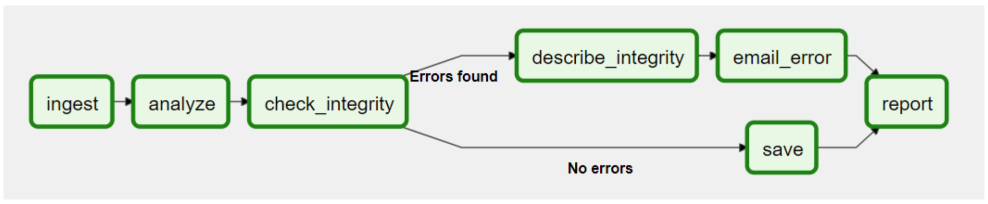

Figure 2.1: A DAG-based pipeline implementation in Apache Airflow [45].

- **scheduling**: If a step needs to be executed periodically at given times, it might be suitable to opt for an orchestrator that supports scheduling. This feature is commonly implemented using cron expressions.

- **dynamic workflows**: Most pipelines are defined statically, and their behaviour does not change during their run. In some cases, however, parameters based on which we logically decide how a task should be performed are only known at runtime.

- **step containerization**: The containerization of steps is often used for transferring parts of a pipeline to be executed in remote environments, such as cloud-based GPU machines or Kubernetes clusters.

- **versioning**: Conceptually, a machine learning pipeline is an arrangement of steps whose order or content may change over time. To keep track of these changes, certain tools implement native pipeline versioning features that automatically detect when a pipeline definition has changed, registering a new versioning.

- **caching**: To avoid unnecessary execution of steps, such as when the inputs and outputs of one remain unaltered, orchestration tools often implement caching features which can automatically detect if a step needs to be executed or if it can be skipped.

2.1.2 Overview of Selected Tools

2.1.2.1 Apache Airflow

Logo	Apache Airflow
Maintainer	Apache Software Foundation
Release Year	2014
License	Apache License 2.0
Website	`www.airflow.apache.org`

Table 2.1: The basic information on Apache Airflow [45].

Key Features & Benefits

- Airflow (Table 2.1) is one of the most mature workflow management tools on the market, dating back to the pre-MLOps era. Due to its maturity and being scoped to orchestration only, it provides many time-tested features and integrations, making it an ideal solution for companies looking for a stable open-source solution with a well-established community behind it.
- The pipelines in Airflow are written in Python. Their atomic building blocks (so-called operators) are widely available from official and community sources, alleviating the need to re-invent code for common tasks. Python functions can be turned into tasks using a simple decorator, while more complex tasks (e.g. operations on cloud storage buckets) usually already have a designated operator implemented and ready for use.
- It offers a feature-rich user interface that allows for pipeline management and visualization using a variety of themed views (e.g. Gannt chart, calendar view).
- As one of few orchestrator tools, it supports plugins that allow the design of pipelines using Jupyter notebooks as steps.

Drawbacks

- As the tool is not specifically meant for MLOps purposes, it may be questionable whether it will keep up with the future trends in the field.
- It does not provide native support for pipeline versioning.

2.1.2.2 DVC

Logo	DVC
Maintainer	Iterative
Release Year	2017
License	Apache License 2.0
Website	`www.dvc.org`

Table 2.2: The basic information on DVC [46].

Key Features & Benefits

- DVC (Table 2.2) provides a unique way of designing pipelines via the command line or directly as YAML files (the former gets converted into the latter automatically).
- It is closely built around the principles of working with Git. Along with its sister tools from Iterative — CML and MLEM – they form an ideal solution for teams who need to build an MLOps stack on top of an existing Git infrastructure.
- Apart from workflow orchestration, it comes with data versioning features, allowing the users to version their data in a Git-like fashion using integrations with a variety of storage solutions.
- For developers using Visual Studio Code, DVC offers an extension which allows the execution of experiments directly from the IDE, with subsequent support for their visualization, along with associated metrics.
- Iterative provides a platform-based solution called Iterative Studio, which serves as a centralized platform for DVC and associated tools.
- It supports live tracking of metrics using DVCLive.

Drawbacks

- Although the approach DVC takes to pipeline design is unique, developers might find it inconvenient to design pipelines using the CLI or YAML files rather than using Python code. While there is a Python API available, it is primarily for the retrieval of data from DVC repositories, and its usage for experiment management is very limited.

2.1.2.3 Flyte

Logo	Flyte
Maintainer	Flyte
Release Year	2019
License	Apache License 2.0
Website	`www.flyte.org`

Table 2.3: The basic information on Flyte [47].

Key Features & Benefits

- Flyte (Table 2.3) is a Kubernetes-grade orchestrator suitable for more complex use cases. It comes with a multitude of advanced features such as intra-task checkpointing, visualizations of data and its lineage, and more.
- It is language-agnostic, providing ways of implementing functions in various languages as tasks (e.g. decorators in Python, base classes in Java).
- It is suitable as an alternative to Kubeflow, as it offers similar features but provides a more user-friendly way of designing pipelines.
- Like Airflow, it can utilize extensions to execute Jupyter notebooks as pipeline steps.
- It provides a variety of production-grade integrations in comparison to other reviewed tools.

Drawbacks

- Due to its complexity, it might have a higher learning curve than the other alternatives and may be an excessive choice for simpler use cases.

2.1.2.4 ZenML

Logo	ZenML
Maintainer	ZenML
Release Year	2021
License	Apache License 2.0
Website	`www.zenml.io`

Table 2.4: The basic information on ZenML [48].

Key Features & Benefits

- Apart from providing workflow orchestration, ZenML (Table 2.4) differentiates itself from the competition by facilitating the creation of entire MLOps stacks through abstractions called stack components. These stack components are effectively abstractions of the code needed to make ZenML interact with other MLOps tools. Once configured, the integrations can be used using the unified ZenML API, rather than setting them up manually and using the API of the respective tool. Additionally, one can extend one of the provided base component classes, introducing support for a framework of their own choice.
- To avoid being completely dependent on third-party tools, ZenML comes with a native implementation of all the components needed for local development. Furthermore, it supports the automatic creation of an MLflow Tracking Server during local pipeline runs, making it an ideal out-of-the-box solution for simple use cases.
- Over the course of writing this thesis, we have registered a high frequency of useful updates, releasing new features approximately every other week, making it a prospective tool to use in the future.

Drawbacks

- The tool is still in its relative infancy and undergoes frequent API changes, making it unsuitable for those looking for a stable solution.
- The integrations with other tools may not support the tools' entire feature set (e.g. ZenML only supports two out of six deployment scenarios of the MLflow Tracking Server). Therefore, it is necessary to check whether it can satisfy the requirements of the project, or whether the integration has to be implemented manually.

2.1.3 Summary

To summarize the information about the individual frameworks, we provide a comparison of the reviewed tools in the following table:

		Airflow	DVC	Flyte	ZenML
DESIGN	**D1**	○	●	○	○
	D2	●	○	●	●
	D3	○	●	○	○
	D4	⦿	○	⦿	○
	D5	○	○	●	○
FEATURES	**F1**	●	○	●	●
	F2	●	○	●	⦿
	F3	○	○	●	●
	F4	○	●	●	●
	F5	○	●	●	●

D1: Design pipelines using the command line.
D2: Design pipelines using Python code (e.g. classes, decorators).
D3: Design pipelines as YAML files.
D4: Use Jupyter Notebook files as pipeline steps.
D5: Design pipelines using other programming languages (e.g. R, Java, etc.).

F1: Create pipelines dynamically at runtime.
F2: Schedule your pipelines to run at specific times.
F3: Run your pipeline steps in isolated containers.
F4: Keep track of how your workflow changes over time with pipeline versioning.
F5: Avoid redundant computations by automatically caching pipeline artifacts.

Legend: ● supported ⦿ supported using plugins ○ not supported

Table 2.5: The comparison of the reviewed workflow orchestration tools.

2.1.4 Additional Tools

The subset of reviewed orchestration tools is only a small part of the current offering. There are other notable implementations, including:

- **Kedro** [49], a Python framework with pipeline visualizations, built-in connectors for retrieving data from various storage solutions, and project templating support.

- **Kubeflow** [50], a Kubernetes-native orchestrator from Google with built-in Jupyter notebook support, hyperparameter tuning component and multi-tenancy.

- **Metaflow** [51], an orchestration tool for Python and R from Facebook that is similar to Flyte.

- **MLflow Recipes** [52], an experimental component of MLflow which allows defining pipelines with a special YAML structure and integrates natively with other features of MLflow.

- **Orchest** [53], a workflow orchestrator that allows the design of pipelines by connecting Jupyter notebooks and Python scripts inside a drag-and-drop style user interface.

2.2 Experiment Trackers

The development of machine learning models is rarely a matter of a single run. Finding the right parameters for a model is an exhaustive process, and as the number of experiments increases, it becomes increasingly important that developers have a way of mapping runs to their parameters for later retrieval and inspection in an easily comprehensible user interface. This is facilitated by tools known as experiment trackers.

2.2.1 Key Features

The most common features of experiment tracking tools include the following:

- **Autologging.** The need for manual logging of every artifact, parameter or metric would prolong the development process and increase the risk of human error. Therefore, many experiment trackers have implemented autologging support, which enables the tool to automatically log various metadata from supported ML libraries using a single line of code.

- **Visualization tools.** The availability of a user interface is without a doubt one of the most important aspects of a tracking tool. The feature set of user interfaces varies by tool, but usually includes various interactive graph visualizations and run comparisons.

- **Filtering.** To quickly filter out runs based on values of metrics or parameters, some trackers have implemented a Pythonic way of searching using conditions such as:

  ```
  run.learning_rate in [0.001, 0.002] and run.batch_size == 32
  ```

- **Integrations with storage solutions.** Each team may take a different approach to storing the outputs of their experiment runs. Whether it is on-premise or cloud-based storage, pre-built integrations with storage solutions are a crucial factor in the selection of a tracking tool.

2.2.2 Overview of Selected Tools

2.2.2.1 Aim

Logo	Aim
Maintainer	AimStack
Release Year	2019
License	Apache License 2.0
Website	`www.aimstack.io`

Table 2.6: The basic information on Aim [54].

Key Features & Benefits

- The user interface of Aim (Table 2.6) is partitioned into sections called explorers, each of which contains adjustable visualization tools tailored for a specific type of artifact, such as metrics, hyperparameters, images, audio or text.
- For easier filtering, developers can use the built-in query language, AimQL, to filter runs based on metrics in a Python-like way.
- Developers can bookmark commonly used visualization configurations (e.g. queries, graph settings) for subsequent access.
- Runs can be marked with color tags, categorized into named experiments for easier querying, live-tracked, logged, send an alert on stagnation or failure, or trigger callbacks on specified events.
- The user interface of Aim can be rendered inside a Jupyter notebook cell, without having to open it in a separate window.
- It can be used to explore runs tracked by both MLflow and W&B.
- It provides multiple interactive demonstrations of the user interface directly on their website.

Drawbacks

- It does not currently support authentication or user isolation.

2.2.2.2 MLflow

Logo	mlflow
Maintainer	Databricks
Release Year	2018
License	Apache License 2.0
Website	`www.mlflow.org`

Table 2.7: The basic information on MLflow [52].

Key Features & Benefits

- MLflow (Table 2.7) is one of the most commonly used experiment tracking solutions, with a managed solution available and backed by Databricks.
- It is a fairly mature, well-documented solution with a large community behind it.
- It supports a variety of deployment scenarios, both local and remote.
- Apart from experiment tracking, it provides additional functionalities such as project management (MLflow Projects), workflow orchestration (MLflow Recipes), model registry (MLflow Model Registry) and deployment (MLflow Models), making it an end-to-end solution.
- It is language-agnostic, and provides an API for Python, R and Java. For all other purposes, language agnosticism is ensured via the REST API.

Drawbacks

- It does not currently support user isolation.
- The support for server authentication has been added very recently, and at the time of the publishment of this work, it has not been fully implemented yet.

2.2.2.3 Weights & Biases

Logo	Weights & Biases
Maintainer	Weights & Biases
Release Year	2018
License	MIT
Website	`www.wandb.ai`

Table 2.8: The basic information on Weights & Biases [55].

Key Features & Benefits

- Weights & Biases (Table 2.8) supports real-time tracking of experiments.
- It has a built-in feature for tracking hardware usage during experiment runs.
- The free tier includes unlimited usage and 100 gigabytes of storage for artifacts.
- Once registered on the platform, the user can use other features of Weights & Biases, such as dataset and model versioning, hyperparameter optimization, automatic report generation, model registry and more.

