Mastering MLOps Architecture

From Code to Deployment

Manage the production cycle of continual learning ML models with MLOps

Raman Jhajj

bpb

Mastering MLOps Architecture

From Code to Deployment

Manage the production cycle of continual learning ML models with MLOps

Raman Jhajj

bpb

Mastering MLOps Architecture: From Code to Deployment

Manage the production cycle of continual learning ML models with MLOps

Raman Jhajj

www.bpbonline.com

First published: 2024

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55519-498

www.bpbonline.com

Dedicated to

My family, that gave me the gift of dreams

and

Friends, who became family.

About the Author

Raman Jhajj is a passionate leader in the data and software engineering space with experience building high-performing teams and leading organizations to become data-driven. He has experience in leading the development of SaaS applications, modern data platforms and MLOps infrastructure. He brings technical expertise across the data stack including AWS, Python, Django, Java, PostgreSQL, Hadoop, Spark, Kafka, Docker, CI/CD, SQL, NoSQL, and more.

Raman holds a master's degree in applied computer science from Georg-August University, Germany as well as a bachelor's in computer science from ICFAI University, India. After living in India, Germany, Austria, and Malta, he now calls Canada home.

Over the course of his career, Raman has driven key initiatives around modernizing data infrastructure, establishing data engineering capabilities, and building MLOps platforms.

Raman thrives on bringing cross-functional teams together to ensure alignment between technology and business goals. He has a proven track record of mentoring engineers and nurturing their potential.

When he is not working, you can often find him reading, writing, or exploring new places and cultures. He is passionate about using technology for social good, driven by a mission to leverage data engineering and AI for positive change.

About the Reviewer

Ashish Patel, an accomplished author, data scientist and researcher with over 11 years of experience. He is a luminary in predictive modeling, data preprocessing, feature engineering, machine learning, and deep learning. Notably, Ashish has taken center stage as a keynote speaker at prestigious events like AWS Community Day, AWS AI ML Days, Faculty Development Programs (FDPs), and IIT Techfest, captivating audiences with his insights. Currently serving as the Sr. AWS AI ML Solution Architect at IBM India Pvt Ltd, he architects innovation by collaborating with IBM and AWS specialists to craft enterprise solutions on Red Hat OpenShift, AWS Infrastructure, and IBM Software technology, aligning seamlessly with the AWS Well-Architected Framework. Ashish is a five-time LinkedIn Top Voice and an AI Research Scientist, with expertise spanning MLOps and a multitude of LLMs and FM Models. Recognized on LinkedIn for his contributions in Statistics, Data Science, Data Analytics, AI, and Machine Learning, Ashish is also a GitHub sensation with over 5k+ followers, marking his profound impact in open-source communities. In the realm where data reigns supreme, Ashish Patel crafts, speaks, and influences the future. He a Quantum Machine Learning practitioner and researcher working with international research community.

Acknowledgement

Writing a book is harder than I thought and more rewarding than I could have ever imagined. None of this would have been possible without the support of my family and friends, whom I would like to acknowledge and thank.

I would like to start by thanking my awesome wife, Simran for being the constant support from those late-night writing sessions and frustration-filled days to my ramblings of how hard it is to put thoughts on paper.

I want to thank my parents - Dad for constantly guiding and showing me that writing a book is an achievable target and Mom for her unwavering belief in me.

I thank Kanwar, Kuljeet and Garima for their constant support throughout the ups and downs of life and for always being there for me. I thank Kaisha for those video calls and for filling the days with laughter.

To all those friends who have been a part of my getting here: Parminder, Kiran, Anmol, Harman, Jagvir and Sukhpreet, I thank you for your heartfelt support and ready smiles, shared meals, advice, perspectives, and friendships. I thank Baani and Ravtaj for the playtime and for reminding me of what it is like to be a child again.

To my mentors throughout this journey: Malaika, Dean Chen, Michiah, and Tovah, I thank you for being the leaders I trust, honour, and respect.

To everyone at BPB Publications who enabled me to write this book. Thank you for the guidance and expertise in bringing this book to fruition. It was a long journey of revising this book, with valuable participation and collaboration of reviewers, technical experts, and editors.

I would also like to acknowledge the valuable contributions of my colleagues and co-workers who have taught me so much and provided

valuable feedback on my work, during many years working in the tech industry.

Finally, I want to thank you, my cherished readers, for taking an interest in my book. To have it received by you is an unexpected gift that keeps me grounded in the moment.

Preface

MLOps is the intersection of DevOps, data engineering and machine learning. Working in the field of machine learning is highly dependent on ever-changing data, whereas MLOps is needed to deliver excellent ML and AI results. This book provides a practical guide to MLOps for data scientists, data engineers, and other professionals involved in building and deploying machine learning systems. It introduces MLOps, explaining its core concepts like continuous integration and delivery for machine learning. It outlines MLOps components and architecture, providing an understanding of how MLOps supports robust ML systems that continuously improve.

By covering the end-to-end machine learning pipeline from data to deployment, the book helps readers implement MLOps workflows. It discusses techniques like feature engineering, model development, A/B testing, and canary deployments.

The book equips readers with knowledge of MLOps tools and infrastructure for tasks like model tracking, model governance, metadata management, and pipeline orchestration. Monitoring and maintenance processes to detect model degradation are covered in depth. With its comprehensive coverage and practical focus, this book enables data scientists, data engineers, DevOps engineers, and technical leaders to effectively leverage MLOps. Readers can gain skills to build efficient CI/CD pipelines, deploy models faster, and make their ML systems more reliable and production-ready.

Overall, the book is an indispensable guide to MLOps and its applications for delivering business value through continuous machine learning and AI.

Chapter 1: Getting Started with MLOps - This chapter introduces MLOps, explaining how it combines machine learning, DevOps, and data engineering to enable continuous delivery of ML models. It covers the importance of MLOps, its principles like reproducibility and auditability,

best practices, and strategies for implementation. The difference between MLOps and the traditional software engineering and the unique challenges of productionizing machine learning are also discussed. The chapter provides a foundation for understanding the MLOps methodology.

Chapter 2: MLOps Architecture and Components - This chapter covers the architecture and components of MLOps systems. It discusses the building blocks like data pipelines, model training, deployment, monitoring, and orchestration. The chapter outlines reference architectures for different maturity levels, from basic to enterprise-grade. It explains environment semantics and model deployment patterns. Finally, it walks through an end-to-end workflow integrating all components across development, staging, and production environments. The goal is to provide a foundation for designing and implementing MLOps solutions suitable for various use cases.

Chapter 3: MLOps Infrastructure and Tools - This chapter explores the infrastructure and tools needed for MLOps. It covers key components like storage, compute, containers, orchestration platforms, and ML platforms for deployment, model registries, and feature stores. The chapter discusses public cloud versus on-premises options, standardized development environments, and build versus buy decisions. It aims to provide guidance on setting up a robust, scalable infrastructure tailored to an organization's specific use cases and resources.

Chapter 4: What are Machine Learning Systems? - This chapter explains what machine learning systems are and how they differ from ML research. It covers an implementation roadmap with phases for initial development, transition to operations, and ongoing operations. The chapter discusses using standardized project structures like cookiecutter data science to facilitate eventual productionization. It aims to provide a foundation for taking a full systems approach to developing real-world ML applications, not just algorithms. The goal is to equip readers with an understanding of all components needed to build successful ML systems.

Chapter 5: Data Preparation and Model Development - This chapter covers data preparation and model development within the MLOps lifecycle. It discusses best practices for version control, preparing data, performing exploratory analysis, feature engineering, training models, and

tracking experiments with MLflow. The chapter shows how these steps fit into a standardized project structure to enable collaboration and reproducibility. It aims to provide guidance on implementing key phases of the machine learning lifecycle in a way that facilitates eventual operationalization and automation.

Chapter 6: Model Deployment and Serving - This chapter covers model deployment and serving in the MLOps lifecycle. It explores strategies like static, dynamic, and streaming deployment, comparing deployment on devices versus servers using VMs, containers, or serverless technologies. The chapter discusses inference options like batch processing versus real-time APIs. It also looks at deployment patterns like canary releases and multi-armed bandits for controlled model rollout.

Chapter 7: Continuous Delivery of Machine Learning Models - This chapter explores methods for implementing continuous integration, continuous training, and continuous delivery in machine learning systems. It examines ML/AI pipelines and architectural maturity levels. Key topics include continuous integration tools like GitHub Actions, strategies for determining when and what to retrain models on, and considerations for rapidly deploying updated models into production through continuous delivery.

Chapter 8: Continual Learning - This chapter explores continual learning in machine learning systems, which involves models perpetually learning and adapting to new data without forgetting past knowledge. It covers principles like stateful training, challenges around obtaining fresh data and evaluating updates, and implementing continual learning in MLOps through triggers and robust monitoring. The goal is to enable frequent automated model updates while maintaining safety, transparency and control.

Chapter 9: Continuous Monitoring, Logging, and Maintenance - This chapter covers principles and best practices for monitoring machine learning models across environments. It examines why continuous monitoring matters, integrating it into MLOps workflows, logging model metadata and performance data, using frameworks like Evidently and Alibi Detect, and evaluating models with techniques like A/B testing.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/mn9abap

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Mastering-MLOps-Architecture-From-Code-to-Deployment**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

CHAPTER 1
Getting Started with MLOps

Introduction

Being an emerging field, **Machine Learning Operations** (**MLOps**) is rapidly gaining momentum with data scientists, **Machine Learning** (**ML**) engineers, and **Artificial Intelligence** (**AI**) enthusiasts. In this chapter, we will go over the premise and background of the MLOps ecosystem. We will try to understand what it is, why it is useful, and what the principles and best practices are when it comes to MLOps. We will also go over what are the pillars of a successful MLOps strategy and how MLOps fits with the ROI requirements of a business.

When looking at MLOps, we can easily relate it to DevOps. DevOps did to software engineering what MLOps is aiming to do to machine learning engineering. DevOps is a culture, philosophy, and set of practices that seek to break down the barriers between development and operations teams, improve collaboration, and deliver software continuously and reliably. It involves the use of various tools and techniques for developing, testing, deploying, monitoring, and operating software engineering systems. DevOps was able to achieve the following for software engineering:

- Shorter development cycles
- Increased deployment velocity
- Automated testing before each deployment

- Auditable system releases
- Continuous monitoring of the system for stability and scalability

This brings us to MLOps. It is similar to DevOps, but with a focus on the unique requirements of machine learning and data-specific workflows. It involves the use of practices and tools for developing, testing, deploying, monitoring, and operating machine learning systems, while incorporating many of the same principles and practices of DevOps. No single solution is going to either make or break a plan. Instead, it is essential to understand the unique requirements of what frameworks might fit into your workflow and have a comprehensive strategy to implement that. In this chapter and throughout the book, we will learn how that is achieved. Next, we will discuss the principles and fundamentals of MLOps and how to use them effectively to get models into production successfully.

