MLOps Engineering

Building, Deploying, and Managing Machine Learning Workflows with Airflow and MLflow on Kubernetes.

Sebastian Blum

2023-08-05

Contents

Summary

MLOps Engineering: Building, Deploying, and Managing Machine Learning Workflows with Airflow and MLflow on Kubernetes" is a comprehensive guide to understanding the principles and roles of MLOps, as well as operational principles such as CI/CD and versioning. Through the use of an exemplary MLOps platform utilizing Airflow, MLflow, and JupyterHub on AWS EKS, readers will learn how to deploy this platform using infrastructure as code Terraform. The book is structured with chapters that introduce the various tools such as Airflow, MLflow, Kubernetes, and Terraform, and also includes a use case that demonstrates the utilization of the platform through Airflow workflows and MLflow tracking. Whether you are a data scientist, engineer, or developer, this book provides the necessary knowledge and skills to effectively implement MLOps practices and platforms in your organization.

Preamble

This project started out of an interest in multiple domains. At first, I was working on a Kubeflow platform at the time and haven't had much experience in the realm of MLOps. I was starting with K8s and Terraform and was interested to dig deeper. I did so by teaching myself and I needed a project. What better also to do than to build "my own" MLOps plattform. I used Airflow since it is widely uses for workflow management and I also wanted to use it from the perspective of a data scientist, meaning to actually build some pipelines with it. This idea of extending the project to actually have a running use case expanded this work to include MLFlow for model tracking.

Topics

The overall aim is to build and create a MLOps architecture based on Airflow running on AWS EKS. Ideally, this architecture is create using terraform. Model tracking might be done using MLFlow, Data tracking using DVC. Further mentioned might be best practices in software development, CI/CD, Docker, and pipelines. I might also include a small Data Science use case utilizing the Airflow Cluster we built.

A work in progress

This project / book / tutorial / whatever this is or will be, startet by explaining the concept of Kubernetes. The plan is to continuously update it by further sections. Since there is no deadline, there is no timeline, and I am also not sure whether there will exist something final to be honest.

This document is written during my journey in the realm of MLOps. It is therefore in a state of continuous development.

1 Introduction

Imagine being able to effortlessly deploy, manage, and monitor your machine learning models with ease. No more headaches from version control issues, data drift, and model performance degradation. That's the power of MLOps. *"MLOps Engineering: Building, Deploying, and Managing Machine Learning Workflows with Airflow and MLflow on Kubernetes"* takes you on a journey through the principles, practices, and platforms of MLOps. You'll learn how to create an end-to-end pipeline for machine learning projects, using cutting-edge tools and techniques like Kubernetes, Terraform, and GitOps, and working with tools to ease your machine learning workflow such as Apache Airflow and MLflow Tracking. Before we begin, let's have a more closer look on what MLOps actually is, what principles it incorporates, and how it distinguished from traditional DevOps.

1.1 Machine Learning Workflow

A machine learning workflow typically involves several stages. These stages are closely related and sometimes overlap as some stages may involve multiple iterations. In the following, the machine learning workflow is broken down to five different stages to make things easier, and give an overview.

Data Preparation → Model Building → Model Evaluation → Model Deployment → Model Monitoring and...
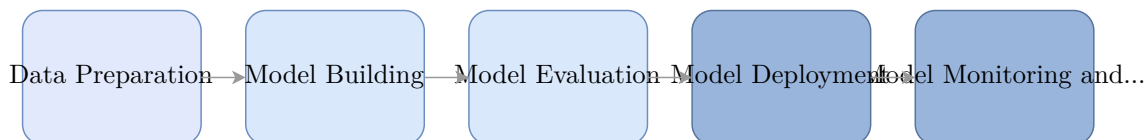
Figure 1.1: ML lifecycle

1. Data Preparation In the first stage, data used to train a machine learning model is collected, cleaned, and preprocessed. Preprocessing includes tasks to remove missing or duplicate data, normalize data, and split data into a training and testing set.

2. Model Building In the second stage, a machine learning model is selected and trained using the prepared data. This includes tasks such as selecting an appropriate algorithm as a

machine learning model, training the model, and tuning the model's parameters to improve its performance.

3. Model Evaluation Afterward, the performance of the trained model is evaluated using the test data set. This includes tasks such as measuring the accuracy and other performance metrics, comparing the performance of different models, and identifying potential issues with the model.

4. Model Deployment Finally, the selected and optimized model is deployed to a production environment where it can be used to make predictions on new data. This stage includes tasks like scaling the model to handle large amounts of data, and deploying the model to different environments to be used in different contexts

5. Model Monitoring and Maintenance It is important to monitor the model performance and update the model as needed, once the model is deployed. This includes tasks such as collecting feedback from the model, monitoring the model's performance metrics, and updating the model as necessary.

Each stage is often handled by the same tool or platform which makes a clear differentiation across stages and tools fairly difficult. Further, some machine learning workflows will not have all the steps, or they might have some variations. A machine learning workflow is thereby not a walk in the park and the actual model code is just a small piece of the work.

Configuration Data Collection Data Verification Machine Resource M Monitoring ML Code Analysis Tools Serving Infrastr... Feature Extraction Process Management Tools

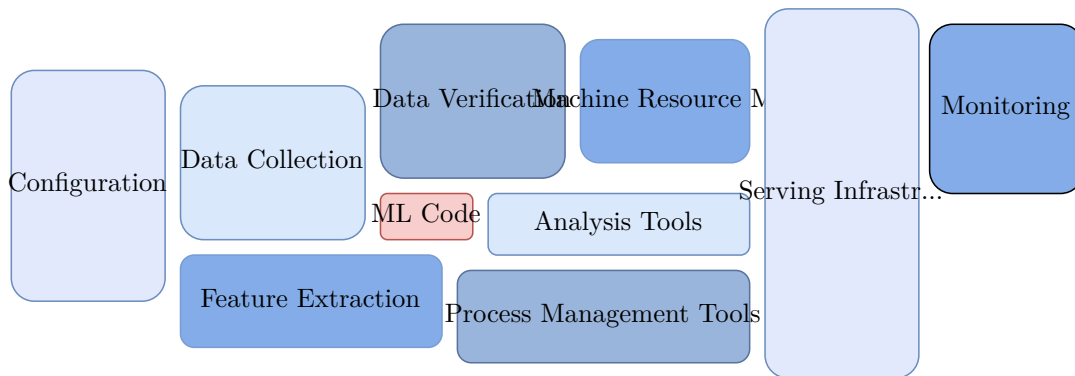Figure 1.2: Only a small fraction of real-world ML systems is composed of the ML code, as shown by the small black box in the middle. The required surrounding infrastructure is vast and complex. (D. Sculley, et al., 2015)

Working with and developing machine learning models, monitoring their performance, and continuously retraining it on new data with possible alternative models can be challenging and involves the right tools.

1.1.1 ML Worflow Tools

There are several machine learning workflow tools that can help streamline the process of building and deploying machine learning models. By Integrating them into the machine learning workflow there can be three main processes and functionalities of tools derived: (1) *Workflow Management*, (2) *Model Tracking*, and (3) *Model Serving*. All three of these processes are closely related to each other and are often handled by the same tool or platform.

1.1.1.1 Workflow Management

Workflow Management is the process of automating and streamlining the stages involved in building, training, and deploying machine learning models. This includes tasks such as data preprocessing, model training, and model deployment. Workflow management tools allow for the coordination of all the necessary stages in the machine learning pipeline, and the tracking of the pipeline's progress.

Apache Airflow is an open-source platform for workflow management and is widely used to handle ETL-processes of data, or automate the training and deployment of machine learning models.

1.1.1.2 Model Tracking

Model Tracking is the process of keeping track of different versions of a machine learning model, including their performance, and the parameters and data used to train it. Model Tracking tools are often used at the development and testing stages of the machine learning workflow. During development, tracking allows to keep track of the different versions of a model, compare their performance and learning parameters, and finally help to select the best version of the model to deploy. Model tracking also allows to check the performance of a machine learning model during testing, and to assure it meets industry requirements.

MLflow is an open-source platform to manage the machine learning lifecycle, including experiment tracking, reproducibility, and deployment. Similarly, *DVC* (Data Version Control) is a tool that allows to version control, manage, and track not only models but also data.

1.1.1.3 Model Serving

Model Serving refers to the process of deploying a machine learning model in a production environment, so it can be used to make predictions on new data. This includes tasks such as scaling the model to handle large amounts of data, deploying the model to different environments, and monitoring the performance of the deployed model. Model serving tools are

specifically used at the deployment stage of the machine learning workflow and can handle the necessary tasks mentioned beforehand.

There are multiple tools that integrate the funtionality of serving models, each different in its specific use cases, for example *TensorFlow*, *Kubernetes*, *DataRobot*, or also the already mentioned tools *MLflow* and *Airflow*.

1.1.2 Developing Machine Learning Models

In the development phase of a machine learning model, many stages of the machine learning workflow are carried out manually. For instance, testing machine learning code is often done in a notebook or script, rather than through an automated pipeline. Similarly, deploying the model is typically a manual process, and only involves serving the model for inference, rather than deploying an entire machine learning system.

This manual deployment process can result in infrequent deployments and the management of only a few models that do not change frequently. Additionally, the deployment process is usually not handled by Data Scientists but is instead managed by an operations team. This can create a disconnection between the development and production stages, and once deployed, there is typically no monitoring of the model's performance, meaning no feedback loop is established for retraining.

Performing stages manually and managing models in production at scale can become exponentially more challenging. To overcome these issues, it is recommended to adopt a unifying framework for production, known as MLOps. This framework will shift the focus from managing existing models to developing new ones, making the process more efficient and effective.

1.2 Machine Learning Operations (MLOps)

Machine Learning Operations (MLOps) combines principles from developing and deploying machine learning models to enable an efficient operation and management of machine learning models in a production environment. The goal of MLOps is to automate and streamline the machine learning workflow as much as possible, while also ensuring that each machine learning model is performing as expected. This allows organizations to move quickly and efficiently from prototyping to production.

The infrastructure involved in MLOps includes both, hardware and software components. Hardware components aim to make the deployment of machine learning models scalable. This includes servers and storage devices for data storage and model deployment, as well as specialized hardware such as GPUs for training and inferencing. Software components

include version control systems, continuous integration and continuous deployment (CI/CD) tools, containerization and orchestration tools, and monitoring and logging tools. There are several cloud-based platforms that provide pre-built infrastructure to manage and deploy models, automate machine learning workflows, and monitor and optimize the performance of models in production. For example AWS SageMaker, Azure Machine Learning, and Google Cloud AI Platform.

Overall, the key concepts and best practices of MLOps, and the use of tools and technologies to support the MLOps process are closely related to *regular* DevOps principles which are a standard for the operation of software in the industry.

1.2.1 ML + (Dev)-Ops

Machine Learning Operations (MLOps) and Development Operations (DevOps) both aim to improve the efficiency and effectiveness of software development and deployment. Both practices share major similarities, but there are some key differences that set them apart.

The goal of DevOps is to automate and streamline the process of building, testing, and deploying software, to make the management of the software lifecycle as quick and efficient as possible. This can be achieved by using tools, such as:

1. *Containerization*: Tools such as Docker and Kubernetes are used to package and deploy applications and services in a consistent and reproducible manner.
2. *Version Control*: Tools such as Git, SVN, and Mercurial are used to track and manage changes to code and configuration files, allowing teams to collaborate and roll back to previous versions if necessary.
3. *Continuous Integration and Continuous Deployment (CI/CD)*: Tools such as Jenkins, Travis CI, and Github Actions are used to automate the building, testing, and deployment of code changes.
4. *Monitoring and Logging*: Tools such as Prometheus, Grafana, and ELK are used to monitor the health and performance of applications and services, as well as to collect and analyze log data.
5. *Cloud Management Platforms*: AWS, Azure, and GCP are used to provision, manage, and scale infrastructure and services in the cloud. Scaling and provisioning infrastructure can be automated by using Infrastructure as Code (IaC) Tools like Terraform.

DevOps focuses on the deployment and management of software in general (or *traditional* sofware), while MLOps focuses specifically on the deployment and management of machine learning models in a production environment. The goal is basically the same as in DevOps, yet deploying a machine learning model. While this is achieved by the same tools and best

practices used in DevOps, deploying machine learning models (compared to software) adds a lot of complexity to the process.

Traditional software
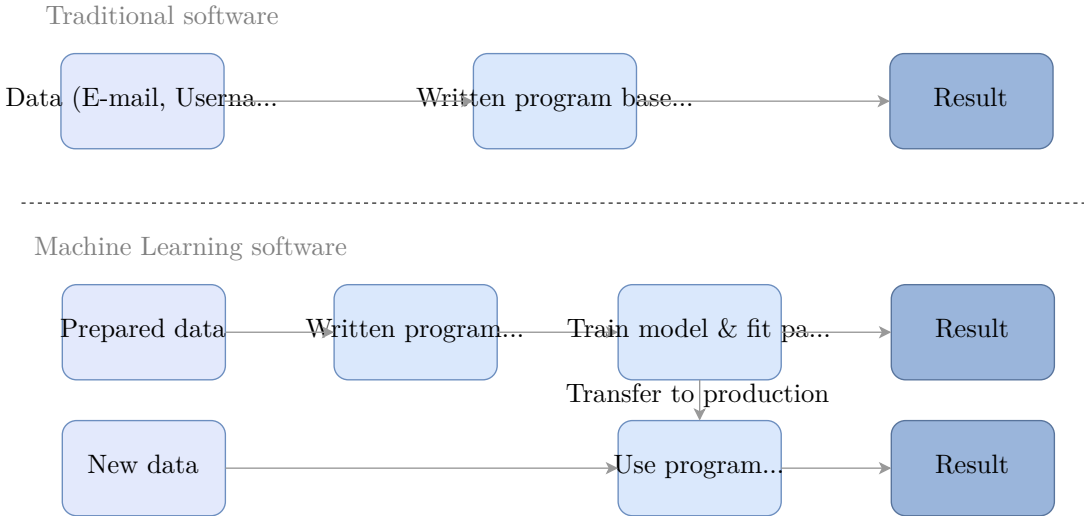
Machine Learning software

Figure 1.3: Traditional vs ML software

Machine learning models are not just lines of code, but also require large amounts of data, and specialized hardware, to function properly. Further, machine learning models and their complex algorithms might need to change when there is a shift in new data. This process of ensuring that machine learning models are accurate and reliable with new data leads to additional challenges. Another key difference is that MLOps places a great emphasis on model governance, which ensures that machine learning models are compliant with relevant regulations and standards. The above list of tools within DevOps can be extended to the following for MLOps.

1. *Machine Learning Platforms*: Platforms such as TensorFlow, PyTorch, and scikit-learn are used to develop and train machine learning models.
2. *Experiment Management Tools*: Tools such as MLflow, Weights & Biases, and Airflow are used to track, version, and reproduce experiments, as well as to manage the model lifecycle.
3. *Model Deployment and Management Tools*: Tools such as TensorFlow Serving, Clipper, and Seldon Core are used to deploy and manage machine learning models in production.
4. *Data Versioning and Management Tools*: Tools such as DVC (Data Version Control) and Pachyderm are used to version and manage the data used for training and evaluating models.

5. *Automated Model Evaluation and Deployment Tools*: Tools such as AlgoTrader and AlgoHub are used to automate the evaluation of machine learning models and deploy the best-performing models to production.

It's important to note that the specific tools used in MLOps and DevOps may vary depending on the organization's needs. Some of the tools are used in both but applied differently in each, e.g. container orchestration tools like Kubernetes. The above lists do also not claim to be complete and there are of course multiple more tools.

1.2.2 MLOps Lifecycle

Incorporating the tools introduced by DevOps and MLOps can extend the machine learning workflow outlined in the previous section, resulting in a complete MLOps lifecycle that covers each stage of the machine learning process while integrating automation practices.
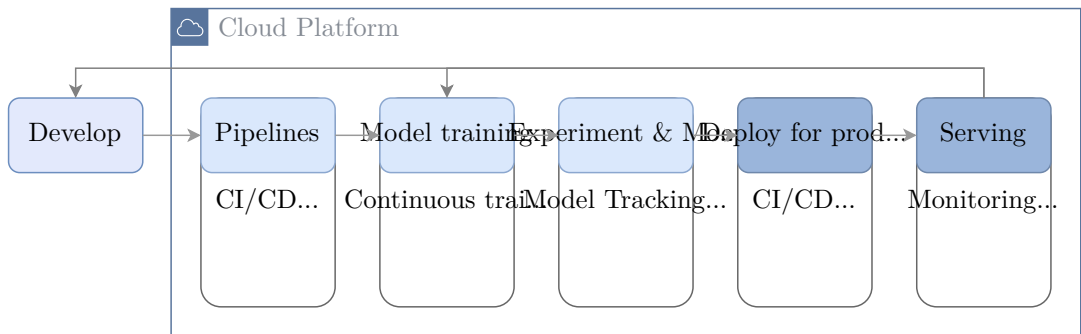
Integrating MLOps into machine learning projects introduces additional complexity into the workflow. Although the development stage can be carried out on a local machine, subsequent stages are typically executed within a cloud platform. Additionally, the transition from one stage to another is automated using tools like CI/CD, which automate testing and deployment.

MLOps also involves integrating workflow management tools and model tracking to enable monitoring and ensure reproducible model training. This enables proper versioning of code, data, and models, providing a comprehensive overview of the project.

1.2.3 MLOps Engineering

MLOps Engineering is the discipline that applies the previously mentioned software engineering principles to create the necessary development and production environment for machine learning models. It usually comines the skills and expertise of Data Scientists and -Engineers,

Machine Learning Engineers, and DevOps engineers to ensure that machine learning models are deployed and managed efficiently and effectively.

One of the key aspects of MLOps Engineering is infrastructure management. The infrastructure refers to the hardware, software and networking resources that are needed to support the machine learning models in production. The underlying infrastructure is crucial for the success of MLOps as it ensures that the models are running smoothly, are highly available and are able to handle the load of the incoming requests. It also helps to prevent downtime, ensure data security and guarantee the scalability of the models.

MLOps Engineering is responsible for designing, building, maintaining and troubleshooting the infrastructure that supports machine learning models in production. This includes provisioning, configuring, and scaling the necessary resources like cloud infrastructure, Kubernetes clusters, machine learning platforms, and model serving infrastructure. It is useful to use configuration management and automation tools like Infrastructure as Code (e.g. Terraform) to manage the infrastructure in a consistent, repeatable and version controlled way. This allows to easily manage and scale the infrastructure as needed, and roll back to previous versions if necessary.

It is important to note that one person can not exceed on all of the above mentioned tasks of designing, building, maintaining, and troubleshooting. Developing the infrastructure for machine learning models in production usually requires multiple people working with different and specialized skillsets and roles.

1.3 Roles and Tasks in MLOps

The MLOps lifecycle typically involves several key roles and responsibilities, each with their own specific tasks and objectives.

1.3.1 Data Engineer

A Data Engineer designs, builds, maintains, and troubleshoots the infrastructure and systems that support the collection, storage, processing, and analysis of large and complex data sets. Tasks include designing and building data pipelines, managing data storage, optimizing data processing, ensuring data quality, and monitoring the necessary data pipelines and systems. They usually work closely with Data Scientists to understand their data needs and ensure the infrastructure supports their requirements.
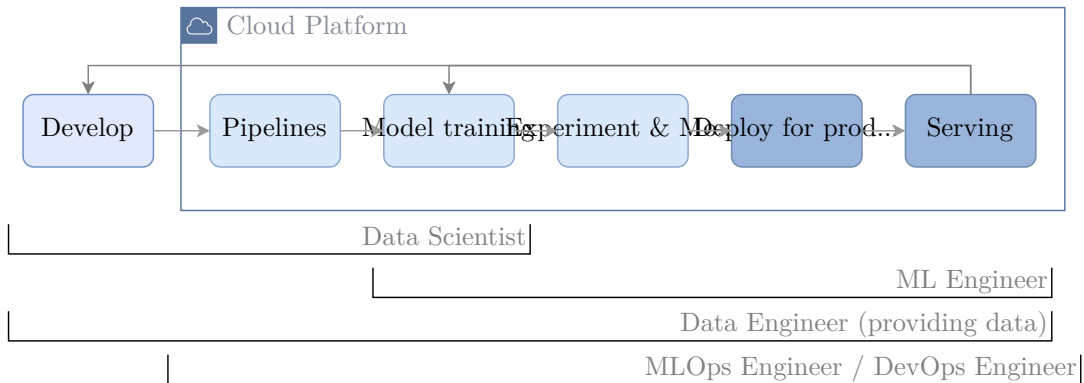
Figure 1.4: Roles and their operating areas within the MLOps lifecycle

1.3.2 Data Scientist

Data Scientists are usually responsible for developing and testing machine learning models within the development stage. Their work typically involves investigating large amounts of data, the necessary preprocessing steps, and choosing a suitable machine learning algorithm that solves the business need. They also develop a functioning ML model respective to the data.

1.3.3 ML Engineer

Machine Learning (ML) Engineers work closely with Data Scientists to develop and deploy machine learning models in a production environment. They are responsible for creating and maintaining the infrastructure and tools needed to run machine learning models at scale and in production. They are also responsible for the day-to-day management and monitoring of machine learning models in a production environment. They use tools and technologies to track model performance and make sure that models are running smoothly and produce accurate results. This also includes taking measures in case of data or model shifts[1].

1.3.4 MLOps Engineer

The MLOps Engineer is responsible to create and maintain the infrastructure and tooling needed to train and deploy models, to monitor the performance of deployed models. They also

---

[1]Model shift refers to the change of a deployed model compared to the model developed in a previous stage, while data shift refers to a change in the distribution of input data compared to the data used during development and testing.

implement processes for collaboration and version control. Overall, this includes tasks such as setting up and configuring the necessary hardware and software environments, creating and maintaining documentation, and implementing automated testing and deployment processes. An MLOps Engineer must be able to navigate this complexity and ensure that the models are deployed and managed in a way that is both efficient and effective.

1.3.5 DevOps Engineer

DevOps Engineers are responsible for automating and streamlining the process of building, testing, and deploying software. They use best practices from software development and operations, such as version control, continuous integration and delivery (CI/CD), and monitoring and observability, to ensure that the software is performing as expected in production.

1.3.6 Additional roles & function

The previous roles only show a small portion of all contributors within data projects. Additional roles within an industry context might also include *Model Governance*, which is responsible to ensure that models are accurate, reliable, and compliant with relevant regulations and standards, or *Business Stakeholders* who provide feedback and requirements for the models in a business context.

It's also worth noting that some of these roles may overlap and different organizations may have different ways of structuring their teams and responsibilities. The most important thing is to have clear communication and collaboration between all the different teams, roles, and stakeholders involved in the MLOps lifecycle.

1.4 Ops Tools & Principles

MLOps integrates a range of DevOps techniques and tools to enhance the development and deployment of machine learning models. By promoting cooperation between development and operations teams, MLOps strives to improve communication, enhance efficiency, and reduce delays in the development process. Advanced version control systems can be employed to achieve these objectives.

Automation plays a significant role in achieving these goals. For instance, CI/CD pipelines streamline repetitive tasks like building, testing, and deploying software. The management of infrastructure can also be automated, by using infrastructure as code to facilitate an automated provisioning, scaling, and management of infrastructure.

To enhance flexibility and scalability in the operational process, containers and microservices are used to package and deploy software. Finally, monitoring and logging tools are used to track the performance of deployed and containerized software and address any issues that arise.

1.4.1 Containerization

Containerization is an essential component in operations as it enables deploying and running applications in a standardized, portable, and scalable way. This is achieved by packaging an application and its dependencies into a container image, which contains all the necessary code, runtime, system tools, libraries, and settings needed to run the application, isolated from the host operating system. Containers are lightweight, portable, and can run on any platform that supports containerization, such as Docker or Kubernetes.

All of this makes them beneficial compared to deploying an application on a virtual machine or traditionally directly on a machine. Virtual machines would emulate an entire computer system and require a hypervisor to run, which introduces additional overhead. Similarly, a traditional deployment involves installing software directly onto a physical or virtual machine without the use of containers or virtualization. Not to mention the lack of portability of both.
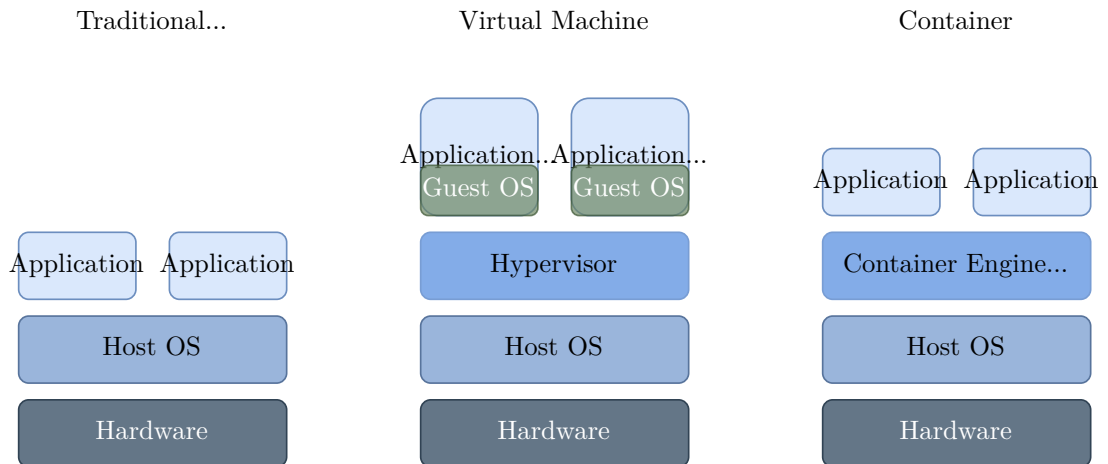
The concept of container images is analogous to shipping containers in the physical world. Like shipping containers can be loaded with different types of cargo, a container image can be used to create different containers with various applications and configurations. Both the physical containers and container images are standardized, just like blueprints, enabling multiple operators to work with them. This allows for the deployment and management of

applications in various environments and cloud platforms, making containerization a versatile solution.

Containerization offers several benefits for MLOps teams. By packaging the machine learning application and its dependencies into a container image, reproducibility is achieved, ensuring consistent results across different environments and facilitating troubleshooting. Containers are portable which enables easy movement of machine learning applications between various environments, including development, testing, and production. Scalability is also a significant advantage of containerization, as scaling up or down compute resources in an easy fashion allows to handle large-scale machine learning workloads and adjust to changing demand quickly. Additionally, containerization enables version control of machine learning applications and their dependencies, making it easier to track changes, roll back to previous versions, and maintain consistency across different environments. To effectively manage model versions, simply saving the code into a version control system is insufficient. It's crucial to include an accurate description of the environment, which encompasses Python libraries, their versions, system dependencies, and more. Virtual machines (VMs) can provide this description, but container images have become the preferred industry standard due to their lightweight nature. Finally, containerization facilitates integration with other DevOps tools and processes, such as CI/CD pipelines, enhancing the efficiency and effectiveness of MLOps operations.

1.4.2 Version Control

Version control is a system that records changes to a file or set of files over time, to be able to recall specific versions later. It is an essential tool for any software development project as it allows multiple developers to work together, track changes, and easily rollback in case of errors. There are two main types of version control systems: centralized and distributed.

1. Centralized Version Control Systems (CVCS) : In a centralized version control system, there is a single central repository that contains all the versions of the files, and developers must check out files from the repository in order to make changes. Examples of CVCS include Subversion and Perforce.

2. Distributed Version Control Systems (DVCS) : In a distributed version control system, each developer has a local copy of the entire repository, including all the versions of the files. This allows developers to work offline, and it makes it easy to share changes with other developers. Examples of DVCS include Git, Mercurial and Bazaar

Version control is a vital component of software development that offers several benefits. First, it keeps track of changes made to files, enabling developers to revert to a previous version in case something goes wrong. Collaboration is also made easier with version control, as it allows multiple developers to work on a project simultaneously and share changes with others.

In addition, it provide backup capabilities by keeping a history of all changes, allowing you to retrieve lost files. Version control also allows auditing of changes, tracking who made a specific change, when, and why. Finally, it enables developers to create branches of a project, facilitating simultaneous work on different features without affecting the main project, with merging later.

Versioning all components of a machine learning project, such as code, data, and models, is essential for reproducibility and managing models in production. While versioning code-based components is similar to typical software engineering projects, versioning machine learning models and data requires specific version control systems. There is no universal standard for versioning machine learning models, and the definition of "a model" can vary depending on the exact setup and tools used.

Popular tools such as Azure ML, AWS Sagemaker, Kubeflow, and MLflow offer their own mechanisms for model versioning. For data versioning, there are tools like Data Version Control (DVC) and Git Large File Storage (LFS). The de-facto standard for code versioning is Git. The code-versioning system Github is used for this project, which will be depicted in more detail in the following.

1.4.2.1 Github

GitHub provides a variety of branching options to enable flexible collaboration workflows. Each branch serves a specific purpose in the development process, and using them effectively can help teams collaborate more efficiently and effectively.
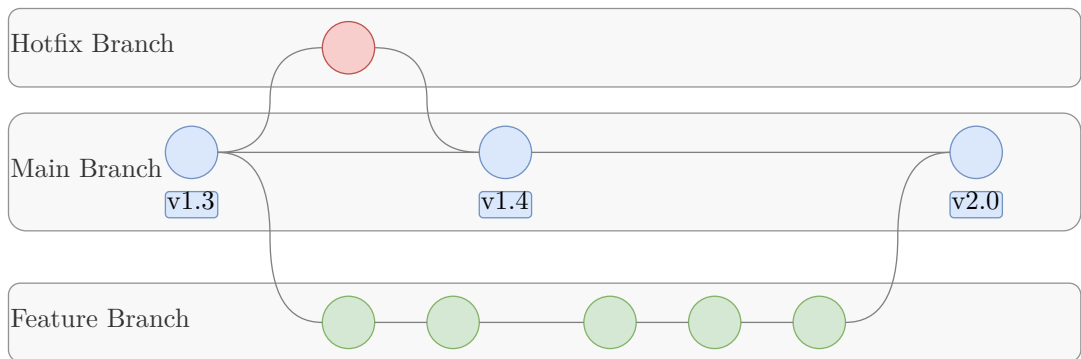
Main Branch: The main branch is the default branch in a repository. It represents the latest stable version and production-ready state of a codebase, and changes to the code are merged into the main branch as they are completed and tested. *Feature Branch:* A feature branch is used to develop a new feature or functionality. It is typically created off the main branch, and once the feature is completed, it can be merged back into the main branch. *Hotfix Branch:*

A hotfix branch is used to quickly fix critical issues in the production code. It is typically created off the main branch, and once the hotfix is completed, it can be merged back into the main branch. *Release Branch:* A release branch is a separate branch that is created specifically for preparing a new version of the software for release. Once all the features and fixes for the new release have been added and tested, the release branch is merged back into the main branch, and a new version of the software is tagged and released.

1.4.2.2 Git lifecycle

After a programmer has made changes to their code, they would typically use Git to manage those changes through a series of steps. First, they would use the command `git status` to see which files have been changed and are ready to be committed. They would then stage the changes they want to include in the commit using the command `git add <FILE-OR-DIRECTORY>`, followed by creating a new commit with a message describing the changes using `git commit -m "MESSAGE"`.

After committing changes locally, the programmer may want to share those changes with others. They would do this by pushing their local commits to a remote repository using the command `git push`. Once the changes are pushed, others can pull those changes down to their local machines and continue working on the project by using the command `git pull`.

```
● ~/Github/mlops-engineering-book (chapter/ops_principles*) » git status
  On branch chapter/ops_principles
  Your branch is up to date with 'origin/chapter/ops_principles'.

  Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git restore <file>..." to discard changes in working directory)
          modified:    manuscript/02.1-Ops-Overview.md

  no changes added to commit (use "git add" and/or "git commit -a")
● ~/Github/mlops-engineering-book (chapter/ops_principles*) » git add manuscript/02.1-Ops-Overview.md
● ~/Github/mlops-engineering-book (chapter/ops_principles*) » git commit -m "fixed spelling mistakes"
  [chapter/ops_principles 89778fe] fixed spelling mistakes
   1 file changed, 15 insertions(+), 28 deletions(-)
   rewrite manuscript/02.1-Ops-Overview.md (93%)
● ~/Github/mlops-engineering-book (chapter/ops_principles) » git push
  Enumerating objects: 7, done.
  Counting objects: 100% (7/7), done.
  Delta compression using up to 2 threads
  Compressing objects: 100% (4/4), done.
  Writing objects: 100% (4/4), 894 bytes | 894.00 KiB/s, done.
  Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
  remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
  To https://github.com/seblum/mlops-engineering-book
     1d21084..89778fe  chapter/ops_principles -> chapter/ops_principles
● ~/Github/mlops-engineering-book (chapter/ops_principles) » git pull
  Already up to date.
○ ~/Github/mlops-engineering-book (chapter/ops_principles) » █
```

If the programmer is collaborating with others, they may need to merge their changes with changes made by others. This can be done using the `git merge <BRANCH-NAME>` command, which combines two branches of development history. The programmer may need to resolve any conflicts that arise during the merge.

If the programmer encounters any issues or bugs after pushing their changes, they can use Git to revert to a previous version of the code by checking out an older commit using the command git checkout. Git's ability to track changes and revert to previous versions makes it an essential tool for managing code in collaborative projects.

While automating the code review process is generally viewed as advantageous, it is still typical to have a manual code review as the final step before approving a pull or merge request to be merged into the main branch. It is considered a best practice to mandate a manual approval from one or more reviewers who are not the authors of the code changes.