Drawbacks

- As a platform-based solution released under the MIT license, it requires the user to register on their website and obtain a license key to use the tool, which complicates the initial setup.
- The free plan can only be used for personal projects.
- Only the client application is open-source, meaning that the server cannot be generally self-hosted (although this option is offered for specific scenarios).

2.2.3 Summary

Due to fairly marginal differences between the reviewed frameworks in terms of key features, we omit the comparison table. Overall, MLflow strikes a balance between maturity and availability. It is a well-established solution that provides additional features on top of experiment tracking, and is completely open-source and ready for self-hosting, as opposed to Weights & Biases. However, if the use case of experiment tracking is for personal projects, choosing W&B may be the most comfortable solution. Furthermore, Aim can be used to display experiments tracked using both MLflow and W&B, should the developer prefer the user interface of the former, but would like to retain the functionalities of the latter.

2.2.4 Additional Tools

As opposed to workflow orchestrators, there is a lack of experiment tracking solutions that are not a part of a platform. Regardless, there are other notable solutions, such as:

- **DVC** [46], which provides lightweight experiment tracking features on the command line or in Visual Studio Code and is best suited for use in conjunction with CML [56] and MLEM [57].
- End-to-end ML platforms, e.g. **ClearML** [58] or **Neptune.ai** [59], which provide extensive experiment tracking features, along with other components to fill in the rest of the MLOps stack.

2.3 Other Categories

Apart from workflow orchestrators and experiment trackers, there are other categories of MLOps tools, each of which is designed to facilitate a certain subset of the machine learning development cycle discussed in Section 1.2.

2.3.1 Data Versioning Tools

Code versioning has long been an integral part of the development process in software engineering. The data used in ML model development — whether it is tables, documents, media or even pre-trained models — consists of the same ”ones and zeroes” as code, only with a more complex structure. Consequently, it is reasonable to demand that the same versioning practices are applied to data and models as to code.

Unfortunately, typical version control platforms have yet to adjust their platforms to accommodate such requests, as they usually impose file limits that are easily exceeded by most data sets in commercial settings. As a result, it is common to use services provided by cloud service providers which offer a wide array of scalable, versatile storage solutions, often at a fraction of the cost it would take to host the same data on other platforms. These limitations have created a niche in the market for data versioning tools, which combine the power of versioning with the versatility of such storage solutions, allowing developers to easily track, store and version data sets of virtually any kind or size.

- **DVC** [46] offers Git-like versioning of data by storing the physical files in one of many supported storage locations, and using Git to only store versioned 'pointers' to the actual data. To make it easier to use, its CLI commands are almost identical to the ones used by git.
- **Pachyderm** [60] offers not only data versioning features but doubles as a production-grade data pipeline creation tool with extensive integrations and native support for both batch and data streaming. It also offers a user interface for better ease of use. Although its recent versions have been released under a proprietary license, the community edition offers a generous free tier.

2.3.2 Feature Stores

Feature stores act as a centralized store for designed features, allowing them to be easily managed, reused and shared across different machine learning projects.

- **Feast** [61] is an open-source feature store that supports integrations with numerous batch and streaming data sources, as well as serving of said features to deployed models for training and inference.

2.3.3 Hyperparamer Optimization Tools

Hyperparameter optimization tools help facilitate the homonymous process during the model tuning phase. They implement common algorithms used for HPO and allow them to be used directly in the code.

- **Optuna** [62] is a tool which provides simple abstractions of the search process through trials and studies, which refer to individual runs and entire experiments on objective functions, respectively. It implements a wide selection of algorithms, uses an imperative approach that makes it easy to understand, supports parallelism, and offers a dashboard for visualizing parameter searches.

2.3.4 Data Annotators

To aid with mundane tasks such as data annotation, some tools provide features that help organize and speed up the annotation process.

- **Label Studio** [63] is a data annotation tool that supports the labelling of data types such as text, audio, video, images or time series. It provides a collaborative user interface and uses machine learning to assist in the process.

2.3.5 Data Validators

Data validators often come with built-in checks that can be run on data sets to confirm the correctness of the data preprocessing pipeline.

- **Deepchecks** [44] offers running various validation suites directly from your Python code, such as data integration and train-test validation checks. This helps with the early detection of errors like label mismatches or data leakage. It supports tabular, computer vision and NLP data. It also provides an interface for creating custom checks.

- **Evidently** [64] offers similar features to Deepchecks in terms of creating and running pre-built test suites using Python code, but is currently limited to tabular data. However, it supports the automatic generation of validation reports in multiple formats.

- **Great Expectations** [65] is another feature-rich tool for tabular data validation, which defines the requirements on data in terms of self-documenting assertions called ”expectations”. To alleviate the need to define every single assertion about the data manually, it can generate expectations through automatic data profiling.

2.3.6 Model Testers & Validators

Model testing and validation tools are used to evaluate the performance of a selected model using a selection of test suites. **Deepchecks** [44] and **Evidently** use the same checks for both model testing and monitoring. To learn more about them, refer to Section 2.3.8.

2.3.7 Model Registries & Deployers

Model registries are centralized stores for registered models and their associated metadata. They are used to persist development, staging and production models in a specialized format that can be later used for deployment.

Model deployers (or model serving components) facilitate the deployment of production-ready models to on-premise or cloud infrastructure and expose REST API endpoints for inference purposes (1.2.3).

The tools below double as both registries and deployers, and also support model serving and integrations with commonly used ML tools:

- **BentoML** [66] is a feature-rich solution for storing and deploying models which also comes with Yatai, a UI-enabled tool for easy management of models deployed in Kubernetes clusters.
- **MLEM** [57] is a Git-friendly model registry and deployment tool that works best in conjunction with DVC [46] and CML [56]. It can be used to build a model registry on top of existing Git infrastructure.
- **MLflow** [52] provides both model registry and deployment features and is an ideal choice for those who already use it for experiment tracking.

2.3.8 Monitoring Tools

Monitoring tools are used to track the live performance of models in production, provide visualization dashboards to inspect their metrics, and often include features that can automatically detect and alert the developers if the performance of a model begins to deteriorate (Section 1.2.3).

- **Deepchecks** [44] provides a well-rounded user interface for model monitoring. It supports various checks for tabular and computer vision data, automatic sending of alerts and built-in analysis features for when an issue occurs.
- **Evidently** [64] currently offers early access to its monitoring features, which include a dashboard with visualizations, a wide selection of built-in test suites and metrics, and automatic report generation.

2.4 Chapter Summary

In this chapter, we:

- described different categories of MLOps tools and their purpose,
- provided an in-depth look at selected workflow orchestration and experiment tracking tools,
- enumerated additional notable tools in all categories.

Chapter 3

Analysis

This chapter delineates the purposes and scope of the implementation part of this thesis. First, we identify the typical user roles in a classroom setting. Finally, we describe the functional and non-functional requirements which will be used to design an appropriate solution.

3.1 Purpose and Scope

In recent years, there has been a steady growth in the number of artificial intelligence subjects taught at the Czech Technical University in Prague. As the popularity of the field grows and more students apply to study in this programme, teachers maximize their effort to broaden students' horizons by introducing specialized subjects that discuss exciting subfields of AI and ML. However, it was not until this year that a subject was first introduced that covered the topic of machine learning operationalization. Although typical use cases of machine learning in classroom settings do not require extensive operationalization, we are confident that spreading the word of its advantages and applying a subset of its principles on the processes would benefit both students and teachers alike.

Rather than being a fully-featured implementation based on the analysis of requirements and resources, the practical part is scoped as a complement to the theoretical review of benefits that come with the adoption of MLOps, serving as a proof of concept that can be used a basis for a more focused approach in the future as part of another student's coursework, study project or a thesis. Its main purpose is to demonstrate the possibility of exposing the university's computing resources to students and academics in a safe and simple manner, standardizing the model development process by utilizing some of the features provided by open-source MLOps frameworks, and simplifying the process of coursework evaluation through use of experiment tracking tools.

3.2 User Requirements

From the perspective of a student, there are two typical scenarios in which they work on a machine learning project as part of their course:

Homework and semestral projects. In this scenario, the student works:

- on multiple simpler tasks with shorter deadlines,
- on a more complex task over the course of the whole semester.

The data sets related to the task are usually static and do not require advanced processing. The work is often done in Jupyter notebooks, and users utilize their own computing resources or platforms, such as Google Colaboratory or Kaggle, which provide limited access to free GPUs.

After the student finishes working on the task, they usually submit the notebooks, along with a commentary or report, to the teacher who then evaluates their approach and the correctness of their results.

Theses. In this scenario, the scope and complexity of projects is usually larger. This may include performing extensive research or cooperating with a company. Consequently, there might be additional requirements for computing resources beyond what free platforms can provide; the subsequent deployment of the model into production; or the report of experiments taken during research.

These scenarios clearly identify the main actors in our setting:

1 **Students or academics**, who are the primary contributors to the code, run most of the experiments and are in charge of maintaining the pipeline. They need to have an easy way of writing the code in the form of an executable pipeline; running and visualising experiments; and storing, evaluating and deploying desirable iterations of the model for evaluation and inference purposes.

2 **Teachers or advisors**, who mainly evaluate the development progress by viewing the experiment runs and running inference on the model. They may also be responsible for overseeing and ensuring the proper use of computing resources provided by the university and need a way of exposing them to the researcher.

3.3 Functional Requirements

The functional requirements define the expectations placed on the feature set of the implementation, and describe its underlying logic and resources.
In cooperation with the supervisor, we established the following:

- **FR1: Workflow Orchestration**
  The user(s) of the stack can write new or modify their existing code into runnable pipelines, whose execution will be handled by one of the reviewed workflow orchestration tools.
- **FR2: Experiment Tracking**
  The user(s) of the stack can visualize their past experiment runs, along with their associated metadata and artifacts, using one of the reviewed experiment management tools.
- **FR3: Resource Management**
  The administrator(s) can limit the computing and storage resources available to the stack users in the remote development scenario. This includes specifying the timeout for individual pipeline runs.
- **FR4: User Management**
  The administrator(s) can create and delete user accounts to grant them explicit access to the underlying resources.

3.4 Non-Functional Requirements

The non-functional requirements define the expectations of the qualitative properties of the implementation in terms of usability, reliability, availability and security. We identified the following requirements for our work:

- **NFR1: Availability via GitLab FIT**
  The stack should be deployable in the GitLab instance self-hosted by the faculty (`https://gitlab.fit.cvut.cz`).
- **NFR2: Agnosticity of Development Approach**
  The stack should not force the developer to use a specific execution environment or approach to writing their code. The pipelines must be executable in any environment that supports the execution of Python code (e.g. IDEs, command line, Jupyter notebooks).
- **NFR3: Security**
  The external security layer is assumed to be provided by the faculty's virtual private network. The implementation should provide a way of restricting access to resources from within the university network.
- **NFR4: Caching**
  The workflow orchestration should support caching to save computing resources by avoiding the unnecessary re-execution of unchanged steps.
- **NFR5: Isolation**
  The individual pipeline runs will be executed in isolated environments in order to protect the host machine's resources and prevent tampering.

3.5 Chapter Summary

In this chapter, we:

- briefly described the setting in which machine learning is taught at the university's Faculty of Information Technology,
- identified the main actors in the process of machine learning project development in the specified setting,
- explained the purpose and scope of the implementation part of this work,
- clearly defined the functional and non-functional requirements for the implementation based on a previous agreement with the supervisor.

Chapter 4

Design

The following chapter describes our stack design process, justifies individual choices and summarizes the challenges we encountered, along with workarounds that needed to be implemented to overcome them. We begin by presenting a diagram of our architecture and an overview of the individual elements. Next, we describe the selection process of each component, summarizing its advantages over the alternatives and the ways they integrate with the rest of the stack. Finally, we provide an overview of resource management, such as accounts, credentials and storage buckets.

4.1 Architecture Proposal

To aid in understanding the architecture of the proposed stack, we provide a simple diagram (Figure 4.1) depicting the main components, actors and some of the main interactions between them.

The primary component of the proposal is the **virtual machine** (VM).