Structure

In this chapter, we will discuss the following topics:

- Understanding MLOps
- Importance of MLOps
- The evolution of MLOps
- Software engineering projects versus machine learning projects
- DevOps versus MLOps
- Principles of MLOps
- MLOps best practices
- MLOps in an organization
- MLOps strategy
- Implementing MLOps
- Overcoming challenges of MLOps

Objectives

By the end of this chapter, you will have a solid understanding of MLOps and the reason behind its hype. We will also learn about the fundamental principles and best practices of MLOps, including reproducibility, transparency, auditability, and scalability.

You will understand the difference between software engineering projects and machine learning projects and how that impacts the need for MLOps versus traditional DevOps. We will also cover the evolution of MLOps over time.

We will discuss the role of MLOps in an organization and why having a good MLOps strategy matters for successful implementation, and how organizations can unlock business value from MLOps while overcoming inherent challenges in the machine learning system implementation.

The chapter will also provide an overview of implementing MLOps in different environments and how vendors and open-source solutions can accelerate implementation.

Understanding MLOps

MLOps is a set of practices designed for collaboration between data scientists, machine learning engineers, data engineers, and operations professionals. MLOps is the answer to the questions:

- Why is machine learning deployment not quick?
- How can we quickly productionize our machine learning models?
- Why can machine learning model deployment be ten times faster?

MLOps is a combination of **Machine Learning** (**ML**) and **Operations** (**Ops**). It refers to the processes and practices for designing, building, enabling, and supporting the efficient deployment of ML models in production and continuously iterating and improving upon these models.

Similar to DevOps, MLOps is heavily dependent on automation and integrations. MLOps aims to standardize the deployment and management of ML models alongside the operationalization of the ML pipeline. It supports the release, activation, monitoring, performance tracking, management, reuse, maintenance, and governance of ML artifacts.

Following and applying this set of practices simplifies the management of models and artifacts, automates the deployment of machine learning models, allows us to maintain data and artifact lineage, and improves the quality and speed of deployment. Implementation of these practices makes it easier to iterate over model development quickly and better align models with business needs/requirements.

MLOps combines and is at the intersection of **Machine Learning**, **DevOps**, and **Data Engineering**, as shown in *Figure 1.1*, with the goal of reliably and efficiently building, deploying, and maintaining ML systems in production. It is at the intersection of **DevOps**, **Data Engineering**, and **Machine Learning**. Machine learning projects and overall systems are experimental in nature. It consists of components that are comparatively more complex to build and operate than DevOps components. Other than the building and deployment, MLOps also needs to account for new components like data drift, the delta between changes in the data from the last model training and current model training, and so on. Refer to *Figure 1.1:*

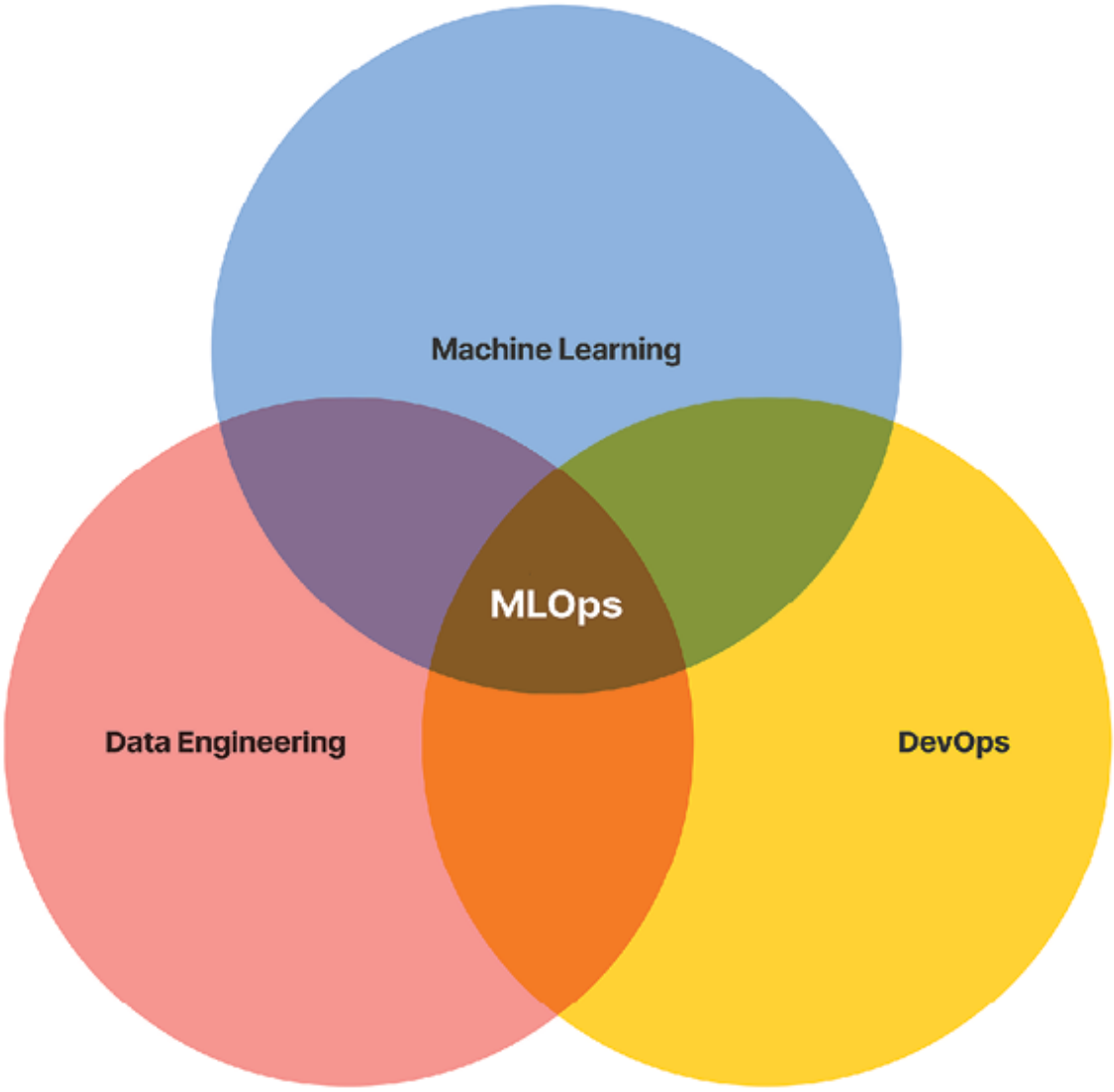

Figure 1.1: *MLOps as an intersection of three domains*

With the base driven by DevOps, MLOps is now slowly evolving into an independent approach to machine learning lifecycle management. It applies to the entire lifecycle and key phases being:

- Data gathering, collecting, and processing raw data
- Data analysis
- Data preparation
- Model training and development
- Model evaluation and validation

- Model serving
- Model health monitoring
- Model re-training and iterations
- Orchestration
- Governance

These key phases indicate how work-intensive the entire process can get, especially since it will most likely need to be repeated multiple times. While it is possibly easier the second time around since we only must update the model on new data patterns and trends, it is still a problem that can take up hours of manual labor. After all, the maintenance of applications in the software development process is usually where most of the money and resources go, not the initial development and release of the application. The same can apply to machine learning models and processes, worsening the overall maintenance costs.

Figure 1.2 shows the relationship between all the key phases of a machine learning pipeline and how these phases fit together to allow us to build a complete pipeline. Refer to the following figure:

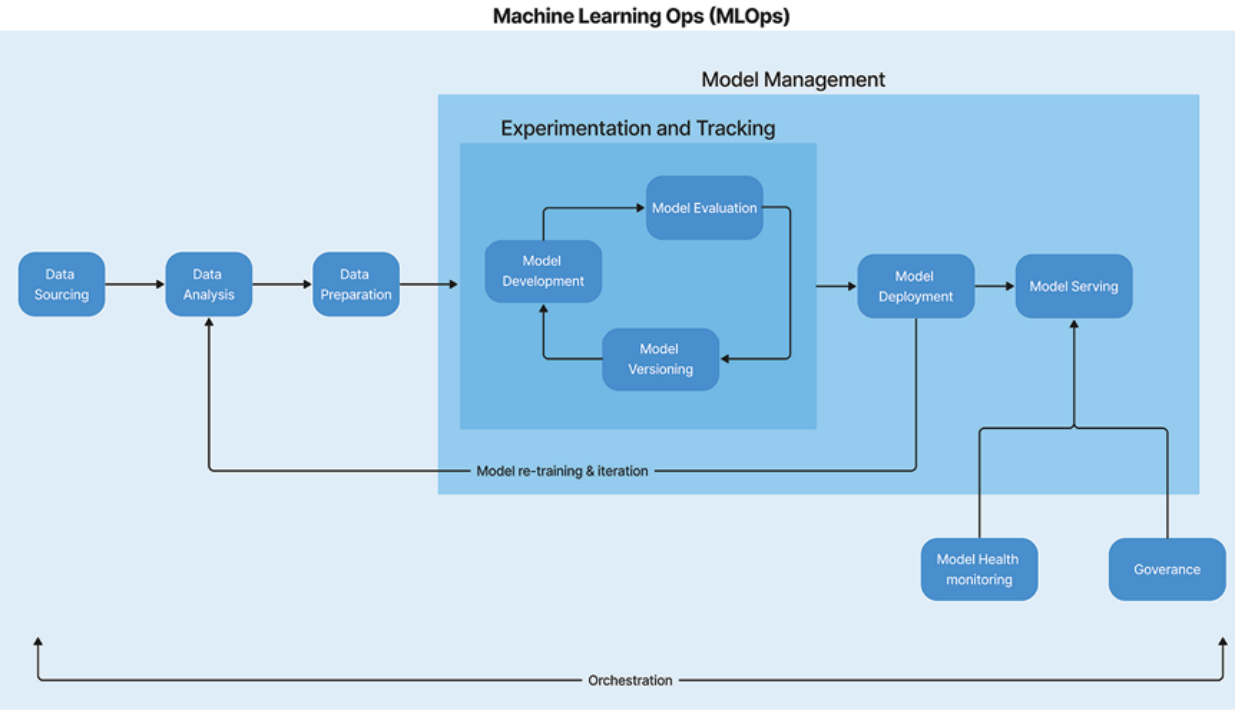

Figure 1.2: *Machine learning project lifecycle*

Imagine if we could simply automate this entire process away, allowing us to take full advantage of high-performance machine learning models without all the hassle. This is where MLOps comes in.