1.4.3 CI/CD

Continuous Integration (CI) and Continuous Delivery / Continuous Delivery (CD) are related software development practices that work together to automate and streamline the software

development and deployment process of code changes to production. Deploying new software and models without CI/CD often requires a lot of implicit knowledge and manual steps.
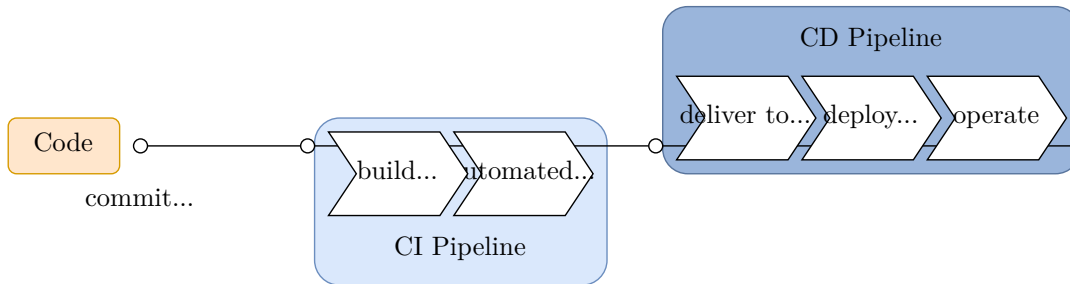
1. *Continuous Integration (CI)*: is a software development practice that involves frequently integrating code changes into a shared central repository. The goal of CI is to catch and fix integration errors as soon as they are introduced, rather than waiting for them to accumulate over time. This is typically done by running automated tests and builds, to catch any errors that might have been introduced with new code changes, for example when merging a Git feature branch into the main branch.

2. *Continuous Delivery (CD)*: is the practice that involves automating the process of building, testing, and deploying software to a production-like environment. The goal is to ensure that code changes can be safely and quickly deployed to production. This is typically done by automating the deployment process and by testing the software in a staging environment before deploying it to production.

3. *Continuous Deployment (CD):* is the practice of automatically deploying code changes to production once they pass automated tests and checks. The goal is to minimize the time it takes to get new features and bug fixes into the hands of end-users. In this process, the software is delivered directly to the end-user without manual testing and verification.

The terms *Continuous Delivery* and *Continuous Deployment* are often used interchangeably, but they have distinct meanings. Continuous delivery refers to the process of building, testing, and running software on a production-like environment, while continuous deployment refers specifically to the process of running the new version on the production environment itself. However, fully automated deployments may not always be desirable or feasible, depending on the organization's business needs and the complexity of the software being deployed. While continuous deployment builds on continuous delivery, the latter can offer significant value on its own.

CI/CD integrates the principles of continuous integration and continuous delivery in a seamless workflow, allowing teams to catch and fix issues early and quickly deliver new features to users. The pipeline is often triggered by a code commit. Ideally, a Data Scientist would push

the changes made to the code at each incremental step of development to a share repository, including metadata and documentation. This code commit would trigger the CI/CD pipeline to build, test, package, and deploy the model software. In contrast to the local development, the CI/CD steps will test the model changes on the full dataset and aiming to deploy for production.

CI and CD practices help to increase the speed and quality of software development, by automating repetitive tasks and catching errors early, reducing the time and effort required to release new features, and increasing the stability of the deployed software. Examples for CI/CD Tools that enable automated testing with already existing build servers are for example GitHub Actions, Gitlab CI/CD, AWS Code Build, or Azure DevOps

The following code snippet shows an exemplary GitHub Actions pipeline to test, build and push a Docker image to the DockerHub registry. The code is structured in three parts. At first, the environment variables are defined under `env`. Two variables are defined here which are later called with by the command `env.VARIABLE`. The second part defines when the pipeline is or should be triggered. The exampele shows three possibilites to trigger a pipelines, when pushing on the master branch `push`, when a pull request to the master branch is granted `pull_request`, or when the pipeline is triggered manually via the Github interface `workflow_dispatch`. The third part of the code example introduces the actual jobs and steps performed by the pipeline. The pipeline consists of two jobs `pytest` and `docker`. The first represents the CI part of the pipeline. The run environment of the job is set up and the necessary requirements are installed. Afterward unit tests are run using the pytest library. If the `pytest` job was successful, the `docker` job will be triggered. The job builds the Dockerfile and pushes it automatically to the specified Dockerhub repository specified in `tags`. The step introduces another variable just like the `env.Variable` before, the `secrets.`. Secrets are a way by Github to safely store classified information like username and passwords. They can be set up using the Github Interface and used in the Github Actions CI using `secrets.SECRET-NAME`.

```
name: Docker CI base

env:
  DIRECTORY: base
  DOCKERREPO: seblum/mlops-public

on:
  push:
    branches: master
    paths: $DIRECTORY/**
  pull_request:
```

```yaml
    branches: [ master ]
  workflow_dispatch:

jobs:
  pytest:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./${{ env.DIRECTORY }}
    steps:
    - uses: actions/checkout@v3
    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.x'
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
        pip install pytest
        pip install pytest-cov
    - name: Test with pytest
      run: |
        pytest test_app.py --doctest-modules --junitxml=junit/test-results.xml --cov=com --co
  docker:
    needs: pytest
    runs-on: ubuntu-latest
    steps:
      - name: Set up QEMU
        uses: docker/setup-qemu-action@v2
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2
      - name: Login to DockerHub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - name: Build and push
        uses: docker/build-push-action@v3
```

```
      with:
    file: ./${{ env.DIRECTORY }}/Dockerfile
    push: true
    tags: ${{ env.DOCKERREPO }}:${{ env.DIRECTORY }}
```

1.4.4 Infrastructure as code

Infrastructure as Code (IaC) is a software engineering approach that enables the automation of infrastructure provisioning and management using machine-readable configuration files rather than manual processes or interactive interfaces.

This means that the infrastructure is defined using code, instead of manually setting up servers, networks, and other infrastructure components. This code can be version controlled, tested, and deployed just like any other software code. It also allows to automate the process of building and deploying infrastructure resources, enabling faster and more reliable delivery of services, as well as ensuring to provide the same environment every time. It also comes with the benefit of an increased scalability, improved security, and better visibility into infrastructure changes.

It is recommended to utilize infrastructure-as-code to deploy an MLOps platform. Popular tools for implementing IaC are for example Terraform, CloudFormation, and Ansible. Chapter 6 gives a more detailed description and a tutorial on how to use Infrastructure as code using *Terraform*.
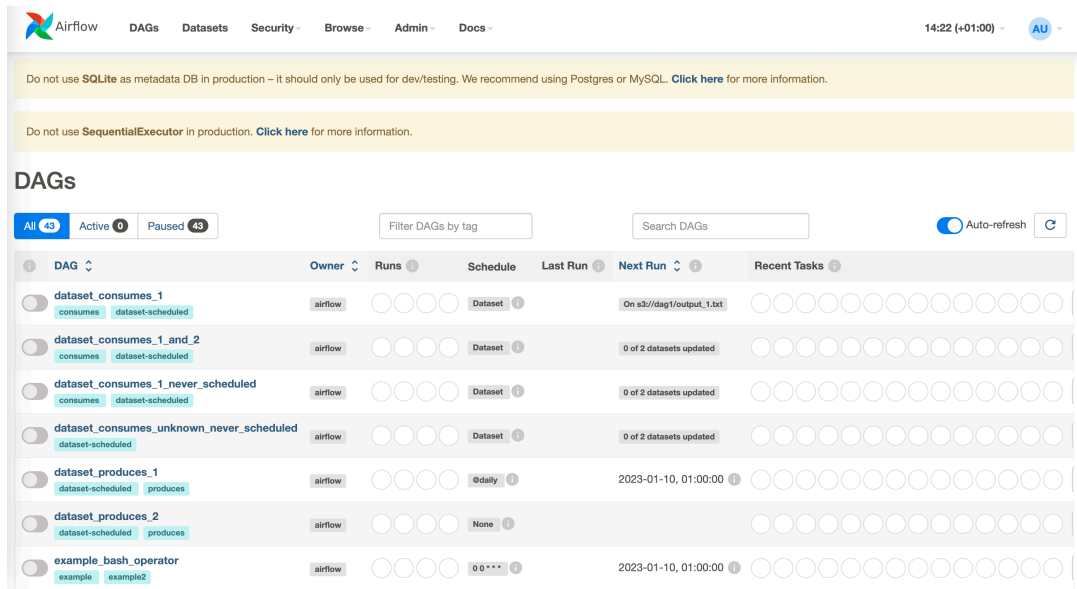
2 Overview about book tutorials

The book contains two sections with distinct focuses. The first section comprises Chapters 3 to 6, which consist of tutorials on the specific tools aforementioned. These chapters also serve as prerequisites for the subsequent sections. Among these tutorials, the chapters dedicated to *Airflow* and *MLflow* are oriented towards Data Scientists, providing insights into their usage. The chapters centered around *Kubernetes* and *Terraform* target Data- and MLOps Engineers, offering detailed guidance on deploying and managing these tools.

The second section, comprising Chapters 7 to 9, delves into an exemplary ML Platform. This section demands a strong background in engineering due to its complexity. While these chapters cover the essential tools introduced in the previous section, they may not explore certain intricate aspects used like OAuth authentication and networking details in great depth. Moreover, it is crucial to note that the ML Platform example presented is not intended for production deployment, as there should be significant security concerns considered. Instead, its main purpose is to serve as an informative illustration of ML platforms and MLOps engineering principles.

3 Airflow

Apache Airflow[1] is an open-source platform to develop, schedule and monitor workflows. Airflow comes with a web user interface that aims to make managing workflows as easy as possible and provides a good overview of each workflow over time and the ability to inspect logs and manage tasks, for example retrying a task in case of failure.

However, the philosophy of Airflow is to define workflows as code, so coding will always be required. Thus, Airflow can also be referred to as a *"Workflows as code"*-tool that allows for a dynamic, extensible, and flexible management of its workflows.

The Airflow platform contains different operators to easily extend and connect with many other technologies. Being able to manage a workflow for all stages of the training of ML models, and the possibility to combine Airflow with other tools like MLflow for model tracking, make Apache Airflow a great tool to incorporate in an MLOps architecture.

---

[1]https://github.com/apache/airflow

The aim of this chapter is to give a tutorial on how to use Airflow from a user perspective, as well as give a short overview of its deployment. Airflow can be deployed in multiple ways, starting from a single processing unit on a local machine to a distributed setup with multiple compute resources for large workflows in a production setting. A detailed description of what an Airflow deployment involves is shown in the section Airflow Infrastructure. The usage tutorial is based on the local installation of Airflow. Please refer to the prerequisits on what is needed to follow through.

Prerequisites

The main prerequisites to follow this tutorial to have an Apache Airflow instance installed. The official documentation gives a good overview on how to do[2]. It is sufficient to run Airflow on a local deployment and there is no need to to set up a complex Airflow deployment on a cluster or else. Further needed is intermediate knowledge of the programming language Python and basic knowledge of bash.

3.1 Core Components

Airflow serves as a batch-oriented workflow orchestration plattform. Workflows vary in their levels of complexity and in general it is a term with various meaning depending on context. When working with Airflow, a workflow usually describes a set of steps to accomplish a given data engineering tasks, e.g. downloading files, copying data, filtering information, or writing to a database, etc.

An exemplary use case might be to set up an ETL pipeline that extracts data from multiple sources, the transformation of the data, as well as loading them into a machine learning model. Even the training itself of a ML model can triggered via Airflow. Another workflow step might involve the generation of a report or backups.

Even though Airflow can implement programs from any language, the workflows are written and defined as Python code. Airflow allows to access each workflow via the previously mentioned web user interface, or via a command-line interface. Writing workflows in code has the benefit to use version control systems like *Git* to ensure roll backs to previous versions as well as to develop a workflow with a team of developers simultaneously. Using *Git* also allows to include further DevOps principles such as pipelines, and testing and validating the codes functionality.

---

[2]https://airflow.apache.org/docs/apache-airflow/stable/start.html

3.1.1 DAGs

A workflows in Airflow is implemented as a DAG, a *Directed Acyclic Graph*. A *graph* describes the actual set of components of a workflow. It is *directed* because it has an inherent flow representing dependencies between its components. It is *acyclic* as it does not loop or repeat.

The DAG object is needed to nest the separate tasks intp a workflow. A workflow specified in code, e.g. python, is often also referred to as a *pipeline*. This terminology can be used synonymosly when working with Airflow. The following code snippet depicts how to define a DAG object in python code. The `dag_id` string is a unique identifier to the DAG object. The `default_args` dictionary consists of additional parameters that can be specified. There are only shown two additional parameters. There are a lot more though which can be seen in the official documentation[3].

```python
from airflow.models import DAG
from pendulum import datetime

# Using extra arguments allows to customize in a clear structure
# e.g. when setting the time zone in a DAG.
default_args = {
    'start_date': datetime(2023, 1, 1, tz="Europe/Amsterdam"),
    'schedule_interval': 'None'
}


example_dag = DAG(
    dag_id='DAG_fundamentals',
    default_args=default_args
)
```

The state of a workflow can accessed via the web user inteface, such as shown in below images. The first image shows Airflows overview of all DAGs currently "Active" and further information about them such as their "Owner", how many "Runs" have been performed and whether they were successful, and much more. The second image depicts a detailed overview of the DAG "xcom_fundamentals". Besudes looking into the *Audit Logs* of the DAG and the *Task Duration*, it is also possible to check the *Code* of the DAG.

---

[3]https://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/models/dag/index.html#airflow.models.dag.DAG

The Airflow command line interface also allows to interact with the DAGs. Below command shows its exemplary usage and how to list all active DAGs. Further examples of using the CLI in a specific context are show in the subsection about Tasks.

```
# Create the database schema
airflow db init


# Print the list of active DAGs
airflow dags list
```

People sometimes think of a DAG definition file as a place where the actual data processing

is done. That is not the case at all! The scripts purpose is to only define a DAG object. It needs to evaluate quickly (in seconds, not minutes) since the scheduler of Airflow will load and execute it periodically to account for changes in the DAG definition. A DAG usually consists of multiple steps it runs through, also names as *tasks*. Tasks themselves consist of *operators*. This will be outlined in the following subsections.

3.1.2 Operators

An Operator represents a single predefined task in a workflow. It is basically a unit of work that Airflow has to complete. Operators usually run independently and generally do not share any information by themselves. There are different categories of operators to perform different tasks with, for example *Action operators*, *Transfer operators*, or *Sensors*. *Action Operators* execute a basic task based on the operators specifications. Examples are the `BashOperator`, the `PythonOperator`, or the `EmailOperator`. The operator names already suggest what kind of executions they provide; the PythonOperator runs python tasks. *Transfer Operators* are designed to transfer data from one place to another, for example to copy data from one cloud bucket to another. Those operators are often stateful, which means the downloaded data are first stored locally and then uploaded to a destination storage. The principle of a stateful execution defines them as an own category of operator. Finally, *Sensors* are a special subclass of operators that are triggered when an external event is happening, for example the creation of a file.

The `PythonOperator` is actually declared a deprecated function. Airflow 2.0 promotes to use the `@task`-decorator of *Taskflow* to define tasks in a more pythonic context. Yet, it still works in Airflow 2.0 and is still a very good example on how operators work.

3.1.2.1 Action Operators

BashOperator

As its name suggests, the `BashOperator` executes commands in the bash shell.

```
from airflow.operators.bash_operator import BashOperator

bash_task = BashOperator(
    task_id='bash_example',
    bash_command='echo "Example Bash!"',
    dag=action_operator_fundamentals
)
```

PythonOperator

The `PythonOperator` expects a python callable. Airflow passes a set of keyword arguments from the `op_kwargs` dictionary to the callable as input.

```python
from airflow.operators.python_operator import PythonOperator

def sleep(length_of_time):
    time.sleep(length_of_time)

sleep_task = PythonOperator(
    task_id='sleep',
    python_callable=sleep,
    op_kwargs={'length_of_time': 5},
    dag=action_operator_fundamentals
)
```

EmailOperator

The `EmailOperator` allows to send predefined emails from an Airflow DAG run. For example, this could be used to notify if a workflow was successfull or not. The `EmailOperator` does require the Airflow system to be configured with email server details as a prerequisite. Please refer to the official docs on how to do this.

```python
from airflow.operators.email_operator import EmailOperator

email_task = EmailOperator(
    task_id='email_sales_report',
    to='sales_manager@example.com',
    subject='automated Sales Report',
    html_content='Attached is the latest sales report',
    files='latest_sales.xlsx',
    dag=action_operator_fundamentals
)
```

3.1.2.2 Transfer Operators

GoogleApiToS3Operator

The `GoogleApiToS3Operator` makes requests to any Google API that supports discovery and uploads its response to AWS S3. The example below loads data from Google Sheets and saves it to an AWS S3 file.

```
from airflow.providers.amazon.aws.transfers.google_api_to_s3 import GoogleApiToS3Opera

google_sheet_id = <GOOGLE-SHEET-ID >
google_sheet_range = <GOOGLE-SHEET-RANGE >
s3_destination_key = <S3-DESTINATION-KEY >

task_google_sheets_values_to_s3 = GoogleApiToS3Operator(
    task_id="google_sheet_data_to_s3",
    google_api_service_name="sheets",
    google_api_service_version="v4",
    google_api_endpoint_path="sheets.spreadsheets.values.get",
    google_api_endpoint_params={
        "spreadsheetId": google_sheet_id, "range": google_sheet_range},
    s3_destination_key=s3_destination_key,
)
```

DynamoDBToS3Operator

The `DynamoDBToS3Operator` copies the content of an AWS DynamoDB table to an AWS S3 bucket. It is also possible to specifiy criteria such as `dynamodb_scan_kwargs` to filter the transfered data and only replicate records according to criteria.

```
from airflow.providers.amazon.aws.transfers.dynamodb_to_s3 import DynamoDBToS3Operator

table_name = <TABLE-NAME >
bucket_name = <BUCKET-NAME >

backup_db = DynamoDBToS3Operator(
    task_id="backup_db",
    dynamodb_table_name=table_name,
    s3_bucket_name=bucket_name,
    # Max output file size in bytes.  If the Table is too large, multiple files will b
    file_size=20,
)
```

AzureFileShareToGCSOperator

The `AzureFileShareToGCSOperator` transfers files from the Azure FileShare to the Google Storage. Even though the storage systems are quite similar, this operator is beneficial when the cloud provider needs to be changed. The `share_name`-parameter denotes the Azure FileShare share name to transfer files from. Similarly, the `dest_gcs`specifies the destination bucket on the Google Cloud.

```python
from airflow.providers.google.cloud.transfers.azure_fileshare_to_gcs import AzureFileShareToGC

azure_share_name = <AZURE-SHARE-NAME >
bucket_name = <BUCKET-NAME >
azure_directory_name = <AZURE-DIRECTORY-NAME >

sync_azure_files_with_gcs = AzureFileShareToGCSOperator(
    task_id="sync_azure_files_with_gcs",
    share_name=azure_share_name,
    dest_gcs=bucket_name,
    directory_name=azure_directory_name,
    replace=False,
    gzip=True,
    google_impersonation_chain=None,
)
```

3.1.2.3 Sensors

A *sensor* is a special subclass of an operator that is triggered when an external event is happening or a certain condition is be true. Such conditions are for example the creation of a file, an upload of a database record, or a certain response from a web request.

It is also possible to specify further requirements to check the condition. The `mode` argument sets how to check for a condition. `mode='poke'` denotes to run a task repeatedly until it is successful (this is the default), whereas `mode='reschedule'` gives up a task slot and tries again later. Simultaneously, the `poke_interval` defines how long a sensor should wait between checks, and the `timeout` parameter defines how long to wait before letting a task fail. Below example shows a `FileSensor` that checks the creation of a file with a `poke_interval` defined.

```python
from airflow.contrib.sensors.file_sensor import FileSensor

file_sensor_task = FileSensor(task_id='file_sense',
                              filepath='salesdata.csv',
```

```
                              poke_interval=30,
                              dag=sensor_operator_fundamentals
                              )
```

Other sensors are for example: * `ExternalTaskSensor` - waits for a task in another DAG to complete * `HttpSensor` - Requests a web URL and checks for content * `SqlSensor` - Runs a SQL query to check for content

3.1.3 Tasks

To use an operator in a DAG it needs to be instantiated as a task. Tasks determine how to execute an operator's work within the context of a DAG. The concepts of a *Task* and *Operator* are actually somewhat interchangeable as each task is actually a subclass of Airflow's `BaseOperator`. However, it is useful to think of them as separate concepts. Tasks are Instances of operators and are usually assigned to a python variable. The following code instantiates the `BashOperator` to two different variables `task_1` and `task_2`. The `depends_on_past` argument ensures that the previously scheduled task has succeeded before the current task is triggered.

```
task_1 = BashOperator(
    task_id="print_date",
    bash_command="date",
    dag=task_fundamentals
)


task_2 = BashOperator(
    task_id="set_sleep",
    depends_on_past=False,
    bash_command="sleep 5",
    retries=3,
    dag=task_fundamentals
)


task_3 = BashOperator(
    task_id="print_success",
    depends_on_past=True,
    bash_command='echo "Success!"',
    dag=task_fundamentals
)
```

Tasks can be referred to by their `task_id` either using the web interface or using the CLI within the airflow tools. `airflow tasks test` runs task instances locally, outputs their log to stdout (on screen), does not bother with dependencies, and does not communicate the state (running, success, failed, …) to the database. It simply allows testing a single task instance. The same accounts for `airflow dags test`

```
# Run a single task with the following command
airflow run <dag_id> <task_id> <start_date>

# Run tasks locally for testing
airflow tasks test <dag_id> <task_id> <input-parameter>

# Testing the task print_date
airflow tasks test task_fundamentals print_date 2015-06-01

# Testing the task sleep
airflow tasks test task_fundamentals sleep 2015-06-01
```

3.1.3.1 Task dependencies

A machine learning or data workflow usually has a specific order in which its tasks should run. Airflow alloes to define a specific order of task completion using task dependencies that are either referred to as upstream or downstream tasks. In Airflow 1.8 and later, this can be defined using the *bitshift operators* in python.

- » - or the upstream operator (before)
- « - or the downstream operator (after)

An exemplary code and chaining examples of tasks would look like this:

```
# Simply chained dependencies
task_1 >> task_2 >> task_3
```



```
# Mixed dependencies
task_1 >> task_3 << task_2
```

```
# which is similar to
task_1 >> task_3
task_2 >> task_3

# or
[task_1, task_2] >> task_3
```



```
# It is also possible to define dependencies with
task_1.set_downstream(task_2)
task_3.set_upstream(task_1)
```



```
# It is also possible to mix it completely wild
task_1 >> task_3 << task_2
task_1.set_downstream(task_2)
```


It is possible to list all tasks within a DAG using the CLI. Below commands show two approaches.

```
# Prints the list of tasks in the "task_fundamentals" DAG
airflow tasks list task_fundamentals
```

```
# Prints the hierarchy of tasks in the "task_fundamentals" DAG
airflow tasks list task_fundamentals --tree
```

In general, each task of a DAG runs on a different compute resource (also called worker). It may require an extensive use of environment variables to achieve running on the same environment or with elevated privileges. This means that tasks naturally cannot cross communicate which impedes the exchange of information and data. To achieve cross communication an additional feature of Airflow needs to be used, called *XCom*.

3.1.4 XCom

XCom (short for "cross-communication") is a mechanism that allows information passing between tasks in a DAG. This is beneficial as by default tasks are isolated within Airflow and may run on entirely different machines. The goal of using XComs is not to pass large values like dataframes themselves. XComs are used to store all relevant metadata that are needed to pass data from one task to another, including infomration about the sender, the recipient, and the data itself.

If there is the need to pass a big data frame from *task_a* to *task_b*, the data is stored in a persistent storage solution (a bucket, or database) and the information about the storage location is stored in the XCom. This means that *task_a* pushes the data to a storage solution and writes the information where the data is stored (e.g. a AWS S3 URI) within an XCom to the Airflow database. Afterward, *task_b* can access this information and retrieve the data from the external storage using the AWS .

The XCom is identified by a *key-value* pair stored in the Airflow Metadata Database. The `key` of an XCom is basically its name and consists of a tuple (`dag_id`, `task_id`, `execution_date`,`key`). *key* within the XComs name denotes the name of the stored data and is a configurable string (by default it's `return_value`). The XComs `value` is a json serializable. It is important to note that it is only designed for small amounts of data, depending on the attached metadata database (e.g. 2GB for SQLite, or 1GB Postgres database). XComs are explicitly pushed and pulled to/from the metadata database using the `xcom_push` and `xcom_pull` methods. While the `push_*`-methods upload the XCom, the `pull`-method downloads information from XCom. The XCom element can be viewed within Airflows web interface which is quite helpful to debug or monitor a DAG.

Below example shows this mechanics. However, when looking at the `push` one can see a difference in their functionality. The method `push_by_returning` uses the operators' auto-push functionality that pushes their results into an default XCom key (the default is `return_value`). Using the auto-push functionality allows to only use python `return` statements and is enabled

by setting the `do_xcom_push` argument to `True`, which it also is by default (`@task` functions do this as well). To push an XCom with a specific key, the `xcom_push`-method needs to be called explicitly. In order to access the `xcom_push` one needs to access the task instance (ti) object. It can be accessed by passing the `"ti"` parameter to the python callable of the PythonOperator. Its usage can be seen in the `push` method, where also a custom key is given to the XCom. Similarly, the `puller` function uses the `xcom_pull` method to pull the previously pushed values from the metadata databes.

```python
from airflow.models import DAG
from pendulum import datetime
from airflow.operators.python_operator import PythonOperator


xcom_fundamentals = DAG(
    dag_id='xcom_fundamentals',
    start_date=datetime(2023, 1, 1, tz="Europe/Amsterdam"),
    schedule_interval=None
)

# DAG definition etc.
value_1 = [1, 2, 3]
value_2 = {'a': 'b'}

def push(**kwargs):
    """Pushes an XCom without a specific target"""
    kwargs['ti'].xcom_push(key='value from pusher 1', value=value_1)

def push_by_returning(**kwargs):
    """Pushes an XCom without a specific target, just by returning it"""
    # Airflow does this automatically as auto-push is turned on.
    return value_2

def puller(**kwargs):
    """Pull all previously pushed XComs and check if the pushed values match the pulle
    ti = kwargs['ti']

    # get value_1
    pulled_value_1 = ti.xcom_pull(key=None, task_ids='push')

    # get value_2
```

```python
    pulled_value_2 = ti.xcom_pull(task_ids='push_by_returning')

    # get both value_1 and value_2 the same time
    pulled_value_1, pulled_value_2 = ti.xcom_pull(
        key=None, task_ids=['push', 'push_by_returning'])

    print(f"pulled_value_1: {pulled_value_1}")
    print(f"pulled_value_2: {pulled_value_2}")


push1 = PythonOperator(
    task_id='push',
    # provide context is for getting the TI (task instance ) parameters
    provide_context=True,
    dag=xcom_fundamentals,
    python_callable=push,
)

push2 = PythonOperator(
    task_id='push_by_returning',
    dag=xcom_fundamentals,
    python_callable=push_by_returning,
    # do_xcom_push=False
)

pull = PythonOperator(
    task_id='puller',
    # provide context is for getting the TI (task instance ) parameters
    provide_context=True,
    dag=xcom_fundamentals,
    python_callable=puller,
)

# push1, push2 are upstream to pull
[push1, push2] >> pull
```

3.1.5 Scheduling

A workflow can be run either triggered manually or on a scheduled basis. Each DAG maintains a state for each workflow and the tasks within the workflow, and specifies whether it is *running*, *failed*, or a *success*. Airflow scheduling is designed to run as a persistent service in an production environment and can be customized. When running Airflow locally, executing the `airflow scheduler` via the CLI will use the configuration specified in `airflow.cfg` and start the service.

The Airflow scheduler monitors all DAGs and tasks, and triggers those task instances whose dependencies have been met. A scheduled tasks needs several attributes specified. When looking at the first example of how to define a DAG we can see that we already defined the attributes `start_date`, and `schedule_interval`. We can also add optional attributes such as `end_date`, and `max_tries`.

```python
from airflow.models import DAG

default_args = {
    'start_date': '2023-01-01',
    # (optional) when to stop running new DAG instances
    'end_date': '2023-01-01',
    # (optional) how many attempts to make
    'max_tries': 3,
    'schedule_interval': '@daily'
}

example_dag = DAG(
    dag_id='scheduling_fundamentals',
    default_args=default_args
)
```

3.1.6 Taskflow

Defining a DAG and using operators as shown in the previous sections is the classic approach to define a workflow in Airflow. However, *Airflow 2.0* introduced the TaskFlow API which allows to work in a more pythonic way using decorators. DAGs and tasks can be created using the `@dag`or `@task` decorators. The function name itself acts as the unique identifier for the DAG or task respectively.

All of the processing in a TaskFlow DAG is similar to the traditional paradigm of Airflow, but it is all abstracted from the developers. This allows developers to focus on the code. There is also no need to specify task dependencies as they are automatically generated within TaskFlow based on the functional invocation of tasks.

Defining a workflow of an ETL-pipeline using the TaskFlow paradigm is shown in belows example. The pipeline invokes an *extract* task, sends the ordered data to a *transform* task for summarization, and finally invokes a *load* task with the previously summarized data. Its quite easy to catch that the Taskflow workflow contrasts with Airflow's traditional paradigm in several ways.

```python
import json
import pendulum
from airflow.decorators import dag, task


# Specify the dag using @dag
# The Python function name acts as the DAG identifier
# (see also https://airflow.apache.org/docs/apache-airflow/stable/tutorial_taskflow_api.html)

@dag(
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def taskflow_api_fundamentals():
    # set a task using @task
    @task()
    def extract():
        data_string = '{"1001": 301.27, "1002": 433.21, "1003": 502.22}'
        order_data_dict = json.loads(data_string)
        return order_data_dict

    @task(multiple_outputs=True)
    def transform(order_data_dict: dict):
        total_order_value = 0
        for value in order_data_dict.values():
            total_order_value += value
        return {"total_order_value": total_order_value}
```

```python
    @task()
    def load(total_order_value: float):
        print(f"Total order value is: {total_order_value:.2f}")

    # task dependencies are automatically generated
    order_data = extract()
    order_summary = transform(order_data)
    load(order_summary["total_order_value"])

# Finally execute the DAG
taskflow_api_fundamentals()
```

Even the passing of data between tasks which might run on different workers is all handled by TaskFlow so there is no need to use XCom. However, XCom is still used behind the scenes, but all of the XCom usage passing data between tasks is abstracted away. This allows to view the XCom in the Airflow UI as before. Belows example shows the `transform` function written in the traditional Airflow using XCom and highlights the simplicity of using TaskFlow.

```python
def transform(**kwargs):
    ti = kwargs["ti"]
    extract_data_string = ti.xcom_pull(task_ids="extract", key="order_data")
    order_data = json.loads(extract_data_string)

    total_order_value = 0
    for value in order_data.values():
        total_order_value += value

    total_value = {"total_order_value": total_order_value}
    total_value_json_string = json.dumps(total_value)
    ti.xcom_push("total_order_value", total_value_json_string)
```

As it is clearly visible, using TaskFlow is an easy approach to workflowing in Airflow. It takes away a lot of worries when it comes to building pipelines and allows for a flexible programing experience using decorators. It allows for several more functionalities, such as reusing decorated tasks in multiple DAGs, overriding task parameters like the `task_id`, custom XCom backends to automatically store data in e.g. AWS S3, and using TaskGroups to group multiple tasks for a better overview in the Airflow Interface.

However, even though Taskflow allows for a smooth way of developing workflows, it is beneficial to learn the traditional Airflow API to understand the fundamental concepts of how to

create a workflow. The following section will build up on that knowledge and depict a full machine learning workflow as an example.

3.2 Exemplary ML workflow

Please refer to the previous section. This involves everything

```python
# build dag that includes everything from before
def to_be_done():
    pass
```

3.3 Airflow infrastructure

Before Data Scientists and Machine Learning Engineers can utilize the power of Airflow Workflows, Airflow obviously needs to be set up and deployed. There are multiple ways an Airflow deployment can take place. It can be run either on a single machine or in a distributed setup on a cluster of machines. As stated in the prerequisites, a local Airflow setup is on a single machine can be used for this tutorial to give an introduction on how to work with airflow. Although Airflow can be run on a single machine, it should be deployed as a distributed system to utilize its full power when working with Airflow in production.

3.3.1 Airflow as a distributed system

Airflow consists of several separate architectural components. While this separation is somewhat simulated on a local deployment, each unit of Airflow can be set up separately when deploying Airflow in a distributed manner. A distributed architecture comes with benefits of availability, security, reliability, and scalability.

An Airflow deployment generally consists of five different compontens:

- **Scheduler:** The schedules handles triggering scheduled workflows and submitting tasks to the executor to run.
- **Executor:** The executor handles running the tasks themselves. In a local installation of Airflow, the tasks are run by the executor itself. In a production ready deployment of Airflow the executor pushes the task execution to separate worker instance which then runs the task.
- **Webserver:** The webserver provides the web user interface of Airflow that allows to inspect, trigger, and debug DAGs and tasks.

- **DAG Directory:** The DAG directory is a directory that contains the DAG files which are read by the scheduler and executor.
- **Metadata Database:** The metadata database is used to store data of the scheduler, executor, and the webserver, such as scheduling- or runtime, user settings, or XCom.

The following graphs shows how the components build up the Airflow architecture.