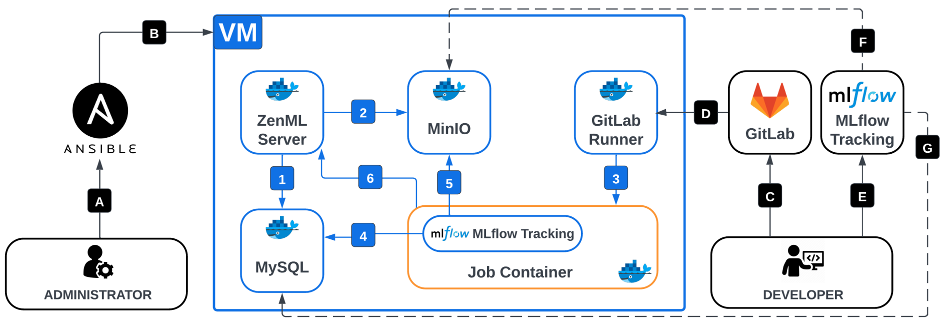

Figure 4.1: The components and interactions of the proposed architecture.

It serves as the host for the tools and services that make up the stack and are deployed in the form of Docker containers. This choice of deployment provides extra security through isolation [**NFR5**] and allows the administrator to specify resource limits for specific containers [**FR3**]. This is especially useful in the case of the GitLab Runner container to prevent it from consuming the entire resource pool of the host machine during pipeline execution. To make the provisioning of these services and resources easier and relieve as much configuration burden as possible, we simplified the process to the following form:

1 The administrator triggers the corresponding Ansible playbook **(A)** on their computer.

2 Ansible will connect to the host machine and execute all tasks in the order specified in the playbook **(B)**.

Each playbook has its specific purpose, such as putting the host machine into a ready-to-use state by deploying all the services using Docker, setting up a root account in each service, and creating default resources. Every time a new, more complex operation is to be defined, an Ansible playbook should be written for it. Otherwise, the configuration process can easily become very complex, and manual execution, or even execution using Bash scripts, can lead to unexpected errors.

Within the VM, there are four permanently running containers. Additional containers are created temporarily for each pipeline run (depicted in orange). The permanent containers include the following:

- a MySQL database, which serves as the backend store for ZenML's pipelines, stacks, users, secrets and other resources **(1)**; the storage for MLflow's run metadata such as parameters and metrics **(4)**; and stores the custom database schema used to keep track of currently registered users (not depicted),
- a ZenML Server deployment, which is used to orchestrate workflows using the machine's computing resources,
- a MinIO server, which serves primarily as the storage for the artifacts and metadata generated by ZenML **(2)** and MLflow **(5)**, but can be used for any purpose, such as for storing study materials or data sets,
- a GitLab Runner deployment, which is used to provision job containers for eligible pipeline runs.

To further elaborate on the temporary containers, GitLab Runner creates a separate Docker container for each job **(3)**, up to the number configured by the administrator. A virtual environment is created within this job container, and all the project dependencies are installed. The job container further connects to the ZenML Server **(6)** and runs a tracking server that connects to the MinIO and MySQL containers so that the outputs of the experiments can be persisted. While this might sound counter-intuitive at first, we provide a deeper explanation for this design choice later in this chapter.

At last, we provide an explanation of basic operations performed by the consumers of the stack. To embrace user-friendliness, we aimed to minimize the need to learn new concepts. Users can keep using GitLab as their code repository the same way they are used to, while the technicalities are handled in the CI/CD pipeline definition file. In the simplest case, each push to a code repository **(C)** should result in a pipeline run execution on the remote server when there is enough capacity for the GitLab Runner to create a job container **(D)**. Otherwise, the pipeline should be queued until there is an empty slot. These functionalities are built into GitLab and require no explicit configuration on our side. Finally, to inspect their previous runs, users can launch an instance of the MLflow Tracking Server **(E)** on their machine using a specific command, which configures the server to connect to the corresponding storage bucket in MinIO **(F)** and database schema in MySQL **(G)**.

4.2 Component Selection

Due to the fairly low complexity of the machine learning projects in question (Section 3.2), at least in terms of not requiring Kubernetes-grade execution or monitoring features, we built our stack using three main components: a workflow orchestrator, an artifact store and an experiment tracker.

To decide on the best tool in each category, we asked ourselves the following questions where applicable:

- Does the tool satisfy our requirements established in the analysis?
- Can the tool be used for both local and remote development?
- How difficult is integrating the tool into our stack compared to alternatives? If it is more difficult, do the advantages of this tool over others outweigh the added complexity of implementation?
- Does the tool integrate well with libraries commonly used in university courses, such as Pandas, sklearn, TensorFlow, PyTorch or Keras?
- Does the tool provide sufficient documentation? How large is the community behind the project?
- If a decision was made in the future to swap the tool for another one, how difficult would the transition be?
- What does the tool's learning curve look like from a developer's perspective?

4.2.1 Workflow Orchestration

Selecting the right orchestration tool was one of the primary objectives [**FR1**]. To ensure that writing new projects or converting old ones is as user-friendly as possible, we needed to select an easy-to-use tool that does not require the user to learn new, complicated concepts. After weighing the benefits and limitations of all reviewed tools, we made the decision to use ZenML. Given the assumption of being limited to a single virtual machine and the scope of machine learning projects in university courses and theses, we established that there was no immediate need for a complex, Kubernetes-grade orchestrator such as Flyte or Kubeflow. To further justify our selection, we provide a list of the main advantages of ZenML for our use case:

- It can be configured for local development purposes in no more than three simple commands: `pip install zenml; zenml init; zenml up`. Apart from providing built-in workflow orchestrator and artifact store components, it also supports the automatic deployment of a local MLflow Tracking Server.
- As mentioned in Section 2.1.2.4, it can be expanded at any time in the future to support a different implementation of workflow orchestration, or any other stack component, without making any changes to the actual code. If used properly, the only changes needed will be made to the infrastructure the code is executed on.
- If the users want to experiment with different tools on their own, they are not bound to the configuration of the server. They can define their own stack for local development, which may use components that are not deployed on the server. Most importantly, there is little-to-none configuration needed to run the same code in both environments.
- Apart from scheduling, its built-in orchestrator supports all the key features described in Section 2.1.1 (including caching [**NFR4**]. However, there is no significant use for scheduling in our setting. If needed, the stack can be modified to use the Airflow orchestrator (Section 2.1.2.1), which supports scheduling using cron expressions.
- It provides a simple-to-understand way of creating and executing pipelines using both a functional and class-based Python API [**NFR2**].

Originally, we intended to create a separate ZenML account for each stack user. This would allow users to connect to the dashboard and use its features, such as visualizing their pipelines and stack configurations. After deeper analysis, this idea had to be cancelled due to the incompleteness of user management features in ZenML. While it supports the creation of users, groups and roles, as well as isolation of user-owned stacks, pipelines and runs within the dashboard, it is currently possible to remove other users without any restrictions once a user has connected to the ZenML Server using any credentials, regardless of their permissions. To prevent potential misuse of this flaw, we had to resort to a temporary workaround, which involves not using the built-in user management features of ZenML and restricting the users' direct access to the ZenML Server deployment by using a single, shared account to execute pipelines without exposing the account's credentials to the developer. The implementation details of this temporary workaround are discussed in more detail in the following chapter.

4.2.2 Experiment Tracking

The next decision we needed to make was to select a tool allowing the users to track the metrics of their pipeline runs and visualize them in a user interface [**FR2**]. Additionally, we needed to consider the tool's capability to track metrics of the most commonly used machine learning frameworks in an automated fashion to minimize the need for manual logging of models and artifacts. Due to the relative lack of tools that are not bound to a hosted platform (e.g. Weights & Biases, which requires the user to register on their website, acquire a free license and use a client to connect to their servers), we ended up choosing MLflow. The following advantages further backed this decision:

- It is a well-established tool that many world-renowned companies use. On top of that, the enterprise version of MLflow is backed by Databricks.
- It supports many deployment scenarios, including a local deployment using an SQLite database and local folders.
- It provides autologging features for various machine learning libraries, alleviating the need to manually enumerate all parameters, artifacts and models to be saved.
- ZenML, which we had chosen as the workflow orchestrator, provides integration for MLflow for both local and remote scenarios, making it easier to set up.
- It is completely open-source and self-hosted, unlike many of the other experiment tracking tools.
- It also doubles as a model registry and deployment component.

Initially, we planned to run a multi-tenant tracking server permanently, allowing both the pipeline runs to connect to it for tracking purposes and the users to inspect and visualize their experiments as needed. However, at the time of writing the thesis, MLflow did not implement user isolation. Had we chosen to deploy a permanently running tracking server connected to a shared database schema, users would have not only had unlimited access to all experiments tracked using the server but also possess the rights to deletion of any resources at will. This forced us to look for an alternative solution, one of which was to run a tracking server temporarily after a remote pipeline run has finished, with the possibility of running a certain number of tracking servers on-demand by creating a manually executable GitLab job. However, this was not viable due to configuration issues related to GitLab's self-hosted runners and the added resource requirements it would introduce. At last, we decided to deploy the tracking server in the following manner:

- For the purposes of tracking experiment runs using the server's resources, each job container would launch its own private instance of the MLflow Tracking Server and have it connect to the database schema and storage bucket owned by the user that executed the pipeline. The artifacts and metadata for would be stored in their corresponding locations for further inspection.
- For the purposes of inspecting past experiment runs, each user can run an instance of the tracking server locally on their machine and have it connect to their respective schema and bucket.

This approach not only helped us simulate user isolation (as the artifacts and metadata are stored in places to which only the owner and administrators have access to), but also reduce the performance requirements on the server, as the number of concurrently deployed tracking servers is equal to the number of running pipelines, which is expected to be low (or even equal to 1). Although the computational requirements of running a tracking server are delegated to the user, they are generally negligible. Furthermore, this allows the user to visualize their experiments on-demand and alleviates the need to implement waiting for empty slots on the server.

4.2.3 Continuous Integration & Delivery

The selection of the continuous integration and delivery platform was mostly influenced by the non-functional requirements [**NFR1**] resulting from the use use of a self-hosted instance of GitLab by the university. This instance is used as the main source code repository for course pages, study materials and students' projects. Therefore, running the pipeline using GitLab CI/CD was more or less a natural choice which allows the users to execute pipeline runs automatically when code is pushed into the repository.

However, the main issue with implementing CI/CD was the ability to use on-premise resources for pipeline execution. By default, GitLab executes pipelines on shared runners that are not powerful enough for machine learning use cases. Furthermore, the number of minutes a user or organization can use these shared runners free of charge is limited. By combining these two factors together, it is not difficult to imagine that a single execution of a pipeline could easily consume a large portion of this budget. Working under the assumption of having at least one GPU available on the host machine, we needed a way of providing these resources to pipeline jobs. For these purposes, GitLab implements a way of registering and using on-premise machines as self-hosted runners via GitLab Runner, which then can be associated with a specific project, group or the entire instance of GitLab.

During the registration process of a new runner, the administrator can specify various options related to the job executor type (i.e. the type of environment used for jobs, such as shell, Docker or Kubernetes), the runner's behaviour and the resources it has access to. While the full list of available options is extensive, we provide a subset of the options we found useful for our use case:

- the number of concurrent jobs that can be run across all runners or within a specific runner,
- the base Docker image to use in job containers (this only applies to the Docker executor),
- options allowing the use of MinIO for caching `mamba` and `pip` packages,
- any additional options that are supported by Docker containers (such as `--oom-kill-disable` to prevent the container from being shut down if the memory limit is surpassed).

Once we have registered our runners, we must provide a link between the code and the registered runners. This is done using a pipeline definition file that specifies the steps to prepare the build environment and execute the pipeline code. The default name for this file is `gitlab-ci.yml`, and it's located in the repository root by default. It is important that administrators keep in mind that exposing the pipeline file to users might be potentially dangerous if an unprivileged user has malicious intent. While the damage is not catastrophic, users can potentially edit the pipeline to include jobs that would block a runner until it times out or change tags in a job to make the step execute on a runner that the administrator has registered for specific purposes. This can be prevented by introducing pre-receive hooks that prevent certain users from committing changes to the pipeline file. On the other hand, such a step could become counterproductive in certain situations, such as when the user's code depends on a package that needs to be installed using a script or the command line. The specific requirements may differ from project to project, so it is recommended that consequences of malicious intentions are communicated to the users and boundaries are set in place for what changes to the pipeline are allowed, should a developer need to make some.