In *Figure 1.2*, you will notice there are two more components that are part of MLOps: **Experimentation and Tracking** and **Model Management**. What are those, and how can we define them?

Experimentation and tracking

Experimentation and tracking are parts of MLOps, which focus on collecting, organizing, and tracking model training information and artifacts across multiple runs, and using multiple configurations. As machine learning is experimental in nature, using experiment-tracking tools to track and benchmark different models and configurations becomes important.

Model management

To ensure that we can maintain consistency while iterating over the model deployments and maintaining lineage, especially at space, an easy-to-follow pattern needs to be in place. This is the part of MLOps when model management comes into the picture. MLOps includes processes for streamlining and optimizing model development, training, packaging, validating, deployment, and monitoring. This way, we can manage ML projects consistently from end to end.

We have briefed the details of MLOps so far. In the next section, we will understand its importance.

Importance of MLOps

As we covered in the last section, traditional DevOps methods or other software engineering management methods are not sufficient for machine learning applications. To understand the reason for this, we will have to analyze what differentiates ML from traditional software applications.

Note: Machine Learning is not just code, like in traditional software development, but code plus data which generates the value of Machine Learning applications.

The data is a fundamental first-class citizen of the development and deployment process with respect to machine learning models. The code enables us to process and evaluate the data to derive insights and business value. As indicated in *Figure 1.3*, we get a good and feasible machine learning model when we integrate data with relevant code. Refer to the following figure:

Figure 1.3: *Components of Machine Learning*

Now, as we understand this relationship between **Code** and **Data** to create **Machine Learning** applications, we must ensure to combine and use the two in unison, so they evolve in a controlled way towards the goal of a robust and scalable machine learning system. Data for training, testing, and inference will change over time across different sources and needs to be met with changing code. Without a systematic MLOps approach, there can be a divergence in how code and data evolve that causes problems in production, gets in the way of smooth deployment, and leads to results that are hard to trace or reproduce.

MLOps streamlines the development, deployment, and monitoring pipeline of ML applications, unifying the contributions from different teams involved and ensuring that all steps of the process are repeatable and reproducible.

MLOps is fundamental to the development and deployment of machine learning and data science applications. Machine learning helps to deploy solutions that unlock previously untapped sources of revenue, save time, and reduce cost by creating more efficient workflows, leveraging data analytics for decision-making, and improving customer experience. These goals are hard to accomplish in the absence of a solid framework to enable this value generation. Automating model training, development, deployment, and monitoring with MLOps means faster iterations and lower operational costs. It helps machine learning and data science teams to be more agile and strategic in their decisions, similar to how software engineering teams are.

MLOps is crucial because it serves as a framework to guide teams for achieving their goals by using the ideal approach for machine learning

development and deployment. MLOps is also a highly flexible framework in the sense that we can decide how big and complex or small and easy you want a set of processes to be. You can decide to go all in and automate every step of MLOps and its practices, or you can decide to keep the set of it manually and automate the others.

The evolution of MLOps

MLOps as a term originated in the early 2010s, as companies began to realize the potential of machine learning. However, the adoption of MLOps has accelerated in recent years as businesses have increasingly relied on machine learning to power their operations. In 2019, *Forrester* predicted that MLOps would become a mainstream practice within two years. And indeed, MLOps is popular among organizations in a variety of fields. Financial services firms use MLOps to automate the deployment of fraud detection and prevention models in real time. Retailers use MLOps to personalize customer experiences and improve product recommendations. Healthcare organizations use MLOps to develop predictive models for various activities from disease diagnosis to patient care. As ML becomes more commonplace, it is likely that MLOps will continue to evolve and grow in popularity and demand. But let us look back and see how and from where MLOps evolved.

Early 2000s was the time when organizations started adopting machine learning into their technology stack and started using it for different use cases. That is where we saw implementations of the early versions of machine learning-based solutions. The only solution available was using **commercial off the shelf** (**COTS**) solutions and vendors like SAS, IBM SPSS, and FICO. Further, with the rise of open-source technologies, availability of large datasets in the public domain, and advancement of programming languages to adapt machine learning-specific libraries such as Python (SciPy stack, scikit-learn, TensorFlow, and so on) or R (dplyr, ggplot2 and so on) among others, the development of ML models became easier. However, the usage of the models in production was still a problem because of all the processes and steps involved, as we saw in the previous section.

With the emergence of the containerization technologies like Docker containers and Kubernetes, the issue with the deployment of the models in a scalable way, was solved. In the next iteration of this problem-solving, we

see ML deployment and management platforms that cover the whole iteration of machine learning deployment, including experimentation, training, deployment, and monitoring. These platforms help in deploying the models on-premise or on the Cloud, making it easier to manage. *Figure 1.4* shows the timeline of the evolution of MLOps from early 2000s starting with the proprietary interfaces till now with dedicated MLOps platforms. Refer the following figure:

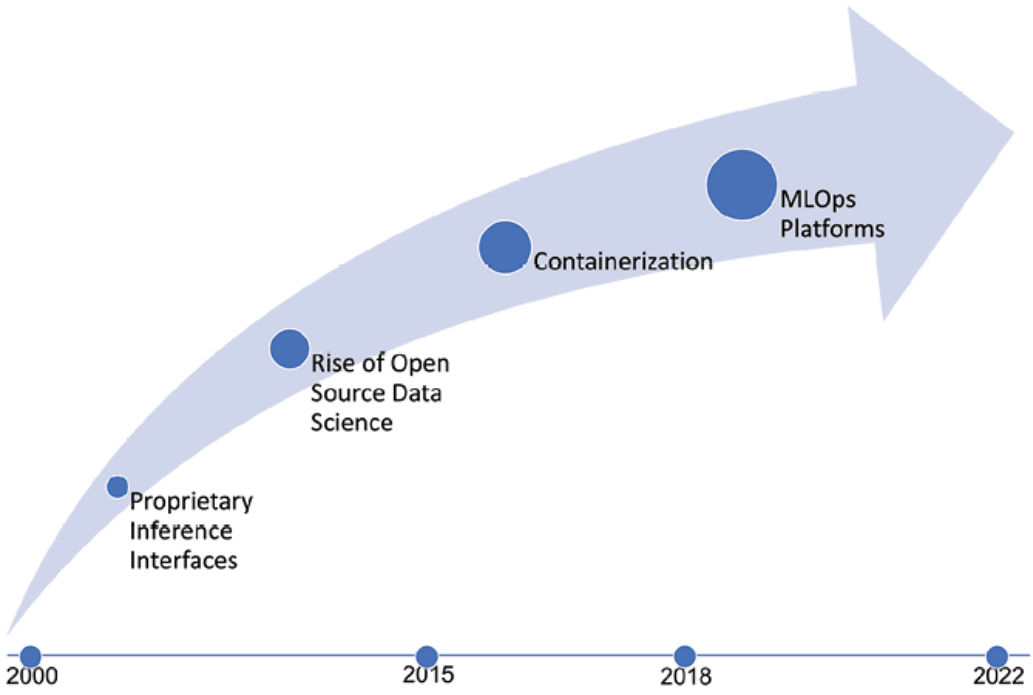

Figure 1.4: *Timeline of the evolution of MLOps*

Software engineering projects versus machine learning projects

To better understand the magnitude and importance of these advancements in MLOps, it is important to learn about the difference between software engineering and machine learning projects. This knowledge will help us understand the limitations of normal DevOps tools in machine learning projects and the reason for the emergence of MLOps.

No doubt, there are many tools in software engineering that could be useful for machine learning projects. The actual code and configurations can be easily managed in a source code management system like **Git**, and all the related concepts, like branches and pull requests, can be used to better

manage the course code, followed by CI/CD systems/platforms, which can be used in automating actual project runs.

But as we learned in the previous sections, ML projects have differences preventing regular software developer tools from serving every need. Some of the differences and requirements are:

- **Dependency on large datasets and trained models**: Machine learning development depends heavily on a large amount of data and datasets for training the model, and the resultant trained model can also be enormous. Normal source code management tools like **Git** et. al. do not handle large files very well on its own. It required extension like **Git Large File Storage** (**Git LFS**) to be able to handle large files more efficiently.
- **Huge resources to train the ML models**: A regular software project organizes the files and compiles them together as a software product; a machine learning project instead trains a model that describes an AI algorithm. In most cases, deploying a software product release is quicker and more straight forward. Therefore, many organizations follow a continuous integration strategy. Training an ML model takes time and dedicated resources, and requires a different strategy for continuous integration.
- **Metrics-driven development versus feature-driven development**: In software engineering release and development decisions are based on whether the team has reached the feature development goals. By contrast, ML development is a metric development such as looking at the predictive value of the machine learning model. ML engineers will iteratively generate multiple models with different parameters and configurations followed by measuring the accuracy of each. Since the goal of the machine learning project is to find the most accurate model, the project is generally guided by metrics achieved in those experiments.
- **Pipelines**: Pipelines are present in software projects as a part of CI/CD pipelines. However, ML projects have a separate set of pipelines which consists of series of steps such as downloading data, preparing

data, separating data into training/validating sets, training a model, and validating the model.

- **Dedicated hardware**: Software projects can be hosted on a variety of software infrastructure and sets of servers ranging from Cloud infrastructure or on-premises. Whereas ML projects have huge computation needs which warrant specific high compute server. Most of the time these high-power servers are GPU intensive which increases the ML algorithm computation speeds and this significantly reducing the time required to train ML models.

Understanding these similarities and differences in software engineering projects and machine learning projects makes it easy for us to understand the differences and similarities in DevOps approach and MLOps approach to the respective project structures. We will look into in details of this difference in the next section.

DevOps versus MLOps

Let us first start with revising DevOps and MLOps.

DevOps is a set of practices that combines the processes of software developers with those of infrastructural teams to create a set of practices to that enables the continuous delivery and deployment of the software projects. As a result, the development cycles of software is expedited, and continuous delivery of software products is ensured.

Similarly, MLOps adopts DevOps principles and applies them to machine learning models instead of software, uniting the development cycles. Which then enables data scientists, machine learning engineers, data engineers and infrastructure teams to work together, deliver, and deploy machine learning models to production environments. DevOps is a foundational building block for performing MLOps. Without DevOps, we cannot do MLOps.