Figure 3.1: Architecture of Airflow as a distributed system

3.3.2 Scheduler

The *scheduler* is basically the brain and heart of Airflow. It handles triggering and scheduling of workflows, as well as submitting tasks to the executor to run. To be able to do this, the scheduler is responsible to parse the DAG files from the *DAG directory*, manage the database states in the *metadata database*, and to communicate with the *executor* to schedule tasks. Since the release of Airflow 2.0 it is possible to run multiple schedulers at a time to ensure a high availability and reliability of this centerpiece of Airflow.

3.3.3 Webserver

The *webserver* runs the web interface of Airflow and thus the user interface every Airflow user sees. This allows to inspect, trigger, and debug DAGs and tasks in Airflow (and much more!), such as seen in the previous chapters. Each user interaction and change is written to the *DAG directory* or the *metadata database*, from where the *scheduler* will read and act upon.

3.3.4 Executor

The *executor* defines where and how the Airflow tasks should be executed and run. This crucial component of Airflow can be configured by the user and should be chosen to fit the users specific needs. There are several different executors, each handling the run of a task a bit differently. Choosing the right executor also relies on the underlying infrastructure Airflow is build upon.

In a production ready deployment of Airflow the executor pushes the task execution to separate worker instances that run the tasks. This allows for different setups such as a Celery-like executors or an executor based on Kubernetes. The benefit of a distributed deployment is its reliability and availability, as it is possible to have many workers in different places (for example using separate virtual machines, or multiple kubernetes pods). It is further possibility to run tasks on different instances based on their needs, for example to run the training step of a machine learning model on a GPU node.

- The `SequentialExecutor` is the default executor in Airflow. This executor only runs one task instance at a time and should therefore not used in a production use case.
- The `LocalExecutor`executes each task in a separate process on a single machine. It's the only non-distributed executor which is production ready and works well in relatively small deployments. If Airflow is installed locally as mentioned in the prerequisites, this executor will be used.
- In contrast, the `CeleryExecutor` uses under the hood the Celery queue system that allows users to deploy multiple workers that read tasks from the broker queue (Redis or RabbitMQ) where tasks are sent by scheduler. This enables Airflow to distribute tasks between many machines and allows users to specify what task should be executed and where. This can be useful for routing compute-heavy tasks to more resourceful workers and is the most popular production executor.
- The `KubernetesExecutor` is another widely used production-ready executor and works similarly to the `CeleryExecutor`. As the name already suggests it requires an underlying Kubernetes cluster that enables Airflow to spawn a new pod to run each task. Even

though this is a robust method to account for machine or pod failure, the additional overhead in creating pods or even nodes can be problematic for short running tasks.

- The `CeleryKubernetsExecutor` uses both, the `CeleryExecutor` and `KubernetesExecutor` (as the name already says). It allows to distinguish whether a particular task should be executed on kubernetes or routed to the celery workers. This way users can take full advantage of horizontal auto scaling of worker pods, and to delegate computational heavy tasks to kubernetes.

- The additional `DebugExecutor` is an executor whose main purpose is to debug DAGs locally. It's the only executor that uses a single process to execute all tasks.

Examining which executor is currently set can be done by running the following command.

```
airflow config get-value core executor
```

3.3.5 DAG Directory

The *DAG directory* contains the DAG files written in python. The DAG directory is read and used by each Airflow component for a different purpose. The *web interface* lists all written DAGs from the directory as well as their content. The *scheduler* and *executor* run a DAG or a task based on the input read from the DAG directory.

The DAG directory can be of different nature. It can be a local folder in case of a local installation, or also use a separate repository like Git where the DAG files are stored. The scheduler will recurse through the DAG Directory so it is also possible to create subfolders for example based on different projects.

3.3.6 Metadata Database

The *metadata database* is used to store data of the *scheduler*, *executor*, and the *webserver*, such as scheduling- or runtime, user settings, or XCom. It is beneficial to run the metadata database as a separate component to keep all data safe and secure in case there are erroneous other parts of the infrastructure. Such considerations account for the architectural decisions of an Airflow deployment and the metadata database itself. For example, it is possible to run Airflow and all of its components on a Kubernetes cluster. However, this is not necessarily recommended as it is prone to the cluster and its accessibility itself. It is also possible to outsource Airflow components such as the *metadata database* to use a clouds' distinguished database resources for example. For example to store all metadata, in the Relational Database Service (RDS) on the AWS Cloud.

4 MLflow

MLflow is an open source platform to manage the machine learning lifecycle end-to-end. This includes the experimentation phase of ML models, their development to be reproducible, their deployment, and the registration of a ML model to be served. MLflow provides four primary components to manage the ML lifecycle. They can be either used on their own or they also to work together.

- **MLflow Tracking** is used to log and compare model parameters, code versions, metrics, and artifacts of an ML code. Results can be stored to local files or to remote servers, and can be compared over multiple runs. MLflow Tracking comes with an API and a web interface to easily observe all logged parameters and artifacts.
- **MLflow Models** enables to manage and deploy machine learning models from multiple libraries. It allows to package your own ML model for later use in downstream tasks, e.g. real-time serving through a REST API. The package format defines a convention that saves the model in different *"flavors"* that can be interpreted by different downstream tools.
- **MLflow Registry** provides a central model store to collaboratively manage the full lifecycle of a MLflow Model, including model versioning, stage transitions, and annotations. It comes with an API and user interface for easy use of such funtionalities and each of those aspects can be checked in MLflows' web interface.
- **MLflow Projects** packages data science code in a standard format for a reusable, and reproducible form to share your code with other data scientists or transfer it to production. A project might be a local directory or a Git repository which uses a descriptor file to specify its dependencies and entrypoints. An existing MLflow Project can be also run either locally or from a Git repository.

MLflow is library-agnostic, which means one can use it with any ML library and programming language. All functions are accessible through a REST API[1] and CLI[2], but quite conveniently the project comes with a Python API, R API, and a Java API already included. It is even possible to define your own plugins[3]. The aim is to make its use as reproducible and reusable as possible so Data Scientists require minimal changes to integrate MLflow into their existing

---

[1] https://MLflow.org/docs/latest/rest-api.html#rest-api
[2] https://MLflow.org/docs/latest/cli.html#cli
[3] https://mlflow.org/docs/latest/plugins.html#mlflow-plugins

codebase. MLflow also comes with a user web interface to conveniently view and compare
models and metrics.

Figure 4.1: Web Interface of MLflow

Prerequisites

To go through this chapter it is necessary to have python and MLflow installed. One can
install MLflow locally via `pip install MLflow`. This tutorial is based on MLflow v2.1.1. It
is also recommended to have knowledge of VirtualEnv, Conda, or Docker when working with
MLflow Projects.

4.1 Core Components

The four primary components of MLflow are shown in more detail and with exemplary code
in the following sections.

4.1.1 MLflow Tracking

MLflow Tracking allows to log and compare parameters, code versions, metrics, and artifacts of a machine learning model. This can be easily done by minimal changes to your code using the MLflow Tracking API. The following examples depict the basic concepts and show how to use it. To use MLflow within your code it needs to be imported first.

```
import mlflow
```

4.1.1.1 MLflow experiment

MLflow experiments are a part of MLflow's tracking component that allow to group runs together based on custom criteria. For example multiple model runs with different model architectures might be grouped within one experiment to make it easier for evaluation.

```
experiment_name = "introduction-experiment"
mlflow.set_experiment(experiment_name)
```

4.1.1.2 MLflow run

An MLflow run is an execution environment for a piece of machine learning code. Whenever parameters or performances of a ML run or experiment should be tracked, a new MLflow run is created. This is easily done using `MLflow.start_run()`. Using `MLflow.end_run()` the run will similarly be ended.

```
run_name = "example-run"

mlflow.start_run()
run = mlflow.active_run()
print(f"Active run_id: {run.info.run_id}")
mlflow.end_run()
```

It is a good practice to pass a run name to the MLflow run to identify it easily afterwards. It is also possible to use the context manager as shown below, which allows for a smoother style.

```python
run_name = "context-manager-run"

with mlflow.start_run(run_name=run_name) as run:
    run_id = run.info.run_id
    print(f"Active run_id: {run_id}")
```

Child runs It is possible to create child runs of the current run, based on the run ID. This can be used for example to gain a better overview of multiple run. Belows code shows how to create a child run.

```python
# Create child runs based on the run ID
with mlflow.start_run(run_id=run_id) as parent_run:
    print("parent run_id: {}".format(parent_run.info.run_id))
    with mlflow.start_run(nested=True, run_name="test_dataset_abc.csv") as child_run:
        mlflow.log_metric("acc", 0.91)
        print("child run_id : {}".format(child_run.info.run_id))

with mlflow.start_run(run_id=run_id) as parent_run:
    print("parent run_id: {}".format(parent_run.info.run_id))
    with mlflow.start_run(nested=True, run_name="test_dataset_xyz.csv") as child_run:
        mlflow.log_metric("acc", 0.90)
        print("child run_id : {}".format(child_run.info.run_id))
```

4.1.1.3 Logging metrics & parameters

The main reason to use MLflow Tracking is to log and store parameters and metrics during a MLflow run. *Parameters* represent the input parameters used for training, e.g. the initial learning rate. *Metrics* are used to track the progress of the model training and are usually updated over the course of a model run. MLflow allows to keep track of the model's train and validation losses and to visualize their development across the training run. Parameters and metrics can be easily logged by calling `MLflow.log_param` and `MLflow.log_metric`. One can also specify a tag to identify the run by using `MLflow.set_tag`. Belows example show how to use each method within a run.

```python
run_name = "tracking-example-run"
experiment_name = "tracking-experiment"
mlflow.set_experiment(experiment_name)
```

```python
with mlflow.start_run(run_name=run_name) as run:

    # Parameters
    mlflow.log_param("learning_rate", 0.01)
    mlflow.log_params({"epochs": 0.05, "final_activation": "sigmoid"})

    # Tags
    mlflow.set_tag("env", "dev")
    mlflow.set_tags({"some_tag": False, "project": "xyz"})

    # Metrics
    mlflow.log_metric("loss", 0.001)
    mlflow.log_metrics({"acc": 0.92, "auc": 0.90})

    # It is possible to log a metrics series (for example a training history)
    for val_loss in [0.1, 0.01, 0.001, 0.00001]:
        mlflow.log_metric("val_loss", val_loss)

    for val_acc in [0.6, 0.6, 0.8, 0.9]:
        mlflow.log_metric("val_acc", val_acc)

    run_id = run.info.run_id
    experiment_id = run.info.experiment_id
    print(f"run_id: {run_id}")
    print(f"experiment_id: {experiment_id}")
```

It is also possible to add information after the experiment ran. One just needs to specify the run ID from the previous run to the current run. The example below shows how to do this, and uses the `mlflow.client.MLflowClient`. The `mlflow.client` module provides a Python CRUD interface, which is a lower level API directly translating to the MLflow REST API[4] calls. It can be used similarly to the `mlflow`-module of the higher level API. It is mentioned here to give a hint of its existence.

```python
from mlflow.tracking import MLflowClient

# add a note to the experiment
MLflowClient().set_experiment_tag(
    experiment_id, "MLflow.note.content", "my experiment note")
```

---

[4]https://mlflow.org/docs/latest/rest-api.html

```python
# add a note to the run
MLflowClient().set_tag(run_id, "MLflow.note.content", "my run note")

# Or we can even log further metrics by calling MLflow.start_run on a specific ID
with mlflow.start_run(run_id=run_id):
    run = mlflow.active_run()
    mlflow.log_metric("f1", 0.9)
    print(f"run_id: {run.info.run_id}")
```

4.1.1.4 Display & View metrics

How can the logged parameters and metrics be used and viewed afterwards? It is possible to give an overview of the currently stored runs using the MLflow API and printing the results.

```python
current_experiment = dict(mlflow.get_experiment_by_name(experiment_name))
mlflow_run = mlflow.search_runs([current_experiment['experiment_id']])
print(f"MLflow_run: {mlflow_run}")
```


Figure 4.2: MLflow Model Tracking CLI Run Overview

Yet, viewing all the results in the web interface of MLflow gives a much better overview. By default, the tracking API writes the data to the local filesystem of the machine it's running on under a `./mlruns` directory. This directory can be accessed by the MLflow's Tracking web interface by running `MLflow ui` via the command line. The web interface can be viewed in the browser under http://localhost:5000 (The port: 5000 is the MLflow default). The metrics dashboard of a run looks like the following:

It is also possible to configure MLflow to log to a remote tracking server. This allows to manage results on in a central place and share them across a team. To get access to a remote tracking server it is needed to set a MLflow tracking URI. This can be done multiple way. Either by setting an environment variable `MLflow_TRACKING_URI` to the servers URI, or by adding it to the start of our code.

```python
import mlflow
mlflow.set_tracking_uri("http://YOUR-SERVER:YOUR-PORT")
mlflow.set_experiment("my-experiment")
```

Figure 4.3: MLflow Model Tracking Dashboard

4.1.1.5 Logging artifacts

Artifacts represent any kind of file to save during training, such as plots and model weights. It is possible to log such files as well, and place them within the same run as parameters and metrics. This means everything created within a ML run is saved at one point. Artifact files can be either single local files, or even full directories. The following example creates a local file and logs it to a model run.

```python
import os

mlflow.set_tracking_uri("http://127.0.0.1:5000/")

# Create an example file output/test.txt
file_path = "outputs/test.txt"
if not os.path.exists("outputs"):
    os.makedirs("outputs")
with open(file_path, "w") as f:
    f.write("hello world!")

# Start the run based on the run ID and log the artifact
# we just created
```

```python
with mlflow.start_run(run_id=run_id) as run:
    mlflow.log_artifact(
        local_path=file_path,
        # store the artifact directly in run's root
        artifact_path=None
    )
    mlflow.log_artifact(
        local_path=file_path,
        # store the artifact in a specific directory
        artifact_path="data/subfolder"
    )

    # get and print the URI where the artifacts have been logged to
    artifact_uri = mlflow.get_artifact_uri()
    print(f"run_id: {run.info.run_id}")
    print(f"Artifact uri: {artifact_uri}")
```

4.1.1.6 Autolog

Previously, all the parameters, metrics, and files have been logged manually by the user. The *autolog*-feature of MLflow allows for automatic logging of metrics, parameters, and models without the need for an explicit log statements. This feature needs to be activated previous to the execution of a run by calling `MLflow.sklearn.autolog()`.

```python
import mlflow.sklearn
import numpy as np
from sklearn.ensemble import RandomForestRegressor

params = {"n_estimators": 4, "random_state": 42}

mlflow.sklearn.autolog()

run_name = 'autologging model example'
with mlflow.start_run(run_name=run_name) as run:
    rfr = RandomForestRegressor(
        **params).fit(np.array([[0, 1, 0], [0, 1, 0], [0, 1, 0]]), [1, 1, 1])
    print(f"run_id: {run.info.run_id}")

mlflow.sklearn.autolog(disable=True)
```

Even though this is a very convenient feature, it is a good practice to log metrics manually, as this gives more control over a ML run.

4.1.2 MLflow Models

MLflow Models manages and deploys models from various different ML libraries such as scikit-learn, TensorFlow, PyTorch, Spark, and many more[5]. It includes a generic `MLmodel` format that acts as a standard format to package ML models so they can be used in different projects and environments. The `MLmodel` format defines a convention that saves the model in so called *"flavors"*. For example `mlflow.sklearn` allows to load mlflow models back into scikit-learn. The stored model can also be served easily and conveniently using these *flavors* as a python function either locally, in Docker-based REST servers containers, or on commercial serving platforms like AWS SageMaker or AzureML. The following example is based on the scikit-learn library.

```python
# Import the sklearn models from MLflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestRegressor

mlflow.set_tracking_uri("http://127.0.0.1:5000/")

run_name = "models-example-run"
params = {"n_estimators": 4, "random_state": 42}

# Start an MLflow run, train the RandomForestRegressor example model, and
# log its parameeters. In the end the model itself is logged and stored in MLflow
run_name = 'Model example'
with mlflow.start_run(run_name=run_name) as run:
    rfr = RandomForestRegressor(**params).fit([[0, 1, 0]], [1])
    mlflow.log_params(params)
    mlflow.sklearn.log_model(rfr, artifact_path="sklearn-model")

model_uri = "runs:/{}/sklearn-model".format(run.info.run_id)
model_name = f"RandomForestRegressionModel"
```

---

[5]https://MLflow.org/docs/latest/models.html

```python
print(f"model_uri: {model_uri}")
print(f"model_name: {model_name}")
```

Once a model is stored in the correct format it can be identified by its `model_uri`, loaded, and used for prediction.

```python
import mlflow.pyfunc

# Load the model and use it for predictions
model = mlflow.pyfunc.load_model(model_uri=model_uri)
data = [[0, 1, 0]]
model_pred = model.predict(data)
print(f"model_pred: {model_pred}")
```


Figure 4.4: MLflow Models

4.1.3 MLflow Model Registry

The MLflow Model Registry provides a central model store to manage the lifecycle of an ML Model. This allows to register MLflow models like the *RandomForestRegressor* from the previous section to the Model Registry and include model versioning, stage transitions, and

annotations. In fact, by running `MLflow.sklearn.log_model` we already did exactly that. Look at how easy the MLflow API is to use. Let's have a look at the code again.

```python
import mlflow.sklearn
import mlflow.pyfunc
from sklearn.ensemble import RandomForestRegressor

mlflow.set_tracking_uri("http://127.0.0.1:5000/")

run_name = "registry-example-run"
params = {"n_estimators": 4,
          "random_state": 42}

run_name = 'model registry example'
with mlflow.start_run(run_name=run_name) as run:
    rfr = RandomForestRegressor(**params).fit([[0, 1, 0]], [1])
    mlflow.log_params(params)
    # Log and store the model and the MLflow Model Registry
    mlflow.sklearn.log_model(rfr, artifact_path="sklearn-model")

model_uri = f"runs:/{run.info.run_id}/sklearn-model"
model_name = f"RandomForestRegressionModel"

model = mlflow.pyfunc.load_model(model_uri=model_uri)
data = [[0, 1, 0]]
model_pred = model.predict(data)
print(f"model_pred: {model_pred}")
```

Yet, it is also possible to register the MLflow model in the model registry by calling `MLflow.register_model` such as show in belows example.

```python
# The previously stated Model URI and name are needed to register a MLflow Model
mv = mlflow.register_model(model_uri, model_name)
print("Name: {}".format(mv.name))
print("Version: {}".format(mv.version))
print("Stage: {}".format(mv.current_stage))
```

Once registered to the model registry the model is versioned. This enables to load a model based on a specific version and to change a model version respectively. A registered model

can be also modified to transition to another version or stage. Both use cases are shown in the example below.

```python
import mlflow.pyfunc

# Load model for prediction. Keep note that we now specified the model version.
model = mlflow.pyfunc.load_model(
    model_uri=f"models:/{model_name}/{mv.version}"
)

# Predict based on the loaded model
data = [[0, 1, 0]]
model_pred = model.predict(data)
print(f"model_pred: {model_pred}")
```

Let's stage a model to `'Staging'`. The for-loop below prints all registered models and shows that there is indeed a model with a `'Staging'`-stage.

```python
# Transition the model to another stage
from mlflow.client import MLflowClient

client = MlflowClient()

stage = 'Staging'  # None, Production

client.transition_model_version_stage(
    name=model_name,
    version=mv.version,
    stage=stage
)

# print registered models
for rm in client.search_registered_models():
    pprint(dict(rm), indent=4)
```

4.1.4 MLflow Projects

MLflow Projects allows to package code and its dependencies as a *project* that can be run reproducible on other data. Each project includes a *MLproject* file written in the *YAML*

syntax that defines the projects dependencies, and the commands and arguments it takes to run the project. It basically is a convention to organizes and describe the model code so other data scientists or automated tools can run it conveniently. MLflow currently supports four environments to run your code: *Virtualenv*, *Conda*, *Docker Container*, and *system environment*. A very basic `MLproject` file is shown below that is run in an *Virtualenv*

```
name: mlprojects_tutorial

# Use Virtualenv: alternatively conda_env, docker_env.image
python_env: <MLFLOW_PROJECT_DIRECTORY>/python_env.yaml

entry_points:
  main:
    parameters:
      alpha: {type: float, default: 0.5}
      l1_ratio: {type: float, default: 0.1}
    command: "python wine_model.py {alpha} {l1_ratio}"
```

A project is run using the `MLflow run` command in the command line. It can run a project from either a local directory or a GitHub URI. The `MLproject` file shows that two parameters are passed to the `MLflow run` command. This is optional in this case as they have default values. It is also possible to specify extra parameters such as the experiment name or to specify the tracking uri (check the official documentation[6] for more). Below is a possible CLI command show to run the MLflow Project. By setting the `MLFLOW_TRACKING_URI` environment variable it is possible to also specify an execution backend for the run.

```
# Run the MLflow project from the current directory
# The parameters are optional in this case as the MLproject file has defaults
mlflow run . -P alpha=5.0

# It is also possible to specify an experiment name or to specify the
# Tracking_URI, e.g.
MLFLOW_TRACKING_URI=http://localhost:<PORT> mlflow run . --experiment-name="models-exp

# Run the MLflow project from a Github URI and use the localhost as backend
MLFLOW_TRACKING_URI=http://localhost:<PORT> MLflow run https://github.com/seblum/mlops
```

---

[6]https://mlflow.org/docs/latest/python_api/mlflow.projects.html

The MLflow Projects API allows to chain projects together into workflows and also supports launching multiple runs in parallel. Combining this with for example the MLflow Tracking API enables an easy way of hyperparameter tuning to develop a model with a good fit.

4.2 MLFflow Architecture

While MLflow can be run locally for your personal model implementation, it is usually deployed on a distributed architecture for large organizations or teams. The MLflow backend consists of three different main components, *Tracking Server*, *Backend Store*, and *Artifact Store*, all of which can reside on remote hosts.

The MLflow client can interface with a variety of backend and artifact storage configurations. The official MLflow documentation[7] outlines several detailed configurations. The example below depicts the main interaction between the different architectural components of a remote MLflow Tracking Server, a Postgres database for backend storage, and an S3 bucket for artifact storage.

Figure 4.5: MLflow Architecture Diagram

4.2.1 MLflow Tracking Server

The MLflow *Tracking Server* is the main component that handles the communication between the REST API to log parameters, metrics, experiments and metadata to a storage solution. The server uses both, the backend store and the artifact store to store and read data from. Although it is possible to track parameters without running a server (e.g. locally), it is recommended to create a MLflow tracking server to log your data to. Some of the functionality of the API is also available via the web interface, for example to create an experiment. Further, the Tracking web user interface allows to view runs easily in the web browser.

Running the CLI command `mlflow ui` starts a web server on your local machine serving the MLFlow UI. Alternatively, a remote MLflow tracking server[8] serves the same UI which can be accessed using the server's URL `http://<TRACKING-SERVER-IP-ADDRESS>:5000` from any machine that can connect to the tracking server.

4.2.2 MLflow Backend Store

The MLflow *Backend Store* is where MLflow stores experiment and run data like parameters, and metrics. It is usually a relational database which means that all metadata will be stored, but no large data files.

MLflow supports two types of backend stores: *file store* and *database-backed store*. By default, the backend store is set to the local file store backend at the `./mlruns` directory. A database-backed store must be configures using the `--backend-store-uri`. MLflow supports encoded Databases like *mysql*, *mssql*, *sqlite*, and *postgresql*, and it is possible to use a variety of externally hosted metadata stores like Azure MySQL, or AWS RDS. To be able to use the *MLflow Model Registry* the server must use a database-backed store.

4.2.3 MLflow Artifact Store

In addition to the Backend Store the *Artifact Store* is another storage place for the MLflow tracking server. It is the location to store large data of an ML run that are not suitable for the Backend Store or a relational database respectively. This is where MLflow users log their artifact outputs, or data and image files to. The user can access these artifacts via HTTP requests to the MLflow Tracking Server.

The location to the server's artifact store defaults to local `./mlruns` directory. It is possible to specify another artifact store server using `--default-artifact-root`. The MLflow client

---

[8]https://mlflow.org/docs/latest/tracking.html#tracking-server

caches the artifact location information on a per-run basis. It is therefore not recommended to alter a run's artifact location before it has terminated.

The *Artifact Store* needs to be configured when running MLflow on a distributed system. In addition to local file paths, MLflow supports to configure the following cloud storage resources as an artifact stores: Amazon S3, Azure Blob Storage, Google Cloud Storage, SFTP server, and NFS.

5 Kubernetes

Kubernetes (short: K8s) is greek and means pilot. K8s is an applications orchestrator that originated from Google and is open source. Beeing an application orchestrator, K8s deploys and manages application containers. It scales up and down so called Pod (A Pod manages a container) as needed and allows for zero downtime as well as the possibility of rollbacks. Thus, the meaning of *pilot* relates to its functionining in piloting containers.

Prerequisites

As a prerequisites to go through this tutorial and implement the scripts one has to have Docker installed, Kubectl to interact via the command line with K8s, as well as Minikube to run a K8s cluster on a local machine. Please refer to the corresponding sites.

5.1 Core Components

5.1.1 Nodes

A K8s Cluster usually consistes of a set of nodes. A Node can hereby be a virtual machine (VM) in the cloud, e.g. AWS, Azure, or GCP, or a node can also be of course a physical on-premise instance. K8s distinguishes the nodes between a *master node* and *worker nodes*. The master node is basically the brain of the cluster. This is where everything is organized, handled, and managed. In comparison, a worker nodes is where the heavy lifting is happening, such as running application. Both, master and worker nodes communicate with each other via the so called kubelet. One cluster has only one master node and usually multiple worker nodes.

5.1.1.1 Master & Control Plane

To be able to work as the brain of the cluster, the master node contains a controll plane made of several components, each of which serves a different function.

Figure 5.1: K8s Cluster

- Scheduler
- Cluster Store
- API Server
- Controller Manager
- Cloud Controller Manager

Figure 5.2: Master Node

5.1.1.1.1 API Server   The api servers serves as the connection between the frontend and the K8s controll plane. All communications, external and interal, go through it. Frontend to Kubernetes Controll Plane. It exposes a restful api on port 443 to allow communication, as well as performes authentication and authorization checks. Whenever we perform something on the K8s cluster, e.g. using a command like `kubectl apply -f <file.yaml>`, we communicate with the api server (what we do here is shown in the section about pods).

5.1.1.1.2 Cluster store   The cluster store stores the configuration and state of the entire cluster. It is a distributed key-value data store and the single source of truth database of the

cluster. As in the example before, whenever we apply a command like `kubectl apply -f <file.yaml>`, the file is stored on the cluster store to store the configuration.

5.1.1.1.3 Scheduler   The scheduler of the control plane watches for new workloads/pods and assigns them to a node based on several scheduling factors. These factors include whether a node is healthy, whether there are enough resources available, whether the port is available, or according to affinity or anti-affinity rules.

5.1.1.1.4 Controller manager   The controller manager is a daemon that manages the control loop. This means, the controller manager is basically a controller of other controllers. Each controller watches the api server for changes to their state. Whenever a current state of a controller does not match the desired state, the control manager administers the changes. These controllers are for example replicasets, endpoints, namespace, or service accounts. There is also the cloud controller manager, which is responsible to interact with the underlying cloud infrastructure.

5.1.1.2 Worker Nodes

There worker nodes are the part of the cluster where the heavy lifting happens. Their VMs (or physical machines) often run linux and thus provide a suitable and running environment for each application.

Figure 5.3: Worker Node

A worker node consists of three main components.

- Kubelet
- Container runtime
- Kube proxy

5.1.1.2.1 Kubelet   The kubelet is the main agent of a worker node that runs on every single node. It receives pod definitions from the API server and interacts with the container runtime to run containers associated with the corresponding pods. The kubelet also reports node and pod state to the master node.

5.1.1.2.2 Container runtime   The container runtime is responsible to pull images from container registries, e.g. from DockerHub, or AWS ECR, as well as starting, and stoping containers. The container runtime thus abstracts container management for K8s and runs a Container Runtime Interface (CRI) within.

5.1.1.2.3 Kube-proxy   The kube-proxy runs on every node via a DaemondSet. It is responsible for network communications by maintaining network rules to allow communication to pods from inside and outside the cluster. If two pods want to talk to each other, the kube-proxy handles their communication. Each node of the cluster gets its own unique IP adress. The kube-proxy handels the local cluster networking as well as routing the network traffic to a load balanced service.

5.1.2 Pods

A pod is the smallest deployable unit in K8s (In contrast to K8s, the smallest deployable unit for docker are containers.). Therefore, a pod is a running process that runs on a clusters' node. Within a pod, there is always one *main container* representing the application (in whatever language written, e.g. JS, Python, Go). There also may or may not be *init containers*, and/or *side containers*. Init containers are containers that are executed before the main container. Side containers are containers that support the main containers, e.g. a container that acts as a proxy to your main container. There may be volumes specified within a pod, which enables containers to share and store data.

Figure 5.4: Pod

The containers running within a pod communicate with each other using localhost and whatever port they expose. The port itself has a unique ip adress, which enables outward communication between pods.

The problem is that a pods does not have a long lifetime (also denoted as ephemeral) and is disposable. This suggests to never create a pod on its own within a K8s cluster and to rather use controllers instead to deploy and maintain a pods lifecycle, e.g. controllers like *Deployments*. In general, managing ressources in K8s is done via an imperative or declarative management.

5.1.3 Imperative & Declarative Management

Imperative management means managing the pods via a CLI and specifying all necessary parameters using it. It is good for learning, troubleshooting, and experimenting on the cluster. In contrast, the declarative approach uses a yaml file to state all necessary parameters needed for a ressource, and then using the CLI to administer the changes. The declarative approach is reproducible, which means the same configuration can be applied in different environments (prod/dev). This is best practice to use when building a cluster. As stated, this differentiation does not only hold for pods, but for all ressources within a cluster.

5.1.3.1 Imperative Management

```
# start a pod by specifying the pods name,
# the container image to run, and the port exposed
kubectl run <pod-name> --image="<image-name>" --port=80


# run following command to test the pod specified
# It will forward to localhost:8080
kubectl port-forward pod/<pod-name> 8080:80
```

5.1.3.2 Declarative Management / Configuration

Declarative configuration is done using a *yaml* format, which works on key-value pairs.

```
# pod.yaml
apiVersion: v1
# specify which kind of configuration this is
# lets configure a simple pod
```

```yaml
kind: Pod
# metadata will be explained later on in more detail
metadata:
  name: hello-world
  labels:
  name: hello-world
spec:
  # remember: a pod is a selection of one or more containers
  # we could therefore specify multiple containers
  containers:
    # specify the container name
  - name: hello
  # specify which container image should be pulled
  image: seblum/mlops-public:cat-v1
  # ressource configurations will be handled later as well
  ressources:
    limits:
    memory: "128Mi"
    cpu: "500m"
  # specify the port on which the container should be exposed
  # similar to the imperative approach
  ports:
      ContainerPorts: 80
```

Appyl this declarative configuration using the following kubectl command via the CLI.

```bash
kubectl apply -f "file-name.yaml"


# similar to before, run following to test your pod on localhost:8080
kubectl port-forward pod/<pod-name> 8080:80
```

5.1.3.3 Kubectl

One word to interacting with the cluster using the CLI. In general, *kubectl* is used to interact
with the K8s cluster. This allows to run and apply pod configurations such as seen before,
as well as the already shown port forwarding. We can also inspect the cluster, see what
ressources are running on which nodes, see their configurations, and watch their logs. A
small selection of commands are shown below.

```
# forward the pods to localhost:8080
kubectl port-forward <ressource>/<pod-name> 8080:80

# show all pods currently running in the cluster
kubectl get pods

# delete a specific pod
kubectl delete pods <pod-name>

# delete a previously applied configuration
kubectl delete -f <file-name>

# show all instances of a specific resource running on the cluster
# (nodes, pods, deployments, statefulsets, etc)
kubectl get <resource>

# describe and show specific settings of a pods
kubectl describe pod <pod-name>
```

5.2 Application Deployment & Design

5.2.1 Deployments

We should never deploy a pod using `kind:Pod`. Pods are ephemeral, so never treat them like pets. They do not heal on their own and if a pod is terminated, it does not restart by themselves. This is dangerous as there should always be one replica running of and application. This demands for a mechanism for the application to self heal and this is exactly where *Deployments* and *ReplicaSets* come in to solve the problem.

In general, Pods should be managed through Deployments. The purpose of a Deployment is to facilitate software deployment. They manage releases of a new application, they provide zero downtime of an application and create a ReplicaSet behind the scenes. K8s will take care of the full deployment process when applying a Deployment, even if we want to make a rolling update to change the version.