Given that all pipeline runs are executed in a newly created environment, there is a need to download all package dependencies every time a pipeline is run. Depending on the project, this could amount to hundreds of megabytes that are downloaded from public package repositories. Therefore, we experimented with the possibility of introducing a per-user distributed package cache, which would cache the dependencies of a user's project and publish them to a designated storage bucket. With caching enabled, we would prevent the re-downloading of all packages every time a pipeline is run, which comes at the cost of additional storage requirements. Once again, the benefits of either option might be project-specific and need to be reconsidered by the administrator.

4.2.4 Storage

Every time a pipeline is run, unless configured otherwise, ZenML and MLflow generate outputs (e.g. artifacts, metrics, metadata) that need to be persisted somewhere. In our scenario, this implied that we needed a way of allocating a certain portion of the available disk storage to each user, which would primarily serve as storage for run outputs, and secondarily as storage for additional content such as study materials or package cache for GitLab CI/CD.

Upon analyzing the possibilities, we found that the most convenient way of provisioning storage would be by using MinIO, an S3-compatible storage solution with extensive user and bucket management capabilities. This choice would enable us to:

- Create an account for each individual stack user. The user could access the console on-demand to view or modify the contents of buckets they have been granted access to using their designated credentials.
- Define granular bucket access policies. For administrators, we would define a policy that grants them full access to all resources on the server. For developers, we designed a policy that would grant them read-only access to buckets with a specific prefix, and read-write access to their private storage bucket identified by their username.
- Organize stack users into groups and simplify the management of policies by using roles with pre-attached corresponding bucket access policies.
- Define object lifetime rules to preserve disk storage. For example, we can set all objects to be automatically deleted after a given period of time has passed since their creation.

Due to MinIO's compatibility with S3, all components that provide integrations with the latter will also support integrations with MinIO. This way, we were able to configure the S3 Artifact Store stack component in ZenML to push artifacts to specific buckets, and the MLflow Tracking Server to push its outputs in a likewise manner.

For local development, users have two distinct options to select from. They can either utilize the built-in capabilities of ZenML and MLflow and configure their stack(s) to output artifacts and metadata to local folders and databases, which are provisioned automatically by the tools; or use the resources they have been allocated on the remote server by configuring the stack components to connect to these resources using their assigned credentials. It is important to remember that once the outputs have been pushed to either destination, they can only be visualized if the dashboard or server is configured to point to the corresponding storage. Simply put, a pipeline run locally can only be inspected if we set the backend and artifact locations to the corresponding run folder. Likewise, to inspect runs on the remote machine, the tracking server must be connected to the remote database and storage.

In connection with the previous section, if we choose to use distributed cache for package dependencies, we must provide the runner(s) with MinIO credentials at runner registration time. For these purposes, we created a separate bucket access policy that only grants read-write access on the specific path where the cache will be stored.

4.2.5 Database

In order for the deployments of ZenML and MLflow to function properly, we needed to incorporate an SQLAlchemy-compatible database into our stack. We opted for MySQL, as the same database is used in the recommended Docker image for remote ZenML Server deployments. As the purposes of the database were previously discussed in the first section, we will only discuss some caveats associated with the database deployment.

Due to the way we incorporated MLflow into our stack architecture, i.e. by using a separate database schema for each user, we must ensure the associated schema is created along with the user account. Additionally, the database schema must be properly protected from foreign access. We can ensure this by giving administrators full access to all schemas and limiting the access of developers to only have access to their own schema.

In order to keep track of currently active user accounts, we use a separate database schema with a single table for users. Depending on the use case, we can at the very least store the user's login and role. This has many advantages, such as letting Ansible know not to attempt to create a user that already exists or delete one that is not registered.

4.3 Identity and Resource Management

In a multi-tenant environment, e.g. when students and teachers utilize the resources of a dedicated virtual machine, it is important to provide each user with their own isolated workspace and resources. This is mainly to protect the users' privacy and prevent malicious activity (whether it is accidental or intentional). The amount of private resources, the degree of access to the shared ones, and the user's visibility of other users' content is usually tied to the user's role. In the context we are working with, this can be simplified to two main roles: students and teachers (or in broader terms, developers and administrators). Generally, the former usually plays the role of a consumer, whereas the latter is assumed to have executive rights over the underlying resources and infrastructure.

4.3.1 Users

The concept of isolated user accounts has become a standard in software applications and is available in all services that we have selected. However, due to the workarounds we mentioned, we must identify the cases in which we do not want to create an account for specific users.

- In MySQL, we want to create a separate account for each user, regardless of whether it is a developer or administrator. Every user will use their assigned credentials to log in; the difference between the two roles will be in the privileges. While developers will be limited to their own database schema used for MLflow, administrators will have access to all resources. This will allow the latter to inspect any user's experiments.
- In MinIO, we want to take the same approach. In this case, the privileges will be handled using bucket policy definitions. Developers will be able to make changes to their own bucket and read the contents of buckets with a specified prefix, which can be used by administrators for sharing materials with developers. Administrators will have access to all resources, once again allowing them access to developers' artifacts.
- In ZenML Server, we must take a more reserved approach, mostly due to the drawbacks mentioned in Section 4.2.1). For this reason, we will resort to using the root account for all operations. To safely expose the root credentials to developers' CI/CD pipeline definitions, we will use GitLab CI/CD masked variables. The administrators may use the credentials directly. Optionally, a separate non-root account can be created and used, but it currently provides no added benefit apart from the fact that the root user account cannot be deleted using the dashboard (which developers do not have access to).

4.3.1.1 User Resources

To properly implement user isolation, we must first identify all the resources a user needs to be able to operate on the stack seamlessly. In our case, the enumeration goes as follows:

1 a MySQL user account and private DB schema for MLflow metadata,

2 a MinIO user account and private MinIO bucket for artifacts and cache,

3 an entry in the stack management schema to mark the user's existence,

4 a private ZenML stack comprising of private components configured to work with the user's database schema and storage bucket,

5 any ZenML secrets needed to connect to the components safely.

4.3.2 Credentials

In order to access the running services, users must be provided with credentials [**NFR3**]. Their generation will be performed in the following manner:

- By default, root credentials will be generated randomly at the beginning of the server provisioning process. If needed, values can be specified explicitly. These credentials will be used to set up the services.
- The credentials for other users will be generated by the user management playbook on account creation. This password will be used for all the user's resources [**FR4**].

All credentials will be saved to the administrator's computer upon their generation. For now, the responsibility of delivering passwords will be left to the administrator.

4.4 Chapter Summary

In this chapter, we:

- illustrated and explained the architecture of our stack,
- enumerated the selected components and justified their selection,
- elaborated on the inner workings of individual components,
- explained the abstractions used for managing accounts and credentials.

Chapter 5

Implementation

This chapter contains the implementation details of the stack. First, we list and describe the technologies used to provision and run the resources on the host machine. Second, we explain the configuration process. Third, we outline the Ansible playbooks used to set up the host machine and manage users and their resources. Next, we describe the individual Docker containers and their interactions. Finally, we provide an example of a pipeline definition file that can be used to link the host machine with the GitLab infrastructure.

5.1 Tools and Libraries

5.1.1 Ansible

Ansible is an open-source tool that facilitates various IT tasks such as configuration management, resource provisioning and software deployment [67]. It works in an agentless manner, i.e. without the need for a daemon or service to be present on the target nodes. Instead, it connects to the nodes using SSH (or WinRM if the node runs on Windows).

The scripts that execute tasks in Ansible are called playbooks and are written in YAML, making it simpler to learn and use than similar infrastructure-as-code utilities such as Puppet [68], which uses a custom declarative language, or Chef [69], which uses Ruby. A playbook consists of one or more tasks, each of which calls an Ansible module. This encapsulation of common tasks in modules makes it more robust, as they are written in Python and often implemented as wrappers around APIs, making them platform-independent. Developers can use built-in models, acquire new ones from additional sources such as Ansible Galaxy (Ansible's built-in module repository), source code repositories like GitHub, or they can develop their own. The listing below shows a slightly modified excerpt from one of our playbooks and how the `community.mysql.mysql_user` module can be used to create a new user in the MySQL database deployment.

```yaml
- name: Create a user account in MySQL
  community.mysql.mysql_user:
    # The credentials of the user to create.
    name: "{{ username }}"  # Ansible supports Jinja templating.
    password: "{{ password }}"
    # The credentials of the account
    # which executes the user creation.
    login_user: "root"
    login_password: "root"
    host: "%"
    # Specify the privileges of the new user.
    priv: "mlflowdb_{{ username }}.*:ALL"
    state: present  # Use 'absent' to delete the user.
```

Listing 5.1: The use of a community Ansible module in practice.

Our choice of Ansible was motivated by the following reasons:

- The provisioning of resources for the stack is a complicated process. Were this process only to be documented as a step-by-step guide, it could only discourage people from using it but would also be prone to human error. Instead of dedicating time to writing the instructions, we can write an Ansible playbook that can be executed repeatedly, taking the burden away from the administrator. Additionally, as Ansible uses YAML, the playbooks themselves are, to a degree, self-documenting.

- In comparison to the use of regular shell scripts, Ansible provides a more flexible and robust way of writing infrastructure code. The availability of open-source community modules saves time and eliminates the need to write boilerplate code for every routine task in the process.

- Ansible provides many useful features, such as Jinja templating support, organization of target nodes into groups by functionality, and encapsulation of tasks, files and variables into reusable roles.

- By using Ansible, we can easily add support for multiple hosts and various platforms with minimal changes to existing infrastructure code.

- The execution of playbooks using Python API via the `ansible-runner` package, the command line or directly from some IDEs such as Visual Studio Code.

A more detailed look at the playbook we use for provisioning resources will be provided in Section 5.3.

5.1.2 Docker

Docker [70] is a virtualization platform for running applications in lightweight container environments which are isolated from the host system, apart from utilizing its system resources. Running applications in containers instead of installing them directly on the host machine helps us not only with the deployment process, as all of the services we use are also available in the form of Docker images but also helps us protect the host machine from malicious intent through encapsulation. The use of Docker further provides us with these advantages:

- Ansible provides a collection of community modules (`community.docker`) for interacting with Docker containers, including their deployment and the execution of commands within them.
- It provides functionalities for limiting resources available to containers, which we can use to limit the computing power available to the pipeline executors and prevent them from consuming too many resources and influencing the functionality of other services.
- Docker containers are platform-independent, placing no additional requirements on the selection of the operating system on the host machine.
- The deployment of services as containers simplifies the installation process, as we can use the same API for all deployments rather than having to follow instructions based on the operating system and service.

We provide an overview of the configuration of individual containers in Section 5.4.

5.2 Configuration Files

There are several configuration files that need to be adjusted for our implementation to work properly. They are located at the root of the `ansible/` folder. For better understanding, we provide an overview of each other configuration file's purpose below.

5.2.1 ansible.cfg

This file is used to configure Ansible itself and provides granular control over its behaviour. However, for most use cases, overriding the default values is unnecessary. In our work, we only override the following values:

- We set the inventory file path to `./inventory.yml`, instead of the default `/etc/ansible/hosts`, to prevent the administrator from having to edit their global Ansible configuration.

- We use a custom standard output logging callback [71] to make the console output more compact. This callback also makes potential errors more readable. This is a matter of personal preference and may be commented out or replaced with another callback. The full list of official callbacks in available in the Ansible documentation [67].

For a full list of configuration options, one can run the `ansible-config init --disabled` command (and optionally pipe it into a new file). This requires Ansible to be installed, so we have included an example in the `extras/` subdirectory. Please note that the file format may change with future Ansible versions, so it is only intended for illustration purposes.

5.2.2 inventory.yml

This file is used by Ansible to identify individual hosts and allows the administrator to define a hierarchy of managed nodes. In our use case, we only assume the existence of a single managed node, but altering the inventory file makes it simple to register additional host machines.

The inventory file used in our implementation goes as follows:

```yaml
all:
  hosts:
    localhost:
      ansible_connection: local
      ansible_python_interpreter: "{{ansible_playbook_python}}"
  children:
    remote:
      hosts:
        vm1:
          ansible_host: ...
          ansible_port: ...
          ansible_user: ...
          ansible_password: ...
          ansible_ssh_private_key_file: ...
```

Listing 5.2: The included Ansible inventory file with basic hierarchy.

Ansible allows the declaration of hosts on multiple levels. For example, we use `hosts: remote` to implicitly run all tasks in a playbook on the host machine and override this setting with `delegate_to: localhost` when we need a specific task to execute on the control node.

The hierarchy established in Listing 5.2 allows us to do the following:

- If we want to run tasks on both the control node and all managed node, we can use `hosts: all`.