DevOps and MLOps are fundamentally similar in their objectives because MLOps principles were derived from DevOps principles. Both the processes are similar when it comes to features like continuous integration of source control, unit testing, integration testing, and continuous delivery of the new software releases. But they are quite different in implementation as MLOps covers both components of machine learning, code and data:

- **Machine Learning is experimental is nature**: Due to experimental nature of machine learning projects, MLOps unlike DevOps is needed to address that experimentation component. Machine learning engineers and data scientists have to tweak various features like hyperparameters, parameters, and models while keeping track of and managing the data and code behind all this training and making sure all results are reproducible.
- **Testing**: Testing an ML system involves model validation, model training, and so on in addition to the conventional code tests, such as unit tests and integration tests.
- **Automated deployment**: We cannot merely deploy an offline-trained ML model as a prediction service. It will require a pipeline to automatically retrain and deploy a model. This pipeline adds complexity because we need to automate the steps that data scientists do manually before deployment, to train and validate new models.
- **Taking into account model performance degradation or data-drift**: ML models in production can have reduces performances due to data involved in training-serving skew or simply due to data-drift. Models can decay in more ways than conventional software systems, and we need to track, monitor, and plan for it.
- **Monitoring**: Models in production need to be monitored continuously along with the statistics of data that built the model at the point in time to refresh the model when needed and maintain its lineage in case we need to audit or go back in time to use old models. The models in production degrade over time thus we need a way to notify the stockholders of the same so that necessary steps can be taken to improve the quality.
- **Continuous integration** (**CI**) is the component which is present both in DevOps and MLOps. However, in machine learning it is no longer only about testing and validating the code and components. It also involves testing and validating data, data schemas, and models.
- **Continuous deployment** (**CD**) is no longer about a single software package or service, but a system (an ML training pipeline) that should

automatically deploy another service (model prediction service) or roll back changes from a model.

Continuous testing (**CT**) is another difference in DevOps and MLOps. Continuous testing in DevOps is part of continuous integration component where it is used to run the tests to make sure the code is still in a good shape. However, in MLOps, it involves automatically retraining, testing and evaluating the model, making it an independent and important component.

Principles of MLOps

In this section, we will go over some of the principles of MLOps that might come in handy and will be useful to improve the state of software management tools to account for the machine learning projects and requirements.

Machine learning engineers or data scientists run many experiments to develop the best trained model for the target scenario. These experiments contain:

- **Code**: The actual code used in the experiments to develop the model.
- **Configuration**: The configuration which defines all the different sets of parameters and variables for these experiments.
- **Dataset**: The data used for training and validation of the model being developed. These datasets can easily be many gigabytes or even hundreds of gigabytes depending on the project.
- **Artifacts**: The trained ML models and any other outputs from the experiments.

A machine learning project is a running software. But often there are difficulties in sharing files with colleagues or reproducing the results. Getting repeatable results that can be shared with colleagues, and where you can go back in time to evaluate earlier stages of the project, requires more comprehensive management tools due to which the MLOps solution needs to encompass ideas and principles like:

- **Reproducibility**: It refers to the ability of precisely re-executing the project at any stage of its development. Some of the requirements to

enable proper reproducibility are:

  - Recording the pre-processing steps such that they are automatically rerunnable by anyone and the state of the project as the project progresses. State refers to any kind of project artifacts including code, configuration, datasets, graphs, test results and so on.
  - Ability to recreate the exact datasets available at any time in the project history is crucial to make the projects not just reproducible but also audit the project at any time.

- **Transparency**: Transparency includes but is not merely limited to being able to inspect every aspect of the project like:
  - What code, configuration and data files are used.
  - What processing steps are used in the project, and the order of the steps.
- **Auditability**: It refers to inspecting the intermediate results of a pipeline. Auditability not only includes looking at the results, but also any intermediate results. It is also closely related to reproducibility because with the ability to reproduce the results, we will not be able to audit them.
- **Scalability**: It is the ability to support multiple co-workers on a project, and the ability to work on multiple projects simultaneously.

Having these principles in place allows us to answer questions like what is the intermediate result generated three months ago using the specific dataset and what data has changed since then? Has the dataset been overwritten or changed? A system supporting transparency, auditability and reproducibility for a machine learning project accounts for all these things.

Now that we have a list of principles, let us look at some of the best practices we should try to maintain while working on machine learning projects and MLOps implementation.

MLOps best practices

MLOps is not merely a single straightforward framework. It consists of different components. In this section, we would cover the best practices for

different components of an machine learning pipeline. These major components are:

- Code
- Data
- Model
- Metrics and KPIs
- Deployment
- Team

Let us dig deep into these components and go over the best practices for each one of them.

Code

Machine learning engineering needs grounding in software engineering and its best practices, especially when handling the code. Some of the best practices from software engineering which are generally missing from machine learning engineering pipelines and code but should be adopted to improve the overall process are:

- **Running automated regression tests**: When making changes, new defects can easily be introduced in existing code. A set of regression tests that can run automatically on each commit helps to spot such defects as early as possible.
- **Using static analysis to check code quality**: High-quality code is easier to understand, test, maintain, reuse, and extend. The universally accepted and implemented way of ensuring high code quality gets checked into the code repository is to use static analysis tools.
- **Using continuous integration**: Code changes and additions may introduce problems in the software system. This can be avoided by running an automated build script each time that code is committed to the versioning repository to ensure it passes all the tests and feature criteria.

Data

A machine learning model can only be as good as the data it is trained upon, it is extremely important to adhere to some of the best practices when it comes to building machine learning systems so that the data pipelines are usable and scalable. Some of the most important best practices are:

- **Using sanity checks for all external data sources**: Data is at the heart of every machine learning model. Therefore, avoiding data errors and making sure data quality is maintained is crucial for model quality. Wherever external non-reliable data sources are used, in case the data provided is incomplete or ill formatted, it is important to verify the data quality before it can be used in the pipeline. Invalid or incomplete data may cause outages in production or lead to inaccurate models.
- **Writing reusable scripts for data cleaning and merging**: Data cleaning and merging are exploratory processes and tend to lack structure. Many times these processes involve manual steps, or poorly structured code which cannot be reused later. Needless to mention, such code cannot be integrated in a processing pipeline. Reusable data cleaning scripts should be written for any machine learning application that does not use raw or standard data sets.
- **Ensuring data labelling is performed in a strictly controlled process**: Controlling the data labelling process ensures label quality – an important quality driver for supervised learning algorithms. Data label control should be applied to any machine learning application that uses labels that is in supervised learning or flavours of supervised learning such as semi-supervised learning.
- **Making data sets available on shared infrastructure**: The amount of data processed by machine learning models is higher than usual software systems, raising concerns related to duplication, transfer, storage, and traceability. Making data set available on shared infrastructure helps mitigate these issues. Good data management and storage is important and one of way to make sure it is to have the following features in place: proper access control (with proper authorization and authentication in place), virtualization, versioning, maintainability, and freshness. They help to avoid duplication and

unnecessary transfers and save time. Moreover, it facilitates the adoption of access control policies and provides traceability by keeping a data access log. Adopting standard naming conventions for the data sets to reflect the version is also considered a best practice.

Model

Model management is the core of MLOps, which makes is important for us to have consistency and follow standard best practices when it comes to model development and maintenance. Best practices for model development and management are:

- **Keeping the first model simple and get the infrastructure right**: This allows us to have the proper processing steps and infrastructure in place and that goes a long way when the model becomes complex over the time.
- **Actively removing or archiving features that are not used**: This helps to avoids tech debt and makes it easier to keep the code fresh especially when it is used for multiple experimentation setups.
- **Peer reviewing training scripts**: We all can make human errors in writing code sometimes, that is why software engineering also uses a review process before merging and feature code in the main codebase. Similarly, the training scripts written for the project should also be peer-reviewed.
- **Enabling parallel training experiments**: As we know, training models is a time intensive process. Especially during experimentation when we might need to run code with different parameters, it goes a long way to be able to train and run those experiments in parallel.
- **Continuously measuring model quality and performance**: Monitoring and measuring model quality once it is deployed in production is important to make sure we can track the degradation of the model and work on improving the quality continuously.

Metrics and KPIs

In any experiment or research-based system, it is extremely important to set proper metrics and KPIs in place before embarking of the project. Having proper metrics and KPIs in place will help in tracking and progress and improvements to the system and guide us in keeping us on the right path. When it comes to usable metrics and KPIs, following are some of the best practices:

- **Choosing a simple, observable and attributable metrics for your first objective**: The ML objective should be something that is easy to measure and is aligned with the business objective, but it may not always be a direct measure of it. In fact, there is often no true objective to train on a simple ML objective. Consider having a policy layer on top that allows us to add additional logic to do the final ranking.
- **Setting governance objectives**: Machine learning applications can severely impact human lives. Avoiding negative impacts or biases in ML models, even without malicious intent, requires all stakeholders to operate according to the same ethical values. It is important to explicitly align all stakeholders on the ethical values and constraints of the machine learning application.
- **Enforcing fairness and privacy**: When processing personal information or when developing decision-making systems that can negatively impact individuals or groups, it is important to enforce requirements for fairness and privacy.

Deployment

A commonly shared statistic regarding machine learning is that 90% of the models never make it to production. One of the reasons for that is that model deployment is the most crucial and complex part. This is a major concern which MLOps tries to address. Following the given best practices makes the deployment process as scalable and manageable:

- **Planning to launch and iterate**: Do not expect that the model you are currently working on will be the last one that you will launch, or even that you will ever stop launching models. Consider whether the

complexity you are adding with this launch will slow down future launches.

- **Automating model deployment**: Deploying and orchestrating different components of an application can be a tedious task. Instead of manually packaging and delivering models, and in order to avoid manual interventions or errors, one can automate the deployment and monitoring task. Increase the ability to deploy models on demand as it increases availability and scalability.
- **Continuously monitoring the behavior of deployed models**: Once a model is promoted to production, the team has to understand how it performs. Monitoring also avoids unintended behavior in production models. It should be implemented in any production-level ML applications.
- **Enabling automatic rollbacks for production models**: Similar to deployment, rolling back models can be a tedious process. Instead of manually performing this task, it is recommended to define an automatic process for it. Avoid sub-optimal models in production.
- **Enabling shadow deployments**: Before pushing a model into production, it is wise to test its quality and performance on data from production. In order to facilitate this task, one can deploy multiple models to *shadow* each other. It allows us to test a model's behavior on production data, without any impact on the service it provides.
- **Logging production predictions with the model's version, code version and input data**: Tracing decisions back to the input data and the model's version can be difficult. It is therefore recommended to log production predictions together with the model's version and input data. Enhance debugging, and enable traceability, reproducibility, compliance and incident management.