5.2.1.1 ReplicaSets

A ReplicaSet makes sure that a desired number of pods is running. When looking at Pods'
name of a Deployment, it usually has a random string attached. This is because a deployment
can have multiple replicas and the random suffix ensures a different name after all. The way
ReplicaSets work is that they implement a background control loop that checks the desired
number of pods are always present on the cluster. We can specify the number of replicas by
creating a yaml-file of a Deployment, similar to previous specifications done to a Pod. As a
reminder, the Deployment can be applied using the `kubectl apply -f` as well.

```yaml
# deployment.yaml
apiVersion: apps/v1
# specify that we want a deployment
kind: Deployment
metadata:
  name: hello-world
spec:
  # specify number of replicas
  replicas: 3
  selector:
    matchLabels:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-world
        image: seblum/mlops-public:cat-v1
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 5000
```

5.2.1.2 Rolling updates

A rolling update means that a new version of the application is rolled out. In general, a basic deployments strategy will delete every single pod before it creates a new version. This is very dangerous since there is downtime. The preferred strategy is to perform a rolling update. This ensures keeping traffic to the previous version until the new one is up and running and alternates traffic until the new version is fully healthy. K8s perfoms the update of an application while the application is up and running. For example, when there are two replicasets running, one with version v1 and one with v2, K8s performs the update such that it only scales v1 down when v2 is already up and running and the traffic has been redirected to v2 as well. How do the deployments need to be configured for that?

```yaml
# deployment_rolling-update.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  replicas: 3
  # a few new things have been added here
  revisionHistoryLimit: 20
  # specify the deployments strategy
  strategy:
    type: RollingUpdate
    rollingUpdate:
      # only one pod at a time to become unavailable
      # in our case scaling down of v1
      maxUnavailable: 1
      # never have more than one pod above the mentioned replicas
      # with three replicas, there will never be 5 pods running during a rollout
      maxSurge: 1
  selector:
    matchLabels:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
      annotations:
        # just an annotation the get the version change
```

```yaml
        kubernetes.io/change-cause: "seblum/mlops-public:cat-v2"
    spec:
      containers:
      - name: hello-world
      # only change specification of the image to v2, k8s performs the update itself
        image: seblum/mlops-public:cat-v2
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 5000
```

The changes can be applied as well using `kubectl apply -f "file-name.yaml"`. Good to know, K8s is not deleting the replicasets of previous versions. They are still stored on the Cluster Store. The `spec.revisionHistory: <?>` state in the yaml denoted this. The last ten previous versions are stored on default. However, it doesn't really make sense to keep more such for example in the previous yaml where there are the last 20 versions specified. This enables to perform **Rollbacks** to previous versions. To not have discrepancies in a cluster, one should always update using the declarative approach. Below stated are a number of commands that trigger and help with a rollback or with rollouts in general.

```bash
# check the status of the current deployment process
kubectl rollout status deployments <name>


# pause the rollout of the deployment.
kubectl rollout pause


# check the rollout history of a specific deployment
kubectl rollout history deployment <name>


# undo the rollout of a deployment and switch to previous version
kubectl rollout undo deployment <name>


# goes back to a specific revision
# there is a limit of history and k8s only keeps 10 previous versions
kubectl rollout undo deployment <name> --to-revision=
```

5.2.2 Resource Management

Besides the importance of a healthy application itself, there should be also enough resources allocated so the application can perform well, e.g. memory & CPU. Yet, it should also only consume the resources needed and not block unneeded ones. It might be dangerous, as one application using a lot of ressources, leaving nothing left for other applications and eventually starving them. To prevent this from happening in K8s. there can be a minimum amount of resources defined a container needs (request) as well as the maximum amount of resources a container can have (limit). Configuring limits and requests for a container can be done within the spec for a Pod or Deployment. Actually, we have been using them all the time previously.

```yaml
# resource-management.yaml
apiVersion: apps/v1
# specify that we want a deployment
kind: Deployment
metadata:
  name: rm-deployment
spec:
  # specify number of replicas
  replicas: 3
  selector:
    matchLabels:
      app: rm-deployment
  template:
    metadata:
      labels:
        app: rm-deployment
    spec:
      containers:
      - name: rm-deployment
        image: seblum/mlops-public:cat-v1
        requests:
          memory: "512Mi"
          cpu: "1000m"
        # Limits
        limits:
          memory: "128Mi"
          cpu: "500m"
        ports:
```

```
            - containerPort: 5000
```

5.2.3 DaemonSets

The master node of K8s decides on what worker nodes a pod is scheduled or not. However, there are times where we want to have a copy of a pod across the cluster. A *DaemonSet* ensures a copy of the specified Pod is exactly doing this. This can be useful for example to deploy system daemons such as log collectors and monitoring agents. DaemonSets are automatically deployed on every single node, unless specified on which node to run. They therefore do not need a specification of nodes and can scale up and down with the cluster as needed. They will automatically scheduled a pod on each new node.

The given example deploys a DaemonSet to cover logging using K8s FluendID.

```yaml
# daemonsets.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: logging
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: logging
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      # this toleration is to have the daemonset runnable on master nodes
      # remove it if your masters can't run pods
```

```yaml
    - key: node-role.kubernetes.io/master
      operator: Exists
      effect: NoSchedule
    # specify the containers as done in Pods or Deployments
    volumes:
    - name: varlog
      hostPath:
        path: /var/log
    - name: varlibdockercontainers
      hostPath:
        path: /var/lib/docker/containers
    containers:
    - name: fluentd-elasticsearch
      # allows to collect logs from nodes
      image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
      resources:
        limits:
          memory: 200Mi
        requests:
          cpu: 100m
          memory: 200Mi
      volumeMounts:
      - name: varlog
        mountPath: /var/log
      - name: varlibdockercontainers
        mountPath: /var/lib/docker/containers
        readOnly: true
    terminationGracePeriodSeconds: 30
```

5.2.4 StatefulSets

StatefulSets are used to deploy and manage stateful applications. Stateful applications are applications which are long lived, for example databases. Most applications of K8s are stateless as they only run for a specific task. However, a database is a state of truth and should be present at all time. StatefulSets manage the pods based on the same container specifications such as Deployments.

Lets assume we have a StatefulSet with 3 replicas. Each Pod has a PV attached.

5.2.5 Jobs & Cron Jobs

Using the *busybox* image in the section about volumes we experienced that the image is very short lived. K8s is not aware of this and runs into a CrashLoopBackOff-Error. K8s will try and restart the container itself though until it BackOffs completley. Because the image is so short live, a job within the image has to be executed such as done with a shell command previously. However, what if we have a simple task that only should run like every 5 minutes, or every single day? A good idea is to use CronJobs for such tasks that start the image if needed. When comparingJobs jobs and CronJobs, jobs execute only once, whereas CronJobs execute depending on an specified expression.

The following job simulates a backup to a database that runs 30 seconds in total. The part in the `args` specifies that the container will sleep for 20 seconds (the hypothetical backup). Afterward, the container will wait 10 seconds to shut down, as specified in `ttlSecondsAfterFinished`.

```yaml
# job.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: db-backup-job
spec:
  # time it takes to terminate the job for one completion
  ttlSecondsAfterFinished: 10
  template:
    spec:
      containers:
      - name: backup
        image: busybox
        command: ["/bin/sh", "-c"]
        args:
          - "echo 'performing db backup...' && sleep 20"
      restartPolicy: Never
```

The CronJob below runs run every minute. Given the structure of (* * * * * *) - (Minutes Hours Day-of-month Month Day-of-week Year), the cronjob expression defines as follows:

```
# cronjob.yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: db-backup-cron-job
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: backup
            image: busybox
            command: ["/bin/sh", "-c"]
            args:
              - "echo 'performing db backup...' && sleep 20"
          restartPolicy: Never
```

5.3 Services & Networking

5.3.1 Services

To overall question when deploying an application is how we can access it. Each individual Pod has its own IP address. Thereby, the client can access this Pod via its IP address. Previously, we have made the app available via `kubectl port-forward` by forwarding the Pods' IP to localhost. However, should only be done for testing purposes and is not a reliable and stable way to enable access. Since Pods are ephemeral, the client cannot rely on the ip address alone. For example, if an application is scaled up or down, there will be new IPs associated with each new Pod.

Instead, *Services* should be used. A service is an abstract way to expose an application as a network service. The service can connect access to a pod via an interal reference, so a change of the Pods IP will not interfere with its accessibility. The service itself has a stable IP adress, a stable DNS name, and a stable port. This allows for a reliable and stable connection from the client to the service, which can then direct the traffic to the pod. There are different types of Services.

- ClusterIP (Default)

Figure 5.5: Services

- NodePort
- ExternalName
- LoadBalancer

5.3.1.1 ClusterIP

The ClusterIP is the default K8s service. This service type will be chosen if no specific type is selected. The ClusterIP is used for cluster internal access and does not allow for external communication. If one Pod wants to talk to another Pod inside the cluster, it will use ClusterIP to do so. The service will allow and send traffic to any pod that is healthy.

5.3.1.2 NodePort

The NodePort service allows to open a static port simultaneously on all nodes. Its range lies between between 30.000/32.767. If a client wants to communicate with a node of the cluster, the client directly communicates with the node via its IP address. When the request reaches the port of the node, the NodePort service handles the request and forwards it to the specifically marked pod. This way, an application running on a pod can be exposed directly on a nodes' IP under a specific port. You'll be able to contact the NodePort Service from outside the cluster by requesting `<NodeIP>:<NodePort>`.

Using a NodePort is beneficial for example when a request is sent to a node without a pod. The NodePort service will forward the request to a node which has a healthy associated pod running. However, only having one service specified per port is also a disadvantage. Having one ingress and multiple services is more desireable. The point of running K8s in the cloud is to scale up and down and if the NodeIP address changes, then we have a problem. So we should not aim to access a Node IP directly in the first place

If applying below example using the frontend-deployment and the backend-deployment, we can access the frontend using the nodeport. Since using minikube, we can access the service by using `minikube service frontend-node --url`. Using the given IP adress it is possible to access the frontend using the NodePort service. We can also test the NodePort service when inside of a node. When accessing a node e.g. via `minikube ssh`, we can run `curl localhost:PORT/` inside the node to derive the to derive the website data from the frontend.

5.3.1.3 LoadBalancer

Loadbalancers are a standard way of exposing applications to the extern, for example the internet. Loadbalancers automatically distribute incoming traffic across multiple targets to balance the load in an equal level. If K8s is running on the cloud, e.g. AWS or GCP, a Network Load Balancer (NLB) is created. The Cloud Controller Manager (remember the Controller Manager of a Node) is resposible to talk to the underlying cloud provier. In Minikube, the external IP to access the application via the LoadBalancer can be exposed using the command `minikube tunnel`.

5.3.1.4 default kubernetes services

There are also default K8s services created automatically to access K8s with the K8s API. Check the endpoints of the *kubernetes* service and the endpoints of the *api-service* pod within kube-system namespace. They should be the same.

It is also possible to show all endpoints of the cluster using

```
kubectl get endpoints
```

5.3.1.5 Exemplary setup of database and frontend microservices

The following example show the deployment and linking of two different deployments. A *frontend-deployment.yaml* that pulls a container running a Streamlit App[1], and a *database-deployment.yaml* that runs a flask application exposing a dictionary as an exemplary and very basic database. The frontend accesses the flask database using a ClusterIP Service linked to the database-deployment. It also exposes an external IP via a Loadbalancer Service, so the streamlit app can be accesses via the browser and without the use of `kubectl port-forward`. Since minikube is a closed network, use `minikube tunnel` to allow access to it using the LoadBalancer.

---

[1]https://streamlit.io/

When looking at the ClusterIP service with `kubectl describe service backendflask` the IP address of the service to exposes, as well as the listed endpoints that connect to the database-deployments are shown. One can compare them to the IPs of the actual deployments - they are the same.

```yaml
# services_frontend-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: seblum/mlops-public:frontend-streamlit
        imagePullPolicy: "Always"
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        # enviroment variable defined in the application and dockerfile
        # value is ip adress of the order
        env:
            # using the ip adress would be a bad idea.
            # use the service ip adress.
            # value: "<order-service-ip-adress>:8081"
            # how to do it should be this.
            # we reference to the order service
          - name: DB_SERVICE
            value: "backendflask:5001"
        ports:
          # we can actually use the actual ip of the service or
```

```yaml
          # use the dns, as done in the example above.
        - containerPort: 8501
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-lb
spec:
  type: LoadBalancer
  selector:
    app: frontend
  ports:
  - port: 80
    targetPort: 8501
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-node
spec:
  type: NodePort
  selector:
    app: frontend
  ports:
  - port: 80
    targetPort: 8501
    nodePort: 30000
```

```yaml
# services_backend-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backendflask
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backendflask
  template:
```

```yaml
    metadata:
      labels:
        app: backendflask
    spec:
      containers:
      - name: backendflask
        image: seblum/mlops-public:backend-flask
        imagePullPolicy: "Always"
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: backendflask
spec:
  # send traffic to any pod that matches the label
  type: ClusterIP # does not need to be specified
  selector:
    app: backendflask
  ports:
    # port the service is associated with
  - port: 5001
    # port to access targeted by the access
    # in our case has to be the same as in backendflask.
    targetPort: 5000
```

5.3.2 Service Discovery

Service Discovery is a mechanism that lets applications and microservices locate each other on a network. In fact, we have aready used Service Discovery in the previous sections, they just haven't been mentioned yet. If a client wants to communicate with the application, it should not use the IP of an individual Pod should not use the individual pod ip. Instead, we should rely on services as they have a stable IP address. We have already seen this in the

section about Services. Yet, each pod has also an individual DNS (Domain Name System). A DNS translates a domain names to an IP address, just one lookes up a number in a telephone book, so it's much easier to reference to a resource online. This is where service Discovery enters the game.

Figure 5.6: Service Discovery

Whenever a service is created, it is registered in the service registry with the service name and the service IP. Most clusters use CoreDNS as a service registry (this would be the telephone book itself). When having a look at the minikube cluster one should see are *core-dns* service running. Now you know what it is for. Having a closer look using `kubectl describe svc <name>`, the core-dns service has only one endpoint. If you want to have an even closer look, you can dive into a pod itself and check the file /etc/resolv.conf. There you find a nameserver where the IP is the one of the core-dns.

```
# when querying services, it necessary
# to specify the corresponding namespace
kubectl get service -n kube-system

# command for queriying the dns
nslookup <podname>
```

5.3.2.1 kube-proxy

As mentioned earlier, each node has three main components: Kubelet, Container Runtime, and the Kube-Proxy. *Kube-Proxy* is a network proxy running on each node and is responsible for internal network communications as well as external.

It also implements a controller that watches the API server for new services and endpoints. Whenever there is a new service or endpoint, the kube-proxy creates a local IPVS rule (IP Virtual Server) that tells the node to intercept traffic destined to the ClusterIP Service. IPVS

is built on top of the network filter and implements a transport-layer load balancing. This gives the ability to load balance to real service as well as redirecting traffic to pods that match service label selectors.

This means, kube-proxy is intercepting all the requests and makes sure that when a request to the ClusterIP service is sent using endpoints, the request is forwarded to the healthy pods behind the endpoint.

5.4 Volume & Storage

Since Pods are ephemeral, any data associated is deleted when a Pod or container restarts. Applications are run stateless the majority of the times, meaning the data does not needs to be kept on the node and the data is stored on an external database. However, there are times when the data wants to be kept, shared between Pods, or when it should persist on the host file system (disk). As described in the section about Pods, a Pod can contain volumes. Volumes are exactly what is needed for such tasks. They are used to store and access data which can be persistent or long lived on K8s.

There are different types of volumes, e.g.:

- EmptyDir
- HostPath Volume
- awsElasticBlockStore: AWS EBS volumes are persistent and originally unmounted. They are read-write-once-only tough.
- There are multiple other types of volumes, a full list can be found here: https://kube rnetes.io/docs/concepts/storage/volumes/#volume-types

5.4.1 EmptyDir Volume

An EmptyDir Volume is initially empty (as the name suggests). The volume is a temporary directory that shares the pods lifetime. If the pod dies, the contents of the emptyDir are lost as well. The EmptyDir is also used to share data between containers inside a Pod during runtime.

```yaml
# volume_empty-dir.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: emptydir-volume
```

```yaml
spec:
  selector:
    matchLabels:
      app: emptydir-volume
  template:
    metadata:
      labels:
        app: emptydir-volume
    spec:
      # add a volume to the deployment
      volumes:
        # mimics a caching memory type
        - name: cache
          # specify the volume type and the temp directory
          emptyDir: {}
        # of course there could also be a second volume added
      containers:
      - name: container-one
        image: busybox
        # image used for testing purposes
        # since the testing image immediately dies, we want to
        # execute an own sh command to interact with the volume
        volumeMounts:
          # The name must match the name of the volume
          - name: cache
            # interal reference of the pod
            mountPath: /foo
        command:
          - "/bin/sh"
        args:
          - "-c"
          - "touch /foo/bar.txt && sleep 3600"
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        # create a second container with a different internal mountPath
      - name: container-two
        image: busybox
```

```
      volumeMounts:
        - name: cache
          mountPath: /footwo
      command:
        - "sleep"
        - "3600"
      resources:
        limits:
          memory: "128Mi"
          cpu: "500m"
```

As stated in the yaml, the busybox image immediately dies. If the Containers where created without the shell commands, the pod would be in a crashloopbackoff-state. To prevent the Pod to do so it is caught with the `sleep`commands until it scales down. Accessing a container using `kubectl exec`, it can be checked whether the foo/bar.txt has been created in *container-one*.

When checking the second container *container-two*, the same file should be visible as well. This is because both containers refer to the same volume. Keep in mind though that the mountPath of the *container-two* is different.

```
# get in container
kubectl exec -it <emptydir-volume-name> -c container-one -- sh
# check whether bar.txt is present
ls

# accessing the second container, there is also a file foo/bar.txt
# remember, both containers share the same volume
kubectl exec -it <emptydir-volume-name> -c container-two -- sh
ls
```

5.4.2 HostPath Volume

THe HostPath Volume type is used when an application needs to access the underlying host file system, meaning the file system of the node. HostPath represents a pre-existing file or directory on the host machine. However, this can be quite dangerous and should be used with caution. If having the right access, the application can interfere and basically mess up the host. It is therefore recommended to set the rights to read only to prevent this from happening.

```yaml
# volume_hostpath.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hostpath-volume
spec:
  selector:
    matchLabels:
      app: hostpath-volume
  template:
    metadata:
      labels:
        app: hostpath-volume
    spec:
      volumes:
        - name: var-log
          # specify the HostPath volume type
          hostPath:
            path: /var/log
      containers:
      - name: container-one
        image: busybox
        volumeMounts:
          - mountPath: /var/log
            name: var-log
            readOnly: true
        command:
          - "sleep"
          - "3600"
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
```

Similar to the EmptyDir Volume example, you can check the implementation of the HostPath
Volume by accessing the volume. When comparing the file structures of the *hostpath-volume*
deployment and the directory `path: /var/log` on the node the deployment is running, they
should be the same. All the changes made to either on of them will make the changes available
on the other. By making changes via the pod we can directly influence the Node. Again, this

is why it is important to keep it read-only.

```
# access the
kubectl exec -it <hostpath-volume-name> -- sh

# ssh into node
minikube ssh <node>
```

5.4.3 Persistent Volumes

Persistent Volumes allow to store data beyond a Pods lifecycle. If a Pod fails, dies or moves to a different node, the data is still intact and can be shared between pods. Persistent Volume types are implemented as plugins that K8s can support(a full list can be found online). Different types of Persistent Volumes are:

- NFS
- Local
- Cloud Network storage (AWS EBS, Azure File Storage, Google Persistent Disk)

The following example show how the usage of Persistent Volumes works on the AWS cloud. K8s is running on an AWS EKS Cluster and AWS EBS Volumes attached to it. The Container storage interface (CSI) of K8s to use Persistent Volumes is implemented by the EBS provider, e.g. the aws-ebs-plugin. This enables a the use of Persistent Volumes in the EKS cluster. Therefore, a Persistent Volume (PV) is rather the mapping between the storage provider (EBS) and the K8s cluster, than a volume itself. The storage class of a Persistent Volume can be configured to the specific needs. Should the storage be fast or slow, or do we want to have each as a storage? Or might there be other parameters to configure the storage?

If a Pods or Deployments want to consume storage of the PV, they need to get access to the PV. This is done via a so called persistent volume claim (PVC).

All of this is part of a Persistent Volume Subsystem. The Persistent Volume Subsystem provides an API for users and administrators. The API abstracts details of how storage is provided from how it is consumed. Again, the provisioning of storage is done via a PV and the consumption via a PCV.

Listed below are again the three main components when dealing with Persistent Volumes in K8s

- **Persistent Volume**: is a storage resource provisioned by an administrator
- **PVC**: is a user's request for and claim to a persistent volume.

Figure 5.7: Persistent Volume Subsystem

- **Storage Class**: describes the parameters for a class of storage for which Persistent Volumes can be dynamically provisioned.

So how are Persistent Volumes specified in our deployments yamls? As there are `kind:Pod` ressources, there can similarly `kind:PersistentVolume` and `kind:PersistentVolumeClaim` be specified. At first, a PV is created. As we run on minikube and not on the cloud, a local storage in the node is allocated. Second, a PVC is created requesting a certain amount of that storage. This PVC is then linked in the specifications of a Deployment to allow its containers to utilized the storage.

Before applying the yaml-files we need to allocate the local storage by claiming storage on the node and set the paths specified in the yamls. To do this, we ssh into the node using `minikube ssh`. We can then create a specific path on the node such as `/mnt/data/`. We might also create a file in it to test accessing it when creating a PVC to a Pod. Since we do not know yet on what node the Pod is scheduled, we should create the directory on both nodes. Below are all steps listed again.

```
# ssh into node
minikube ssh
# create path
sudo mkdir /mnt/data
# create a file with text
sudo sh -c "echo 'this is a pvc test' > /mnt/data/index.html"
# do this on both nodes as pod can land on either one of them
```

Afterward we can apply the yaml files and create a PV, PVC, and the corresponding Deployment utilizing the PVC. The yaml code below shows this process.

```yaml
# volume_persistent-volume.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mypv
spec:
  # specifiy the capacity of the PersistentVolume
  capacity:
    storage: "100Mi"
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: manual
  hostPath:
    path: "/mnt/data"
    # specify the hostPath on the node
    # that's the path we specified on our node
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc
spec:
  resources:
    requests:
      # we request the same as the PV is specified
      # so we basically request everything
      storage: "100Mi"
  volumeMode: Filesystem
  storageClassName: "manual"
  accessModes:
    - ReadWriteOnce
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pv-pvc-deployment
spec:
```

```yaml
    selector:
      matchLabels:
        app: pv-pvc
    template:
      metadata:
        labels:
          app: pv-pvc
      spec:
        volumes:
          - name: data
            # define the use of the PVC by specifying the name
            # specify the pod/deployment can use the PVC
            persistentVolumeClaim:
              claimName: mypvc
        containers:
        - name: pv-pvc
          image: nginx:latest
          volumeMounts:
            - mountPath: "/usr/share/nginx/html"
              # since the PVC is stated, the container needs to
              # mount inside it
              # name is equal to the pvc name specified
              name: data
          resources:
            limits:
              memory: "128Mi"
              cpu: "500m"
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: pv-pvc
spec:
  type: LoadBalancer
  selector:
    app: pv-pvc
  ports:
```

```
  - port: 80
    targetPort: 80
```

By accessing a pod using `kubectl exec -it <persistent-volume-name> -- sh` we can check whether the path is linked using the PVC. Now, the end result may seem the same as what we did with the HostPath Volume. But it actually is not, it just looks like it since both, the PersistentVolume and the HostPath connect to the Host. Yet, the locally mounted path would be somewhere else when running in the cloud. The PV configuration would point to another storage source instead of a local file system, for example an attached EBS of EFS storage. Since we also created a LoadBalancer service, we can run `minikube tunnel` to expose the application deplyment under localhost:80. It should show the input of the index.html file we created on the storage.

5.5 Environment, Configuration & Security

5.5.1 Namespaces

Namespaces allow to organize resources in the cluster, which makes it more overseeable when there are multiple resources for different needs. Maybe we want to organize by team, department, or according to a development environment (dev/prod), etc. By default, K8s will use the *default*-namespace for resources that have not been specified otherwise. Similarly, kubectl interacts with the default namespace as well. Yet, there are already different namespace in a basic K8s cluster

- **default** - The default namespace for objects with no other namespace
- **kube-system** - The namespace for objects created by the Kubernetes system
- **kube-public** - This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.
- *kube-node-lease* - This namespace for the lease objects associated with each node which improves the performance of the node heartbeats as the cluster scales.

Of course, there is also the possibility of creating ones own namespace and using it by attaching a e.g. Deployment to it, such as seen in the following example.

```yaml
# namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: monitoring

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: monitoring-deployment
  namespace: monitoring
spec:
  replicas: 1
  selector:
    matchLabels:
      app: monitoring-deployment
  template:
    metadata:
      labels:
        app: monitoring-deployment
    spec:
      containers:
      - name: monitoring-deployment
        image: "grafana/grafana:latest"
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 5000
```

When creating a Service, a corresponding DNS entry is created as well, such as seen in the Services section when calling `backendflask` directly. This entry is created according to the namespace which is denoted to the service. This can be useful when using the same configuration across multiple namespaces such as development, staging, and production. It is also possible to reach across namespaces. One needs to use the fully qualified domain name (FQDN) tough, such as `<service-name>.<namespace-name>.svc.cluster.local`.

5.5.2 Labels, Selectors and Annotations

In the previous sections we already made use of labels, selectors, and annotations, e.g. when matching the ClusterIP service to the back-deployments. Labels are a key-value pair that can be attached to objects such as Pods, Deployments, Replicaset, Services, etc. Overall, they are used to organize and select objects.

Annotations are an unstructured key-value mapping stored with a resource that may be set by external tools to store and retrieve any metadata. In contrast to labels and selectors, annotations are not used for querying purposes but rather to attach arbitrary non-identifying metadata. These data are used to assist tools and libraries to work with the K8s ressource, for example to pass configuration around between systems, or to send values so external tools can perform more informed decisions based on the annotations provided.

Selectors are used to filter K8s objects based on a set of labels. A selector basically simply uses a boolean language to select pods. The selector matches the labels under a an all or nothing principle, meaning everything specified in the selector must be fulfilled by the labels. However, this works not the other way around. If there are multiple labels specified and the selector matches only one of them, the selector will match the ressource itself. How a selector matches the labels can be tested using the `kubectl` commands as seen below.

```
# Show all pods including their labels
kubectl get pods --show-labels

# Show only pods that match the specified selector key-value pairs
kubectl get pods --selector="key=value"
kubectl get pods --selector="key=value,key2=value2"

# in short one can also write
kubectl get pods -l key=value
# or also look for multiple
kubectl get pods -l 'key in (value1, value2)'
```

When using ReplicaSets in a Deployment, their selector matches the labels to a specific pod (check e.g. the section describing Deployments). Any Pods matching the label of the selector will be created according to the specified replicas. Of course, there can also be multiple labels specified. The same principle accounts when working with Services. Below example shows two different Pods and two NodePort services. Each service matches to a Pod based on their selector-label relationship. Have a look at their specific settings using `kubectl`. The Nodeport Service *labels-and-selectors-2* has no endpoints, as it is a all-or-none-principle and none of the created Pods matches the label `environment=dev`. In contrast, even though the

Pod *cat-v1* has multiple labels specified `app: cat-v1; version: one`, the NodePort Service *labels-and-selectors* is linked to it. It is also linked to the second Pod *cat-v2*.

```yaml
# labels.yaml
apiVersion: v1
kind: Pod
metadata:
  name: cat-v1
  labels:
    app: cat-v1
    version: one
spec:
  containers:
  - name: cat-v1
    image: "seblum/mlops-public:cat-v1"
    resources:
      limits:
        memory: "128Mi"
        cpu:  "500m"
---
apiVersion: v1
kind: Pod
metadata:
  name: cat-v2
  labels:
    app: cat-v1
spec:
  containers:
  - name: cat-v2
    image: "seblum/mlops-public:cat-v2"
    resources:
      limits:
        memory: "128Mi"
        cpu:  "500m"
---
apiVersion: v1
kind: Service
metadata:
  name: labels-and-selectors
spec:
```

```
  type: NodePort
  selector:
    app: cat-v1
  ports:
  - port: 80
    targetPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: labels-and-selectors-2
spec:
  type: NodePort
  selector:
    app: cat-v1
    environment: dev
  ports:
  - port: 80
    targetPort: 5000
```

5.5.3 ConfigMaps

When building software, the same container image should be used for development, testing, staging, and production stage. Thus, container images should be reusable. What usually changes are only the configuration settings of the application. *ConfigMaps* allow to store such configurations as a simple mapping of key-value pairs. Most of the time, the configuration within a config map is injected using environment variables and volumes. However, ConfigMaps should only be used to store configuration files, not sensitive data, as they do not secure them. Besides allow for an easy change of variables, another benefit of using ConfigMaps is that changes in the configuration are not disruptive, meaning the application can still run while the configuration changes without affecting the application. However, one needs to keep in mind that change made to ConfigMaps and environment variables will not be reflected on already and currently running containers.

The following example creates two different ConfigMaps. The first one includes three environment variables as data. The second one include a more complex configuration of an nginx server.

```yaml
# configmaps.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-properties
data:
  app-name: kitty
  app-version: 1.0.0
  team: engineering
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-conf
data:
  # configuration in .conf
  nginx.conf: |
    server {
        listen       80;
        server_name  localhost;

        location / {
            root   /usr/share/nginx/html;
            index  index.html index.htm;
        }

        # redirect server error pages to the static page /50x.html
        #
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   /usr/share/nginx/html;
        }

        location /health {
            access_log off;
            return 200 "healthy\n";
        }
    }
```

Additionally, a Deployment is created which uses both ConfigMaps. A ConfigMap is declared under `spec.volumes` as well. It is also possible to state a reference to both ConfigMaps simultaneously. The Deployment creates two containers. The first container mounts each ConfigMap as a Volume. Container two uses environment variables to access and configure the key-value pairs of the ConfigMaps and store them on the container.

```yaml
# configmaps_deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: config-map
spec:
  selector:
    matchLabels:
      app: config-map
  template:
    metadata:
      labels:
        app: config-map
    spec:
      volumes:
        # specify ConfigMap nginx-conf
        - name: nginx-conf
          configMap:
            name: nginx-conf
        # specify ConfigMap app-properties
        - name: app-properties
          configMap:
            name: app-properties
        # if both configmaps shall be mounted under one directory,
        # we need to use projected
        - name: config
          projected:
            sources:
              - configMap:
                  name: nginx-conf
              - configMap:
                  name: app-properties
      containers:
        - name: config-map-volume
```

```
      image: busybox
      volumeMounts:
        - mountPath: /etc/cfmp/ngnix
        # is defined here in the nginx-volume to mount
          name: nginx-conf
        # everything from that configMap is mounted as a file
        # the file content is the value themselves
        - mountPath: /etc/cfmp/properties
          name: app-properties
        - mountPath: etc/cfmp/config
          name: config
      command:
        - "/bin/sh"
        - "-c"
      args:
        - "sleep 3600"
      resources:
        limits:
          memory: "128Mi"
          cpu: "500m"
  - name: config-map-env
    image: busybox
    resources:
      limits:
        memory: "128Mi"
        cpu: "500m"
    # as previously, keep the busybox container alive
    command:
      - "/bin/sh"
      - "-c"
    args:
      - "env && sleep 3600"
    env:
      # environment variables to read in from config map
      # for every data key-value pair in config Map, an own
      # environment variable is created, which gets
      # the value from the corresponding key
      - name: APP_VERSION
        valueFrom:
```

```
                configMapKeyRef:
                   name: app-properties
                   key: app-version
            - name: APP_NAME
              valueFrom:
                configMapKeyRef:
                   name: app-properties
                   key: app-name
            - name: TEAM
              valueFrom:
                configMapKeyRef:
                   name: app-properties
                   key: team
            # reads from second config map
            - name: NGINX_CONF
              valueFrom:
                configMapKeyRef:
                   name: nginx-conf
                   key: nginx.conf
```

We can check for the attached configs by accessing the containers via the shell, similar to what we did in the section about Volumes. In the container *config-map-volume*, the configs are saved under the respective `mountPath` of the volume. In the *config-map-env*, the configs are stored as environment variables.

```
# get in container -volume or -env
kubectl exec -it <config-map-name> -c >container-name< -- sh
# check subdirectories
ls

# print environment variables
printenv
```

5.5.4 Secrets

Secrets, as the name suggests, store and manage sensitive information. However, secrets are actually not secrets in K8s. They can quite easily decoded using `kubectl describe` on a secret and decode it using the shell command `echo <password> | base64 -d`. Thus,

sensitive information like database password should never be stored in secrets. There are much better ressources to store such data, for example a Vault on the cloud provider itself. However, secret can be used so that you don't need to include confidential data in your application code. Since they are stored and created independently of the Pods that use them, there is less risk of being exposed during the workflow of creating, viewing, and editing Pods.

It is possible to create secrets using imperative approach as shown below.

```
# create the two secrets db-password and api-token
kubectl create secret generic mysecret-from-cli --from-literal=db-password=123 --from-

# output the new secret as yaml
kubectl get secret mysecret -o yaml

# create a file called secret with a file-password in it
echo "super-save-password" > secret

# create a secret from file
kubectl create secret generic mysecret-from-file --from-file=secret
```

Similar to ConfigMaps, secrets are accessed via an environment variable or a volume.

```
# secrets.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secrets
spec:
  selector:
    matchLabels:
      app: secrets
  template:
    metadata:
      labels:
        app: secrets
    spec:
      volumes:
        # get the secret from a volume
        - name: secret-vol
          secret:
```

```yaml
          # the name of the secret we created earlier
          secretName: mysecret-from-cli
    containers:
    - name: secrets
      image: busybox
      volumeMounts:
        - mountPath: /etc/secrets
          name: secret-vol
      env:
        # nane of the secret in the container
        - name: CUSTOM_SECRET
          # get the secret from an environment variable
          valueFrom:
            secretKeyRef:
              # name and key of the secret we created earlier
              name: mysecret-from-file
              key: secret
      command:
        - "sleep"
        - "3600"
      resources:
        limits:
          memory: "128Mi"
          cpu: "500m"
```

5.5.4.1 Exemplary use case of secrets

When pulling from a private dockerhub repository, applying the deployment will throw an error since there are no username and password specified. As they should not be coded into the deployment yaml itself, they can be accessed via a secret. In fact, a specific secret can be specified for docker registry. The secret can be specified using the imperative approach.

```
kubectl create secret docker-registry docker-hub-private \
--docker-username=YOUR_USERNAME \
--docker-password=YOUR_PASSWORD \
--docker-email=YOUR_EMAIL
```

Finally, the secret is specified in the deployment configuration where it can be accessed during application.

```yaml
# secret_dockerhub.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secret-app
spec:
  selector:
    matchLabels:
      app: secret-app
  template:
    metadata:
      labels:
        app: secret-app
    spec:
      # specifiy the docker-registry secret to be accessed
      imagePullSecrets:
        - name: docker-hub-private
      containers:
      - name: secret-app
        # of course you need an own private repository
        # to pull and change the name accordingly
        image: seblum/private:cat-v1
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 80
```

5.6 Observability & Maintenance

5.6.1 Health Checks

Applications running in service need to be healthy at all times so they are ready to receive traffic. K8s uses a process called health checks to test whether an application is alive. If there are any issues and the application is unhealthy, K8s will restart the process. Yet, checking only whether a process is up and running might be not sufficient. What if, e.g., a client wants to connect to a database and the connection cannot be established, even though the app is

up and running? To solve more specific issues like this, health checks like a *liveness probe* or *readiness probe* can be used. If there have not been specified, K8s will use the default checks to test whether a process is running.