- If we want to run tasks on the control node, we can use `hosts: localhost`. The same applies for `hosts: remote` and managed nodes.
- If we had multiple managed nodes (e.g. `vm1, vm2, vm3`) and we wanted to execute tasks on a specific one, we could use `hosts: vm2`.

The inventory system in Ansible is a powerful tool that can not be fully utilized with a single host machine, but at the very least, it helps with organization. For each node, we then define the following properties:

- **ansible_host**: The IP address of the server.
- **ansible_port**: The server port to use for SSH. If not specified, it defaults to 22.
- **ansible_user**: The account that will be implicitly used to execute tasks, unless explicitly overridden or using elevated privileges via one of the `become` options.
- **ansible_password**: The password to use for SSH connection (not recommended).
- **ansible_ssh_private_key_file**: The path to the private key on the control node to use for SSH connection (recommended).

These settings correspond to the following `ssh` command call:

```
ssh [ansible_user]@[ansible_host]:[ansible_port] \
    -i [ansible_ssh_private_key_file]

# If not using '-i', the system prompts for [ansible_password].
```

5.2.3 runners.yml

This file is used to define the properties of the GitLab Runner container, as well as individual runners to provision on the server. Effectively, it works as an Ansible variable file that is consumed by the Ansible role used for GitLab Runner container provisioning (see Section 5.4). Our implementation provides a simplified version of the full file, which can be found under `defaults/main.yml` in the role's GitHub repository [72]. If the administrator does not explicitly override these values, the default ones will be used from the currently installed version of the role on the control node. Due to the size of the file, we only summarize the properties that can be adjusted:

- The URL of the GitLab instance and the GitLab Runner registration token. Upon registration, the runner will automatically link itself to the project, group or instance it is configured for.

- If GitLab Runner is configured to deploy as a container, we can configure its properties (e.g. name, tags, volume mount path, networks, etc.).
- The enumeration of individual runners and their properties (e.g. executors, job concurrency, image, tags, cache, etc.).

5.2.4 stack.yml

For other configuration variables which are used in the playbooks but not necessarily bound to a specific tool, we created a separate configuration file, and currently lets us specify the following:

- A custom root username and/or password. If not specified, it uses `root` and a randomly generated password, respectively.
- The name for the Docker bridge network to which all containers will be connected, and the list of user accounts that will have elevated privileges when running `docker` commands.
- The list of `pip` packages to install on the managed node.
- The list of buckets to create after MinIO has been set up.
- The size and object lifetime of the developers' buckets.
- The list of ZenML integrations to install (i.e. those required by the configured stacks).

5.3 Ansible Playbooks

Our implementation uses Ansible primarily to set up the infrastructure on the virtual machine and the subsequent creation and deletion of user accounts and their respective resources. These functionalities are split into two separate playbooks:

- `run_server.yml`, which puts the host machine in a state where it is ready to execute users' pipelines,
- `manage_users.yml`, which adds or removes users along with their associated resources.

5.3.1 run_server.yml

This playbook handles the installation of prerequisites on the host, setting up root credentials, deploying and configuring the containerized services and creating the administrator's account and resources on the server.

In more detail, the process goes as follows:

1 A check for the presence of root credentials on the control node (`credentials/root.passwd`) is performed. If not found, it will either use the user-provided values from `stack.yml`, or will use `root` as the username, along with a randomly generated password, which will be stored in previously mentioned file.

2 The `geerlingguy.docker` and `geerlingguy.pip` roles are used to install and set up Docker and Pip, along with the latter's dependencies. This puts the host into a state where it can run Docker containers. To allow containers to communicate with each other without having to resolve the IP addresses manually, a custom bridge network is created.

3 The custom role `mysql` is used to deploy the MySQL container. Before that, the database initialization script is copied into the container so that it can be deployed during its creation.

4 The custom role `minio` is used to deploy the MinIO container. Once deployed, the MinIO Client binaries are downloaded to the host and an alias is set up for the container's MinIO API endpoint. Afterwards, the bucket policy definitions are copied to the host and registered; the `shared_buckets` are created as defined in `stack.yml`, along with an account through which GitLab Runner jobs will access the package cache.

5 The role `riemers.gitlab-runner` is used to deploy the GitLab Runner container and register the runners as specified in `runners.yml`. At this point, the runner should be visible as active in the CI/CD section in your GitLab project, group or instance.

6 The custom role `zenml-server` is used to deploy the ZenML Server container. Once deployed, a connection is established to the ZenML Server, a ZenML repository is initialized, and default ZenML integrations are installed.

7 An account with administrative privileges is created for the executor of the playbook.

Once the playbook successfully finishes execution, the server should be ready to serve pipeline requests. The administrator can verify the correctness of the deployment by connecting to the machine, inspecting the running Docker containers and their logs, and connecting to the individual services (e.g. the database, MinIO console or ZenML dashboard) using their or the root user credentials. At this point, they can create or delete administrator and developer accounts using the `manage_users.yml` playbook.

5.3.2 manage_users.yml

This playbook facilitates the creation and deletion of user accounts along with their associated resources. It utilizes the custom role `user`, more specifically the two task definition files `create.yml` and `delete.yml` located in `roles/user/tasks`.

The process of account creation comprises the following steps:

1 A check is performed to ensure the user does not already exist. Otherwise, the operation is aborted.

2 A random password is generated for the user and is stored in the `credentials/{username}.passwd` file. This can then be distributed to the user so that they can access their resources.

3 A MySQL account is created, followed by database schema with the name `mlflow_{username}`, to which the user is assigned full access to it.

4 A MinIO user account with corresponding privileges is created, along with a storage bucket with the name `{username}`. The user account is also assigned to the corresponding MinIO group, from which it inherits the bucket policies associated with it.

5 The user is provisioned with the following ZenML resources:

   - a ZenML secret with the user's credentials for connecting to their storage bucket,
   - an empty ZenML secret for connecting to the MLflow Tracking Server (as a workaround for ZenML's requirement to supply a secret even if the tracking server does not require authentication),
   - a local ZenML orchestrator component (to allow for recursive deletion using ZenML CLI),
   - an S3 artifact store component, which is configured to connect to the user's storage bucket using the previously mentioned secret,
   - a MLflow experiment tracker component used to connect to the tracking server instance launched during pipeline execution run on GitLab CI/CD,
   - a ZenML stack comprising of the three previously mentioned components.

6 The user's addition is acknowledged by adding them to the `users` table of the `stack_management` database schema.

The deletion process follows a similar scenario in reverse, prompting the administrator to confirm the deletion process before the resources are deleted irreversibly.

5.4 Docker Containers

As previously mentioned in Section 4.1, there are four Docker containers that run continuously on the server. Their deployment is handled by the `run_server.yml` playbook, more specifically:

- MySQL, MinIO and ZenML Server are deployed using the `community.docker.docker_container` module, which allows us to specify the properties of the container in the following way:

```
- name: Deploy ZenML Server as a Docker container
  community.docker.docker_container:
  name: zenml-server
  image: "zenmldocker/zenml-server:{{ zenml_server_image_version }}"
  state: started
  restart_policy: always
  networks:
    - name: "{{ docker_bridge_name }}"
  env:
    ZENML_STORE_URL:
        "mysql://{{ root_username}}:{{ root_password }}@mysql/zenml"
    ZENML_DEFAULT_USER_NAME: "{{ root_username }}"
    ZENML_DEFAULT_USER_PASSWORD: "{{ root_password }}"
  published_ports:
    - 8080:8080
  etc_hosts:
    host.docker.internal: "host-gateway"
```

Listing 5.3: The deployment of ZenML Server using Ansible.

- GitLab Runner is deployed using the `riemers.gitlab-runner` role [72], which, apart from deploying GitLab Runner in a container, also helps automate its configuration and the provisioning of individual runners.

5.4.1 MySQL

The first containerized service running on the server is the MySQL database. Its deployment and configuration are encapsulated in the `mysql` role, using the Ansible task shown in Listing 5.4. Additionally, we define the `init.sql` file (Listing 5.5) located in `roles/mysql/files`, which creates a simple database schema `stack_management` for keeping track of registered users, and creates an empty `zenml` schema, which is then filled in during the deployment of the ZenML Server container. This file is moved to the `/etc/mysql/scripts` folder on the host, which is mapped to the `docker-entrypoint-initdb.d` folder in the container and automatically executed after the container is deployed.

```
- name: "Deploy MySQL as a Docker container"
  community.docker.docker_container:
    name: mysql
    hostname: mysql
    image: mysql:{{ mysql_docker_image_version }}
    restart_policy: always
    networks:
      - name: "{{ docker_bridge_name }}"
    published_ports:
      - "3306:3306"
    volumes:
      - /etc/mysql/data:/var/lib/mysql
      - /etc/mysql/scripts:/docker-entrypoint-initdb.d
    env:
      MYSQL_ROOT_USER: "{{ root_username }}"
      MYSQL_ROOT_PASSWORD: "{{ root_password }}"
    state: started
```

Listing 5.4: The Ansible task used for MySQL deployment.

```
CREATE DATABASE IF NOT EXISTS stack_management;

CREATE TABLE IF NOT EXISTS stack_management.users (
    username VARCHAR(32),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    account_type ENUM('administrator', 'developer')
);

CREATE DATABASE IF NOT EXISTS zenml;
```

Listing 5.5: The schema initialization file used during MySQL deployment.

The container is exposed via the standard port 3306 on the host machine, allowing registered users to access their resources on demand using a client of their own choice. However, developer accounts only possess access to their MLflow database schema, contents of which may not be comprehensible in their raw form and will mostly be accessed in a structured form via the UI provided by the MLflow Tracking Server. The administrators, who have access to all database schemas, can therefore launch a tracking server that connects to any developer's schema and inspect their experiments without manually needing this information from the developer.

5.4.2 MinIO

The second container in order of deployment is MinIO, whose deployment is handled by the task shown in Listing 5.6. The container exposes two ports to the host:

- port 9000, which is the API port used to perform resource management operations. As developer buckets are used for artifact storage, this port will be used to access resources from ZenML and the MLflow Tracking Server. Administrators, who have access to all buckets, will be able to connect their tracking server instances to any developer's bucket and visualize their experiments,
- port 9090, which hosts a web server that users can access to interact with the MinIO deployment in a visual, user-friendly way, after entering their credentials. Administrators can use this console to manage resources if preferred over using the CLI client.

```yaml
- name: "Deploy MinIO as a Docker container"
  community.docker.docker_container:
    name: minio
    hostname: minio
    image: minio/minio:{{ minio_docker_image_version }}
    restart_policy: always
    published_ports:
      - "9000:9000"
      - "9090:9090"
    volumes:
      - /etc/minio/data:/mnt/minio/data
    command:
        ["server", "--console-address", ":9090", "/mnt/minio/data"]
    networks:
      - name: "{{ docker_bridge_name }}"
    env:
      MINIO_ROOT_USER: "{{ root_username }}"
      MINIO_ROOT_PASSWORD: "{{ root_password }}"
    state: started
```

Listing 5.6: The Ansible task used for MinIO deployment.

The deployment of resources themselves is handled using the `mc` client, which allows resource management using the command line. To allow the automatic application of bucket policies, we have defined three default policies in the `roles/minio/files` folder. These policies are in the form of JSON files and are written using the same API as in Amazon S3. Listing 5.7 shows the default bucket access policy for developers, who have read-only access to buckets that use the prefix `shared`, and read-write access to buckets whose name is identical to their username. After the deployment, these files are copied to the host machine, applied using the MinIO client and attached to corresponding roles.

```json
{
"Version": "2012-10-17",
"Statement": [
    {
        "Sid": "ReadObjectsInSharedBuckets",
        "Action": [
            "s3:ListBucket",
            "s3:GetObject"
        ],
        "Effect": "Allow",
        "Resource": [
            "arn:aws:s3:::shared*",
            "arn:aws:s3:::shared*/*"
        ]
    },
    {
        "Sid": "ModifyObjectsInUserBuckets",
        "Action": [
            "s3:ListBucket",
            "s3:GetObject",
            "s3:PutObject",
            "s3:DeleteObject",
            "s3:GetObjectRetention"
        ],
        "Effect": "Allow",
        "Resource": [
            "arn:aws:s3:::${aws:username}/*"
        ]
    }
]
}
```

Listing 5.7: The default bucket access policy used for developer accounts.