Team

Machine learning and data science is a collaborative effort in any organization and needs to the collaboration of multiple teams including data engineering, data science, operations and/or software engineering. Following

the mentioned best practices makes it possible to manage the backlog and workload manageable:

- **Using a collaborative development platform**: Collaborative development platforms provide easy access to data, code, information, and tools. They also help teams to keep each other informed, make and record decisions, and work together asynchronously or remotely. Broadly used collaborative development environments include Github, GitLab, Bitbucket and so on.
- **Working against a shared backlog**: An actively maintained list of agreed-upon work items (backlog) enables coordination of tasks within the team and with external stakeholders. It also helps in planning ahead and performing retrospective evaluations. It avoids misunderstandings on the content, priority, and status of tasks.
- **Communicating, aligning and collaborating with others**: It is extremely important to ensure alignment with other teams, management and external stakeholders. The system that the team develops is meant to integrate with other systems within the context of a wider organization. This requires communication, alignment, and collaboration with others outside the team.

MLOps in an organization

MLOps is becoming a useful tool for organizations that uses or depend upon machine learning or data science as either its core business offering or depends on these domains for acceleration of the business value of the already existing offerings. The reason to use machine learning is not to pivot the organization to becoming machine learning researchers competing with companies that specialize in research; it is to accelerate the strategic advantages of the organization through technology.

The calculated risk of adopting machine learning domain as a business accelerator is acceptable if integrated and done in a manner that allows an organization to minimize the downsides of technology change management and MLOps can address that concern. Many options exist to accelerate technological advancement in the organizations even if they do not want to entirely setup and spend on the research teams. Some of those options

include pre-trained models like **Hugging Face**[1] or **TensorFlow Hub**[2], computer vision APIs like **AWS Rekognition**[3] or open-source AutoML solutions like **Ludwig**[4] or MLOps orchestration frameworks like **MLRun**[5] or **MLFlow**[6]. Organizations that adopt MLOps with an approach of using the right level of abstraction give themselves an advantage over organizations that hired 15 data scientists who only focus on the research component of a machine learning system and ignore the overall system design. In the latter example, for standard machine learning systems it is often the case that after years of focusing on research and ignoring the end-to-end solution. In the best case, nothing is done, but in the worst case, a lousy solution creates a worse outcome than doing nothing.

For organizations which uses technology and machine learning to support its growth need to do so by having proper strategy in place and quick adaption of the technology. Thus it is crucial to have proper strategy in place to integrate right level of MLOps at right times. Let us look at some more details on how MLOps strategy can help the organizations and what should be the building blocks of the MLOps strategy of an organization.

MLOps strategy

With all the considerations so far with MLOps, it is time to turn to strategy and see how we can setup the building blocks of the proper MLOps strategy. There are four key categories to consider when implementing an MLOps strategy:

- Cloud
- Training and talent
- Vendor
- Executive focus on ROI

Let us discuss each of these four categories.

Cloud

As we know, there are multiple general-purpose Cloud providers and dedicated machine learning Cloud service providers. However, there is no one or perfect as to which cloud platform to use. It completely depends on

the organization's requirements and what kind of platform the organization is already using. Any central platform will offer advantages of economics of scale, so that is not the major driving factor. In an MLOps strategy, it is essential to be aware of how a Cloud platform fits into the unique goals of each specific component, like platform deployment, integrations with other existing platforms and vendors organizations are already using, hiring the talent to work with these platforms, and long-term viability.

Training and talent

Often, organizations look at the power of new technology and how they can take advantage of it. However, they also fail to consider the training and talent component of using a specific technology at times. In almost all cases, an organization should use a less powerful technology or a commonly used technology platform if hiring and training for that are better and easy. This fact means the widespread use of technology is crucial when implementing new technology. That is why it is crucial to check the support mechanism provided by the vendor for any propriety technology and the overall community and support available for any open-source technology. Ultimately, the latest technology is dead on arrival if you cannot hire or train your staff. This is why training and talent are crucial parts of not just MLOps strategy but any technical strategy.

Vendor

An often-overlooked issue with using Cloud computing is that it usually needs to be augmented by specialized vendors to help an organization reach its goals with the technology. Generic Cloud platforms are sometimes too generic to provide a specific solution as they need to address a wide set of requirements. This is where specialized vendors play a crucial role in filling those gaps for specific technology requirements, which can be used if the organization is unwilling to or has the resources to build the solution using generic Cloud platforms. So, it is important to properly define the organization's requirements from machine learning clearly and include those as part of the overall MLOps strategy. These strategic choices can lead to better **Return on investment** (**ROI**) for both the Cloud and the business strategies. Examples include using vendor technology specializing in Hadoop, Kubernetes, on pre-trained models to accelerate the speed to value.

The vendors will be unique to each organization based on their requirements, level of abstraction, the talent available in the organization to work on these vendor platforms, as well as their business goals.

Executive focus on Return on Investment

Return on investment is the most crucial part of any MLOps strategy. Ultimately, the preceding three categories do not mean anything if the executive focus is not on ROI. If we club the other three categories under the technology focus part of the strategy, ROI is definitely where the executive focus should be. The purpose of any technology is to ultimately drive long-term business value, accelerating the time to value. This justifies the investment in any technology.

Implementing MLOps

At this point, it seems that MLOps will help us deploy machine learning models rapidly and maintain them once they are deployed. Now, the biggest hurdle seems to be the question of how to get there, and where to start. The level of automation described and discussed in this chapter so far required significant work from machine learning, data engineering and infrastructure/operational teams to achieve it. It almost seems better in the short run to build and deploy the models manually rather than devote resources to setting up the entire infrastructure, but this is simply unsustainable in the mid to long run.

One way to approach this is to start small and automate one step at a time in an abstracted manner till we reach full automation. The only thing we need to keep in mind while doing so is that any automation being worked upon is scalable and will fit in the overall architecture of the MLOps. For example, data scientists and machine learning engineers generally start any project experimentation with notebooks like Jupyter which are great for performing experiments, so let us start with thinking is there a way to track them? Such a functionality would be extremely useful especially when teams are implementing advanced machine learning architectures from scratch, as it would let them share and compare the new models across all the relevant metrics with deployed models on current architectures.

The takeaway from here is that accounting for this kind of abstraction to start with small portions first and use them as building blocks to achieve the final goals of an automated, near realtime MLOps platform makes it manageable and achievable. No doubt these factors and step-by-step implementation require significant resources to plan, develop, and test while keeping the final vision in mind. This line of thinking will also help when we go out and look at all the great tools and platforms available to use which essentially implement either part of or all of the automation for us. Several examples of such tools and platforms are available like **MLFlow**, **Databricks**, **Kubeflow**, **AWS SageMaker**, **Microsoft Azure**, **Google Cloud**, and **DataRobots**. With these tools, implementing MLOps principles into our workflow will be significantly easier. We will look at the overall architecture and how to make these decisions in later chapters. For now, let us check how different environments can help us in overcoming some of these challenges with MLOps in the next section.

Overcoming challenges of MLOps

By now we know that there are significant challenges for most machine learning models to make it to production. These challenges and hurdles differ significantly depending upon the organization's plan on level of abstraction and implementation but there are options out there to accelerate this implementation timeframe and reduce the time to value with MLOps. Let us look at some of those ways to overcome the challenges; MLOps is generally either implemented on the Cloud, on-premise or in a hybrid environment. Each of these environments provides ways to handle the challenges.

MLOps in Cloud

There are several critical advantages of Cloud computing that MLOps methodologies leverage:

- Cloud is elastic resource that enables both efficient use of computing and storage and the ability to scale on demand to meet almost any requirement. This capability means that Cloud computing has on-demand access to essentially infinite resources.

- Second, the Cloud had a network effect in that Cloud technologies benefit from integrating other cloud technologies. A great example is **AWS Lambda**, a serverless technology. AWS Lambda is a valuable service to build applications because of the deep integration with other AWS services like **AWS Step Functions**. **AWS Sagemaker**, or **AWS S3**. For any active Cloud platform, you can assume that the integrated network of services further strengthens its capabilities as the platform develops more features.
- Third, in this day and age almost all Cloud vendors have MLOps platforms. AWS has Sagemaker, Azure has Azure Machine Learning, and Google has Vertex AI. Even smaller niche vendors have offerings and platforms to address the needs of machine learning and MLOps. By using a Cloud platform, an organization will likely use some of the offerings of the native ML platform and potentially augment it with custom solutions and third-party solutions which further accelerate the time to value from implanting MLOps.
- Fourth, all Cloud vendors have Cloud development environments. A significant trend is the use of a combination of lightweight Cloud Shell environments like AWS CloudShell, heavier full IDE environment options like **AWS Cloud9**, and notebook environments, both free like **Sagemaker Studio Lab** or **Google Colab** and those with rich IDE integration like **Sagemaker Studio**.
- Finally, depending on what a company is doing, they may have no option but to use Cloud computing. Some Cloud computing components are a hard requirement for organizations specializing in building be spoke deep learning solutions because deep learning requires extensive storage and computing capabilities.

In addition to the public Cloud vendors, several additional players offer MLOps solutions in the Cloud. These vendors can operate on the public Cloud or on private Clouds. The advantage of using a smaller vendor is the customization level that such a company provides its customers. In addition, an MLOps vendor will have more in-depth expertise in MLOps since that is its only focus. This often ensures a lot more relevant features and a lot more integrations. Finally, by choosing a vendor that is not tied to a specific cloud

provider, customers are not tied to it either. Rather, customers can use the vendor across multiple clouds or on additional infrastructure.

While public cloud providers offer their own solutions, sometimes organizations might need a solution that is more tailored to their specific needs. Let us look at two more deployment options: on-premises deployment and hybrid cloud deployment.

MLOps on-premises

In some use cases, organizations cannot use the public Cloud. Business restrictions like the need to secure sensitive data or having to adhere to strict regulations (for example, data localization privacy regulations) require an MLOps solution that can operate on-premises. Many MLOps solutions offer the ability to deploy them either in the Cloud or on-premises. The only downside to this approach is that on-premises solutions require an organization to procure the servers and equipment that will support the intense computing power needed to run ML algorithms at scale. Further, it will not be that easy to scale up or down the capabilities on demand. The organizations will also need significant infrastructure/operations teams to update and maintain the infrastructure.

On the other hand, an on-premises deployment will almost certainly require some sort of customization. This gives organizations more control over the product and they can make specific requests to tailor it to their needs. More specifically, if the deployed solution is a startup solution they will be attentive and work hard to ensure satisfaction and adoption. If it is an open source product, then not only can the organization leverage the development power of the community, they can also go inside with their own developers and tinker with the product to ensure it suits their needs and requirements.

MLOps in hybrid environments

Similar to the on-premises deployment, some organizations might prefer a hybrid Cloud deployment approach. This would involve deploying on the public Cloud(s), on-premises, and perhaps even on a private Cloud or on edge devices. Naturally, this complicates the process since the MLOps solution should ensure that data and control path are appropriately separated based on the specific requirements of the project. Further, they must be

delivered by a highly available, scalable entity that orchestrates, tracks, and manages ML pipelines across types of infrastructure deployments. And all this has to take place at a high speed and availability, with optimal performance. Finally, the solution would ideally provide a single development and deployment stack for engineers across all infrastructure types.