5.6.1.1 Liveness Probe

The Kubelet of a node uses *Liveness Probes* to check whether an application runs fine or whether it is unable to make progress and its stuck on a broken state. For example, it could catch a deadlock, a database connection failure, etc. The Liveness Probe can restart a container accordingly. To use a Liveness Probe, an endpoint needs to be specified. The benefit of this is, that it is simple to define what it means for an application to be healthy just by defining a path.

5.6.1.2 Readiness Probe

Similar to a Liveness Probe, the *Readniness Probe* is used by the kubelet to check when the container is ready to start accepting traffic. A Pod is considered ready when all of its containers are ready. Its configuration is also done by specifying a path to what it means the application is healthy. A lot of frameworks, like e.g. springboot, actually provide a path to use.

Belows configuration shows a Deployment which includes a Liveness and a Readiness Probe. The image of the deployment is set up so its process is killed after a given number of seconds. This has been passed using environment variables such as seen in the script. Both, the Liveness and the Readiness Probe have the same parameters in the given example.

- initialDelaySeconds: The probe will not be called until x seconds after all the containers in the Pod are created.
- timeoutSeconds: The probe must respond within a x-second timeout and the HTTP status code must be equal to or greater than 200 and less than 400.
- periodSeconds: The probe is called every x seconds by K8s
- failureThreshold: The container will fail and restart if more than x consecutive probes fail

```yaml
# healthchecks.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backendflask-healthcheck
spec:
```

```yaml
replicas: 1
selector:
  matchLabels:
    app: backendflask-healthcheck
template:
  metadata:
    labels:
      app: backendflask-healthcheck
      environment: test
      tier: backend
      department: engineering
  spec:
    containers:
      - name: backendflask-healthcheck
        # check whether I have to change the backend app to do this.
        image: "seblum/mlops-public:backend-flask"
        imagePullPolicy: "Always"
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        # Specification of the the Liveness Probe
        livenessProbe:
          httpGet:
            # path of the url
            path: /liveness
            port: 5000
          # time the liveness probe starts after pod is started
          initialDelaySeconds: 5
          timeoutSeconds: 1
          failureThreshold: 3
          # period of time when the checks should be performed
          periodSeconds: 5
        # Specification of the Readiness Probe
        readinessProbe:
          httpGet:
            path: /readiness
            port: 5000
          initialDelaySeconds: 10
```

```yaml
            # change to 1 seconds and see the pod not going to go ready
            timeoutSeconds: 3
            failureThreshold: 1
            periodSeconds: 5
          env:
            # variable for the container to be killed after 60 seconds
            - name: "KILL_IN_SECONDS"
              value: "60"
          ports:
            - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: backendflask-healthcheck
spec:
  type: NodePort
  selector:
    app: backendflask-healthcheck
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30000
---
apiVersion: v1
kind: Service
metadata:
  name: backendflask-healthcheck
spec:
  type: ClusterIP
  selector:
    app: backendflask-healthcheck
  ports:
    - port: 80
      targetPort: 8080
```

5.6.2 Debugging

5.7 Helm

The previous sections showed the complexity of working with Kubernetes. As in other programming languages, there are easier ways to manage an application rather than writing all of the deployments and services of an application by hand. Among others, *Helm* is a package manager for Kubernetes that enables to template and group Kubernetes manifests as versioned packages.

Helm packages applications into so called **, which are a collection of yaml and helper templates. Once an application is packaged, it can be installed onto a Kubernetes cluster using a single command. Helm s are packaged as `tar`files and stored in repositories similar to registries like Docker or PyPi, and some repository registries like Artifactory automatically index Helm Charts. This ease of working allows Helm to provide a large collection of open-source charts to easily deploy applications such as PostgreSQL[2], Redis[3], Nginx[4], and Prometheus[5].

5.7.1 Helm Chart Structure

As previously mentioned, a Helm package consists of a *Helm Chart* that includes a collection of yaml and helper templates. This chart is finally packaged as a `.tar` file. The following structure shows how the chart `my-custom-chart` is organized in the directory `custom_chart`. The `Chart.yaml` file consists of the major metadata about the chart, such as name, version, and description, as well as dependencies if multiple charts are packaged. The `/templates` directory contains all the Kubernetes manifests that define the behavior of the application and are deployed automatically by installing the chart. Exemplary variables able to be specified are denoted in the `values.yaml` file, which also incorporates default values. Just like a *.gitignore* file it is also possible to add a *.helmignore*.

```
custom_chart/
 .helmignore        # Contains patterns to ignore
 Chart.yaml         # Information about your chart
 values.yaml        # The default values for your templates
 templates/         # The template files
     ingress.yaml   # ingress.yaml manifest,
     ...            # and others...
```

---

[2]https://github.com/bitnami/charts/tree/master/bitnami/postgresql
[3]https://github.com/bitnami/charts/tree/master/bitnami/redis
[4]https://github.com/kubernetes/ingress-nginx
[5]https://github.com/prometheus-community/helm-charts

When wanting to create a own Helm Chart, there is no need to create all the files on your own. To bootstrap a Helm Chart there is the in-built command:

```
helm create <my-chart>
```

that provide all the common Kubernetes manifests (`deployment.yaml`, `hpa.yaml` , `ingress.yaml` , `service.yaml` , and `serviceaccount.yaml`) as well as helper templates to circumvent resource naming constraints and labels/annotations. The command will provide a scalable deployment for `nginx` on default, which can be simply modified to deploy a custom docker image by editing the `values.yaml` file.

5.7.2 Working with Helm

As there is large collection of open-source and public charts already available, there is no need to create your own Helm Chart. One simply can use helms build in command to search for a specific Helm Chart, such as shown below by searching for a `redis` deployment, or have a look oneselves by scrolling through for example *artifactory* or *bitnami*, which provide a large collection of public charts.

```
helm search hub redis
```

5.7.2.1 Adding a Helm Chart to the local setup.

After finding the correct Helm Chart, it can simply be added to the local setup. Once added to the local setup, the chart is listed in the local repository and is ready to be installed.

```
# Add a helm chart to the local setup under the name "bitnami"
helm repo add bitnami https://charts.bitnami.com/bitnami

# Search and show the local repository for all charts with the name bitnami
# Once it a helm chart is listed here it can be installed
helm search repo bitnami
```

Since Helm harts are packaged as a `.tar`file, they can also be downloaded locally and modified as needed.

```
# Download the nginx-ingress-controller helm chart to local
helm pull bitnami/nginx-ingress-controller --version 5.3.19
```

5.7.2.2 Installing a Helm Chart

Once a Helm Chart is downloaded or added to the local setup, it can be installed using the `helm install` command followed by a custom release name, and the name of the chart to be installed. It is a best practice to update the list of charts before installing, just like when installing packages in other programming languages such as pip and python.

```
# Make sure we get the latest list of charts
helm repo update


# Installing a Helm Chart from the local repository
# helm install <CUSTOM-RELEASE-NAME> <CHART-NAME>
helm install custom-bitnami bitnami/wordpress


# Installing a downloaded Helm Chart from a directory
helm install -f values.yaml my-custom-chart ./custom_chart
```

Similar to installing a Helm Chart, it can also be uninstalled.

```
# helm uninstall <CUSTOM-RELEASE-NAME>
helm uninstall custom-bitnami
helm uninstall my-custom-chart
```

All installed and released charts can be listed using the following command.

```
helm list
```

6 Terraform

Terraform is an open-source, declarative programming language developed by HashiCorp and allows to create both on-prem and cloud resources in the form of writing code. This is also know as Infrastructure as Code (IAC). There are several IaC tools in the market today and Terraform is only one of them. Yet it is a well known and established tool and during the course of this project we only focus on this.

HashiCorp[1] describes that:

Terraform is an infrastructure as code (IaC) tool that allows you to build, change, and version infrastructure safely and efficiently. This includes both low-level components like compute instances, storage, and networking, as well as high-level components like DNS entries and SaaS features.

This means that users are able to manage and provision an entire IT infrastructure using machine-readable definition files and thus allowing faster execution when configuring infrastructure, as well as enabling full traceability of changes. Terraform comes with several hundred different providers that can be used to provision infrastructure, such as Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, Helm, GitHub, Splunk, DataDog, etc.

The given chapter introduces the concepts & usage of Terraform which will be needed to create the introduced MLOps Airflow deployment in an automated and tracable way. We will learn how to use Terraform to provision ressources as well as to structure a Terraform Project.

Prerequisites

To be able to follow this tutorial, one needs to have the AWS CLI installed as well as the AWS credentials set up. Needles to say an AWS accounts needs to be present. It is also recommended to have basic knowledge of the AWS Cloud as this tutorials used the AWS infrastructure to provision cloud resources. The attached resource definitions are specified to the AWS region `eu-central-1`. It might be necessary to change accordingly if you are

---

[1]https://www.terraform.io/docs

set in another region. Further, Terraform itself needs to be installed. Please refer to the corresponding sites. The scripts are run under Terraform version `v1.2.4`. Later releases might have breaking changes. One can check its installation via `terraform version`.

6.1 Basic usage

A Terraform project is basically just a set of files in a directory containing resource definitions of cloud ressources to be created. Those Terraform files, denoted by the ending `.tf`, use Terraform's configuration language to define the specified resources. In the following example there are two definitions made: a `provider` and a `resource`. Later in this chapter we will dive deeper in the structure of the language. For now, we only need to know this script is creating a file called `hello.txt` that includes the text `"Hello, Terraform"`. It's our Terraform version of Hello World!

```
provider "local" {
  version = "~> 1.4"
}
resource "local_file" "hello" {
  content = "Hello, Terraform"
  filename = "hello.txt"
}
```

6.1.1 terraform init

When a project is run for the first time the terraform project needs to be initialized. This is done via the `terraform init` command. Terraform scans the project files in this step and downloads any required providers needed (more details to providers in a following section). In the given example this is the local procider.

```
# Initializes the working directory which consists of all the configuration files
terraform init
```

```
Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/local from the dependency lock file
- Using previously-installed hashicorp/local v1.4.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

6.1.2 terraform validate

The `terraform validate` command checks the code for syntax errors. This is optional yet a way to handle initial errors or minor careless mistakes

```
# Validates the configuration files in a directory
terraform validate
```

```
Success! The configuration is valid.
```

6.1.3 terraform plan

The `terraform plan` command verifies what action Terraform will perform and what resources will be created. This step is basically a *dry run* of the code to be executed. It also returns the provided values and some permission attributes which have been set.

```
# Creates an execution plan to reach a desired state of the infrastructure
terraform plan
```

```
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following
symbols:
  + create

Terraform will perform the following actions:

  # local_file.hello will be created
  + resource "local_file" "hello" {
      + content              = "Hello, Terraform"
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "hello.txt"
      + id                   = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

─────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run
"terraform apply" now.
```

6.1.4 terraform apply

The command `Terraform apply` creates the resource specified in the `.tf` files. Initially, the
same output as in the `terraform plan` step is shown (hence its *dry run*). The output further
states which resources are added, which will be changed, and which resources will be destroyed.
After confirming the actions the resource creation will be executed.

Modifications to previously deployed ressources can be implemented by using `terraform
apply` again. The output will denote that there are resources to change.

```
# Provision the changes in the infrastructure as defined in the plan
terraform apply
```

```
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following
symbols:
  + create

Terraform will perform the following actions:

  # local_file.hello will be created
  + resource "local_file" "hello" {
      + content             = "Hello, Terraform"
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename            = "hello.txt"
      + id                  = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

local_file.hello: Creating...
local_file.hello: Creation complete after 0s [id=392b5481eae4ab2178340f62b752297f72695d57]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

6.1.5 terraform destroy

To destoy all created ressouces and to delete everything we did before, there is a `terraform
destroy` command.

```
# Deletes all the old infrastructure resources
terraform destroy
```

```
local_file.hello: Refreshing state... [id=392b5481eae4ab2178340f62b752297f72695d57]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following
symbols:
  - destroy

Terraform will perform the following actions:

  # local_file.hello will be destroyed
  - resource "local_file" "hello" {
      - content              = "Hello, Terraform" -> null
      - directory_permission = "0777" -> null
      - file_permission      = "0777" -> null
      - filename             = "hello.txt" -> null
      - id                   = "392b5481eae4ab2178340f62b752297f72695d57" -> null
    }

Plan: 0 to add, 0 to change, 1 to destroy.

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

local_file.hello: Destroying... [id=392b5481eae4ab2178340f62b752297f72695d57]
local_file.hello: Destruction complete after 0s

Destroy complete! Resources: 1 destroyed.
```

6.2 Core Components

The following section will explain the core components and building blocks of Terraform. This
will enable you to build your very first Terraform definition files.

6.2.1 Providers

Terraform relies on plugins called providers to interact with Cloud providers, SaaS providers,
and other APIs. Each provider adds specific resource types and/or data sources that can
be managed by Terraform. For example, the `aws` provider shown below allows to specify
resources related to the AWS Cloud such as S3 Buckets or EC3 Instances.

Depending on the provider it is necessary to supply it with specific parameters. The aws
provier for example needs the `region` as well as username and password. If nothing is specified
it will automatically pull these information from the *AWS CLI* and the credentials specified
under the directory `.aws/config`. It is also a best practice to specify the version of the
provider, as the providers are usually maintained and updated on a regular basis.

```
provider "aws" {
  region = "us-east-1"
}
```

6.2.2 Resources

A *resource* is the core building block when working with Terraform. It can be a `"local_file"` such as shown in the example above, or a cloud resource such as an `"aws_instance"` on aws. The resource type is followed by the custom name of the resource in Terraform. Resource definitions are usually specified in the `main.tf`file. Each customization and setting to a ressource is done within its resource specification. The style convention when writing Terraform code states that the resource name is named in lowercase as well as it should not repeat the resource type. An example can be seen below

```
# Ressource type: aws_instance
# Ressource name: my-instance
resource "aws_instance" "my-instance" {
  # resource specification
  ami           = "ami-0ddbdea833a8d2f0d"
  instance_type = "t2.micro"

  tags = {
    Name = "my-instance"
    ManagedBy = "Terraform"
  }
}
```

6.2.3 Data Sources

Data sources in Terraform are "read-only" resources, meaning that it is possible to get information about existing data sources but not to create or change them. They are usually used to fetch parameters needed to create resources or generally for using parameters elsewhere in Terraform configuration.

A typical example is shown below as the `"aws_ami"` data source available in the AWS provider. This data source is used to recover attributes from an existing AMI (Amazon Machine Image). The example creates a data source called `"ubuntu` that queries the AMI registry and returns several attributes related to the located image.

```
data "aws_ami" "ubuntu" {
  most_recent = true
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-trusty-14.04-amd64-server-*"]
```

```
  }
  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
  owners = ["099720109477"] # Canonical
}
```

Data sources and their attributes can be used in resource definitions by prepending the `data` prefix to the attribute name. The following example used the `"aws_ami"` data source within an `"aws_instace"` resource.

```
resource "aws_instance" "web" {
  ami = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
}
```

6.2.4 State

A Terraform state stores all details about the resources and data created within a given context. Whenever a resource is create terrafrom stores its identifier in the statefile `terraform.tfstate`. Providing information about already existing resources is the primary purpose of the statefile. Whenever a Terraform script is applied or whenever the resource definitions are modified, Terraform knows what to create, change, or delete based on the existing entries within the statefile. Everything specified and provisioned within Terraform will be stored in the statefile. This should be kept in mind and detain to store sensitive information such as initial passwords.

Terraform uses the concept of a backend to store and retrieve its statefile. The default backend is the local backend which means to store the statefile in the project's root folder. However, we can also configure an alternative (remote) backend to store it elsewhere. The backend can be declared within a `terraform` block in the project files. The given example stores the statefile in an AWS S3 Bucket callen `some-bucket`. Keep in mind this needs access to an AWS account and also needs the AWS provider of terraform.

```
terraform {
  backend "s3" {
    bucket = "some-bucket"
    key = "some-storage-key"
```

```
    region = "us-east-1"
  }
}
```

6.3 Modules

A Terraform module allows to reuse resources in multiple places throughout the project. They act as a container to package resource configurations. Much like in standard programming languages, Terraform code can be organized across multiple files and packages instead of having one single file containing all the code. Wrapping code into a module not only allows to reuse it throughout the project, but also in different environments, for example when deploying a *dev* and a *prod* infrastructure. Both environments can reuse code from the same module, just with different settings.

A Terraform module is build as a directory containing one or more resource definition files. Basically, when putting all our code in a single directory, we already have a module. This is exactly what we did in our previous examples. However, terraform does not include subdirectories on its own. Subdirectories must be called explicitly using a terraform `module`parameter. The example below references a module located in a `./network` subdirectory and passes two parameters to it.

```
# main.tf
module "network" {
  source = "./networking"
  create_public_ip = true
  environment = "prod"
}
```

Each module consists of a similar file structure as the root directory. This includes a `main.tf` where all resources are specified, as well as files for different data sources such as `variables.tf` and `outputs.tf`. However, providers are usually configured only in the root module and are not reused in modules. Note that there are different approaches on where to specify the providers. They are either specified in the `main.tf` or a separate `providers.tf`. It does not make a difference for Terraform as it does not distinguish between the resource definition files. It is merely a strategy to keep code and project in a clean and consistent structure.

```
root
   main.tf
```

```
    variables.tf
    outputs.tf

 networking
        main.tf
        variables.tf
        outputs.tf
```

6.3.1 Input Variables

Each module can have multiple *Input Variables*. Input Variables serve as parameters for a Terraform module so users can customize behavior without editing the source. In the previous example of importing a `network` module, there have been two input variables specified, `create_public_ip` and `environment`. Input variables are usually specified in the `variables.tf` file.

```
# variables.tf
variable "instance_name" {
  type = string
  default = "awesome-instance"
  description = "Name of the aws instance to be created"
}
```

Each variable has a type (e.g. `string`, `map`, `set`, `boolen`) and may have a `default` value and `description`. Any variable that has no default must be supplied with a value when calling the `module` reference. This means that variables defined at the root module need values assigned to as a requirement so Terraform will not fail. This can be done by different resources, for example

- a variable's `default` value
- via the command line using the `terraform apply -var="variable=value"`option
- via environment variables starting with `TF_VAR_`; Terraform will check them automatically
- a `.tfvars` file where the variable values are specified; Terraform can load variable definitions from these files automatically (please check online resources for further insights)

Variables can be used in expressions using the `var.`prefix such as shown in below example. We use the resource configuration of the previous example to create an `aws_instance` but this time its name is provided by an input variable.

```
# main.tf
resource "aws_instance" "awesome-instance" {
  ami           = "ami-0ddbdea833a8d2f0d"
  instance_type = "t2.micro"

  tags = {
    Name = var.instance_name
  }
}
```

6.3.2 Output Variables

Similar to Input variables, a terraform module has *output variables*. As their name states, output variables return values of a Terraform module and are denoted in the *outputs.tf* file as expected. A module's consumer has no access to any resources or data created within the module itself. However, sometimes a modules attrivutes are needed for another module or resource. Output variables address this issue by exposing a defined subset of the created resources.

The example below defines an output value *instance_address* containing the IP address of an EC2 instance the we create with a module. Any module that reference this module can use the *instance_address* value by referencing it via *module.module_name.instance_address*

```
# outputs.tf
output "instance_address" {
    value = aws_instance.awesome-instance.private_ip
    description = "Web server's private IP address"
  }
```

```
module.networking.aws_security_group.tf-tutorial-sg: Creation complete after 3s [id=sg-067c41e618217205a]
aws_instance.awesome-instance: Still creating... [20s elapsed]
aws_instance.awesome-instance: Still creating... [30s elapsed]
aws_instance.awesome-instance: Still creating... [40s elapsed]
aws_instance.awesome-instance: Still creating... [50s elapsed]
aws_instance.awesome-instance: Creation complete after 52s [id=i-061ff2728d33ef1eb]

Apply complete! Resources: 7 added, 0 changed, 0 destroyed.

Outputs:

instance_address = "172.31.34.217"
```

6.3.3 Local Variables

Additionally to Input variables and output variables a module provides the use of local variables. Local values are basically just a convenience feature to assign a shorter name to an expression and work like standard variables. This means theor scope is also limited to the module they are declared in. Using local variables reduces code repetitions which can be especially valuable when dealing with output variables from a module.

```
# main.tf
locals {
  vpc_id = module.network.vpc_id
}
module "network" {
  source = "./network"
}
module "service1" {
  source = "./service1"
  vpc_id = local.vpc_id
}
module "service2" {
  source = "./service2"
  vpc_id = local.vpc_id
}
```

6.4 Additional tips & tricks

Of course there is much more to Terraform than these small examples can provide. Yet there are also some contrains when working with Terraform or declarative languages in general. Typically they do not have for-loops or other traditional procedural logic built into the language to repeat a piece of logic or conditional if-statements to configure reasources on demand. However, there are ways there are some ways to deal with this issue and to create multiple respurces without copy and paste.

Terraform comes with different looping constructs, each used slightly different. The *count* and *for_each* meta arguments enable us to create multiple instances of a resource.

6.4.1 count

Count can be used to loop over any resource and module. Every Terraform resource has a meta-parameter *count* one can use. Count is the simplest, and most limited iteration construct and all it does is to define how many copies to create of a resource. When creating multiple instance with one specification, the problem is that each instance must have a unique name, otherwise Terraform would cause an error. Therefore we need to index the meta-parameter just like doing it in a for-loop to give each resource a unique name. The example below shows how to do this on an AWS IAM user.

```
resource "aws_iam_user" "example" {
  count = 2
  name  = "neo.${count.index}"
}
```

```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

all_users_count = [
    {
        "arn" = "arn:aws:iam::728728728728:user/neo.0"
        "force_destroy" = false
        "id" = "neo.0"
        "name" = "neo.0"
        "path" = "/"
        "permissions_boundary" = tostring(null)
        "tags" = tomap(null) /* of string */
        "tags_all" = tomap({})
        "unique_id" = "AIDA4OKBSADPST5SZNUEG"
    },
    {
        "arn" = "arn:aws:iam::728728728728:user/neo.1"
        "force_destroy" = false
        "id" = "neo.1"
        "name" = "neo.1"
        "path" = "/"
        "permissions_boundary" = tostring(null)
        "tags" = tomap(null) /* of string */
        "tags_all" = tomap({})
        "unique_id" = "AIDA4OKBSADPURLBRKSJC"
    },
]
```

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["adam", "eve", "snake", "apple"]
}

resource "aws_iam_user" "example" {
  count = length(var.user_names) # returns the number of items in the given array
  name  = var.user_names[count.index]
}
```

```
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.

Outputs:

all_users_count_list = [
    {
        "arn" = "arn:aws:iam::728728728728:user/adam"
        "force_destroy" = false
        "id" = "adam"
        "name" = "adam"
        "path" = "/"
        "permissions_boundary" = tostring(null)
        "tags" = tomap(null) /* of string */
        "tags_all" = tomap({})
        "unique_id" = "AIDA4OKBSADP4Q6FJDVKK"
    },
    {
        "arn" = "arn:aws:iam::728728728728:user/eve"
        "force_destroy" = false
        "id" = "eve"
        "name" = "eve"
        "path" = "/"
        "permissions_boundary" = tostring(null)
        "tags" = tomap(null) /* of string */
        "tags_all" = tomap({})
        "unique_id" = "AIDA4OKBSADPRYMWEQD6L"
    },
    {
        "arn" = "arn:aws:iam::728728728728:user/snake"
        "force_destroy" = false
        "id" = "snake"
        "name" = "snake"
        "path" = "/"
        "permissions_boundary" = tostring(null)
        "tags" = tomap(null) /* of string */
        "tags_all" = tomap({})
        "unique_id" = "AIDA4OKBSADPQJ7TNJQRB"
    },
    {
        "arn" = "arn:aws:iam::728728728728:user/apple"
        "force_destroy" = false
        "id" = "apple"
        "name" = "apple"
        "path" = "/"
        "permissions_boundary" = tostring(null)
        "tags" = tomap(null) /* of string */
        "tags_all" = tomap({})
        "unique_id" = "AIDA4OKBSADP73Y4SMMA5"
    },
]
```

After using count on a resource it becomes an array of resources rather than one single resource. The same hold when using count on modules. When adding count to a module it turns it into an array of modules.

This can round into problems because the way Terraform identifies each resource within the array is by its index. Now, when removing an item from the middle of the array, all items after it shift one index back. This will result in Terraform deleting every resource after that item and then re-creating these resources again from scratch

So after running `terraform plan` with just three names, Terraform's internal representation will look like this:

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["adam", "eve", "apple"]
}
```

```
resource "aws_iam_user" "example" {
  count = length(var.user_names) # returns the number of items in the given array
  name  = var.user_names[count.index]
}
```

```
Apply complete! Resources: 0 added, 1 changed, 0 destroyed.

Outputs:

all_users_count_list = [
  {
    "arn" = "arn:aws:iam::728728728728:user/adam"
    "force_destroy" = false
    "id" = "adam"
    "name" = "adam"
    "path" = "/"
    "permissions_boundary" = tostring(null)
    "tags" = tomap({})
    "tags_all" = tomap({})
    "unique_id" = "AIDA40KBSADP3UPAYSQAR"
  },
  {
    "arn" = "arn:aws:iam::728728728728:user/ev
    "force_destroy" = false
    "id" = "eve"
    "name" = "eve"
    "path" = "/"
    "permissions_boundary" = tostring(null)
    "tags" = tomap({})
    "tags_all" = tomap({})
    "unique_id" = "AIDA40KBSADP76JWVZWJ7"
  },
  {
    "arn" = "arn:aws:iam::728728728728:user/snake"
    "force_destroy" = false
    "id" = "snake"
    "name" = "apple"
    "path" = "/"
    "permissions_boundary" = tostring(null)
    "tags" = tomap({})
    "tags_all" = tomap({})
    "unique_id" = "AIDA40KBSADP45Z50T7EU"
  },
]
```

Count as conditional Count can also be used as a form of a conditional if statement. This is possible as Terraform supports *conditional expressions*. If `count` is set to one 1, one copy of that resource is created; if set to 0, the resource is not created at all. Writing this as a conditional expression could look something like the follow, where var.enable_autoscaling is a boolean variable either set to `True` or `False`.

```
resource "example-1" "example" {
  count = var.enable_autoscaling ? 1 : 0
  name  = var.user_names[count.index]
}
```

6.4.2 for-each

The *for_each* expression allows to loop over lists, sets, and maps to create multiple copies of a resource just like the *count* meta. The main difference between them is that *count* expects

a non-negative number, whereas *for_each* only accepts a list or map of values. Using the same example as above it would look like this:

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["adam", "eve", "snake", "apple"]
}

resource "aws_iam_user" "example" {
  for_each = toset(var.user_names)
  name     = each.value
}

output "all_users" {
  value = aws_iam_user.example
}
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

short_upper_names = [
  "ADAM",
  "EVE",
]
upper_names = [
  "ADAM",
  "EVE",
  "SNAKE",
  "APPLE",
]
```

Using a map of resource with the *for_each* meta rather than an array of resources as with *count* has the benefit to remove items from the middle of the collection safely and without re-creating the resources following the deleted item. Of course, the same can also be done for modules.

```
module "users" {
  source = "./iam-user"

  for_each  = toset(var.user_names)
  user_name = each.value
}
```

6.4.3 for

Terraform also offers a similar functionality as python list comprehension in the form of a for expression. This should not be confused with the *for-each* expression seen above. The basic syntax is shown below to convert the list of names of previous examples in var.names to uppercase:

```
output "upper_names" {
  value = [for name in var.names : upper(name)]
}

output "short_upper_names" {
  value = [for name in var.names : upper(name) if length(name) < 5]
}
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

short_upper_names = [
    "ADAM",
    "EVE",
]
upper_names = [
    "ADAM",
    "EVE",
    "SNAKE",
    "APPLE",
]
```

Using for to loop over lists and maps within a string can be used similarly. This allows us to use control statements directly withing strings using a syntax similar to string interpolation.

```
output "for_directive" {
  value = "%{ for name in var.names }${name}, %{ endfor }"
}
```

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

for_directive = "adam, eve, snake, apple, "
```

6.4.4 Workspaces

Terraform workspaces allow us to keep multiple state files for the same project. When we run Terraform for the first time in a project, the generated state file will go into the *default* workspace. Later, we can create a new workspace with the `terraform workspace new` command, optionally supplying an existing state file as a parameter.

6.5 Exemplary Deployment

Building a VPC with EC2 & S3

Finally, we want to put everything together and provision our own cloud infrastructure using Terraform. We will create a AWS Virtual private Cloud with EC2 Instances running on it, and S3 Buckets attached to the them. We will use *for_each* and *count* to create multiple instaces. The Terraform infrastructure is separated into three modules *VPS*, *EC2*, and *S3*.

```
root
   main.tf
   variables.tf
   outputs.tf


 networking
      main.tf
      variables.tf
      outputs.tf


 ec2
      main.tf
      variables.tf
      outputs.tf


 s3
       main.tf
       variables.tf
       outputs.tf
```

7 ML Platform Design

ML platforms can be set up in various ways to apply MLOps practices to the machine learning workflow. (1) SaaS tools provide an integrated development and management experience, with an aim to offer an end-to-end process. (2) Custom-made platforms offer high flexibility and can be tailored to specific needs. However, integrating multiple different services requires significant engineering effort. (3) Many cloud providers offer a mix of SaaS and custom-tailored platforms, providing a relatively well-integrated experience while remaining open enough to integrate other services.

This project involves building a custom-tailored MLOps platform focused on MLOps engineering, as the entire infrastructure will be set up from scratch. An exemplary MLOps platform will be developed using Airflow and MLflow for management during the machine learning lifecycle and JupyterHub to provide a development environment.

Even though there are workflow tools better designed for machine learning pipelines, for example Kubeflow Pipelines, Airflow and MLflow can leverage and an combine there functionalities to provide similar capabilites. Airflow provides the workflow management for the platform whilst MLflow is used for machine learning tracking. MLflow further allow to register each model effortlessly. As an MLOps plattform should also provide an environment to develop machine learning model code, JupyterHub will be deployed to be able to develop code in the cloud and without the need for a local setup. The coding environment will synchronize with Airflow's DAG repository to seamlessly integrate the defined models within the workflow management. Airflow and MLflow are very flexible with their running environment and their stack would be very suitable for small scale systems, where there is no need for a setup maintaining a Kubernetes cluster. While it would be possible to run anything on a docker/docker-compose setup, this work will scale the mentioned tools to a Kubernetes cluster in the cloud to fully enable the concept of an MLOps plattform. The infrastructure will be maintained using the Infrastructure as Code tool *Terraform*, and incorporate best Ops practices such as CI/CD and automation. The project will also incorporate the work done by data and machine learning scientists since basic machine learning models will be implemented and run on the platform.

The following chapters give an introductory tutorial on each of the previously introduced tools. A machine learning workflow using Airflow is set up on the deployed infrastructure, including data preprocessing, model training, and model deployment, as well as tracking the experiment and deploying the model into production using MLFlow.

The necessary AWS infrastructure is set up using Terraform. This includes creating an AWS EKS cluster and the associated ressources like a virtual private cloud (VPC), subnets, security groups, IAM roles, as well as further AWS ressources needed to deploy Airflow and MLflow. Once the EKS cluster is set up, Kubernetes can be used to deploy and manage applications on the cluster. Helm, a package manager for Kubernetes, is used to manage the deployment of Airflow and MLflow. The EKS cluster allows for easy scalability and management of the platforms. The code is made public on a Github repository and Github Actions is used for automating the deployment of the infrastructure using CI/CD principles.

Once the infrastructure is set up, machine learning models can be deployed to the EKS cluster as Kubernetes pods, using Airflows scheduling processes. Airflow's ability to scan local directories or Git repositories will be used to import the relevant machine learning code from second Github repository. Similarly, to building Airflow workflows, the machine learning code will also include using the MLFlow API to allow for model tracking. Github Actions is used as a CI/CD pipeline to automatically build, test, and deploy machine learning models

to this repository similarly as it is used in the repository for the infrastructure.

Whereas the deployment of the infrastructure would be taken care of by MLOps-, DevOps-, and Data Engineers, the development of the Airflow workflows including MLFlow would be taken care of by Data Scientist and ML Engineers.