5.4.3 GitLab Runner

The GitLab Runner container deployment and configuration is handled by the `riemers.gitlab-runner` role. It is the only container that does not publish any ports, so it cannot be directly accessed from outside the host machine. Instead, the communication between GitLab and the runner container is handled internally over HTTPS, where the latter constantly polls the server for new tasks (the communication is one-way from the runner to the server). The container launches CI/CD jobs when triggered by GitLab. In our case, it launches additional Docker images on the host machine, once again without exposing any ports, providing an added layer of security and isolation.

5.4.4 ZenML Server

At last, we deploy the ZenML Server using the `zenml-server` role. The task overseeing the deployment is shown in Listing 5.3. ZenML Server exposes port 8080, which can be used to access both the API and the dashboard. However, due to the reasons mentioned in 4.2.1, the access to the ZenML server (and the UI) is only allowed using the root account, whose credentials are hidden from unprivileged users via GitLab masked variables.

5.5 GitLab CI/CD

To let the runners know what to do with the code in the developer's repositories, we must define the GitLab CI/CD pipeline. The one we used can be found in `src/extras/gitlab-ci.yml`, and can be modified to the needs of the group or project. For better clarity, we provide an overview of what happens in individual sections of the pipeline definition file:

- `image`: In this section, we specify the Docker image that will be used in the job container. If not explicitly specified, the default value from the runner configuration file will be used.

- `variables`: Here, we define a unique name for the virtual environment, consisting of the username, project name, and commit hash. This is not mandatory but improves clarity and prevents possible collisions of environment names. We then define the path to the `conda.yaml` file, which each user can edit to define their project dependencies. Finally, we specify the paths to directories where the package manager store downloaded packages.

- `cache`: This section specifies the key which will be used to search the corresponding cache. Right now, we are using the branch name. Alternatively, we can the username or even a common key to force all pipeline runs to use the same cache (note that runners need to be configured accordingly to point to the correct location as well). We also specify the directories whose contents should be cached.

- `stages`: Here, we specify the names of individual stages. As we only have a single job, this can be skipped entirely and will only have cosmetic effects.

Finally, we proceed to the definition of the `Run the pipeline` job. All code within this section is executed inside a job container. The outline of the script goes as follows:

1 Conda is used to install Mamba, a lightweight C++ re-implementation of the former. This has allowed us to slightly increase the package downloading process (along with caching). Alternatively, a Docker image with Mamba pre-installed can be used to avoid this extra step. The tool is initialized, and the shell is restarted to apply changes.

2 The APT repository cache is updated, and required APT dependencies are installed (the two included packages are required for establishing an SQL connection by the tracking server).

3 A virtual environment is created using the supplied requirements file and activated.

4 Environment variables required by the MLflow Tracking Server are exported. These include the MinIO endpoint URL and the access and secret keys (i.e. username and password).

5 An instance of MLflow Tracking Server is configured to connect to the repository owner's database schema and storage bucket, and launched as a background job (to prevent it from blocking the shell, which is the default behaviour of MLflow

6 A connection is established to the ZenML Server on the host, the user's private stack is activated, and the entry point (pipeline execution) command is run, triggering the pipeline.

7 Once the pipeline execution has finished, we kill the MLflow Tracking Server, disconnect from the ZenML Server, and conclude the job.

5.6 Chapter Summary

In this chapter, we:

- described the tools used in our implementation,

- explained the configuration process of the stack,
- outlined the functioning of Ansible playbooks and described the contents of included ones,
- explained the configuration of individual Docker containers,
- elaborated on the individual steps in the GitLab CI/CD pipeline file.

Chapter 6

Conclusion

This diploma thesis discusses the growing importance of MLOps and explores the possibilities of its practical application in university courses and research tasks at our university. In the introductory part of our work, we outlined the benefits that machine learning operationalization has brought to organizations that have successfully integrated its principles into their business processes, supported by surveys that reflect on the state of AI adoption in the business sector. Furthermore, we delineated the main objectives for our work:

- summarizing the key principles of MLOps and identifying its potential for the facilitation of individual phases of the machine learning project life cycle,
- providing an overview of the currently available open-source tools and highlighting their unique features,
- analyzing the setting in which machine learning projects are taught at our university, and identifying processes that could be improved by the use of MLOps tools,
- designing and implementing a proof of concept that demonstrates the potential of MLOps adoption for the benefit of students, researchers and teachers.

To provide strong foundational knowledge of the subject matter, we explored the reasons behind the formation of the paradigm, outlining the key principles by which it differentiates from operationalization in software engineering, and highlighting its hidden potential for the standardization and simplification of processes involved in every stage of a machine learning project's life cycle. We performed an overview of different categories of open-source tools available on the market, pointed out their strengths and weaknesses, and elaborated on a subset of tools that applied to our use case and fell within the scope of our work. Furthermore, we analyzed the typical scenarios of machine learning project development at our university in order to identify the main actors and interactions involved, and established the requirements for the practical part of this thesis. Based on the collected knowledge, we designed an architecture that allows the exposure of computing and storage resources to developers and allows for the future incorporation of additional components. Finally, we provided an implementation using Ansible playbooks, which can automatically deploy the proposed infrastructure on a selected host machine. Additionally, we provide a tutorial for both administrators and developers that elaborates on how to configure the architecture to work with the university's GitLab infrastructure, and how to configure their code and GitLab accounts to be able to utilize the provided resources, respectively.

Evaluation

Based on the enumeration of our achievements, we consider the objectives of this diploma thesis to be fulfilled. We provided the reader with a sufficient amount of foundational knowledge of the subject matter to the extent needed to understand the concepts in the thesis, summarized the current offering of relevant open-source tools and their features, and identified processes at our university which could benefit from their integration. These theoretical foundations were then used to design a suitable architecture and implement a proof of concept that demonstrates some of the possibilities of application in the learning and research processes at our university.

To objectively reflect on the shortcomings, we must acknowledge that the provided example project is somewhat lacklustre and only fit as a quick demonstration of the configuration. At the time of writing the assignment, we planned to include a project that would cover the entire life cycle, as described in Chapter 1. However, it soon became clear that building a project that would utilize such a wide selection of tools could not only easily fall within the scope of a separate thesis, but would also require a proportional architecture of similar complexity. Instead, we adjusted our goals to put more focus on providing a foundational knowledge of the paradigm and the tools that help implement it, so that possible future work could put more focus on the implementation part and use the outputs of this work as a reference.

Future Work

Although our work displays some of the advantages of the adoption of MLOps, a fully-featured implementation based on a methodological analysis of the requirements and preferences of students and teachers is far beyond the scope of this thesis. Instead, our work is meant to advocate this effort, and hopefully marks the first step of a long but rewarding path of incorporating some of its principles into the learning process, perhaps as a result of a joint effort of software engineering and artificial intelligence students' coursework or theses in the future.

To contribute to the pool of ideas in which the implementation could be improved, we provide the following suggestions that can be considered in future works:

Addition of new stack components.

A structured, in-depth analysis of the preferences and requirements of the teaching staff, researchers and students will introduce the need to incorporate additional stack components. We strongly believe that the selection of ZenML as the workflow orchestrator will greatly facilitate the expansion process.

Re-evaluation of the proposed architecture.

The selected tools will likely advance in the future and their limitations which led to the implementation of workarounds may be removed. Therefore, it will be necessary to reconsider the architecture and evaluate the possibility of its simplification.

Further automation of mundane tasks.

Ansible is a powerful and versatile tool that helps with the automation of various tasks, especially from the administrator's perspective. We further advocate its use to standardize the execution of generic tasks such as password distribution or automatic provisioning of user resources on certain events, such as their addition to a GitLab group.

Encapsulation of common tasks.

Although the underlying tools and resources can be used manually through their respective APIs, it would be suitable to identify the most common ones and provide a way to execute them in a user-friendly way, such as a web application or a command line utility. This may include processes such as password management, starting an instance of the tracking server that points to specific resources, and more. This would greatly improve the usability of the stack in practice.

Expansion of the architecture

If the proposed architecture was ever to be used in production, it would be suitable to add additional components such as a reverse proxy that would allow for easier authentication and user management, or monitoring tools that would allow the administrators to keep track of resource usage or identify problems, should any arise.

Bibliography

[1] Taulli, T. MLOps: What You Need To Know. August 2020, [Online; posted August 1, 2020]. Available from: `https://www.forbes.com/sites/tomtaulli/2020/08/01/mlops-what-you-need-to-know/?sh=27f30f712146`

[2] McCulloch, W. S.; Pitts, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, volume 52, no. 1, 1990: pp. 99–115, ISSN 0092-8240, doi:https://doi.org/10.1016/S0092-8240(05)80006-0. Available from: `https://www.sciencedirect.com/science/article/pii/S0092824005800060`

[3] Turing, A. M. Computing Machinery and Intelligence. *Mind*, volume 59, no. 236, 1950: pp. 433–460, ISSN 00264423. Available from: `http://www.jstor.org/stable/2251299`

[4] Rosenblatt, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, volume 65, no. 6, 1958: pp. 386–408, ISSN 0033-295X, doi:10.1037/h0042519. Available from: `http://dx.doi.org/10.1037/h0042519`

[5] Samuel, A. L. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, volume 3, no. 3, July 1959: pp. 210–229.

[6] Press, G. On Thinking Machines, Machine Learning And How AI Took Over Statistics. 2021. Available from: `https://www.forbes.com/sites/gilpress/2021/05/28/on-thinking-machines-machine-learning-and-how-ai-took-over-statistics/?sh=47df816b2513`

[7] Gonsalves, T. *The Summers and Winters of Artificial Intelligence*. 01 2019, ISBN 9781522573692, pp. 168–179, doi:10.4018/978-1-5225-7368-5.ch014.

[8] Rieder, B. Towards a political economy of technical systems: The case of Google. *Big Data & Society*, volume 9, no. 2, 2022: p. 20539517221135162, doi:10.1177/20539517221135162. Available from: `https://doi.org/10.1177/20539517221135162`

[9] Artificial Intelligence Index Report 2022. `https://aiindex.stanford.edu/wp-content/uploads/2022/03/2022-AI-Index-Report_Master.pdf`, 2022.

[10] Dr. Anand S. Rao, G. V. Sizing the prize. What's the real value of AI for your business and how can you capitalise? `https://www.pwc.com/gx/en/issues/analytics/assets/pwc-ai-analysis-sizing-the-prize-report.pdf`, 2020.

[11] McKinsey. The state of AI in 2022—and a half decade in review. December 2022, [Online; posted December 6, 2022]. Available from: `https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai-in-2022-and-a-half-decade-in-review`

[12] The State of Machine Learning Operations 2022. `https://info.verta.ai/report-state-of-mlops-2022`, 2022.

[13] Engel, C.; Ebel, P.; et al. Empirically Exploring the Cause-Effect Relationships of AI Characteristics, Project Management Challenges, and Organizational Change. In *Innovation Through Information Systems*, edited by F. Ahlemann; R. Schütte; S. Stieglitz, Cham: Springer International Publishing, 2021, ISBN 978-3-030-86797-3, pp. 166–181.

[14] Davenport, T. H. From analytics to artificial intelligence. *Journal of Business Analytics*, volume 1, no. 2, 2018: pp. 73–80, doi:10.1080/2573234X.2018.1543535. Available from: `https://doi.org/10.1080/2573234X.2018.1543535`

[15] Herremans, D. aiSTROM–A Roadmap for Developing a Successful AI Strategy. *IEEE Access*, volume 9, 2021: pp. 155826–155838, doi:10.1109/access.2021.3127548. Available from: `https://doi.org/10.48550/arXiv.2107.06071`

[16] Gartner. Gartner Says Nearly Half of CIOs Are Planning to Deploy Artificial Intelligence. 2018.

[17] Artificial Intelligence Index Report 2023. `https://aiindex.stanford.edu/wp-content/uploads/2023/04/HAI_AI-Index-Report_2023.pdf`, 2023.

[18] Tamburri, D. A. Sustainable MLOps: Trends and Challenges. In *2020 22nd International Symposium on Symbolic and Numeric Algorithms*

for Scientific Computing (SYNASC), 2020, pp. 17–23, doi:10.1109/SYNASC51798.2020.00015.

[19] Mäkinen, S.; Skogström, H.; et al. Who Needs MLOps: What Data Scientists Seek to Accomplish and How Can MLOps Help? In *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, 2021, pp. 109–112, doi:10.1109/WAIN52551.2021.00024.