Finding a vendor or open-source solution that answers all these requirements might not be as simple, in such scenarios and custom requirements. The best bet is probably with specific vendors, preferably startups which are open to listening and tweaking their offering as per the requirements or mature open-source solutions that can be customized to the specific needs and requirements.

Conclusion

This chapter sets the stage for understanding the crisis in enterprises getting machine learning and AI in production. MLOps aims to add a methodology that builds off the successful lessons of DevOps while handling the unique characteristics of machine learning.

As we covered in the chapter, MLOps is a set of principles and practices adopted from DevOps and applied to machine learning. We looked into why MLOps is important and covered the basic principles and best practices of MLOps. We also covered the role of MLOps in an organization and why having a good MLOps strategy matters to be able to successfully implement the processes as well as the building blocks of the MLOps strategy for an organization. The chapter also provides an overview of the challenges faced in implementing the MLOps. Ultimately, it comes down to ROI with data science, as technology is an accelerant of value for most organizations, not the value.

In the next chapter, we will look in the overall architecture and the different components of this architecture when it comes to MLOps. This will set us up for success to able to achieve the goal of easily deploying the machine learning models in different environments. We will also look at how pipelines make the core component of MLOps architecture and what different deployment strategies can come in handy while designing the process.

Points to remember

- MLOps combines machine learning, DevOps, and data engineering to enable continuous delivery and deployment of machine learning models.
- MLOps aims to standardize and streamline the machine learning lifecycle, including data processing, model training, deployment, monitoring, and governance.
- Machine learning projects differ from traditional software projects in their focus on experimental iteration guided by metrics rather than features.
- Key principles of MLOps include reproducibility, transparency, auditability, and scalability.
- An MLOps strategy should consider different possible platforms and systems, vendors, required training of the team to onboard into the platforms or vendors, and executive focus on ROI.
- MLOps can be implemented on cloud, on-premises, or in a hybrid model, each with tradeoffs. Vendors and open-source solutions can help accelerate implementation.
- The end goal is the continuous delivery of machine learning models to unlock business value while overcoming the challenges inherent in machine learning systems.

Key terms

- **MLOps**: **Machine Learning Operations** (**MLOps**) is a set of practices designed for collaboration between data scientists, machine learning engineers, data engineers, and operations professionals. It is a combination of **Machine Learning** (**ML**) and **Operations** (**Ops**) which provides us with the processes for designing, building, enabling and supporting the efficient deployment of ML models in production and continuously iterating and improving upon these models.

- **COTS**: **Commercial off-the-shelf** (**COTS**) software and services are built and delivered usually from a third-party vendor. COTS can be purchased, leased or licensed by the sellers to the organizations.
- **Docker**: Docker is a set of platform-as-a-service products that use OS-level virtualization to deliver software in packages called **containers**.
- **Kubernetes**: Kubernetes, also known as **K8s** is an open-source container orchestration system for automating deployment, scaling and management of containerized applications.
- **CI/CD/CT**: It refers to Continuous integration, continuous deployment, continuous testing.
- **Metric-driven development**: Metric-driven development is a methodology that emphasizes the use of data and metrics to drive product development decisions. Since the goal of machine learning projects is to find the most accurate model, the projects are generally guided by metrics achieved in the experiments and are thus metric-driven.
- **Feature-driven development**: Software engineering release and development decisions are based on whether the team has reached the feature development goals. Feature-driven development organizes software development around making progress on features.

---

[1] **https://huggingface.co/**
[2] **https://tfhub.dev/**
[3] **https://aws.amazon.com/rekognition/**
[4] **https://ludwig.ai/0.5/**
[5] **https://www.mlrun.org/**
[6] **https://mlflow.org/**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 2
MLOps Architecture and Components

Introduction

In the previous chapter, we looked at the introduction of MLOps, its fundamental principles and uses. We also looked at how MLOps plays an important role in any machine learning system or application. In this chapter, we will learn about the architecture of an MLOps framework and infrastructure. We will go over the development and deployment strategies for development, staging, and production environments and the process of moving the machine learning models from the development phase all the way to the production system.

A well-defined MLOps architecture is essential for managing machine learning workflows and operationalizing models in production environments. We will dive deeper into the components and architecture of MLOps systems. It is important to look at MLOps architecture from different maturity aspects from basic proof-of-concepts to sophisticated enterprise-grade systems. Understanding these architecture evolution levels provides guidance on how to tailor MLOps implementations based on organizational needs and infrastructure readiness. This foundation will equip us to design and implement MLOps solutions suitable for the different use cases.

Structure

In this chapter, we will discuss the following topics:

- MLOps components
  - Data source and versioning
  - Data analysis and experiment management
  - Code repository
  - Pipeline orchestration
  - Model training and storage
  - Model deployment and serving
  - Monitoring for model, data, and application
  - Feature processing and store
- MLOps architecture
  - Architecture level 1: Minimum viable architecture
  - Architecture level 2: Production grade MLOps
  - Architecture level 3: Enterprise grade MLOps
- The semantics of dev, staging, and production
  - Execution environment
  - Code
  - Models
  - Data
- Machine learning deployment patterns
  - Deploy models
  - Deploy code
- Bringing the architectural components together
  - Development environment

    - Staging environment
    - Production environment

Objectives

By the end of this chapter, you will have a solid understanding of the different components required for a successful MLOps infrastructure and a reference MLOps architecture that can address most of the use cases of MLOps and how it fits in with different development environments.

We will also cover the three major verticals required to set up a scalable and feasible MLOps platform:

- The data preparation vertical
- The model experimentation and training vertical and
- The model operation vertical

We will understand how the first two verticals are more mature than the model operations vertical. As in the last decade, the focus and most of the investment was on data preparation, engineering, training, exploration, and model experimentation tools. The third vertical is also an integral part of any machine learning system, which focuses on productionizing the results of the data preparation, experimentation, and model development, giving organizations the feasible platform to accelerate machine learning adoption and value creation at scale. This chapter will break down all the components required for a mature MLOps architecture and how a well-built architecture can ensure that the machine learning pipelines are built for production at scale.

After this chapter, we will have a solid grasp of MLOps architecture building blocks, maturity levels, environment semantics, deployment patterns, and end-to-end workflows.

MLOps components

Based on the principles we looked into in *Chapter 1: Getting Started with MLOps*, we will look at relevant components that can be used to address those principles and guide us in how things should be realized in MLOps

architecture. The major components for a successful and scalable MLOps architecture are:

- Data source and data versioning
- Data analysis and experiment management
- Code repository
- Pipeline orchestration
  - Workflow orchestration
  - CI/CD automation
- Model training and storage
  - Model training
  - Model registry
- Model deployment and serving
- Monitoring for model, data, and application
  - Training performance tracking
  - Metadata store
- Feature processing and storage

The following section will describe the preceding components and provide an overview of the generic functionality of each of these components.

Data source and data versioning

Data is important for any machine learning model. It is often believed that a machine learning model is as good as the data upon which it is trained; if the training data is not of good quality the model output will never be good. We must be careful and intentional about the data sources and datasets used for training, validation, and evaluation purposes. Having proper and clean data sources and clubbing them with the ability to have versioning and data labeling processes in place can guarantee us to have a good dataset to train the models. It also will allow us to maintain the proper lineage of the data we are using for training purposes.

One of the most time-consuming and lengthy tasks is the initial data preparation. Before it can be used for training and experimentation, we will need balanced, clean, and labeled data. The quantity of data available is also important as the required size of the dataset varies depending on the model type.

While thinking about this component of the architecture, consider how we can use programmatic labeling tools and processes to improve reliability, scalability, and auditability while creating balanced and good training datasets and have processes and components in place to version the datasets accordingly.

When it comes to data sources, some of the available options are:

- Databases (both SQL and NoSQL)
- Data warehouse
- Data lake, and so on
- Publicly available or proprietary datasets

For data versioning, a lot of the custom of-the-shelf MLOps platforms, as well as open-source tools, provide this functionality, some of the options are listed as follows:

- DVC
- Pachyderm
- lakeFS
- dolt
- Neptune
- Delta Lake

Data analysis and experiment management

Data analysis and experiment management infrastructure provide the foundational and underlying compute resources like CPUs, RAM and GPUs for the initial data exploration and analysis. They also facilitate running the experiments for model training, validation, and evaluation. The provided

infrastructure can be either distributed, like Cloud environments or non-distributed, like local machines or servers.

Once we have access to high-quality, labelled training data, we can move to the next phase, which is data analysis and experimentation. As we know, machine learning models are a reflection of the data it is trained with. A good data analysis and experimentation infrastructure is important to empower data scientists and for the project's success.

While considering solutions, keep in mind that machine learning engineers and data scientists should be able to use the platform or tools that make the training and experimentation easy and repeatable. The tools should be flexible and easily integrated with other platforms, tools or libraries.

Some of the technology components which help with performing these tasks are:

- Exploratory data analysis
  - Jupyter notebooks
  - Google Colab notebooks
- Language and libraries:
  - Python (Scikit Learn, PyTorch, Tensorflow, Keras)
- Frameworks:
  - Apache Spark
  - Apache Mahout
- Experiment Tracking:
  - MLFlow
  - Kubeflow
  - Neptune
  - Comet
  - AWS Sagemaker
- Visualization:
  - Seaborn

    - Matplotlib
    - Plotly

Code repository

The source code repository is the most familiar component for any engineer. It ensures code storing, versioning and code lineage and allows the whole team to work, commit and merge the code in parallel without overwriting others' work or any fear of losing the work.

Some of the universally used source code repositories are:

- GitHub
- Bitbucket
- GitLab

Pipeline orchestration

Pipeline orchestration is an umbrella concept which handles the orchestration and automation of the entire process. It is responsible for making sure all the steps of the workflow run properly. Two major components of pipeline orchestration are **workflow orchestration** and **CI/CD automation**.

Workflow orchestration

The workflow orchestration component takes care of the orchestration of each and every task of an ML workflow via **Directed Acyclic Graphs (DAGs)**.

DAGs are directed graphs without any directed cycles. They represent execution order and artifact usage of single steps of the workflow. In a directed acyclic graph, it is impossible to start at one point in the graph and traverse the entire graph. Each edge is directed from an earlier edge to a later edge.