8 ML Platform Deployment

> **_NOTE:_** The chapter discussing the deployment of an ML platform with Airflow
> and MLflow on AWS EKS, utilizing Terraform for deployment, is currently in
> the writing phase. The information provided in this disclaimer is based on the
> current state of knowledge up until July 2023. Thank you for your understanding
> and patience as I work on completing this chapter.

The provided directory structure represents the Terraform project for managing the infrastructure of our ML platform. It follows a modular organization to promote reusability and maintainability of the codebase. The full codebase is also available and can be accessed on github[1]

```
root
    main.tf
    variables.tf
    outputs.tf
    providers.tf

 infrastructure

    vpc

    eks

    networking

    rds

 modules

    airflow
```

---

[1] https://github.com/seblum/mlops-airflow-on-eks

```
    mlflow


    jupyterhub
```

By structuring the Terraform project this way it becomes easier to manage, scale, and maintain the infrastructure as the project grows. Each module can be independently developed, tested, and reused across different projects, promoting consistency and reducing duplication of code and effort.

Root

The *root* directory of the Terraform project contains the general configuration files related to the overall infrastructure setup.

- The `main.tf` Terraform configuration file, where all major resources are defined and organized into modules.
- The `variables.tf` containing the definition of input variables used throughout the project, allowing users to customize the infrastructure setup.
- The `outputs.tf` defining the output variables that expose relevant information about the deployed infrastructure.
- The `providers.tf` that defining and configuring the providers used in the project, for example, AWS, Kubernetes, Helm.

Infrastructure

The *infrastructure* directory holds the individual modules responsible for provisioning specific components of the AWS Cloud and EKS setup.

- `vpc` defines a module that configures resources related to the Virtual Private Cloud (VPC), such as subnets, route tables, and internet gateways.
- The `eks` module is responsible for creating and configuring an Amazon Elastic Kubernetes Service (EKS) cluster, including worker nodes and other related resources like the Cluster Autoscaler, Elastic Block Storage, or Elastic File System.
- `networking` contains networking components that provide access to the cluster using ingresses and DNS records, for example the AWS Application Load Balancer or an External DNS.
- The `rds` module provides resources to deploy and Amazon Relational Database Service (RDS), such as database instances, subnets, and security groups. This module is needed for the specific tools and components of our ML platform.

Modules

The *modules* directory contains Terraform modules that are specific for setting up out ML Platform and provides the components to integrate the MLOps Framework, such as tools for model tracking (MLflow), workflow management (Airflow), or a integrated development environment (JupyterHub).

- `airflow` provides the Terraform module to deploy an Apache Airflow instance based on the Helm provider, which enables to orchestrate our ML workflows. The module is highly customized as it sets up necessary connections to other services, sets airflow variables that can be used by Data Scientists, creates an ingress ressource, and enables user management and authentication using Github.
- The `mlflow` module sets up MLflow to managing machine learning experiments and models. As MLflow does not natively provide a solution to deploy on Kubernetes, a custom Helm deployment is integrated that configures the necessary deployment, services, and ingress ressources.
- `jupyterhub` deploys a JupyterHub environment via Helm that enables multi-user notebook environment, suitable for collaborative data science and machine learning work. The Helm chart is highly customized providing user management and authentication via Github, provisioning ingress resources, and cloning a custom Github repository that provides all our Data Science and Machine Learning code.

8.1 Root directory module

The root directory of the Terraform infrastructure consists of the main module, calling other submodules that deploy specific infrastructure setting or tools. This enables to have an overview about the deployment in one place. At first, the necessary cluster infrastructure is deployed such as the `vpc` and the `eks` cluster itself. Afterward the custom tools to be run on EKS are deployed, such as `airflow`, `mlflow`, and `jupyterhub`.

The following will look a bit more detailed into the call of the module `airflow`, yet a lot of it also applies for the other modules. The module imports are structure in three sections. At first the general information about the module are given, such as `name` of the module, or the `cluster_name`, as well as more specific variables needed for specific Terraform calls in the module, like `cluster_endpoint`. Terraform does not provide the functionality to *activate* or *deactivate* a module by itself. As this is a useful feature, a custom workaround is proposed by setting the count a module as such `count = var.deploy_airflow ? 1 : 0`. This will set the cound of the module to `0` or `1`, depending on the `var.deploy_airflow` variable. This functionality is proposed for all custom modules.

Secondly, as Airflow needs access to and RDS Database, the RDS module is called. Therefore it is needed to pass the relevant information to create the the RDS under the correct settings, like `vpc_id`, `rds_engine`, or `storage_type`.

Third, variable values for the Airflow Helm chart are passed to the module. Using Helm makes the deployment of Airflow very easy. Since there are customizations on the deployment, such as a connection to the Airflow DAG repository on Github, it is necessary to specify these information beforehand, and to integrate them into the deployment.

```
locals {
  cluster_name            = "${var.name_prefix}-eks"
  vpc_name                = "${var.name_prefix}-vpc"
  port_airflow            = var.port_airflow
  port_mlflow             = var.port_mlflow
  mlflow_s3_bucket_name   = "${var.name_prefix}-mlflow-bucket"
  force_destroy_s3_bucket = true
  storage_type            = "gp2"
  max_allocated_storage   = var.max_allocated_storage
  airflow_github_ssh      = var.airflow_github_ssh
  git_username            = var.git_username
  git_token               = var.git_token
  git_repository_url      = var.git_repository_url
  git_branch              = var.git_branch
}


data "aws_caller_identity" "current" {}



# INFRASTRUCTURE
module "vpc" {
  source       = "./infrastructure/vpc"
  cluster_name = local.cluster_name
  vpc_name     = local.vpc_name
}

module "eks" {
  source               = "./infrastructure/eks"
  cluster_name         = local.cluster_name
  eks_cluster_version  = "1.23"
  vpc_id               = module.vpc.vpc_id
```

```
  private_subnets       = module.vpc.private_subnets
  security_group_id_one = [module.vpc.worker_group_mgmt_one_id]
  security_group_id_two = [module.vpc.worker_group_mgmt_two_id]
  depends_on = [
    module.vpc
  ]
}

# CUSTOM TOOLS
module "airflow" {
  count            = var.deploy_airflow ? 1 : 0
  source           = "./modules/airflow"
  name             = "airflow"
  cluster_name     = local.cluster_name
  cluster_endpoint = module.eks.cluster_endpoint

  # RDS
  vpc_id                     = module.vpc.vpc_id
  private_subnets            = module.vpc.private_subnets
  private_subnets_cidr_blocks = module.vpc.private_subnets_cidr_blocks
  rds_port                   = local.port_airflow
  rds_name                   = "airflow"
  rds_engine                 = "postgres"
  rds_engine_version         = "13.3"
  rds_instance_class         = "db.t3.micro"
  storage_type               = local.storage_type
  max_allocated_storage      = local.max_allocated_storage

  # HELM
  helm_chart_repository = "https://airflow-helm.github.io/charts"
  helm_chart_name       = "airflow"
  helm_chart_version    = "8.6.1"
  git_username          = local.git_username
  git_token             = local.git_token
  git_repository_url     = local.git_repository_url
  git_branch            = local.git_branch

  depends_on = [
    module.eks
```

```
  ]
}


module "mlflow" {
  count                 = var.deploy_mlflow ? 1 : 0
  source                = "./modules/mlflow"
  name                  = "mlflow"
  mlflow_s3_bucket_name = local.mlflow_s3_bucket_name
  s3_force_destroy      = local.force_destroy_s3_bucket

  # RDS
  vpc_id                     = module.vpc.vpc_id
  private_subnets            = module.vpc.private_subnets
  private_subnets_cidr_blocks = module.vpc.private_subnets_cidr_blocks
  rds_port                   = local.port_mlflow
  rds_name                   = "mlflow"
  rds_engine                 = "mysql"
  rds_engine_version         = "8.0.30"
  rds_instance_class         = "db.t3.micro"
  storage_type               = local.storage_type
  max_allocated_storage      = local.max_allocated_storage

  depends_on = [
    module.eks
  ]
}


module "jupyterhub" {
  count            = var.deploy_jupyterhub ? 1 : 0
  source           = "./modules/jupyterhub"
  name             = "jupyterhub"
  cluster_name     = local.cluster_name
  cluster_endpoint = module.eks.cluster_endpoint

  # HELM
  helm_chart_repository = "https://jupyterhub.github.io/helm-chart/"
  helm_chart_name       = "jupyterhub"
```

```
  helm_chart_version     = "2.0.0"

  depends_on = [
    module.eks
  ]
}
```

8.2 Infrastructure

The subdirectory `infrastructure` consists of four main modules, `vpc`, `eks`, `networking`, and `rds`. The former three are responsible to create the cluster itself, as well as the necessary tools to implement the platform functionalities. The `rds` module is merely an extension linked to the cluster which is needed to store data of tools like Airflow or Mlflow. The `rds` module is thereby called in the corresponding modules where an AWS RDS is needed, even though the module is placed in the Infrastructure directory.

8.2.1 Virtual Private Cloud

The provided code in the `vpc` module establishes a Virtual Private Cloud (VPC) with associated subnets and security groups. It configures the required networking and security infrastructure to serve as the foundation to deploy an AWS EKS cluster.

The VPC is created using the `terraform-aws-modules/vpc/aws` module version 5.0.0. The VPC is assigned the IPv4 CIDR block `"10.0.0.0/16"` and spans across all three available AWS availability zones within the specified region `eu-central-1`. It includes both public and private subnets, with private subnets associated with NAT gateways for internet access. DNS hostnames are enabled for the instances launched within the VPC.

The VPC subnets are tagged with specific metadata relevant to Kubernetes cluster management. The public subnets are tagged with `"kubernetes.io/cluster/${local.cluster_name}"` set to `"shared"` and `"kubernetes.io/role/elb"` set to 1. The private subnets are tagged with `"kubernetes.io/cluster/${local.cluster_name}"` set to `"shared"` and `"kubernetes.io/role/internal-elb"` set to 1.

Additionally, three security groups are defined to manage access to worker nodes. They are intended to provide secure management access to the worker nodes within the EKS cluster. Two of these security groups, `"worker_group_mgmt_one"` and `"worker_group_mgmt_two"`, allow SSH access from specific CIDR blocks. The third security group, `"all_worker_mgmt,"`

allows SSH access from multiple CIDR blocks, including `"10.0.0.0/8"`, `"172.16.0.0/12"`, `"192.168.0.0/16"`

```
locals {
  cluster_name = var.cluster_name
}

data "aws_availability_zones" "available" {}

module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.0.0"

  name = var.vpc_name

  cidr = "10.0.0.0/16"
  azs  = slice(data.aws_availability_zones.available.names, 0, 3)

  private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  public_subnets  = ["10.0.4.0/24", "10.0.5.0/24", "10.0.6.0/24"]

  enable_nat_gateway   = true
  single_nat_gateway   = true
  enable_dns_hostnames = true

  public_subnet_tags = {
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/elb"                      = 1
  }

  private_subnet_tags = {
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/internal-elb"             = 1
  }
}

resource "aws_security_group" "worker_group_mgmt_one" {
  name_prefix = "worker_group_mgmt_one"
  vpc_id      = module.vpc.vpc_id
```

```
  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"

    cidr_blocks = [
      "10.0.0.0/8",
    ]
  }
}

resource "aws_security_group" "worker_group_mgmt_two" {
  name_prefix = "worker_group_mgmt_two"
  vpc_id      = module.vpc.vpc_id

  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"

    cidr_blocks = [
      "192.168.0.0/16",
    ]
  }
}

resource "aws_security_group" "all_worker_mgmt" {
  name_prefix = "all_worker_management"
  vpc_id      = module.vpc.vpc_id

  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"

    cidr_blocks = [
      "10.0.0.0/8",
      "172.16.0.0/12",
      "192.168.0.0/16",
```

```
    ]
  }
}
```

8.2.2 Elastic Kubernetes Service

The provided Terraform code sets up an AWS EKS (Elastic Kubernetes Service) cluster with specific configurations and multiple node groups. The `"eks"` module is used to create the EKS cluster, specifying its name and version. The cluster has public and private access endpoints enabled, and a managed AWS authentication configuration. The creation of the `vpc` module is a prerequisite for the `"eks"` module, as the latter requires information like the `vpc_id`, or `subnet_ids` for a successful creation.

The EKS cluster itself is composed of three managed node groups: `"group_t3_small"`, `"group_t3_medium"`, and `"group_t3_large"`. Each node group uses a different instance type (`t3.small`, `t3.medium`, and `t3.large`) and has specific scaling policies. All three node groups have auto-scaling enabled. The node group `"group_t3_medium"` has set the minimum and desired sizes of nodes to 4, which ensures a base amount of nodes and thus resources to manage further deployments. The `"group_t3_large"` is tainted with a `NoSchedule`. This node group can be used for more resource intensive tasks by specifiyng a pod's toleration.

The `eks` module also deploys several Kubernetes add-ons, including `coredns`, `kube-proxy`, `aws-ebs-csi-driver`, and `vpc-cni`. The vpc-cni add-on is configured with specific environment settings, enabling prefix delegation for IP addresses.

- `CoreDNS` provides DNS-based service discovery, allowing pods and services to communicate with each other using domain names, and thus enabling seamless communication within the cluster without the need for explicit IP addresses.
- `kube-proxy`: is responsible for network proxying on Kubernetes nodes which ensures that network traffic is properly routed to the appropriate pods, services, and endpoints. It allows for an seamless communication between different parts of the cluster.
- `aws-ebs-csi-driver`(Container Storage Interface) is an add-on that enables Kubernetes pods to use Amazon Elastic Block Store (EBS) volumes for persistent storage, allowing data to be retained across pod restarts and ensuring data durability for stateful applications. The EBS configuration and deployment are describen in the following subsection, but the respective `service_account_role_arn` is linked to the EKS cluster on creation.
- `vpc-cni` (Container Network Interface) is essential for AWS EKS clusters, as it enables networking for pods using AWS VPC (Virtual Private Cloud) networking. It ensures

that each pod gets an IP address from the VPC subnet and can communicate securely
with other AWS resources within the VPC.

```
locals {
  cluster_name                        = var.cluster_name
  cluster_namespace                   = "kube-system"
  ebs_csi_service_account_name        = "ebs-csi-controller-sa"
  ebs_csi_service_account_role_name   = "${var.cluster_name}-ebs-csi-controller"
  autoscaler_service_account_name     = "autoscaler-controller-sa"
  autoscaler_service_account_role_name = "${var.cluster_name}-autoscaler-controller"

  nodegroup_t3_small_label   = "t3_small"
  nodegroup_t3_medium_label  = "t3_medium"
  nodegroup_t3_large_label = "t3_large"
  eks_asg_tag_list_nodegroup_t3_small_label = {
    "k8s.io/cluster-autoscaler/enabled" : true
    "k8s.io/cluster-autoscaler/${local.cluster_name}" : "owned"
    "k8s.io/cluster-autoscaler/node-template/label/role" : local.nodegroup_t3_small_lab
  }

  eks_asg_tag_list_nodegroup_t3_medium_label = {
    "k8s.io/cluster-autoscaler/enabled" : true
    "k8s.io/cluster-autoscaler/${local.cluster_name}" : "owned"
    "k8s.io/cluster-autoscaler/node-template/label/role" : local.nodegroup_t3_medium_l
  }

  eks_asg_tag_list_nodegroup_t3_large_label = {
    "k8s.io/cluster-autoscaler/enabled" : true
    "k8s.io/cluster-autoscaler/${local.cluster_name}" : "owned"
    "k8s.io/cluster-autoscaler/node-template/label/role" : local.nodegroup_t3_large_lab
    "k8s.io/cluster-autoscaler/node-template/taint/dedicated" : "${local.nodegroup_t3_
  }

  tags = {
    Owner = "terraform"
  }
}

data "aws_caller_identity" "current" {}
```

```
#
# EKS
#
module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "19.5.1"

  cluster_name              = local.cluster_name
  cluster_version           = var.eks_cluster_version
  cluster_enabled_log_types = ["api", "controllerManager", "scheduler"]

  vpc_id     = var.vpc_id
  subnet_ids = var.private_subnets

  cluster_endpoint_private_access = true
  cluster_endpoint_public_access  = true
  manage_aws_auth_configmap       = true

  # aws_auth_users              = local.cluster_users # add users in later step

  cluster_addons = {
    coredns = {
      most_recent = true
    },
    kube-proxy = {
      most_recent = true
    },
    aws-ebs-csi-driver = {
      service_account_role_arn = "arn:aws:iam::${data.aws_caller_identity.current.account_id}
    },
    vpc-cni = {
      most_recent              = true
      before_compute           = true
      service_account_role_arn = module.vpc_cni_irsa.iam_role_arn
      configuration_values = jsonencode({
        env = {
          # Reference docs https://docs.aws.amazon.com/eks/latest/userguide/cni-increase-ip-a
          ENABLE_PREFIX_DELEGATION = "true"
          WARM_PREFIX_TARGET       = "1"
```

```
      }
    })
  }


}

eks_managed_node_group_defaults = {
  ami_type                = "AL2_x86_64"
  disk_size               = 10
  iam_role_attach_cni_policy = true
  enable_monitoring       = true
}

eks_managed_node_groups = {
  group_t3_small = {
    name = "ng0_t3_small"

    instance_types = ["t3.small"]

    min_size      = 0
    max_size      = 6
    desired_size  = 0
    capacity_type = "ON_DEMAND"
    labels = {
      role = local.nodegroup_t3_small_label
    }
    tags = {
      "k8s.io/cluster-autoscaler/enabled"                   = "true"
      "k8s.io/cluster-autoscaler/${local.cluster_name}"     = "owned"
      "k8s.io/cluster-autoscaler/node-template/label/role" = "${local.nodegroup_t3_sr
    }
  }
  group_t3_medium = {
    name = "ng1_t3_medium"

    instance_types = ["t3.medium"]

    min_size      = 4
    max_size      = 6
```

```
      desired_size   = 4
      capacity_type = "ON_DEMAND"
      labels = {
        role = local.nodegroup_t3_medium_label
      }
      tags = {
        "k8s.io/cluster-autoscaler/enabled"                      = "true"
        "k8s.io/cluster-autoscaler/${local.cluster_name}"        = "owned"
        "k8s.io/cluster-autoscaler/node-template/label/role"     = "${local.nodegroup_t3_med
      }
    }
    group_t3_large = {
      name = "ng2_t3_large"

      instance_types = ["t3.large"]

      min_size      = 0
      max_size      = 3
      desired_size  = 0
      capacity_type = "ON_DEMAND"
      labels = {
        role = local.nodegroup_t3_large_label
      }
      taints = [
        {
          key    = "dedicated"
          value  = local.nodegroup_t3_large_label
          effect = "NO_SCHEDULE"
        }
      ]
      tags = {
        "k8s.io/cluster-autoscaler/enabled"                        = "true"
        "k8s.io/cluster-autoscaler/${local.cluster_name}"          = "owned"
        "k8s.io/cluster-autoscaler/node-template/label/role"       = "${local.nodegroup_t3_lar
        "k8s.io/cluster-autoscaler/node-template/taint/dedicated" = "${local.nodegroup_t3_lar
      }
    }
  }
  tags = local.tags
```

```
}

#  Role for Service Account
module "vpc_cni_irsa" {
  source  = "terraform-aws-modules/iam/aws//modules/iam-role-for-service-accounts-eks"
  version = "~> 5.0"

  role_name_prefix      = "VPC-CNI-IRSA"
  attach_vpc_cni_policy = true
  vpc_cni_enable_ipv4   = true

  oidc_providers = {
    main = {
      provider_arn               = module.eks.oidc_provider_arn
      namespace_service_accounts = ["kube-system:aws-node"]
    }
  }
}
```

8.2.2.1 Elastic Block Store

The EBS CSI controller (Elastic Block Store Container Storage Interface) is set up by defining an IAM (Identity and Access Management) role using the `"ebs_csi_controller_role"` module. The role allows the EBS CSI controller to assume a specific IAM role with OIDC (OpenID Connect) authentication, granting it the necessary permissions for EBS-related actions in the AWS environment by an IAM policy. The IAM policy associated with the role is created likewise and permits various EC2 actions, such as attaching and detaching volumes, creating and deleting snapshots, and describing instances and volumes.

The code also configures the default Kubernetes StorageClass named `"gp2"` and annotates it as not the default storage class for the cluster, managing how storage volumes are provisioned and utilized in the cluster. Ensuring that the `"gp2"` StorageClass does not become the default storage class is needed as we additionally create an EFS Storage (Elastic File System), which is described in the next subsection.

```
#
# EBS CSI controller
#
module "ebs_csi_controller_role" {
```

```
  source                      = "terraform-aws-modules/iam/aws//modules/iam-assumable-role-
  version                     = "5.11.1"
  create_role                 = true
  role_name                   = local.ebs_csi_service_account_role_name
  provider_url                = replace(module.eks.cluster_oidc_issuer_url, "https://", "")
  role_policy_arns            = [aws_iam_policy.ebs_csi_controller_sa.arn]
  oidc_fully_qualified_subjects = ["system:serviceaccount:${local.cluster_namespace}:${local.
}

resource "aws_iam_policy" "ebs_csi_controller_sa" {
  name        = local.ebs_csi_service_account_name
  description = "EKS ebs-csi-controller policy for cluster ${var.cluster_name}"

  policy = jsonencode({
    "Version" : "2012-10-17",
    "Statement" : [
      {
        "Action" : [
          "ec2:AttachVolume",
          "ec2:CreateSnapshot",
          "ec2:CreateTags",
          "ec2:CreateVolume",
          "ec2:DeleteSnapshot",
          "ec2:DeleteTags",
          "ec2:DeleteVolume",
          "ec2:DescribeInstances",
          "ec2:DescribeSnapshots",
          "ec2:DescribeTags",
          "ec2:DescribeVolumes",
          "ec2:DetachVolume",
        ],
        "Effect" : "Allow",
        "Resource" : "*"
      }
  ] })
}

resource "kubernetes_annotations" "ebs-no-default-storageclass" {
  api_version = "storage.k8s.io/v1"
```

```
  kind       = "StorageClass"
  force      = "true"

  metadata {
    name = "gp2"
  }
  annotations = {
    "storageclass.kubernetes.io/is-default-class" = "false"
  }
}
```

8.2.2.2 Elastic File System

The EFS CSI (Elastic File System Container Storage Interface) driver permits EKS pods to use EFS as a persistent volume for data storage, enabling pods to use EFS as a scalable and shared storage solution.. The driver itself is deployed using a Helm chart through the "helm_release" resource. Of course we also need to create an IAM role for the EFS CSI driver, which is done using the "attach_efs_csi_role" module, which allows the driver to assume a role with OIDC authentication, and grants the necessary permissions for working with EFS, similar to the EBS setup.

For security, the code creates an AWS security group named "allow_nfs" that allows inbound NFS traffic on port 2049 from the private subnets of the VPC. This allows the EFS mount targets to communicate with the EFS file system securely. The EFS file system and access points are created manually for each private subnet mapping the "aws_efs_mount_target" to the "aws_efs_file_system" resource.

Finally, the code defines a Kubernetes StorageClass named "efs" using the "kubernetes_storage_class" resource. The StorageClass specifies the EFS CSI driver as the storage provisioner and the EFS file system created earlier as the backing storage. Additionally, the "efs" StorageClass is marked as the default storage class for the cluster using an annotation. This allows dynamic provisioning of EFS-backed persistent volumes for Kubernetes pods on default, simplifying the process of handling storage in the EKS cluster. This is done for example for the Airflow deployment in a later step.

```
#
# EFS
#
resource "helm_release" "aws_efs_csi_driver" {
  chart      = "aws-efs-csi-driver"
```

```
  name        = "aws-efs-csi-driver"
  namespace   = "kube-system"
  repository  = "https://kubernetes-sigs.github.io/aws-efs-csi-driver/"
  set {
    name  = "controller.serviceAccount.create"
    value = true
  }
  set {
    name  = "controller.serviceAccount.annotations.eks\\.amazonaws\\.com/role-arn"
    value = module.attach_efs_csi_role.iam_role_arn
  }
  set {
    name  = "controller.serviceAccount.name"
    value = "efs-csi-controller-sa"
  }
}

module "attach_efs_csi_role" {
  source = "terraform-aws-modules/iam/aws//modules/iam-role-for-service-accounts-eks"
  role_name           = "efs-csi"
  attach_efs_csi_policy = true
  oidc_providers = {
    ex = {
      provider_arn               = module.eks.oidc_provider_arn
      namespace_service_accounts = ["kube-system:efs-csi-controller-sa"]
    }
  }
}

resource "aws_security_group" "allow_nfs" {
  name        = "allow nfs for efs"
  description = "Allow NFS inbound traffic"
  vpc_id      = var.vpc_id

  ingress {
    description = "NFS from VPC"
    from_port   = 2049
    to_port     = 2049
    protocol    = "tcp"
```

```
    cidr_blocks = var.private_subnets_cidr_blocks
  }
  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
}


resource "aws_efs_file_system" "stw_node_efs" {
  creation_token = "efs-for-stw-node"
}


resource "aws_efs_mount_target" "stw_node_efs_mt_0" {
  file_system_id  = aws_efs_file_system.stw_node_efs.id
  subnet_id       = var.private_subnets[0]
  security_groups = [aws_security_group.allow_nfs.id]
}


resource "aws_efs_mount_target" "stw_node_efs_mt_1" {
  file_system_id  = aws_efs_file_system.stw_node_efs.id
  subnet_id       = var.private_subnets[1]
  security_groups = [aws_security_group.allow_nfs.id]
}


resource "aws_efs_mount_target" "stw_node_efs_mt_2" {
  file_system_id  = aws_efs_file_system.stw_node_efs.id
  subnet_id       = var.private_subnets[2]
  security_groups = [aws_security_group.allow_nfs.id]
}


resource "kubernetes_storage_class_v1" "efs" {
  metadata {
    name = "efs"
    annotations = {
      "storageclass.kubernetes.io/is-default-class" = "true"
    }
```

```
  }

  storage_provisioner = "efs.csi.aws.com"
  parameters = {
    provisioningMode = "efs-ap"                         # Dynamic provisioning
    fileSystemId     = aws_efs_file_system.stw_node_efs.id # module.efs.id
    directoryPerms   = "777"
  }

  mount_options = [
    "iam"
  ]
}
```

8.2.2.3 Cluster Autoscaler

The EKS Cluster Autoscaler ensures that the cluster can automatically scale its worker nodes based on the workload demands, ensuring optimal resource utilization and performance.

The necessary IAM settings are set up prior to deploying the Autoscaler. First, an IAM policy named `"node_additional"` is created to grant permission to describe EC2 instances and related resources. This enables the Autoscaler to gather information about the current state of the worker nodes and make informed decisions regarding scaling. For each managed node group in the EKS cluster (defined by the `"eks_managed_node_groups"` module output), the IAM policy is attached to its corresponding IAM role. This ensures that all worker nodes have the required permissions to work with the Autoscaler. After setting up the IAM policies, tags are added to provide the necessary information for the EKS Cluster Autoscaler to identify and manage the Auto Scaling Groups effectively and to support cluster autoscaling from zero for each node group. The tags are created for each node group (`"nodegroup_t3_small"`, `"nodegroup_t3_medium"` ,and `"nodegroup_t3_large"`) and are based on the specified tag lists defined in the `"local.eks_asg_tag_list_*"` variables.

The EKS Cluster Autoscaler itself is instantiated using the custom `"eks_autoscaler"` module on the bottom of the code snippet. The module is called to set up the Autoscaler for the EKS cluster and the required input variables are provided accordingly. Its components are described in detailed in the following.

```
#
# EKS Cluster autoscaler
#
```

```
resource "aws_iam_policy" "node_additional" {
  name        = "${local.cluster_name}-additional"
  description = "${local.cluster_name} node additional policy"

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = [
          "ec2:Describe*",
        ]
        Effect   = "Allow"
        Resource = "*"
      },
    ]
  })
}


resource "aws_iam_role_policy_attachment" "additional" {
  for_each = module.eks.eks_managed_node_groups

  policy_arn = aws_iam_policy.node_additional.arn
  role       = each.value.iam_role_name
}


# Tags for the ASG to support cluster-autoscaler scale up from 0 for nodegroup2
resource "aws_autoscaling_group_tag" "nodegroup_t3_small" {
  for_each               = local.eks_asg_tag_list_nodegroup_t3_small_label
  autoscaling_group_name = element(module.eks.eks_managed_node_groups_autoscaling_grou
  tag {
    key                 = each.key
    value               = each.value
    propagate_at_launch = true
  }
}


resource "aws_autoscaling_group_tag" "nodegroup_t3_medium" {
  for_each               = local.eks_asg_tag_list_nodegroup_t3_medium_label
  autoscaling_group_name = element(module.eks.eks_managed_node_groups_autoscaling_grou
```

```
  tag {
    key                 = each.key
    value               = each.value
    propagate_at_launch = true
  }
}

resource "aws_autoscaling_group_tag" "nodegroup_t3_large" {
  for_each              = local.eks_asg_tag_list_nodegroup_t3_large_label
  autoscaling_group_name = element(module.eks.eks_managed_node_groups_autoscaling_group_names
  tag {
    key                 = each.key
    value               = each.value
    propagate_at_launch = true
  }
}

module "eks_autoscaler" {
  source                        = "./autoscaler"
  cluster_name                  = local.cluster_name
  cluster_namespace             = local.cluster_namespace
  aws_region                    = var.aws_region
  cluster_oidc_issuer_url       = module.eks.cluster_oidc_issuer_url
  autoscaler_service_account_name = local.autoscaler_service_account_name
}
```

The configurationof the Cluster Autoscaler begins with the creation of a Helm release named `"cluster-autoscaler"` using the `"helm_release"` resource. The Helm chart is sourced from the `"kubernetes.github.io/autoscaler"` repository with the chart version `"9.10.7"`. The settings inside the Helm release include the AWS region, RBAC (Role-Based Access Control) settings for the service account, cluster auto-discovery settings, and the creation of the service account with the required permissions.

The necessary resources for the settings are created accordingly in the following. The service account is created using the `"iam_assumable_role_admin"` module with an assumable IAM role that allows the service account to access the necessary resources for scaling. It is associated with the OIDC (OpenID Connect) provider for the cluster to permit access.

An IAM policy named `"cluster_autoscaler"` is created to permit the Cluster Autoscaler to interact with Auto Scaling Groups, EC2 instances, launch configurations, and tags. The

policy includes two statements: `"clusterAutoscalerAll"` and `"clusterAutoscalerOwn"`. The first statement grants read access to Auto Scaling Group-related resources, while the second statement allows the Cluster Autoscaler to modify the desired capacity of the Auto Scaling Groups and terminate instances. The policy also includes conditions to ensure that the Cluster Autoscaler can only modify resources with specific tags. The conditions check that the Auto Scaling Group has a tag `"k8s.io/cluster-autoscaler/enabled"` set to `"true"` and a tag `"k8s.io/cluster-autoscaler/<cluster_name>"` set to `"owned"`. If you remember it, we have set these tags when setting up the managed node groups for the EKS Cluster in the previous step.

```
resource "helm_release" "cluster-autoscaler" {
  name             = "cluster-autoscaler"
  namespace        = var.cluster_namespace
  repository       = "https://kubernetes.github.io/autoscaler"
  chart            = "cluster-autoscaler"
  version          = "9.10.7"
  create_namespace = false

  set {
    name  = "awsRegion"
    value = var.aws_region
  }
  set {
    name  = "rbac.serviceAccount.name"
    value = var.autoscaler_service_account_name
  }
  set {
    name  = "rbac.serviceAccount.annotations.eks\\.amazonaws\\.com/role-arn"
    value = module.iam_assumable_role_admin.iam_role_arn
    type  = "string"
  }
  set {
    name  = "autoDiscovery.clusterName"
    value = var.cluster_name
  }
  set {
    name  = "autoDiscovery.enabled"
    value = "true"
  }
  set {
```

```
    name  = "rbac.create"
    value = "true"
  }
}

module "iam_assumable_role_admin" {
  source                          = "terraform-aws-modules/iam/aws//modules/iam-assumable-role-
  version                         = "~> 4.0"
  create_role                     = true
  role_name                       = "cluster-autoscaler"
  provider_url                    = replace(var.cluster_oidc_issuer_url, "https://", "")
  role_policy_arns                = [aws_iam_policy.cluster_autoscaler.arn]
  oidc_fully_qualified_subjects = ["system:serviceaccount:${var.cluster_namespace}:${var.auto
}

resource "aws_iam_policy" "cluster_autoscaler" {
  name_prefix = "cluster-autoscaler"
  description = "EKS cluster-autoscaler policy for cluster ${var.cluster_name}"
  policy      = data.aws_iam_policy_document.cluster_autoscaler.json
}

data "aws_iam_policy_document" "cluster_autoscaler" {
  statement {
    sid    = "clusterAutoscalerAll"
    effect = "Allow"

    actions = [
      "autoscaling:DescribeAutoScalingGroups",
      "autoscaling:DescribeAutoScalingInstances",
      "autoscaling:DescribeLaunchConfigurations",
      "autoscaling:DescribeTags",
      "ec2:DescribeLaunchTemplateVersions",
    ]

    resources = ["*"]
  }

  statement {
    sid    = "clusterAutoscalerOwn"
```

```
    effect = "Allow"

    actions = [
      "autoscaling:SetDesiredCapacity",
      "autoscaling:TerminateInstanceInAutoScalingGroup",
      "autoscaling:UpdateAutoScalingGroup",
    ]

    resources = ["*"]

    condition {
      test     = "StringEquals"
      variable = "autoscaling:ResourceTag/k8s.io/cluster-autoscaler/${var.cluster_name}
      values   = ["owned"]
    }
    condition {
      test     = "StringEquals"
      variable = "autoscaling:ResourceTag/k8s.io/cluster-autoscaler/enabled"
      values   = ["true"]
    }
  }
}
```

8.2.3 Networking

The `networking` module of the infrastructure directory integrates an *Application Load Balancer* (ALB) and *External DNS* in the cluster. Both play crucial roles in managing and exposing Kubernetes applications within the EKS cluster to the outside world. The ALB serves as an Ingress Controller to route external traffic to Kubernetes services, while External DNS automates the management of DNS records, making it easier to access services using user-friendly domain names. The root module of network just calls both submodules, which are described in detail in the following sections.

```
module "external-dns" {
  ...
}


module "application-load-balancer" {
```

```
  ...
}
```

8.2.3.1 AWS Application Load Balancer (ALB)

The ALB is a managed load balancer service provided by AWS. In the context of an EKS cluster, the ALB serves as an Ingress Controller and thus is responsible for routing external traffic to the appropriate services and pods running inside your Kubernetes cluster. The ALB acts as the entry point to our applications and enables us to expose multiple services over a single public IP address or domain name, which simplifies access for users and clients.