[20] Kreuzberger, D.; Kühl, N.; et al. Machine Learning Operations (MLOps): Overview, Definition, and Architecture. *IEEE Access*, volume 11, 2023: pp. 31866–31879, doi:10.1109/ACCESS.2023.3262138.

[21] John, M. M.; Olsson, H. H.; et al. Towards MLOps: A Framework and Maturity Model. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2021, pp. 1–8, doi:10.1109/SEAA53835.2021.00050.

[22] Renggli, C.; Rimanic, L.; et al. A Data Quality-Driven View of MLOps. 2021, `2102.07750`.

[23] Sculley, D.; Holt, G.; et al. Hidden Technical Debt in Machine Learning Systems. *NIPS*, 01 2015: pp. 2494–2502.

[24] Appinventiv. Estimating the Time, Cost, & Deliverables of an ML App. `https://appinventiv.com/blog/machine-learning-app-project-estimate/`, 2022.

[25] Ghantous, G. B.; Gill, A. DevOps: Concepts, Practices, Tools, Benefits and Challenges. In *PACIS*, 2017.

[26] Leite, L.; Rocha, C.; et al. A Survey of DevOps Concepts and Challenges. *ACM Computing Surveys*, volume 52, no. 6, nov 2019: pp. 1–35, doi:10.1145/3359981. Available from: `https://doi.org/10.1145/3359981`

[27] Kanstantsin, Z. Multivocal Literature Review on the Security of DevSecOp. *Asian Journal of Research in Computer Science*, volume 14, no. 2, Jul. 2022: p. 1–9, doi:10.9734/ajrcos/2022/v14i230329. Available from: `https://journalajrcos.com/index.php/AJRCOS/article/view/277`

[28] Rijal, L.; Colomo-Palacios, R.; et al. *AIOps: A Multivocal Literature Review.* Cham: Springer International Publishing, 2022, ISBN 978-3-030-80821-1, pp. 31–50, doi:10.1007/978-3-030-80821-1_2. Available from: `https://doi.org/10.1007/978-3-030-80821-1_2`

[29] Duckworth, C.; Chmiel, F.; et al. Using explainable machine learning to characterise data drift and detect emergent health risks for emergency

department admissions during COVID-19. *Scientific Reports*, volume 11, 11 2021, doi:10.1038/s41598-021-02481-y.

[30] Žliobaitė, I.; Pechenizkiy, M.; et al. *An Overview of Concept Drift Applications*. Cham: Springer International Publishing, 2016, ISBN 978-3-319-26989-4, pp. 91–114, doi:10.1007/978-3-319-26989-4_4. Available from: `https://doi.org/10.1007/978-3-319-26989-4_4`

[31] Zhou, Y.; Yu, Y.; et al. Towards MLOps: A Case Study of ML Pipeline Platform. In *2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*, 2020, pp. 494–500, doi:10.1109/ICAICE51518.2020.00102.

[32] Yildiz, B.; Hung, H.; et al. ReproducedPapers.org: Openly Teaching and Structuring Machine Learning Reproducibility. In *Reproducible Research in Pattern Recognition*, edited by B. Kerautret; M. Colom; A. Krähenbühl; D. Lopresti; P. Monasse; H. Talbot, Cham: Springer International Publishing, 2021, ISBN 978-3-030-76423-4, pp. 3–11.

[33] Bischl, B.; Binder, M.; et al. Hyperparameter Optimization: Foundations, Algorithms, Best Practices and Open Challenges. 2021, `2107.05847`.

[34] Kolajo, T.; Daramola, O.; et al. Big data stream analysis: a systematic literature review. *Journal of Big Data*, volume 6, 06 2019: p. 47, doi:10.1186/s40537-019-0210-7.

[35] Garcia, A. M.; Griebler, D.; et al. Evaluating Micro-batch and Data Frequency for Stream Processing Applications on Multi-cores. In *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2022, pp. 10–17, doi:10.1109/PDP55904.2022.00011.

[36] Databases vs. Data Warehouses vs. Data Lakes. `https://www.mongodb.com/databases/data-lake-vs-data-warehouse-vs-database`, accessed on April 27th, 2023.

[37] Studer, S.; Bui, T. B.; et al. Towards CRISP-ML(Q): A Machine Learning Process Model with Quality Assurance Methodology. 2021, `2003.05155`.

[38] TensorFlow (documentation). Available from: `https://www.tensorflow.org/api_docs`

[39] TensorFlow (documentation). Available from: `https://huggingface.co/docs`

[40] Reijonen, J.; Opsenica, M.; et al. Regression Training using Model Parallelism in a Distributed Cloud. In *2019 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and*

Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), 2019, pp. 741–747, doi:10.1109/DASC/PiCom/CBDCom/CyberSciTech.2019.00139.

[41] Company, M. . How COVID-19 is changing consumer behavior—now and forever. 2020. Available from: `https://www.mckinsey.com/industries/retail/our-insights/how-covid-19-is-changing-consumer-behavior-now-and-forever`

[42] KDNuggets. Ravages of Concept Drift in Stream Learning Applications. `https://www.kdnuggets.com/2019/12/ravages-concept-drift-stream-learning-applications.html`, 2019, accessed on April 25th, 2023.

[43] Gama, J. a.; Žliobaitundefined, I.; et al. A Survey on Concept Drift Adaptation. *ACM Comput. Surv.*, volume 46, no. 4, mar 2014, ISSN 0360-0300, doi:10.1145/2523813. Available from: `https://doi.org/10.1145/2523813`

[44] Deepchecks (documentation). Available from: `https://deepchecks.com/docs/`

[45] Apache Airflow (documentation). Available from: `https://airflow.apache.org/docs/`

[46] DVC (documentation). Available from: `https://dvc.org/doc`

[47] Flyte (documentation). Available from: `https://docs.flyte.org/`

[48] ZenML (documentation). Available from: `https://docs.zenml.io/`

[49] Kedro (documentation). Available from: `https://kedro.readthedocs.io/`

[50] Kubeflow (documentation). Available from: `https://www.kubeflow.org/docs/`

[51] Metaflow (documentation). Available from: `https://docs.metaflow.org/`

[52] MLflow (documentation). Available from: `https://mlflow.org/docs/latest/index.html`

[53] Orchest (documentation). Available from: `https://docs.orchest.io/`

[54] Aim (documentation). Available from: `https://aimstack.readthedocs.io/`

[55] Weights & Biases (documentation). Available from: https://docs.wandb.ai/

[56] CML (documentation). Available from: https://cml.dev/doc

[57] MLEM (documentation). Available from: https://mlem.ai/doc/

[58] ClearML (documentation). Available from: https://clear.ml/docs

[59] Neptune.ai (documentation). Available from: https://docs.neptune.ai/

[60] Pachyderm (documentation). Available from: https://docs.pachyderm.com/

[61] Feast (documentation). Available from: https://docs.feast.dev/

[62] Optuna (documentation). Available from: https://optuna.readthedocs.io/

[63] Label Studio (documentation). Available from: https://labelstud.io/

[64] Evidently (documentation). Available from: https://docs.evidentlyai.com/

[65] Great Expectations (documentation). Available from: https://docs.greatexpectations.io/docs/

[66] BentoML (documentation). Available from: https://docs.bentoml.org/

[67] Ansible (documentation). Available from: https://docs.ansible.com/

[68] Puppet (documentation). Available from: https://www.puppet.com/docs/

[69] Chef (documentation). Available from: https://docs.chef.io/

[70] Merkel, D. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, volume 2014, no. 239, 2014: p. 2.

[71] octplane. Ansible Stdout Compact Logger. https://github.com/octplane/ansible_stdout_compact_logger, 2022.

[72] Riemers. Ansible GitLab Runner. https://github.com/riemers/ansible-gitlab-runner, 2021.

[73] Vagrant (documentation). Available from: https://developer.hashicorp.com/vagrant/docs

[74] GitLab CI/CD (documentation). Available from: https://docs.gitlab.com/ee/ci/

[75] Conda (documentation). Available from: `https://docs.conda.io/`

[76] Mamba (documentation). Available from: `https://mamba.readthedocs.io/`

[77] PyTorch Lightning (documentation). Available from: `https://lightning.ai/docs/pytorch/stable/`

Appendix A

Administrator Guide

The following chapter contains a step-by-step guide for setting up the designed stack on the host machine, enabling its use by the developers, and performing basic operations using the provided playbooks and tools.

Note: To save space, please assume that all file names mentioned in this guide have an implicit prefix of `src/provisioning/` (relative to the root of the attachment).

A.1 Prerequisites

A.1.1 Host Machine

This guide works with the following assumptions:

- The administrator has root-level access to the host machine.
- The host is a physical or virtual machine running on one of the following: Debian 11.6, Ubuntu 20.04 or Ubuntu 22.04.

In situations where the host machine is running on a different version or Linux distribution, there might be issues such as missing dependencies or incompatibility of the tasks within Ansible playbooks. One way to test the compatibility with a given OS is to provision a box using Vagrant [73] and try to deploy the server there. The same process was applied to verify the compatibility with the distributions mentioned above. We provide a simple Vagrantfile that spins up these instances in `extras/Vagrantfile`.

A.1.2 Virtual Environment

To prevent conflicts with your base environment, we recommend creating a new one using your preferred environment management system. Ensure both Python and Ansible are present in the environment.

A.2 Configuration

Before you run the `run_server.yml` playbook, ensure that you have undergone the following configuration steps.

A.2.1 Ansible

1 Download all the requirements used by our playbooks by running the following command:

```
ansible-galaxy install -r requirements.yml
```

2 Our implementation includes a very minimal `ansible.cfg` file that overrides some of the default settings. Depending on your use case, you might find it useful to override additional options. Try running the command `ansible-config init --disabled` to see the default configuration file and check if there are any options you would like to override. If so, either copy them into the provided configuration file, or pipe the output of the command into a new one. Do not forget to set the value of `inventory` to point to the correct inventory file.

3 Ensure that the user account under which you want to execute commands on the server exists. If you want to use SSH keys to communicate with the server, ensure the public key is located in the corresponding folder on the server and that you have access to the private key.

4 Fill in the required information about the host machine in the Ansible inventory file (`inventory.yml`). You may change the `vm1` to a desired name, this value is mostly used for naming purposes (e.g. MinIO server alias, ZenML stack names).

```
vm1:
    ansible_host: <enter the IP address of your VM>
    ansible_port: <leave empty unless not using port 22>
    ansible_user: <enter the executing account username>
    ansible_password: <leave empty if using PK>
    ansible_ssh_private_key_file: <path to the PK>
```

A.2.2 General

This section refers to the `stack.yml` configuration file.

1 If you wish to use a custom root username and/or password, uncomment and fill in the corresponding values. If you leave the username commented out, it will default to `root`. If you leave the password commented out, a random password will be generated and placed into `credentials/root.passwd`. If a (root) password file exists, it is not regenerated.

2 Add the `ansible_user`'s username into the `docker_users` list. This will allow the user to execute Docker commands without needing elevated privileges.

3 Add any Pip packages to the list as needed. Do not remove any existing ones, as you may break the playbooks. If needed, you may modify the versions or try using the latest.

4 Add any additional shared buckets that you want to create automatically. Modify the default values for developer buckets. Using the MinIO client or web server, you can add new buckets afterwards.

A.2.3 Runners

This section refers to the `runners.yml` configuration file.

1 The included `runners.yml` is a subset of all available options. To get the original file, visit the original GitHub repository [72] and use the contents of the `defaults/main.yml` file.

2 Set the `gitlab_runner_coordinator_url` with the URL of the GitLab instance, e.g. `https://gitlab.com` or `https://gitlab.fit.cvut.cz`.

3 Obtain the GitLab Runner registration token for your instance, group or project. Set the `gitlab_runner_registration_token` to this value. For more info on how to obtain the registration token, refer to GitLab's documentation [74].

4 Add or remove individual runners in the `gitlab_runner_runners` section as needed. Use the comments to aid you or get inspired by the default values.

Note that GitLab has announced the removal of the registration token architecture in a future release, which will introduce a breaking change. The author will likely update the role to correspond to the new system, but this cannot be guaranteed. Everything should remain functional if your instance runs a version lower than 17.0.

A.3 Server Provisioning

Once you have successfully configured all four configuration files, you can run the `run_server.yml` playbook. Ansible will start setting up the infrastructure and logging the progress to standard output. Once finished, all services should be running and two users should be registered on the server: the root user and the administrator.