Some of the industry standard orchestration platforms for machine learning are:

- Apache Airflow

- Kubeflow Pipelines
- Luigi
- AWS Sagemaker Pipelines
- Azure Pipelines

CI/CD automation

Continuous integrations/continuous delivery (**CI/CD**) component, as the name indicates, ensures continuous integration, delivery and deployment of the latest code to the different environments based on how the automation pipeline is configured. CI/CD also deals with running automated tests. In context of MLOps, it takes care of automatically building, testing, delivering and deploying the models into the desired environment. It also provides rapid feedback to the team about the success or failure of the process with details on which component, which steps failed and for what reasons, thus allowing the team to respond quickly. This not only automates the repetitive steps of the overall pipeline but also increases the overall visibility of the team into the feasibility of the new code.

Suggested technology components to be used for CI/CD automation are:

- GitHub Actions
- CircleCI
- Travis CI
- Jenkins
- Cloudbees Platform

Model training and storage

Model training and storage is a combination of multiple sub-components to train a model, evaluate and version the model and then followed by storing it in a model registry to be used by the deployment pipeline.

Model training

The model training component enables us to efficiently run powerful algorithms for training machine learning models, which are both scalable

and cost-effective. Model training is one of the core components of MLOps architecture. It should be elastic and flexible enough to scale up or down based on the model we are training and the dataset being used. The architecture should be able to perform distributed training using multiple CPUs/GPUs and multiple workers.

Some of the most common platforms which can be considered for model training are:

- AWS Sagemaker
- Databricks unified data analytics platform
- Microsoft Azure machine learning studio
- DataRobot

Choosing the appropriate platform for model training and storage depends on various factors including the specific use case (that is the size and complexity of the dataset being used), type of the machine learning model being trained, the cost of the platform, technical requirements and preferences based on the existing infrastructure. Some of the considerations which can help in making those decisions are:

- Consider the complexity of the model and its training requirements. More complex models require more computational resources.
- Evaluate the size of the dataset. Larger datasets might necessitate more storage and processing power.
- Decide on the machine learning framework (that is TensorFlow, PyTorch) and tools we will use for that.
- Determine how much we can spend on the infrastructure as some of the training platforms can be expensive.
- We need to make sure the platforms can scale and handle any growing needs over time.
- If the models require GPUs/TPUs that should also be considered while deciding on the platform.
- The team’s expertise and experience also play an important role in choosing the platform.

- Consider whether the platform integrates well with other existing tools and workflows.

Model registry

The model registry centrally stores the trained ML models together with their metadata. It has two main functionalities: storing the ML artifacts and storing the ML metadata. Both of these functions are extremely important for governance, explainability, version tracking and lineage maintenance of the machine learning models running in different environments and stored in the registry.

Advanced platform integration model registries include:

- MLFlow
- AWS Sagemaker model registry
- Microsoft Azure ML model registry
- Neptune.ai

Simple storage examples which can be used for the model registry are:

- AWS S3
- Microsoft Azure storage
- Google Cloud storage

Model deployment and serving

Once we have our experiments completed, artifact storage in the registry and proper version tagging is done, the model now needs to be prepared for deployment. Containerization is the most commonly used approach as it gives the flexibility to deploy on many different kinds of infrastructure and provides the most flexibility. We can decide to use our own specifications when it comes to creating containers or use them if we have any standard specifications for the organizations. In case we want to look at some of the standard container specifications for machine learning, we do have a couple of options like **Open Neural Network Exchange** (**ONNX**), TensorFlow saved model, chasis.ml, and so on. These can help in packaging the model in a common specification and expose the model as an API endpoint. Using

these packaging models is optional and can be completely ignored if we want to create our own standards in the specifications of the projects in the organization. However, using established standards and best practices is recommended as it can provide benefits like interoperability, portability, and reproducibility. When we have the model containerized, it can be deployed into different environments. An approach like this allows us to process data in batches or, if we prefer, in real-time using data streams.

What follows after the model deployment is model serving and inference. Now, there are three main types of model-serving approaches. The first one is **online inference** for real-time predictions, where small chunks of data come in near real-time, and predictions are generated for those before returning the response. The second is **batch inference,** which is the preferred method if we want to generate predictions using large volumes of input data and want to store the predictions somewhere and use them from there. Finally, we have **hybrid inference,** where a combination of online and batch inference is used to balance the trade-offs between real-time responsiveness and computational efficiency. Hybrid inference is particularly useful for applications that require real-time and batch processing or for models that can be partitioned into smaller components that can be served independently.

Some of the technologies which can be used for the implementation of this component are:

- Containerization
  - Docker
  - Kubernetes
- Web application frameworks (For rest of APIs):
  - Flask
  - Django
  - Fast API
- Serving frameworks:
  - AWS Sagemaker Endpoints
  - Microsoft Azure ML

- Google Vertex AI
- KServing of Kubeflow
- IBM Watson Studio
- TensorFlow Serving

Monitoring for model, data, and application

The monitoring component, as is obvious from the name, monitors the model training and serving performance. Other than the performance monitoring, we will need to monitor the incoming data for data drift, monitor the infrastructure where the application is deployed, and review pipeline orchestration monitoring to make sure all the moving pieces are working fine.

As we know, model improvement is a continuous process. To improve upon the version of the model which is in production, we need to enable the team and other stakeholders to monitor and see if the performance degrades and data drifts from the original. Further, we must expose the monitoring capabilities to the infrastructure teams to monitor the infrastructure on which machine learning applications are running.

In addition to performance monitoring, we need metrics to track usage and maintenance of the machine learning systems to manage costs.

Outside the model-specific monitoring, processes and controls should be in place to govern the usage and maintenance of the machine learning system in production. It not only helps in keeping the infrastructure up to date but also cost management.

Some examples of monitoring tools which can be used for machine learning projects are:

- Kubeflow in-built monitoring
- MLflow inbuild monitoring
- AWS Sagemaker model monitoring (inbuilt)
- Prometheus and Grafana
- Elasticsearch, Logstash, and Kibana stack (ELK stack)

- AWS Cloudwatch

Training performance tracking

Input data and model predictions are monitored, both for statistical properties like data drift, model performance and so on, and for computational performance like errors during training, throughput and more.

Metadata store

Machine learning metadata stores enable us to track and store various kinds of metadata. A Metadata store is required to keep track of all monitoring and training metadata in the production environment. Depending upon the needs of the project and infrastructure, some metadata stores can be configured within the model registry database. It can also be used to log the metadata of each job like the end result, data, time, duration and model training metadata like the parameters used for training for that specific run, the resulting performance metrics, and data processing metadata, including metadata from the data and code used.

Feature processing and storage

Feature processing components and the corresponding pipeline reads data from different data tables and feature tables, featurize and write to tables in the feature stores. This component generally consists of two steps: **feature processing pipeline** and **feature store** to store the outcome and make them available for model training.

Feature processing

Feature processing consists of a data preparation step which checks for and corrects any data quality issues. It is followed by the featurization step, which consists of logic to produce features to be used for model training and development.

In some organizations, feature processing pipelines are managed separately from machine learning projects. In such cases, this component can be omitted from this reference architecture.

Feature store

A feature store is a database which enables **central storage** of all the parameters or features which are commonly used. There are two types of feature stores:

- Databases that are used as **offline feature** stores and serve features with normal database latency for experimentation.
- Database that is used as **online feature** store to serve features with extremely low latency for online predictions in the production environment.

Offline and online feature stores both cater to different purposes in managing and serving feature data. An offline feature store is primarily focused on batch processing and data preparation for machine learning models. It is designed to store, manage, and serve historical data that has been collected and processed over time. An online feature store, on the other hand, is geared towards real-time or near-real-time use cases in production environments. It serves feature data to machine learning models during inference or prediction.

Offline feature stores are optimized for handling large volumes of data in a batch-oriented manner. They are used to preprocess and transform raw data into features that are ready to be used for training models. Online feature stores are optimized for low-latency retrieval of feature data. They provide quick access to features that models require for making predictions.

In terms of scaling, offline feature stores are typically designed to scale horizontally, accommodating large datasets and processing workloads efficiently. Online feature stores need to be highly scalable to handle concurrent requests from multiple models or applications without causing delays.

In summary, the main difference between offline and online feature stores lies in their use cases and the nature of the data they handle. Offline feature stores are used for data preparation and historical analysis, while online feature stores are essential for serving real-time feature data to models during inference or prediction. In many cases, organizations use both types of feature stores in combination to cover the entire machine learning lifecycle, from data preprocessing to production deployment.

Some commonly used feature stores are:

- Feast
- AWS Sagemaker Feature Store
- Databricks Feature Store

MLOps architecture

Based on the principles and components, we can create an MLOps architecture which is generic and adaptable in nature to fit most of the machine learning use cases. Moreover, it can guide machine learning engineers and data scientists on how to approach an MLOps system to tailor the architecture as per the specific organizational requirements. One goal of having a generic architecture in place for guidance is to have the artifact which is designed to be technology agnostic. It will help the machine learning team to choose the tools and technologies as per their unique requirements and align with the organizational policies and structure.

The first objective while building an MLOps practice in the team is to ensure team members can work efficiently. We do not wish to build a new framework that will increase the workload on the team and is bad for the adoption of any new framework.

To enable the teams to adopt the appropriate components and infrastructure for better experimentation and proper tooling, we need to take one step at a time. We must be careful about what process and tooling we are introducing to the process based on the maturity of the team and the requirements of the organization. With the right infrastructure in place, we will see more and quicker model iteration and which will keep the models fresh in order to generate better results.

In this section, we will divide the production machine learning lifecycle into three possible architectural plans depending upon the maturity level of the team and the organization in adapting these architectural frameworks. Each of these fits a stage of the machine learning lifecycle and the maturity of the team in adopting those processes.

Note: MLOps is a continuous journey of improvement and optimization for productivity and scalability.

Here are the architectural patterns for the different maturity levels of machine learning in an organization:

Level 1: Minimum viable architecture

This is the architectural pattern which can support few smaller models which the organization deploys as the starting point. It consists of several manual steps and processes. If the number of models increase or the complexity rises, this level of architecture with manual steps adds a lot of overhead. This can be considered as the **Proof of Concept** (**POC**) architecture, where the team can start to adapt to the new way of working with the models.

Level 2: Production grade MLOps

This level of architectural pattern is for organizations that are ready for the next step of automation. It fits the organizations that manage more than a couple of models in production and support product or service offerings of the organization. Thus the need for a production-grade MLOps architecture in place to support the production workload.

Level 3: Enterprise grade MLOps

This level is for teams engaged with the use and development of machine learning applications in multiple scenarios, and where machine learning empowers multiple products in the organization. In such cases, organizations need a consistent way to manage all pipelines, artifacts and model lifecycles.

The ability to excel at each level and ultimately progress is based on an organization's infrastructure readiness. As an organization's infrastructure and needs mature, it results in the increase of the number of models managed in production. Consequently, they want to evolve from one architecture level to the next.

With this understanding, we are ready to look into the architectural patterns in detail. We will see the common solutions that help the organization in addressing its requirements at each of the three maturity levels.