The code starts by defining some local variables, followed by creating an assumable IAM role for the AWS Load Balancer Controller service account by the module `aws_load_balancer_controller_controller_role`. The service account holds the necessary permissions and associates with the OIDC provider of the EKS cluster as it is the same module call we already used multiple times beforehand. The IAM policy for the role is defined in the `"aws_iam_policy.aws_load_balancer_controller_controller_sa"` resource.

Since its policy document is quite extensive, it is loaded from a file named `"AWSLoadBalancerControllerPolicy.j` In summary, the AWS IAM document allows the AWS Elastic Load Balancing (ELB) controller, specifically the Elastic Load Balancer V2 (ELBV2) API, to perform various actions related to managing load balancers, target groups, listeners, rules, and tags. The document includes several"Allow" statements that grant permissions for actions like describing and managing load balancers, target groups, listeners, and rules. It also allows the controller to create and delete load balancers, target groups, and listeners, as well as modify their attributes. Additionally, the document permits the addition and removal of tags for ELBV2 resources.

After setting up the IAM role, the code proceeds to install the AWS Load Balancer Controller using Helm. The Helm chart is sourced from the `"aws.github.io/eks-charts"` repository, specifying version `"v2.4.2"`. The service account configuration is provided to the Helm release's values, including the name of the service account and annotations to associate it with the IAM role created earlier. The `"eks.amazonaws.com/role-arn"` annotation points to the ARN of the IAM role associated with the service account, allowing the controller to assume that role and operate with the appropriate permissions.

```
locals {
  aws_load_balancer_controller_service_account_role_name = "aws-load-balancer-controller-role
  aws_load_balancer_controller_service_account_name      = "aws-load-balancer-controller-sa"
}
```

```
data "aws_caller_identity" "current" {}
data "aws_region" "current" {} #

module "aws_load_balancer_controller_controller_role" {
  source                          = "terraform-aws-modules/iam/aws//modules/iam-assumabl
  version                         = "5.11.1"
  create_role                     = true
  role_name                       = local.aws_load_balancer_controller_service_account_r
  provider_url                    = replace(var.cluster_oidc_issuer_url, "https://", "")
  role_policy_arns                = [aws_iam_policy.aws_load_balancer_controller_control
  oidc_fully_qualified_subjects = ["system:serviceaccount:kube-system:${local.aws_load
}

resource "aws_iam_policy" "aws_load_balancer_controller_controller_sa" {
  name        = local.aws_load_balancer_controller_service_account_name
  description = "EKS ebs-csi-controller policy for cluster ${var.cluster_name}"

  policy = file("${path.module}/AWSLoadBalancerControllerPolicy.json")
}

resource "helm_release" "aws-load-balancer-controller" {
  name             = var.helm_chart_name
  namespace        = var.namespace
  chart            = "aws-load-balancer-controller"
  create_namespace = false

  repository = "https://aws.github.io/eks-charts"
  version    = var.helm_chart_version

  values = [yamlencode({
    clusterName = var.cluster_name
    image = {
      tag = "v2.4.2"
    },
    serviceAccount = {
      name = "${local.aws_load_balancer_controller_service_account_name}"
      annotations = {
        "eks.amazonaws.com/role-arn" = "arn:aws:iam::${data.aws_caller_identity.curren
```

```
      }
    }
  })]
}
```

8.2.3.2 External DNS

External DNS is a Kubernetes add-on that automates the creation and management of DNS records for Kubernetes services. It is particularly useful when services are exposed to the internet through the ALB or any other Ingress Controller. When an Ingress resource is created that defines how external traffic should be routed to services within the EKS cluster, External DNS automatically updates the DNS provider with the corresponding DNS records (in our case this is Route 53 in AWS). Automatically configuring DNS records ensures that the records are always up-to-date, which helps maintain consistency and reliability in the DNS configuration, and users can access the Kubernetes services using user-friendly domain names rather than relying on IP addresses.

The code is structured similar to the ALB and defines local variables first, followed by creating a service account to interact with AWS resources. The service account, its role with OIDC and the policy with relevant permissions are created by the `external_dns_controller_role` module same to as we know it from previous implementations. The policy allows the external DNS controller to operate within the specified AWS Route 53 hosted zone, such as changing resource record sets, and listing hosted zones and resource record sets.

Finally, the Helm is used to to deploy the external DNS controller as a Kubernetes resource. The Helm release configuration includes specifying the previously create service account, the IAM `role-arn` associated with it, the `aws.region` where the Route 53 hosted zone exists, and a `domainFilter` which filters to a specific domain provided by us.

```
locals {
  external_dns_service_account_role_name = "external-dns-role"
  external_dns_service_account_name      = "external-dns-sa"
}

data "aws_caller_identity" "current" {}
data "aws_region" "current" {} #

module "external_dns_controller_role" {
  source                     = "terraform-aws-modules/iam/aws//modules/iam-assumable-role-
  version                    = "5.11.1"
```

```
  create_role                     = true
  role_name                       = local.external_dns_service_account_role_name
  provider_url                    = replace(var.cluster_oidc_issuer_url, "https://", "")
  role_policy_arns                = [aws_iam_policy.external_dns_controller_sa.arn]
  oidc_fully_qualified_subjects = ["system:serviceaccount:${var.namespace}:${local.ext
}

resource "aws_iam_policy" "external_dns_controller_sa" {
  name        = local.external_dns_service_account_name
  description = "EKS ebs-csi-controller policy for cluster ${var.cluster_name}"

  policy = jsonencode({
    "Version" : "2012-10-17",
    "Statement" : [
      {
        "Effect" : "Allow",
        "Action" : [
          "route53:ChangeResourceRecordSets"
        ],
        "Resource" : [
          "arn:aws:route53:::hostedzone/*"
        ]
      },
      {
        "Effect" : "Allow",
        "Action" : [
          "route53:ListHostedZones",
          "route53:ListResourceRecordSets"
        ],
        "Resource" : [
          "*"
        ]
      }
    ]
  })
}

resource "helm_release" "external_dns" {
  name             = var.name
```

```
  namespace        = var.namespace
  chart            = var.helm_chart_name
  create_namespace = false

  repository = "https://charts.bitnami.com/bitnami"
  version    = var.helm_chart_version

  values = [yamlencode({
    serviceAccount = {
      create = true
      name   = "${local.external_dns_service_account_name}"
      annotations = {
        "eks.amazonaws.com/role-arn" = "arn:aws:iam::${data.aws_caller_identity.current.accou
      }
    },
    aws = {
      zoneType = "public"
      region   = "${data.aws_region.current.name}"
    },
    policy = "sync"
    domainFilter = [
      "${var.domain_name}"
    ]
    provider   = "aws"
    txtOwnerId = "${var.name}"
  })]
}
```

8.2.4 Relational Database Service

The Amazon RDS (Relational Database Service) instance is provisioned by the
`aws_db_instance` resource. It configures the instance with the specified settings, such
as `allocated_storage`, `storage_type`, `engine`, `db_name`, `username`, and `password`, etc. All
these parameters are provided whenever the module is invoked, e.g. in the Airflow or Mlflow
modules.. The `skip_final_snapshot` set to true states that no final DB snapshot will be
created when the instance is deleted.

The resource `aws_db_subnet_group` creates an RDS subnet group with the name
`"vpc-subnet-group-${local.rds_name}"`. It associates the RDS instance with the private

subnets specified in the `VPC` module, and is used to define the subnets in which the RDS instance can be launched. Similar to the subnet group, the RDS instance uses an own security group. The security group `aws_security_group` is attached to the RDS instance. It specifies `ingress` (inbound)) and `egress` (outbound) rules to control network traffic. In this case, it allows inbound access on the specified port used by the RDS engine (5432 for PostgreSQL) from the CIDR blocks specified in the `private_subnets_cidr_blocks`, and allows all outbound traffic (`0.0.0.0/0`) from the RDS instance.

The `rds` module is not necessarily needed to run a kubernetes cluster properly. It is merely an extension of the cluster and is needed to store relevant data of the tools used, such as airflow or mlflow. The module is thus called directly from the own airflow and mlflow modules.

```
locals {
  rds_name           = var.rds_name
  rds_engine         = var.rds_engine
  rds_engine_version = var.rds_engine_version
  rds_port           = var.rds_port
}


resource "aws_db_subnet_group" "default" {
  name       = "vpc-subnet-group-${local.rds_name}"
  subnet_ids = var.private_subnets
}


resource "aws_db_instance" "rds_instance" {
  allocated_storage     = var.max_allocated_storage
  storage_type          = var.storage_type
  engine                = local.rds_engine
  engine_version        = local.rds_engine_version
  instance_class        = var.rds_instance_class
  db_name               = "${local.rds_name}_db"
  username              = "${local.rds_name}_admin"
  password              = var.rds_password
  identifier            = "${local.rds_name}-${local.rds_engine}"
  port                  = local.rds_port
  vpc_security_group_ids = [aws_security_group.rds_sg.id]
  db_subnet_group_name  = aws_db_subnet_group.default.name
  skip_final_snapshot   = true
}
```

```
resource "aws_security_group" "rds_sg" {
  name    = "${local.rds_name}-${local.rds_engine}-sg"
  vpc_id = var.vpc_id

  ingress {
    description = "Enable postgres access"
    from_port   = local.rds_port
    to_port     = local.rds_port
    protocol    = "tcp"
    cidr_blocks = var.private_subnets_cidr_blocks
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

8.3 Modules

Within the setup, there are multiple custom modules, namely airflow, mlflow, jupyterhub, and monitoring. Each module is responsible for deploying a specific workflow tool.

These module names also align with their corresponding namespaces within the cluster.

8.3.1 Airflow

The `Airflow` module is responsible for provisioning all components related to the deployment of Airflow. Being a crucial workflow orchestration tool in our ML platform, Airflow is tightly integrated with various other components in the Terraform codebase, which requires it to receive multiple input variables and configurations. Airflow itself is deployed in the Terraform code via a Helm chart. The provided Terraform code also integrates the Airflow deployment with AWS S3 for efficient data storage and logging. It also utilizes an AWS RDS instance from the infrastructure section to serve as the metadata storage. Additionally, relevant Kubernetes secrets are incorporated into the setup to ensure a secure deployment.

The code starts by declaring several local variables that store the names of Kubernetes secrets and S3 buckets for data storage and logging. Next, it creates a Kubernetes namespace for Airflow to isolate the deployment.

```
locals {
  k8s_airflow_db_secret_name   = "${var.name_prefix}-${var.namespace}-db-auth"
  git_airflow_repo_secret_name = "${var.name_prefix}-${var.namespace}-https-git-secret
  git_organization_secret_name = "${var.name_prefix}-${var.namespace}-organization-git
  s3_data_bucket_secret_name   = "${var.name_prefix}-${var.namespace}-${var.s3_data_bu
  s3_data_bucket_name          = "${var.name_prefix}-${var.namespace}-${var.s3_data_bu
  s3_log_bucket_name           = "${var.name_prefix}-${var.namespace}-log-storage"
}


data "aws_caller_identity" "current" {}
data "aws_region" "current" {} #


resource "kubernetes_namespace" "airflow" {
  metadata {


    name = var.namespace
  }
}


#
# Log Storage
#
module "s3-remote-logging" {
  source              = "./remote_logging"
  s3_log_bucket_name = local.s3_log_bucket_name
  namespace           = var.namespace
  s3_force_destroy    = var.s3_force_destroy
  oidc_provider_arn   = var.oidc_provider_arn
}


#
# Data Storage
#
module "s3-data-storage" {
  source                 = "./data_storage"
  s3_data_bucket_name    = local.s3_data_bucket_name
```

```
  namespace                  = var.namespace
  s3_force_destroy           = var.s3_force_destroy
  s3_data_bucket_secret_name = local.s3_data_bucket_secret_name
}
```

Afterward, two custom modules, `"s3-remote-logging"` and `"s3-data-storage"` set up S3 buckets for remote logging and data storage. Both modules handle creating the S3 buckets and necessary IAM roles for accessing them. The terraform code of both modules is not depicted here, it is visible on github[2] though. The main difference between the modules are in the in the assume role policies that are needed for the different use cases of storing and reading data, or logging to S3. While the `"s3_log_bucket_role"` allows a Federated entity, specified by an OIDC provider ARN, to assume the role using `"sts:AssumeRoleWithWebIdentity"`, the `"s3_data_bucket_role"` allows both a specific IAM user (constructed from the user's ARN) and the Amazon S3 service itself to assume the role using `"sts:AssumeRole"`.

s3-data-storage role policy

```
# s3-data-storage role policy
resource "aws_iam_role" "s3_data_bucket_role" {
  name                  = "${var.namespace}-s3-data-bucket-role"
  max_session_duration = 28800

  assume_role_policy = <<EOF
  {
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
              "AWS": "arn:aws:iam::${data.aws_caller_identity.current.account_id}:user/${aws_
            },
            "Action": "sts:AssumeRole"
        },
        {
            "Effect": "Allow",
            "Principal": {
              "Service": "s3.amazonaws.com"
            },
```

[2]https://github.com/seblum/mlops-airflow-on-eks

```
            "Action": "sts:AssumeRole"
        }
    ]
  }
  EOF
}
```

s3-remote-logging role policy

```
# s3-remote-logging role policy
resource "aws_iam_role" "s3_log_bucket_role" {
  name                  = "${var.namespace}-s3-log-bucket-access-role"
  max_session_duration = 28800

  assume_role_policy = <<EOF
  {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action" : "sts:AssumeRoleWithWebIdentity",
        "Effect": "Allow",
        "Principal" : {
          "Federated" : [
            "${var.oidc_provider_arn}"
          ]
        }
      }
    ]
  }
  EOF
}
```

After the S3 buckets are set up, the code proceeds to create Kubernetes secrets to store various credentials required for Airflow's operation. These include credentials for PostgreSQL database, GitHub authentication secrets for accessing private repositories, and secrets for accessing GitHub organizations which are required to authenticate users. The `"rds-airflow"` module is used to create the RDS instance for Airflow, which will serve as the external database for the deployment.

The Apache Airflow deployment is defined using a `helm_release` of the *Airflow Community Helm Chart* and is highly customized to cater to our specific needs. The release includes various configurations for Airflow, such as custom environment variables, extra environment variables sourced from the GitHub organization secret, and the `KubernetesExecutor` Airflow executor. The deployment enables DAG synchronization of a dedicated Github repository which includes our Airflow DAGs (see chapter 9). It also configures a persistent volume using Amazon EFS for Airflow logs and a Kubernetes Ingress resource to expose the Airflow web interface using an Application Load Balancer (ALB). Additionally, readiness and liveness probes are configured for the web server.

```
#
# Helm Release Airflow
#
resource "kubernetes_secret" "airflow_db_credentials" {
  metadata {
    name      = local.k8s_airflow_db_secret_name
    namespace = helm_release.airflow.namespace
  }
  data = {
    "postgresql-password" = module.rds-airflow.rds_password
  }
}


resource "kubernetes_secret" "airflow_https_git_secret" {
  metadata {
    name      = local.git_airflow_repo_secret_name
    namespace = helm_release.airflow.namespace
  }
  data = {
    "username" = var.git_username
    "password" = var.git_token
  }
}


resource "kubernetes_secret" "airflow_organization_git_secret" {
  metadata {
    name      = local.git_organization_secret_name
    namespace = helm_release.airflow.namespace
  }
  data = {
```

```
      "GITHUB_CLIENT_ID"     = var.git_client_id
      "GITHUB_CLIENT_SECRET" = var.git_client_secret
    }
}


# RDS
resource "random_password" "rds_password" {
  length  = 16
  special = false
}


module "rds-airflow" {
  source                     = "../../infrastructure/rds"
  vpc_id                     = var.vpc_id
  private_subnets            = var.private_subnets
  private_subnets_cidr_blocks = var.private_subnets_cidr_blocks
  rds_port                   = var.rds_port
  rds_name                   = var.rds_name
  rds_password               = coalesce(var.rds_password, random_password.rds_password
  rds_engine                 = var.rds_engine
  rds_engine_version         = var.rds_engine_version
  rds_instance_class         = var.rds_instance_class
  storage_type               = var.storage_type
  max_allocated_storage      = var.max_allocated_storage
}


# HELM
resource "helm_release" "airflow" {
  name             = var.name
  namespace        = var.namespace
  create_namespace = var.create_namespace

  repository = "https://airflow-helm.github.io/charts"
  chart      = var.helm_chart_name
  version    = var.helm_chart_version
  wait       = false # deactivate post install hooks otherwise will fail

  values = [yamlencode({
    airflow = {
```

```
extraEnv = [
  {
    name = "GITHUB_CLIENT_ID"
    valueFrom = {
      secretKeyRef = {
        name = local.git_organization_secret_name
        key  = "GITHUB_CLIENT_ID"
      }
    }
  },
  {
    name = "GITHUB_CLIENT_SECRET"
    valueFrom = {
      secretKeyRef = {
        name = local.git_organization_secret_name
        key  = "GITHUB_CLIENT_SECRET"
      }
    }
  }
],
config = {
  AIRFLOW__WEBSERVER__EXPOSE_CONFIG = true
  AIRFLOW__WEBSERVER__BASE_URL       = "http://${var.domain_name}/${var.domain_suffix}"

  AIRFLOW__CORE__LOAD_EXAMPLES = false
  # AIRFLOW__LOGGING__LOGGING_LEVEL            = "DEBUG"
  # AIRFLOW__LOGGING__REMOTE_LOGGING           = true
  # AIRFLOW__LOGGING__REMOTE_BASE_LOG_FOLDER = "s3://${module.s3-data-storage.s3_log_bu
  # AIRFLOW__LOGGING__REMOTE_LOG_CONN_ID       = "aws_logs_storage_access"
  AIRFLOW__CORE__DEFAULT_TIMEZONE = "Europe/Amsterdam"
},
users = []
image = {
  repository = "seblum/airflow"
  tag        = "2.6.3-python3.11-custom-light"
  pullPolicy = "IfNotPresent"
  pullSecret = ""
  uid        = 50000
  gid        = 0
```

```
    },
    executor           = "KubernetesExecutor"
    fernetKey          = var.fernet_key
    webserverSecretKey = "THIS IS UNSAFE!"
    connections = [
      {
        id          = "aws_logs_storage_access"
        type        = "aws"
        description = "AWS connection to store logs on S3"
        extra       = "{\"region_name\": \"${data.aws_region.current.name}\"}"
      }
    ],
    variables = [
      {
        key   = "MLFLOW_TRACKING_URI"
        value = "http://mlflow-service.mlflow.svc.cluster.local"
      },
      {
        key   = "s3_access_name"
        value = "${local.s3_data_bucket_secret_name}"
      }
    ]
  },
  serviceAccount = {
    create = true
    name   = "airflow-sa"
    annotations = {
      "eks.amazonaws.com/role-arn" = "${module.s3-remote-logging.s3_log_bucket_role_
    }
  },
  scheduler = {
    logCleanup = {
      enabled = false
    }
  },
  workers = {
    enabled = false
    logCleanup = {
      enables = true
```

```
    }
  },
  flower = {
    enabled = false
  },
  postgresql = {
    enabled = false
  },
  redis = {
    enabled = false
  },
  externalDatabase = {
    type              = "postgres"
    host              = module.rds-airflow.rds_host
    port              = var.rds_port
    database          = "airflow_db"
    user              = "airflow_admin"
    passwordSecret    = local.k8s_airflow_db_secret_name
    passwordSecretKey = "postgresql-password"
  },
  dags = {
    path = "/opt/airflow/dags"
    gitSync = {
      enabled               = true
      repo                  = var.git_repository_url
      branch                = var.git_branch
      revision              = "HEAD"
      repoSubPath           = "workflows"
      httpSecret            = local.git_airflow_repo_secret_name
      httpSecretUsernameKey = "username"
      httpSecretPasswordKey = "password"
      syncWait              = 60
      syncTimeout           = 120
    }
  },
  logs = {
    path = "/opt/airflow/logs"
    persistence = {
      enabled = true
```

```
      storageClass : "efs"
      size : "5Gi"
      accessMode : "ReadWriteMany"
    }
  },
  ingress = {
    enabled    = true
    apiVersion = "networking.k8s.io/v1"
    web = {
      annotations = {
        "external-dns.alpha.kubernetes.io/hostname"  = "${var.domain_name}"
        "alb.ingress.kubernetes.io/scheme"           = "internet-facing"
        "alb.ingress.kubernetes.io/target-type"      = "ip"
        "kubernetes.io/ingress.class"                = "alb"
        "alb.ingress.kubernetes.io/group.name"       = "mlplatform"
        "alb.ingress.kubernetes.io/healthcheck-path" = "/${var.domain_suffix}/health
      }
      path = "/${var.domain_suffix}"
      host = "${var.domain_name}"
      precedingPaths = [{
        path        = "/${var.domain_suffix}*"
        serviceName = "airflow-web"
        servicePort = "web"
      }]
    }
  },
  web = {
    readinessProbe = {
      enabled           = true
      initialDelaySeconds = 45
    },
    livenessProbe = {
      enabled           = true
      initialDelaySeconds = 45
    },
    webserverConfig = {
      stringOverride = file("${path.module}/WebServerConfig.py")
    }
  },
```

```
  })]
}
```

In a final step of the Helm Chart, a custom `WebServerConfig.py` is specified which is set to integrate our Airflow deployment with a Github Authentication provider. The python script consists of two major parts: a custom AirflowSecurityManager class definition and the actual webserver_config configuration file for Apache Airflow's web server.

The custom `CustomSecurityManager` class extends the default AirflowSecurityManager to retrieves user information from the GitHub OAuth provider. The webserver_config configuration sets up the configurations for the web server component of Apache Airflow by indicating that OAuth will be used for user authentication. The `SECURITY_MANAGER_CLASS` is set to the previously defined `CustomSecurityManager` to customizes how user information is retrieved from the OAuth provider. Finally, the GitHub provider is configured with its required parameters like `client_id`, `client_secret`, and API endpoints.

```python
#########################################
# Custom AirflowSecurityManager
#########################################
from airflow.www.security import AirflowSecurityManager
import os


class CustomSecurityManager(AirflowSecurityManager):
    def get_oauth_user_info(self, provider, resp):
        if provider == "github":
            user_data = self.appbuilder.sm.oauth_remotes[provider].get("user").json()
            emails_data = (
                self.appbuilder.sm.oauth_remotes[provider].get("user/emails").json()
            )
            teams_data = (
                self.appbuilder.sm.oauth_remotes[provider].get("user/teams").json()
            )

            # unpack the user's name
            first_name = ""
            last_name = ""
            name = user_data.get("name", "").split(maxsplit=1)
            if len(name) == 1:
                first_name = name[0]
```

```python
        elif len(name) == 2:
            first_name = name[0]
            last_name = name[1]

        # unpack the user's email
        email = ""
        for email_data in emails_data:
            if email_data["primary"]:
                email = email_data["email"]
                break

        # unpack the user's teams as role_keys
        # NOTE: each role key will be "my-github-org/my-team-name"
        role_keys = []
        for team_data in teams_data:
            team_org = team_data["organization"]["login"]
            team_slug = team_data["slug"]
            team_ref = team_org + "/" + team_slug
            role_keys.append(team_ref)

        return {
            "username": "github_" + user_data.get("login", ""),
            "first_name": first_name,
            "last_name": last_name,
            "email": email,
            "role_keys": role_keys,
        }
    else:
        return {}


########################################
# Actual `webserver_config.py`
########################################
from flask_appbuilder.security.manager import AUTH_OAUTH

# only needed for airflow 1.10
# from airflow import configuration as conf
# SQLALCHEMY_DATABASE_URI = conf.get("core", "SQL_ALCHEMY_CONN")
```

```python
AUTH_TYPE = AUTH_OAUTH
SECURITY_MANAGER_CLASS = CustomSecurityManager

# registration configs
AUTH_USER_REGISTRATION = True  # allow users who are not already in the FAB DB
AUTH_USER_REGISTRATION_ROLE = (
    "Public"  # this role will be given in addition to any AUTH_ROLES_MAPPING
)

# the list of providers which the user can choose from
OAUTH_PROVIDERS = [
    {
        "name": "github",
        "icon": "fa-github",
        "token_key": "access_token",
        "remote_app": {
            "client_id": os.getenv("GITHUB_CLIENT_ID"),
            "client_secret": os.getenv("GITHUB_CLIENT_SECRET"),
            "api_base_url": "https://api.github.com",
            "client_kwargs": {"scope": "read:org read:user user:email"},
            "access_token_url": "https://github.com/login/oauth/access_token",
            "authorize_url": "https://github.com/login/oauth/authorize",
        },
    },
]

# a mapping from the values of `userinfo["role_keys"]` to a list of FAB roles
AUTH_ROLES_MAPPING = {
    "github-organization/airflow-users-team": ["User"],
    "github-organization/airflow-admin-team": ["Admin"],
}

# if we should replace ALL the user's roles each login, or only on registration
AUTH_ROLES_SYNC_AT_LOGIN = True

# force users to re-auth after 30min of inactivity (to keep roles in sync)
PERMANENT_SESSION_LIFETIME = 1800
```

8.3.2 Jupyterhub

JupyterHub is utilized in the setup to provide an IDE (Integrated Development Environment). Belows Terraform code defines a `helm_release` that deploys JupyterHub on our EKS cluster. There are no other resources needed to run JupyterHub compared to the other components of our ML platform The Helm configuration specifies various settings and customizations to include a JupyterHub instance with a single-user Jupyter notebook server. For example defining a post-start lifecycle hook to run a Git clone command inside the single-user notebook server container, or defining extra environment variable for the single-user server, namely `"MLFLOW_TRACKING_URI"` pointing to the previous specified MLflow service.

The Airflow configuration enables an Ingress resource to expose JupyterHub to the specified domain, and adds annotations to control routing and manage the AWS Application Load Balancer (ALB). It also includes settings for the JupyterHub proxy and enables a culling mechanism to automatically shut down idle user sessions.

Similar to the Airflow deployment, the JupyterHub instance is configured to use GitHub OAuthenticator for user authentication. The OAuthenticator is configured with the provided GitHub `client_id` and `client_secret`, and the `oauth_callback_url` to set a specific endpoint under the specified domain name.

```
resource "helm_release" "jupyterhub" {
  name             = var.name
  namespace        = var.name
  create_namespace = var.create_namespace

  repository = "https://jupyterhub.github.io/helm-chart/"
  chart     = var.helm_chart_name
  version   = var.helm_chart_version

  values = [yamlencode({
    singleuser = {
      defaultUrl = "/lab"
      image = {
        name = "seblum/jupyterhub-server"
        tag  = "latest"
      },
      lifecycleHooks = {
        postStart = {
          exec = {
            command = ["git", "clone", "${var.git_repository_url}"]
```

```
      }
    }
  },
  extraEnv = {
    "MLFLOW_TRACKING_URI" = "http://mlflow-service.mlflow.svc.cluster.local"
  }
},
ingress = {
  enabled : true
  annotations = {
    "external-dns.alpha.kubernetes.io/hostname" = "${var.domain_name}"
    "alb.ingress.kubernetes.io/scheme"          = "internet-facing"
    "alb.ingress.kubernetes.io/target-type"     = "ip"
    "kubernetes.io/ingress.class"               = "alb"
    "alb.ingress.kubernetes.io/group.name"      = "mlplatform"
  }
  hosts = ["${var.domain_name}", "www.${var.domain_name}"]
},
proxy = {
  service = {
    type = "ClusterIP"
  }
  secretToken = var.proxy_secret_token
}
cull = {
  enabled = true
  users   = true
}
hub = {
  baseUrl = "/${var.domain_suffix}"
  config = {
    GitHubOAuthenticator = {
      client_id          = var.git_client_id
      client_secret      = var.git_client_secret
      oauth_callback_url = "http://${var.domain_name}/${var.domain_suffix}/hub/oauth_call
    }
    JupyterHub = {
      authenticator_class = "github"
    }
```

```
      }
    }
  })]
}
```

8.4 Design Decisions

9 Use Case Development

The ML platform aims to enable the full development cycle of ML algorithms within DevOps and MLOps principles. This includes to enable development within a sophisticated IDE such as VSCode, as well as incorporating best practices like Gitflow, automation by CI/CD, and the containerization of code. Whereas the previous chapter explained the ML platform itself and its deployment, this chapter illustrates the workflow of developing ML models on the platform. Thus, a use case has been selected and implemented such that it integrates each of the previous mentioned principles and best practices.

Skin Cancer Detection

The exemplary use case *CNN for skin cancer detection* is based on a dataset from Kaggle[1] and has been chosen to implement a use case leveraging a Deep Learning model, as well as its data quality which requires little to no data exploration and preprocessing.

Data

The dataset from Kaggle[2] utilized in this project is obtained from the ISIC (International Skin Image Collaboration) Archive. It consists of 1800 images depicting benign moles and 1497 images representing malignant moles that have already been classified. The primary objective of this use case is to develop a model capable of visually classifying moles as either benign or malignant.

All the images in the dataset have been uniformly resized to a lower resolution of 224x224x3 RGB. Additionally, the dataset has already been cleaned and divided into a train set and a test set. This allows for a straightforward implementation that focuses on the crucial aspects of this tutorial, which involve putting machine learning into production and leveraging workflow management and model tracking tools. The data is also conveniently sorted based on the two types, namely benign and malignant.

---

[1]https://www.kaggle.com/code/fanconic/cnn-for-skin-cancer-detection
[2]https://www.kaggle.com/code/fanconic/cnn-for-skin-cancer-detection

```
data

 train

    benign
      ...

    malignant
       ...

 test

    benign
```

The dataset is kept in an AWS S3 Bucket, which was deployed on the pre-existing ML Platform. The deployment of the Bucket was accomplished using an Infrastructure-as-a-Code (IaaC) script, and the dataset was uploaded to the Bucket at the same time. To enable the ML pipeline to interact with AWS for data retrieval and storage, a custom utils script in the code base provides the necessary functionality.

9.1 Integrated Development Environment

Jupyterhub serves as the integrated server environment within the ML platform, providing an Integrated Development Environment (IDE). However, it deviates from the traditional Jupyter Notebooks and instead utilizes VSCode as the IDE. Upon initialization, Jupyterhub clones the GitHub repository *mlops-airflow-DAGs* that contains the code for the use case. This is the same repository that is synchronized with Airflow to load the provided DAGs. The purpose of this approach is to offer a user-friendly and efficient development experience to platform users.

In addition to synchronizing with Airflow, the use of GitHub provides additional tools to streamline development and incorporate DevOps practices effectively. One of these tools is Github Actions, which enables automation through CI/CD (Continuous Integration/Continuous Deployment) and supports the Git workflow. By configuring Github Actions through code, developers can seamlessly integrate these practices into their development processes. The configuration files for Github Actions are also cloned alongside the repository, ensuring that the integration and development processes remain smooth and efficient.

9.1.1 Github Repository

The code for the model pipeline is located in the GitHub repository called *mlops-airflow-DAGs*. It encompasses the setup of an Airflow DAG and the utilization of MLflow. The code responsible for the pipeline functionality can be located in the `src` subdirectory of the repository. The Airflow DAG is written in the `airflow_DAG.py` file situated in the root directory of the repository. Additionally, the repository includes a GitHub Actions workflow file located in the `.github/workflows/` subdirectory. The `Docker` subdirectory contains the Dockerfiles for the various containers used in the ML pipeline. These include a container with the code for data preprocessing and model training, a Dockerfile for serving the model using fastAPI, and a file that runs a streamlit app for sending inferences to the served model.

```
root
   Readme.md

 .github/workflows/

 airflow_docker_DAG.py
 airflow_kubernetes_DAG.py

 src
    preprocessing.py
      train.py
      ...
    model
        utils.py
           ...