A.4 GitLab Configuration

Due to the workaround related to ZenML's lacking enforcement of user privileges, you must expose the credentials to the ZenML Server using GitLab's masked variables, which you can find under `Settings > CI/CD > Variables`. Make sure you mark the variable as masked to obfuscate it in developers' build logs.

If you are using the default GitLab CI/CD pipeline file, define the following variables:

- `ZENML_ROOT_USERNAME`: the name of the ZenML root account (this does not necessarily have to be masked, an short usernames such as `root` cannot be masked in GitLab anyway),
- `ZENML_ROOT_PASSWORD`: the password of the ZenML root account.

A.5 User Management

The management of users is possible using the `manage_users.yml` playbook.

To add or delete a user, run the playbook with the following parameters:

```
ansible-playbook manage_users.yml \
--extra-vars "username=[GITLAB_USERNAME] role=[ROLE]" \
--tags "[OP]"

[GITLAB_USERNAME]: The username as it appears on GitLab.
[ROLE]: One of 'developer', 'administrator'.
[OP]: One of 'add', 'create' to add; 'remove', 'delete' to remove.
```

If you try to do an invalid operation (e.g. delete a non-existing user or create a duplicate), the execution will abort due to an existing/missing entry in the `stack_management` database schema.

Once the playbook finishes execution, the user will be ready to configure their stack according to Section B.4.

A.6 Experiment Inspection

As an administrator, you have unlimited access to all resources on the server. This includes everyone's database schema that stores run metadata and the storage bucket that stores artifacts. To run a tracking server connected to a developer's resources, you only need to do the following:

1 Prior to running the script, make sure to export the following variables:

   - `MY_ACCESS_KEY`: your assigned username on the server,
   - `MY_SECRET_KEY`: your password provided by the administrator,

- `SERVER_IP`: the IP address of the host machine.

2 Run the provided script **`extras/mlflow_track_user.sh`** and pass the username (access key) of the target developer as the first argument, such as:

```
extras/mlflow_track_user.sh bacigmic
```

This command can be used identically to access your own resources by replacing the developer's username with your own.

A.7 Direct Pipeline Executions

The stack currently contains a local orchestrator, meaning that if you need to run a pipeline directly on the host machine, you would have to copy the files to the host and execute them according to the local or hybrid deployment scenario in Sections B.2 and B.3, respectively. Enabling direct access to the host from a remote machine to avoid the overhead of pushing the code to GitLab would require running at least an Airflow server on the host machine, which would introduce additional resource requirements. If the need to perform this action is only occasional, follow the instructions mentioned above.

Appendix B

Developer Guide

This section provides a step-by-step guide to configuring one's machine and project repository to work with the proposed stack in three distinct scenarios described in the following table (Table B.1):

Scenario	local (B.2)	hybrid (B.3)	remote (B.4)
The computing resources are provided by the...	user's machine	user's machine	host machine
Artifacts and metadata are stored in the...	user's machine	remote storage	remote storage

Table B.1: The three different scenarios of stack interaction.

Please note that this guide is not meant to explain the process of building pipelines. You can inspect the provided example or read about it in the corresponding documentation. This guide focuses on the configuration aspect of the stack.

B.1 Prerequisites

Virtual Environment It is highly recommended to use Conda [75] or Mamba [76] as your top-level virtual environment manager. The pre-defined GitLab CI/CD pipeline is configured to install the required packages from a Conda-compatible environment file located in the `environments` subfolder of your project root. You can find an example of this file in the tool's documentation or the `src/demo/environments` folder in the thesis attachment. Following this approach will alleviate the need to modify the CI/CD pipeline file. If you need to make changes, find the corresponding line and alter it to your needs.

Libraries The minimum requirement for working with the stack as designed is to install Python, ZenML and MLflow.

B.2 Local Development

This scenario allows developers to write and execute their code offline without relying on remote resources. ZenML fully enables this by utilizing file-based databases and folders on the local filesystem.
To set up your machine for local development, follow these steps:

1 Create and activate a new virtual environment.

2 Install the required libraries mentioned in Section B.1.

3 Create a new directory for your project.

   - Initialize a new ZenML repository using `zenml init`.
   - Alternatively, for demonstration purposes, you can quickly set up an example project by installing the extra dependencies for ZenML:

     ```
     pip install zenml[templates] && zenml init --template
     ```

     This will also allow you to explore the recommended directory structure for ZenML projects. However, these projects do not include an experiment tracking component.

4 Register the proposed stack with ZenML by running the provided script in `src/demo/stacks/local.sh`. This will install the integration for MLflow and register a local equivalent of the designed stack. ZenML uses a local database to store registered stacks, which can be used across projects. You may, however, need to activate the desired stack when switching to a different virtual environment using the `zenml stack set` command.

5 At this point, the initial configuration process is over. You should be able to run any ZenML pipeline that includes the three registered components.

We will now demonstrate the local workflow on a simple MNIST classification task, adapted from PyTorch Lightning [77] and ZenML [48].

1 Copy the contents of the example project in `src/demo` to the previously created folder, or create a new one and repeat the ZenML repository creation process.

2 Update your virtual environment using the YAML configuration file in the `environments` subfolder or create a new one.

3 Ensure that you have selected our `local_example_stack` by running the `zenml stack get` command.

4 Run the pipeline using `python run_mnist.py`.

5 Once the pipeline has finished running, you should be able to inspect your outputs:

   5.1 ZenML will automatically prompt you to run the `zenml up` command, which will start a local dashboard server where you can inspect your pipeline run. It also has many interesting features you can try. In the remote deployment scenario, you will not be able to access this dashboard due to the limitations we previously discussed.

   5.2 In the case of local MLflow deployments, by default, ZenML stores the runs in a very obscurely named folder which may be difficult to find manually. However, you can add the following code to the script you use to run the pipeline:

```
from zenml.integrations.mlflow.mlflow_utils import (
    get_tracking_uri
)
print(f"mlflow ui --backend-store-uri '{get_tracking_uri()}'")
```

   This will print out the command that can be used to launch the MLflow UI and inspect the past runs of this specific pipeline.

The output for the example project should look like this:

```
Pipeline run mnist_pipeline-2023_05_03-20_21_05_888897 has finished in 26.278s.
Pipeline visualization can be seen in the ZenML Dashboard.
Run zenml up to see your pipeline!
mlflow ui --backend-store-uri 'file:/...(some long path).../mlruns'
```

You can use the provided commands to run the dashboard and access the tracking server.

B.2.1 Advanced Configuration

One of the advantages of local development is that you can try out all the features of ZenML without being limited by the server's configuration. We strongly recommend you visit the ZenML Documentation [48] and explore the possibilities of adding additional stack components based on your needs, using a different orchestrator flavor, or getting inspired by more advanced examples of ZenML usage, some of which we describe in Section B.5.

B.3 Hybrid Development

In this scenario, you will execute the pipelines locally but store the artifacts and metadata in your designated remote storage. This is essentially what happens inside job containers on the remote server.

To successfully set up this scenario, follow these steps:

1 Ensure you have executed the first three steps mentioned in Section B.2.

2 If you have not registered the hybrid stack yet, you can do so by running the provided script in `src/demo/stacks/hybrid.sh`.

3 In this case, ZenML does not handle the provisioning of the tracking server. You must do it manually by running the script in `src/demo/scripts/mlflow_tracking.sh`.
Prior to running the script, make sure to export the following variables:

  - `MY_ACCESS_KEY`: your assigned username on the server,
  - `MY_SECRET_KEY`: your password provided by the administrator,
  - `SERVER_IP`: the IP address of the host machine.

  This spins up an instance of the MLflow Tracking Server, which connects to your remote bucket and database schema. Your registered hybrid stack's experiment tracking component is configured to point to this server running on your local machine and will use push artifacts to the remote server.

The remainder of the process is identical to the previous scenario. You can manage your artifacts by accessing the MinIO Console on the host server's port 9090 and logging in with your credentials, or directly via the API available on port 9000 (the use of MinIO's `mc` client is recommended). You can also access your database schema in MySQL directly by using an SQL client and logging in on port 3306, however, the raw data might not be of much benefit to you.

B.4 Remote Development

In this final scenario, you will execute pipelines by pushing commits to a GitLab repository. Each commit will request a job from the registered runners, which will pick up the job when a free slot is available, and execute it using its allocated resources.

To properly configure your GitLab project, follow these steps:

1 Fill in the `conda.yaml` and `requirements.txt` files accordingly based on your project environment, as shown in `src/demo/environments`. Do not delete the `pip` section in the former, it is used to automatically install the packages from the latter file using pip.

2 If you use the provided GitLab CI/CD pipeline definition file, set the following environment variables in your project (you can find them in `Settings > CI/CD > Variables`):

- `MY_ACCESS_KEY`: your assigned username on the server,
- `MY_SECRET_KEY`: your password provided by the administrator,
- `ZENML_ENTRYPOINT`: the command used to run the pipeline (in our case, it is `python run_mnist.py`). The reason for the use of this environment variable is to avoid altering the pipeline definition file, if possible.

Alternatively, you can embed the values in the pipeline definition file directly. This is only a security measure.

3 At this point, you can execute a pipeline by pushing a commit or manually in the `CI/CD > Pipelines` section. You should see the pipeline start as soon as a slot is available. The outputs of the pipeline will be pushed to your schema and bucket.

4 Use the provided script in `src/demo/scripts/mlflow_tracking.sh` to run a tracking server that connects to your resources.

B.5 Additional Remarks

Due to the unanticipated scope of the task at the time of assignment, resulting from unexpected workarounds and the broadness of the tool offering on the market, the supplied project used for demonstration does not properly reflect all the capabilities provided by the stack. While its purpose was primarily as a quick way to demonstrate the configuration, we feel that we should, at the very least, provide the reader with suggestions on the next steps in exploring everything the selected (or reviewed) tools offer.

Luckily, some of these tools already have a corresponding stack component implemented in ZenML and can be easily added by following the instructions in the documentation. As for DVC and Optuna, ZenML has already added them to their roadmap, and it is likely that their corresponding stack components will be implemented in the following months, making their use even easier.

B.5.1 Data Versioning with DVC

First, we suggest obtaining a (tabular) data set that you can easily edit (add or remove items) and install DVC. As previously reviewed, it offers Git-like commands to version data that does not normally fit into Git repositories. The company which maintains DVC has multiple video tutorials demonstrating the process of working with DVC — all you need is a compatible storage source. For simple use cases, you store your files using Google Drive.

B.5.2 Data Validation with Deepchecks

If your data set from the previous step has not been already cleaned, you can try out the data validation feature of Deepchecks (or another compatible data validation flavor) using ZenML's built-in Data Validator component. ZenML provides built-in steps that you can import and use after registering the component and activating a stack that deploys it (use the included scripts and ZenML's CLI as inspiration). Optionally, you can download a ”toy data set” from a website like Kaggle.

B.5.3 Hyperparameter Optimization with Optuna

There is a readily available example of how to use Optuna in your ZenML pipelines[1]. As there is no corresponding stack component, you must manually set up Optuna, but all information is available in the GitHub repository.

B.5.4 Model Registry and Deployment

While you will probably not be deploying the model to a Kubernetes cluster just for demonstration purposes, using MLflow within our stack enables you to deploy the trained model to a local MLflow server. You can enable this by registering the MLflow flavors of the Model Registry and Model Deployer components and following the documentation which explains how to integrate it into your code.

[1]https://github.com/dnth/zenml-optuna

Appendix **C**

List of Abbreviations

AI	artificial intelligence
API	application programming interface
CI/CD	continuous integration / continuous delivery
CLI	command line interface
DAG	directed acyclic graph
DB	database
DevOps	development and operations
GPU	graphics processing unit
IDE	integrated development environment
HPO	hyper-parameter optimization
ML	machine learning
MLOps	machine learning operations
NLP	natural language processing
OS	operating system
UI	user interface
VM	virtual machine
W&B	Weights & Biases

Appendix D

Enclosed Media Contents

```
README.txt ........... the rendering of the directory structure in ASCII
src ...................................... implementation-related files
  demo ......................... the example project used in the guides
  provisioning ................ implementation of server provisioning
text ................................................ text-related files
  DP_Bacigal_Michal_2023.pdf .......... a PDF rendering of the thesis
  thesis_assignment.pdf ... a PDF rendering of the thesis assignment
    src ........................... the source files for the thesis text
      assets ............................. images used in the thesis
      sections .................. individual chapters in TeX format
```