 Docker

    fastapi-serve-app
       Dockerfile
          ...

    python-base-cnn-model
        Dockerfile
           ...
```

9.2 Pipeline Workflow

The code and machine learning pipeline have been modularized into distinct steps, including preprocessing, model training, model comparison, and model serving. Airflow serves as the model workflow tool, generating DAGs for managing the pipeline. MLflow is integrated to facilitate model tracking, registry, and serving functionalities. To ensure portability and scalability, the codebase has been containerized using Docker, allowing it to be executed in Docker and/or Kubernetes environments.

The `src` code is installed as a Python package within the Docker container, enabling easy invocation within the Airflow DAG. However, it is important to note that although Model Serving is triggered within the Airflow pipeline, it consists of a separate Python code and is not integrated into the `src` package. Likewise, model inferencing has its own distinct description and functionality. The code bases for both model serving and model inferencing can be found in the app/ directory, alongside their respective Dockerfiles. A detailed explanation of how these components function will be provided in the following section.
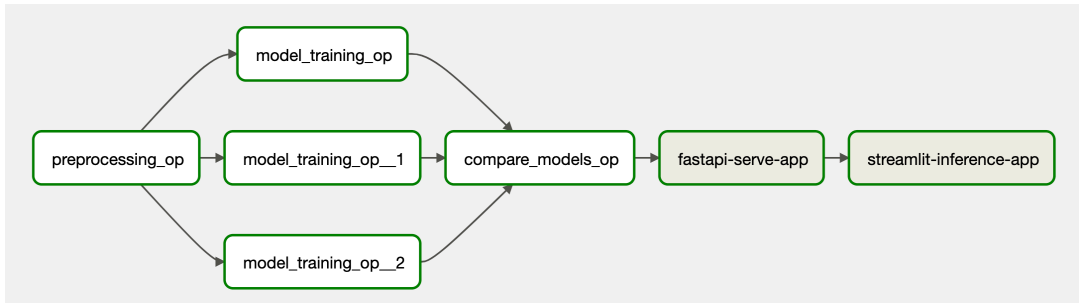
9.2.1 Airflow Workflow

The specification of the Airflow DAG, which includes the DAG structure, tasks, and their dependencies, can be found in the `airflow_DAG.py` file. The DAG is built using the TaskFlow API.

An ML pipeline of this use case consists of three main steps: preprocessing, training, and serving. The preprocessing step involves data processing and storing it in the S3 storage. The training step and code are designed to accommodate different TensorFlow models, allowing for parallel training on multiple models, thereby reducing the time to deployment. Since there are multiple models, it is essential to serve only the model with the best metrics based on the current data. Hence, an intermediate step is incorporated to compare the metrics of all the models and select the best one for serving.

To execute the pipeline steps, the Airflow Docker Operator is employed, which ensures that each step runs in a separate and isolated environment using Docker or Kubernetes jobs. Dockerizing the code is a prerequisite for this process. The Airflow task then invokes the relevant methods of the Python code and executes them accordingly.

Once the model is in the serving phase, a Streamlit app is deployed for applying inference on new data.

The code below defines the `ml_pipeline_dag` function as an Airflow DAG using the `dag` decorator. Each step of the pipeline, including data preprocessing, model training, model comparison, and serving the best model, is represented as a separate task with the `@task` decorator. Dependencies between these tasks are established by passing the output of one task as an argument to the next task. The `ml_pipeline` object serves as a representation of the entire DAG.

Importing Dependencies

At first the necessary dependencies for the code are imported, including libraries for MLflow, Airflow, and other utilities.

```python
# Imports necessary packages
import os
from enum import Enum
import mlflow
import pendulum
from airflow.decorators import dag, task
```

Setting MLflow Tracking URI and Experiment

Secondly, the necessary variables and constants for the whole Airflow DAG are defined and set. Either hard coded as string, or read from environment variables. Also, the MLflow tracking URI is set and a MLflow experiment retrieved, or created if none exists already.

```python
# Define variables and constants
MLFLOW_TRACKING_URI_local = "http://127.0.0.1:5008/"
MLFLOW_TRACKING_URI = "http://host.docker.internal:5008"
EXPERIMENT_NAME = "cnn_skin_cancer"
AWS_BUCKET = os.getenv("AWS_BUCKET")
```

```python
AWS_REGION = os.getenv("AWS_REGION")
AWS_ACCESS_KEY_ID = os.getenv("AWS_ACCESS_KEY_ID")
AWS_SECRET_ACCESS_KEY = os.getenv("AWS_SECRET_ACCESS_KEY")
AWS_ROLE_NAME = os.getenv("AWS_ROLE_NAME")

# Set MLflow tracking URI
mlflow.set_tracking_uri(MLFLOW_TRACKING_URI_local)

try:
    # Creating an experiment
    mlflow_experiment_id = mlflow.create_experiment(EXPERIMENT_NAME)
except:
    pass
# Setting the environment with the created experiment
mlflow_experiment_id = mlflow.set_experiment(EXPERIMENT_NAME).experiment_id
```

Setting Default Arguments and Environment Data

```python
# Set various model params and airflow or environment args
dag_default_args = {
    "owner": "seblum",
    "depends_on_past": False,
    "start_date": pendulum.datetime(2021, 1, 1, tz="UTC"),
    "tags": ["Keras CNN to classify skin cancer"],
}


kwargs_env_data = {
    "MLFLOW_TRACKING_URI": MLFLOW_TRACKING_URI,
    "MLFLOW_EXPERIMENT_ID": mlflow_experiment_id,
    "AWS_ACCESS_KEY_ID": AWS_ACCESS_KEY_ID,
    "AWS_SECRET_ACCESS_KEY": AWS_SECRET_ACCESS_KEY,
    "AWS_BUCKET": AWS_BUCKET,
    "AWS_REGION": AWS_REGION,
    "AWS_ROLE_NAME": AWS_ROLE_NAME,
}


model_params = {
    "num_classes": 2,
```

```
    "input_shape": (224, 224, 3),
    "activation": "relu",
    "kernel_initializer_glob": "glorot_uniform",
    "kernel_initializer_norm": "normal",
    "optimizer": "adam",
    "loss": "binary_crossentropy",
    "metrics": ["accuracy"],
    "validation_split": 0.2,
    "epochs": 2,
    "batch_size": 64,
    "learning_rate": 1e-5,
    "pooling": "avg",  # needed for resnet50
    "verbose": 2,
}
```

Defining the Airflow DAG

After all parameters have been set, the actual Airflow DAG for the CNN skin cancer workflow is defined. As each single task of the ML pipeline of the Airflow DAG is executed as a container run, the container image that is pulled from DockerHub needs to be specified.

```
skin_cancer_container_image = "seblum/cnn-skin-cancer:latest"

@dag(
    "cnn_skin_cancer_docker_workflow",
    default_args=dag_default_args,
    schedule_interval=None,
    max_active_runs=1,
)
def cnn_skin_cancer_workflow():
```

Defining Preprocessing Task

As a first step, the preprocessing task is defined, which performs data preprocessing.

```
    @task.docker(
        image=skin_cancer_container_image,
        multiple_outputs=True,
```

```python
        # Add the previously defined variables and constants as environment variables to the
        environment=kwargs_env_data,
        working_dir="/app",
        force_pull=True,
        network_mode="bridge",
    )
    def preprocessing_op(mlflow_experiment_id):
        """
        Perform data preprocessing.

        Args:
            mlflow_experiment_id (str): The MLflow experiment ID.

        Returns:
            dict: A dictionary containing the paths to preprocessed data.
        """
        import os

        from src.preprocessing import data_preprocessing

        aws_bucket = os.getenv("AWS_BUCKET")

        (
            X_train_data_path,
            y_train_data_path,
            X_test_data_path,
            y_test_data_path,
        ) = data_preprocessing(mlflow_experiment_id=mlflow_experiment_id, aws_bucket=aws_buck

        # Create dictionary with S3 paths to return
        return_dict = {
            "X_train_data_path": X_train_data_path,
            "y_train_data_path": y_train_data_path,
            "X_test_data_path": X_test_data_path,
            "y_test_data_path": y_test_data_path,
        }
        return return_dict
```

Defining Model Training Task

Similarly, the model training task is defined, which trains a machine learning model.

```python
@task.docker(
    image=skin_cancer_container_image,
    multiple_outputs=True,
    environment=kwargs_env_data,
    working_dir="/app",
    force_pull=True,
    network_mode="bridge",
)
def model_training_op(mlflow_experiment_id, model_class, model_params, input):
    """
    Train a model.

    Args:
        mlflow_experiment_id (str): The MLflow experiment ID.
        model_class (str): The class of the model to train.
        model_params (dict): A dictionary containing the model parameters.
        input (dict): A dictionary containing the input data.

    Returns:
        dict: A dictionary containing the results of the model training.
    """
    import os

    from src.train import train_model

    aws_bucket = os.getenv("AWS_BUCKET")
    run_id, model_name, model_version, model_stage = train_model(
        mlflow_experiment_id=mlflow_experiment_id,
        model_class=model_class,
        model_params=model_params,
        aws_bucket=aws_bucket,
        import_dict=input,
    )

    return_dict = {
        "run_id": run_id,
```

```
            "model_name": model_name,
            "model_version": model_version,
            "model_stage": model_stage,
        }
        return return_dict
```

Defining Model Comparison Task

The following code snippet defines the model comparison task, which compares trained models.

```python
@task.docker(
    image=skin_cancer_container_image,
    multiple_outputs=True,
    environment=kwargs_env_data,
    force_pull=True,
    network_mode="bridge",
)
def compare_models_op(train_data_basic, train_data_resnet50, train_data_crossval):
    """
    Compare trained models.

    Args:
        train_data_basic (dict): A dictionary containing the results of training the basi
        train_data_resnet50 (dict): A dictionary containing the results of training the R
        train_data_crossval (dict): A dictionary containing the results of training the C

    Returns:
        dict: A dictionary containing the results of the model comparison.
    """
    compare_dict = {
        train_data_basic["model_name"]: train_data_basic["run_id"],
        train_data_resnet50["model_name"]: train_data_resnet50["run_id"],
        train_data_crossval["model_name"]: train_data_crossval["run_id"],
    }

    print(compare_dict)
    from src.compare_models import compare_models
```

```
        serving_model_name, serving_model_uri, serving_model_version = compare_models(
        return_dict = {
            "serving_model_name": serving_model_name,
            "serving_model_uri": serving_model_uri,
            "serving_model_version": serving_model_version,
        }
        return return_dict
```

Defining Pipeline

After the tasks have been specified, they are connected together to define the workflow pipeline, specifying inputs and outputs.

```
# CREATE PIPELINE

preprocessed_data = preprocessing_op(
    mlflow_experiment_id=mlflow_experiment_id,
)
train_data_basic = model_training_op(
    mlflow_experiment_id=mlflow_experiment_id,
    model_class=Model_Class.Basic.name,
    model_params=model_params,
    input=preprocessed_data,
)
train_data_resnet50 = model_training_op(
    mlflow_experiment_id=mlflow_experiment_id,
    model_class=Model_Class.ResNet50.name,
    model_params=model_params,
    input=preprocessed_data,
)
train_data_crossval = model_training_op(
    mlflow_experiment_id=mlflow_experiment_id,
    model_class=Model_Class.CrossVal.name,
    model_params=model_params,
    input=preprocessed_data,
)

# Similarly, the operations for compare_models, serve_fastapi_app, and serve_strea
# would be added to the pipeline as well.
```

Finally, the Airflow DAG function is called in a last step.

```python
# Call the airflow DAG
cnn_skin_cancer_workflow()
```

9.2.2 MLflow

Mlflow is leveraged in the preprocessing and model training stages to store crucial data parameters, model training parameters, and metrics, while also enabling the saving of trained models in the model registry. In the `airflow_DAG.py` file, Mlflow is invoked to create an experiment, and the experiment ID is passed to each pipeline step to store parameters in separate runs. This ensures a clear distinction between the execution of different models.

The `train_model` pipeline steps serve as a container for the model training procedure. Within the container, the model is trained using specific code. All the relevant information about the model and the model itself are logged using mlflow as well. This workflow ensures the comprehensive tracking of model parameters and metrics, and the saved model can be accessed and compared during the subsequent model comparison step. In fact, during this stage, the best model is transferred to another model stage within the model registry.

9.3 Building the Pipeline Steps

As mentioned previously, the machine learning pipeline for this particular use case comprises three primary stages: preprocessing, training, and serving. Furthermore, only the model that achieves the highest accuracy is chosen for deployment, which introduces an additional step for model comparison. Each of these steps will be further explained in the upcoming sections.

9.3.1 Data Preprocessing

The data processing stage involves three primary processes. First, the raw data is loaded from an S3 Bucket. Second, the data is preprocessed and converted into the required format. Finally, the preprocessed data is stored in a way that allows it to be utilized by subsequent models. The data processing functionality is implemented within the given `data_preprocessing` function. The `utils` module, imported at the beginning, provides the functionality to access, load, and store data from S3. The data is normalized and transformed into a NumPy array to make it compatible with TensorFlow Keras models. The function returns the names and paths of the preprocessed and uploaded data, making it convenient for selecting them for

future model training. Moreover, the data preprocessing stage establishes a connection with MLflow to record the sizes of the datasets.

Importing Required Libraries

The following code imports the necessary libraries and modules required for the code execution. It includes libraries for handling file operations, data manipulation, machine learning, progress tracking, as well as custom modules.

```python
# Imports necessary packages
import os
from datetime import datetime
from typing import Tuple

import mlflow
import numpy as np
from keras.utils.np_utils import to_categorical
from sklearn.utils import shuffle
from src.utils import AWSSession, timeit
from tqdm import tqdm

# Import custom modules
from src.utils import AWSSession
```

Data Preprocessing Function Definition

At first, the `data_preprocessing` function is defined, which performs the data preprocessing steps. The function takes three arguments: `mlflow_experiment_id` (the MLflow experiment ID for logging), aws_bucket (the S3 bucket for reading raw data and storing preprocessed data), and path_preprocessed (the subdirectory for storing preprocessed data, with a default value of "preprocessed"). The function returns a tuple of four strings representing the paths of the preprocessed data.

```python
@timeit
def data_preprocessing(
    mlflow_experiment_id: str,
    aws_bucket: str,
    path_preprocessed: str = "preprocessed",
) -> Tuple[str, str, str, str]:
```

```
    """Preprocesses data for further use within model training. Raw data is read from given S

    Args:
        mlflow_experiment_id (str): Experiment ID of the MLflow run to log data
        aws_bucket (str): S3 Bucket to read raw data from and write preprocessed data
        path_preprocessed (str, optional): Subdirectory to store the preprocessed data on the

    Returns:
        Tuple[str, str, str, str]: Four strings denoting the path of the preprocessed data st
    """
```

Setting MLflow Tracking URI and AWS Session

Afterward, the MLflow tracking URI is set and an AWS session created using the AWS Access
Key obtained from the environment variables and using the custom class `AWSSession()`.

```
    mlflow_tracking_uri = os.getenv("MLFLOW_TRACKING_URI")
    mlflow.set_tracking_uri(mlflow_tracking_uri)

    # Instantiate aws session based on AWS Access Key
    # AWS Access Key is fetched within AWS Session by os.getenv
    aws_session = AWSSession()
    aws_session.set_sessions()
```

Setting Paths and Helper Functions

The paths for storing raw and preprocessed data within the S3 bucket are defined in a
next step. As well as the helper functions `_load_and_convert_images`, `_create_label` and
`_merge_data`. The `_load_and_convert_images` function loads and converts images from an
S3 bucket folder into a NumPy array. The `_create_label` function creates a label array for
a given dataset, while the `_merge_data` function merges two datasets into a single dataset.

```
    # Set paths within s3
    path_raw_data = f"s3://{aws_bucket}/data/"

    folder_benign_train = f"{path_raw_data}train/benign"
    folder_malignant_train = f"{path_raw_data}train/malignant"
```

```python
    folder_benign_test = f"{path_raw_data}test/benign"
    folder_malignant_test = f"{path_raw_data}test/malignant"

    # Inner helper functions to load the data to a NumPy Array, create labels, and mer
    @timeit
    def _load_and_convert_images(folder_path: str) -> np.array:
        ims = [
            aws_session.read_image_from_s3(s3_bucket=aws_bucket, imname=filename)
            for filename in tqdm(aws_session.list_files_in_bucket(folder_path))
        ]
        return np.array(ims, dtype="uint8")

    def _create_label(x_dataset: np.array) -> np.array:
        return np.zeros(x_dataset.shape[0])

    def _merge_data(set_one: np.array, set_two: np.array) -> np.array:
        return np.concatenate((set_one, set_two), axis=0)
```

Preprocessing Steps and MLflow Logging

This section performs the main preprocessing steps. It loads images from the S3 bucket,
creates labels, merges data, shuffles the data, performs data normalization, and uploads the
preprocessed data as NumPy arrays to the S3 bucket. The MLflow logging is also performed,
recording the sizes of the training and testing data.

```python
    # Start a MLflow run to log the size of the data
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    with mlflow.start_run(experiment_id=mlflow_experiment_id, run_name=f"{timestamp}_P
        print("\n> Loading images from S3...")
        # Load in training pictures
        X_benign = _load_and_convert_images(folder_benign_train)
        X_malignant = _load_and_convert_images(folder_malignant_train)

        # Load in testing pictures
        X_benign_test = _load_and_convert_images(folder_benign_test)
        X_malignant_test = _load_and_convert_images(folder_malignant_test)

        # Log train-test size in MLflow
        print("\n> Log data parameters")
```

```python
        mlflow.log_param("train_size_benign", X_benign.shape[0])
        mlflow.log_param("train_size_malignant", X_malignant.shape[0])
        mlflow.log_param("test_size_benign", X_benign_test.shape[0])
        mlflow.log_param("test_size_malignant", X_malignant_test.shape[0])

        print("\n> Preprocessing...")
        # Create labels
        y_benign = _create_label(X_benign)
        y_malignant = _create_label(X_malignant)

        y_benign_test = _create_label(X_benign_test)
        y_malignant_test = _create_label(X_malignant_test)

        # Merge data
        y_train = _merge_data(y_benign, y_malignant)
        y_test = _merge_data(y_benign_test, y_malignant_test)

        X_train = _merge_data(X_benign, X_malignant)
        X_test = _merge_data(X_benign_test, X_malignant_test)

        # Shuffle data
        X_train, y_train = shuffle(X_train, y_train)
        X_test, y_test = shuffle(X_test, y_test)

        y_train = to_categorical(y_train, num_classes=2)
        y_test = to_categorical(y_test, num_classes=2)

        # With data augmentation to prevent overfitting
        X_train = X_train / 255.0
        X_test = X_test / 255.0
```

Uploading preprocessed data

The four preprocessed numpy arrays (X_train, y_train, X_test, y_test) are uploaded to an S3 bucket. The arrays are stored as pickle files with specific file keys in the bucket. Finally, the paths of the preprocessed data are create and and returned as a tuple of strings.

```python
    print("\n> Upload numpy arrays to S3...")
    aws_session.upload_npy_to_s3(
        data=X_train,
        s3_bucket=aws_bucket,
        file_key=f"{path_preprocessed}/X_train.pkl",
    )
    aws_session.upload_npy_to_s3(
        data=y_train,
        s3_bucket=aws_bucket,
        file_key=f"{path_preprocessed}/y_train.pkl",
    )
    aws_session.upload_npy_to_s3(
        data=X_test,
        s3_bucket=aws_bucket,
        file_key=f"{path_preprocessed}/X_test.pkl",
    )
    aws_session.upload_npy_to_s3(
        data=y_test,
        s3_bucket=aws_bucket,
        file_key=f"{path_preprocessed}/y_test.pkl",
    )

    X_train_data_path = f"{path_preprocessed}/X_train.pkl"
    y_train_data_path = f"{path_preprocessed}/y_train.pkl"
    X_test_data_path = f"{path_preprocessed}/X_test.pkl"
    y_test_data_path = f"{path_preprocessed}/y_test.pkl"

    # Return directory paths of the data stored in S3
    return X_train_data_path, y_train_data_path, X_test_data_path, y_test_data_path
```

9.3.2 Model Training

The training step is designed to accommodate different models based on the selected model. The custom `model.utils` package, imported at the beginning, enables the selection and retrieval of models. The chosen model can be specified by passing its name to the `get_model` function, which then returns the corresponding model. These models are implemented using TensorFlow Keras and their code is stored in the `/model` directory. The model is trained using the `model_params` parameters provided to the training function, which include all the

necessary hyperparameters. The training and evaluation are conducted using the preprocessed data from the previous step, which is downloaded from S3 at the beginning. Depending on the selected model, a KFold cross-validation is performed to improve the model's fit.

MLflow is utilized to track the model's progress. By invoking `mlflow.start_run()`, a new MLflow run is initiated. The `model_params` are logged using `mlflow.log_params`, and MLflow autolog is enabled for Keras models through `mlflow.keras.autolog()`. After successful training, the models are stored in the model registry. The trained model is logged using `mlflow.keras.register_model`, with the specified `model_name` as the destination.

The function returns the MLflow run ID and crucial information about the model, such as its name, version, and stage.

Importing Dependencies

This section imports the necessary dependencies for the code, including libraries for machine learning, data manipulation, and utility functions.

```python
# Imports necessary packages
import json
import os
from datetime import datetime
from enum import Enum
from typing import Tuple

import mlflow
import mlflow.keras
import numpy as np
from keras import backend as K
from keras.callbacks import ReduceLROnPlateau
from sklearn.metrics import accuracy_score
from sklearn.model_selection import KFold

# Import custom modules
from src.model.utils import Model_Class, get_model
from src.utils import AWSSession
```

Defining the *train_model* Function

The actual code starts by defining the train_model function, which takes several parameters
for training a machine learning model, logging the results to MLflow, and returning relevant
information. The MLflow tracking URI is retrieved from the environment variable and sets
it as the tracking URI for MLflow.

```python
def train_model(
    mlflow_experiment_id: str,
    model_class: Enum,
    model_params: dict,
    aws_bucket: str,
    import_dict: dict = {},
) -> Tuple[str, str, int, str]:
    """
    Trains a machine learning model and logs the results to MLflow.

    Args:
        mlflow_experiment_id (str): The ID of the MLflow experiment to log the results
        model_class (Enum): The class of the model to train.
        model_params (dict): A dictionary containing the parameters for the model.
        aws_bucket (str): The AWS S3 bucket name for data storage.
        import_dict (dict, optional): A dictionary containing paths for importing data

    Returns:
        Tuple[str, str, int, str]: A tuple containing the run ID, model name, model ve

    Raises:
        None
    """
    mlflow_tracking_uri = os.getenv("MLFLOW_TRACKING_URI")
    mlflow.set_tracking_uri(mlflow_tracking_uri)
```

Loading Data

This section handles the loading of data required for training the model. It retrieves the
file paths for the training and testing data from the `import_dict` parameter and loads the
corresponding NumPy arrays from an AWS S3 bucket using the `AWSSession` class.

```python
print("\n> Loading data...")
X_train_data_path = import_dict.get("X_train_data_path")
y_train_data_path = import_dict.get("y_train_data_path")
X_test_data_path = import_dict.get("X_test_data_path")
y_test_data_path = import_dict.get("y_test_data_path")

# Instantiate aws session based on AWS Access Key
# AWS Access Key is fetched within AWS Session by os.getenv
aws_session = AWSSession()
aws_session.set_sessions()

# Read NumPy Arrays from S3
X_train = aws_session.download_npy_from_s3(s3_bucket=aws_bucket, file_key=X_train_data_pat
y_train = aws_session.download_npy_from_s3(s3_bucket=aws_bucket, file_key=y_train_data_pat
X_test = aws_session.download_npy_from_s3(s3_bucket=aws_bucket, file_key=X_test_data_path]
y_test = aws_session.download_npy_from_s3(s3_bucket=aws_bucket, file_key=y_test_data_path]
```

Training the Model

After the data is loaded, the training process for the machine learning model is started. It begins by printing the model class and generating a timestamp for the run name. Then, it starts an MLflow run with the specified experiment ID and run name. The model parameters are logged using MLflow's log_params function. Additionally, a callback for reducing the learning rate during training is configured using the ReduceLROnPlateau class from Keras.

The model training handles two different scenarios based on the selected `model_class`. If it is set to cross-validation (`Model_Class.CrossVal`), the model is trained using cross-validation. Otherwise, it is trained using the specified model class.

```python
print("\n> Training model...")
print(model_class)
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
with mlflow.start_run(experiment_id=mlflow_experiment_id, run_name=f"{timestamp}-{model_cl
    mlflow.log_params(model_params)
    learning_rate_reduction = ReduceLROnPlateau(monitor="accuracy", patience=5, verbose=1

    # If CrossVal is selected, train BasicNet as Cross-Validated Model
    if model_class == Model_Class.CrossVal.value:
        kfold = KFold(n_splits=3, shuffle=True, random_state=11)
```

```python
        cvscores = []
        for train, test in kfold.split(X_train, y_train):
            model = get_model(Model_Class.Basic.value, model_params)

            # Train Model
            model.fit(
                X_train[train],
                y_train[train],
                epochs=model_params.get("epochs"),
                batch_size=model_params.get("batch_size"),
                verbose=model_params.get("verbose"),
            )
            scores = model.evaluate(X_train[test], y_train[test], verbose=0)
            print("%s: %.2f%%" % (model.metrics_names[1], scores[1] * 100))
            cvscores.append(scores[1] * 100)
            K.clear_session()
    else:
        model = get_model(model_class, model_params)
        mlflow.keras.autolog()

        # Train Model
        model.fit(
            X_train,
            y_train,
            validation_split=model_params.get("validation_split"),
            epochs=model_params.get("epochs"),
            batch_size=model_params.get("batch_size"),
            verbose=model_params.get("verbose"),
            callbacks=[learning_rate_reduction],
        )
        mlflow.keras.autolog(disable=True)
```

Testing and Evaluating the Model

After the model training, the trained model is tested on the test data and its prediction accuracy evaluated. The accuracy score is calculated using the `accuracy_score` function from scikit-learn and logged as a metric using MLflow. Afterward, the trained and evaluated model is registered with MLflow using the `register_model` function. The resulting model name,

version, and stage are obtained to finally return them in the functions `return` statement.

```python
        run_id = run.info.run_id
        model_uri = f"runs:/{run_id}/{model_class}"

        # Testing model on test data to evaluate
        print("\n> Testing model...")
        y_pred = model.predict(X_test)
        prediction_accuracy = accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_pred, axi
        mlflow.log_metric("prediction_accuracy", prediction_accuracy)
        print(f"Prediction Accuracy: {prediction_accuracy}")

        print("\n> Register model...")
        mv = mlflow.register_model(model_uri, model_class)

    # Return run ID, model name, model version, and current stage of the model
    return run_id, mv.name, mv.version, mv.current_stage
```

9.4 Model Serving & Inferencing

The process of serving and making inferences utilizes Docker containers and runs them within
Kubernetes pods.

The concept involves running a Docker container that serves the pre-trained TensorFlow
model using FastAPI. This containerized model is responsible for providing predictions and
responses to incoming requests. Additionally, a Streamlit app is used to interact with the
served model, enabling users to make inferences by sending input data to the model and
receiving the corresponding predictions.

9.4.1 Model Serving

The model serving application is built on FastAPI, which includes various endpoints catering
to our use case. The primary endpoint, `predict`, allows for multiple predictions to be made,
while additional maintenance endpoints such as `info` or `health` provide relevant information
about the app itself.

To initiate the prediction process, the model is first retrieved from the MLflow registry. The
specific model to be fetched and the location of the MLflow server are specified through

environment variables. Once the model is loaded, it is used to generate predictions based on the provided input data. The API call returns the predictions in the form of a Python list.

Importing Dependencies

```python
# Imports necessary packages
import io
import os
from io import BytesIO

import mlflow
import mlflow.keras
import numpy as np
import pandas as pd
from fastapi import FastAPI, File, HTTPException, UploadFile
from fastapi.encoders import jsonable_encoder
from fastapi.responses import JSONResponse
from PIL import Image
from tensorflow import keras
```

Creating the FastAPI Instance

An instance of the FastAPI application is created as well as its name and version defined.

```python
# Create FastAPI instance
app = FastAPI()

model_name = "Skin Cancer Detection"
version = "v1.0.0"
```

Defining API Endpoints

This section defines three API endpoints: /info, /health, and /predict. The /info endpoint returns information about the model, including its name and version. The /health endpoint returns the health status of the service.

```python
@app.get("/info")
async def model_info():
    """
    Endpoint to retrieve information about the model.

    Returns:
        - Dictionary containing the model name and version
    """
    return {"name": model_name, "version": version}


@app.get("/health")
async def service_health():
    """
    Endpoint to check the health status of the service.

    Returns:
        - Dictionary indicating the health status of the service
    """
    return {"ok"}
```

The `/predict` endpoint is used for making predictions and accepts an uploaded file. Within the predict endpoint, two helper functions for image preprocessing are defined. `_read_imagefile` reads the image file from the provided data and returns it as a PIL Image object. `_preprocess_image` performs normalization and reshaping on the image data to prepare it for the model.

```python
@app.post("/predict")
async def predict(file: UploadFile = File(...)):
    """
    Endpoint to make predictions on skin cancer images.

    Parameters:
        - file: Uploaded image file (JPG format)

    Returns:
        - Prediction results as a JSON object
    """
    # Get environment variables
    MLFLOW_TRACKING_URI = os.getenv("MLFLOW_TRACKING_URI")
```

```python
MLFLOW_MODEL_NAME = os.getenv("MLFLOW_MODEL_NAME")
MLFLOW_MODEL_VERSION = os.getenv("MLFLOW_MODEL_VERSION")

def _read_imagefile(data) -> Image.Image:
    """
    Read image file from bytes data.

    Parameters:
        - data: Bytes data of the image file

    Returns:
        - PIL Image object
    """
    image = Image.open(BytesIO(data))
    return image

def _preprocess_image(image) -> np.array:
    """
    Preprocess the input image for model prediction.

    Parameters:
        - image: PIL Image object

    Returns:
        - Processed numpy array image
    """
    np_image = np.array(image, dtype="uint8")
    np_image = np_image / 255.0
    np_image = np_image.reshape(1, 224, 224, 3)
    return np_image
```

The following code snippoet contains the actual logic for the `/predict` endpoint. It checks if the uploaded file has a `.jpg` extension. If it does, it reads the image, loads the MLflow model, preprocesses the image, performs the prediction using the model, and returns the predictions as a JSON response. If the file format is not `.jpg`, it raises a `HTTPException` with a status code of 400 and an error message indicating the invalid file format.

```python
if file.filename.endswith(".jpg"):
    print("[+] Read File")
```

```
        image = _read_imagefile(await file.read())

        print("[+] Initialize MLflow")
        mlflow.set_tracking_uri(MLFLOW_TRACKING_URI)

        print("[+] Load Model")
        model = mlflow.keras.load_model(f"models:/{MLFLOW_MODEL_NAME}/{MLFLOW_MODEL_VERSION}"
        
        print("[+] Preprocess Data")
        np_image = _preprocess_image(image)

        print("[+] Initiate Prediction")
        preds = model.predict(np_image)

        print("[+] Return Model Prediction")
        return {"prediction": preds.tolist()}
    else:
        # Raise a HTTP 400 Exception, indicating Bad Request
        raise HTTPException(status_code=400, detail="Invalid file format. Only JPG Files acce
```

9.4.2 Streamlit App

The Streamlit app offers a simple interface for performing inferences on the served model.
The user interface enables users to upload a `jpg` image. Upon clicking the `predict` button,
the image is sent to the model serving app, where a prediction is made. The prediction results
are then returned as a JSON file, which can be downloaded upon request.

Importing Dependencies This section imports the necessary dependencies for the code,
including libraries for file handling, JSON processing, working with images, making HTTP
requests, and creating the Streamlit application.

```
# Imports necessary packages
import io
import json
import os

import pandas as pd
import requests
```

```python
import streamlit as st
from PIL import Image
```

Setting Up the Streamlit Application

At first, the header and subheader for the Streamlit application are set. Afterward, the FastAPI serving IP and port are retrieved from environment variables. They constructs the FastAPI endpoint URL and are later used to send a POST request to.

```python
st.header("MLOps Engineering Project")
st.subheader("Skin Cancer Detection")

# FastAPI endpoint
FASTAPI_SERVING_IP = os.getenv("FASTAPI_SERVING_IP")
FASTAPI_SERVING_PORT = os.getenv("FASTAPI_SERVING_PORT")
FASTAPI_ENDPOINT = f"http://{FASTAPI_SERVING_IP}:{FASTAPI_SERVING_PORT}/predict"
```

Uploading test image

The `st.file_uploader` allows the user to upload a test image in JPG format using the Streamlit file uploader widget. The type of the uploaded file is limited to `.jpg`. If a test image has been uploaded, the image is processed by opening it with PIL and creating a file-like object.

```python
test_image = st.file_uploader("", type=["jpg"], accept_multiple_files=False)

if test_image:
    image = Image.open(test_image)
    image_file = io.BytesIO(test_image.getvalue())
    files = {"file": image_file}
```

Displaying the uploaded image and performing prediction

A two-column layout in the Streamlit app is created That displays the uploaded image in the first column. In the second columns, a button for the user to start the prediction process is displayed. When the button is clicked, it sends a POST request to the FastAPI endpoint with the uploaded image file. The prediction results are displayed as JSON and can be downloaded as a JSON file.

```python
    col1, col2 = st.columns(2)

    with col1:
        # Display the uploaded image in the first column
        st.image(test_image, caption="", use_column_width="always")

    with col2:
        if st.button("Start Prediction"):
            with st.spinner("Prediction in Progress. Please Wait..."):
                # Send a POST request to FastAPI for prediction
                output = requests.post(FASTAPI_ENDPOINT, files=files, timeout=8000)
            st.success("Success! Click the Download button below to retrieve prediction resul
            # Display the prediction results in JSON format
            st.json(output.json())
            # Add a download button to download the prediction results as a JSON file
            st.download_button(
                label="Download",
                data=json.dumps(output.json()),
                file_name="cnn_skin_cancer_prediction_results.json",
            )
```

10 Acknowledgements

I would like to express my gratitude to the everyone contributing to this project and everyone who has provided invaluable insights and support throughout the development of this bookdown project:

I am grateful to everyone who have contributed their time and expertise in reviewing drafts and providing valuable feedback. Your input has undoubtedly shaped the final outcome of this bookdown, and I am deeply appreciative of your efforts.