Architecture level 1: Minimum viable architecture

Level 1 architecture is for organizations that are getting started with their first model or have 1-2 models in production. From a technical perspective, at this level, engineers train models manually through notebooks (either

Jupyter or other) on the individual local machines or basic cloud resources, storing them manually. They are then handled to the engineering team to integrate into the applications and move the models into production. This architecture is best suited for organizations which are just starting or the single-product, single-model organizations where the model changes are infrequent and the workflows are simple enough for manual processes. The process in this level of architecture is mostly manual data science or machine learning engineer-driven process. This might be enough in scenarios where models are not that frequently changed or trained.

This architecture level, as depicted in *Figure 2.1*, addresses the basic challenges faced by machine learning teams in order to make sure that they can work efficiently. Most of the components in this architecture are managed manually by the use of scripts or pure code without any automation in any of the components.

Some of the challenges level 1 architecture addresses are:

- Consistent development environments
- Elastic scalability for model development
- Parallel processing to handle the high volume of incoming data
- Simplified and manageable model development, versioning and deployment.

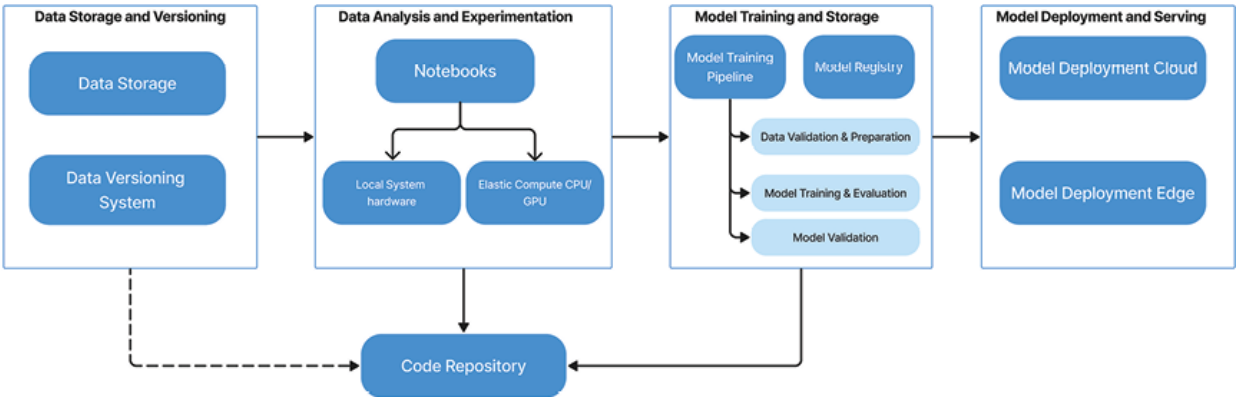

Figure 2.1: *Architecture Level 1: Minimum Viable Architecture*

Characteristics of level 1 architecture are listed as follows:

- **Manual or script-driven process**: Most of the steps are manual, including initial data exploration and analysis, data preparation, experimentation, model training, evaluation and validation.
- **The disconnect between machine learning and operations teams**: Level 1 architecture separates the machine learning team, which creates the model, and the software engineering team, which deploys the model, integrates with the application and serves the model as part of the platform. Machine learning teams train the model and then hand over the model artifact to the software engineering team for deployment purposes.
- **Infrequent retraining and new versions**: This is one of the main characteristics of why level 1 architecture works and provides value for such use cases. This is the scenario where the machine learning team manages only a few models, that change rarely. A new model version is deployed on a monthly or quarterly cadence.
- **No continuous integration**: As we know, changes to these models are infrequent. As a result, no efforts need to be put into continuous integration at this architectural level. Usually, testing the code is part of the notebooks or script execution. We can package the model and manually hand it over to the engineering team.
- **No continuous deployment**: For the same reason, it does not make sense to put in efforts to set up continuous deployment of the models.
- **Lack of continuous performance monitoring**: This architecture level does not track or log model predictions and any actions in order to support retraining efforts.

Architecture level 2: Production grade MLOps

This next level is for organizations which manage multiple models in the production environment. It helps the team that frequently deploys models into production but does not monitor the model performance and use it to make improvements to the model. It can also be useful for the teams which need to know how to best A/B test or roll out new versions of the model frequently, without a lot of friction. It focuses on adding monitoring and automation to re-trigger training pipelines.

This level of architecture, as depicted in *Figure 2.2*, is well suited for organizations which have few complex models in production and continuously work on improving them. It results in frequent updates and deployments of the models. Some of the challenges this architecture maturity level addresses are:

- It is the next iteration on level 1, so it addresses all the challenges addressed in level 1 architecture.
- Model performance tracking and overall infrastructure and metadata monitoring.
- Training pipeline, which can be automated and triggered using new data availability and on a regular time cadence.

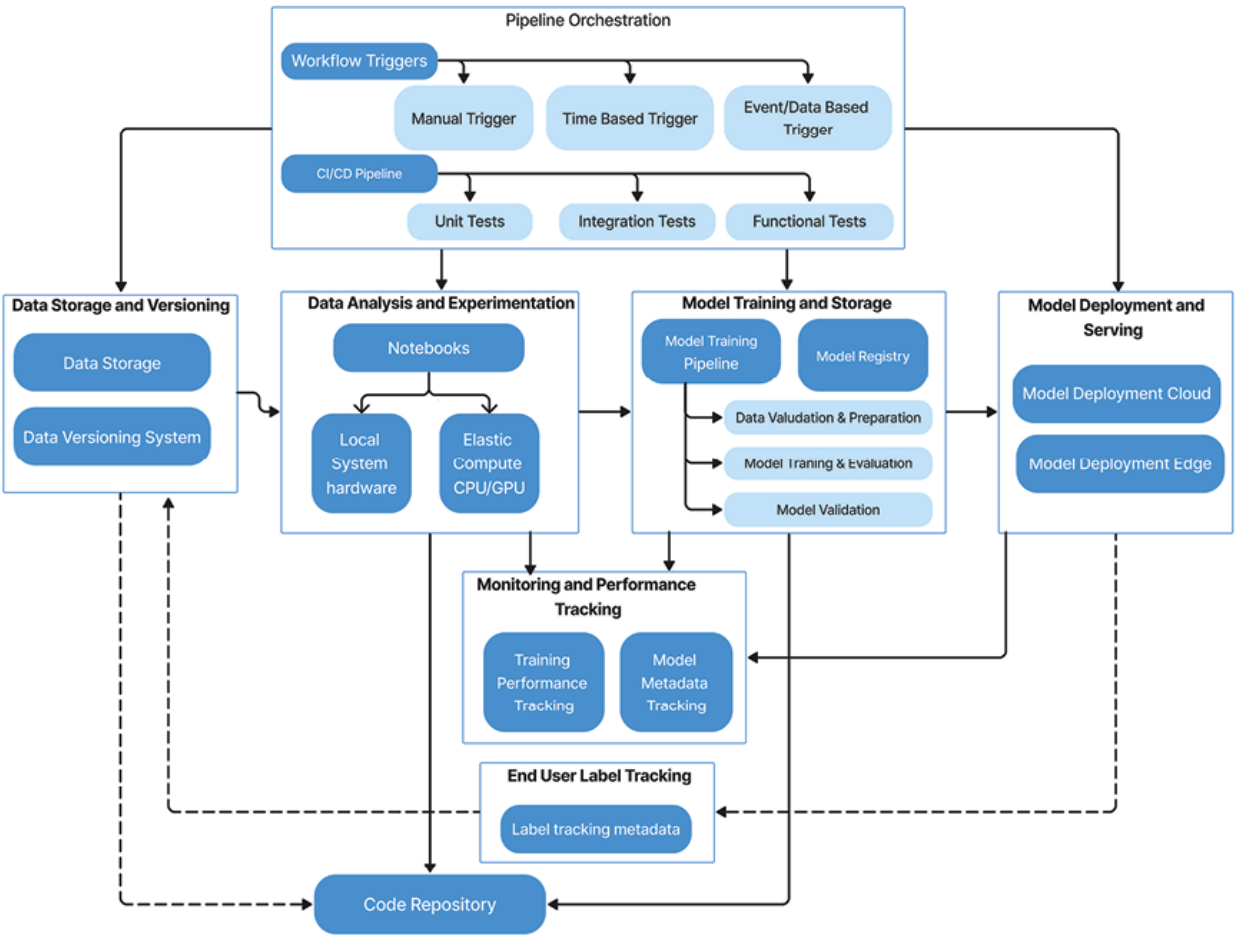

Figure 2.2: *Architecture Level 2: Production grade MLOps*

Characteristics of level 2 architecture are listed as follows:

- **Faster experimentation:** Machine learning experiment steps are reproducible and thus can be orchestrated and done automatically.
- **Continuous training of the model:** The model is automatically trained in production, either on a regular time cadence or based on event triggers when new data is available to be used for training.
- **Modularized code components:** To construct and manage machine learning pipelines, all the different components of the pipeline need to be built in a way that makes them reusable, scalable, and idempotent. If the component code is generic to a certain level, it can be converted to a utility function which can potentially be shared across machine learning pipelines.
- **Continuous deployment of models:** The model deployment step, which serves the trained, evaluated, and validated model as a prediction service, is completely automated with proper monitor capabilities in place.
- **Pipeline deployment:** This is one of the major differences between level 1 and level 2 architecture. In level 1, the trained model is deployed to the production environment. In level 2 architecture, the entire training pipeline is deployed, which then automatically and recurrently runs and generates models. It can then be automatically served as the prediction service.

Architecture level 3: Enterprise grade MLOps

MLOps level 3 architecture is for organizations leveraging machine learning capabilities across multiple products or platforms. These complex environments need a unifying architecture and framework. With the goal of streamlining and managing end-to-end workflows for operations, data and models. It provides the integration required for multiple teams to efficiently manage machine learning lifecycles and supports seamless collaboration between teams and platforms. Suppose an organization runs not only multiple platforms powered by machine learning but also has different teams managing those platforms, running its own machine learning systems. Couple that complexity with data sharing between the different models and systems, which might result in data inconsistency issues across models or

issues of data duplication at multiple places. Each team will end up creating identical data features from the same data to train their models. This architecture level is the answer for them to address all their challenges. In short, this architecture level is for the multi-model, multi-product organizations to enable them to process and retrain the models daily or even hourly if they desire. Some of the challenges level 3 architecture addresses are:

- Similar to the previous levels, level 3 architecture also iterates and builds upon the existing architecture, so it addresses all the challenges covered in Minimum Viable Architecture and production grade MLOps.
- More comprehensive model registry, including features like model cards, model version details and so on.
- Ability to orchestrate all machine learning pipeline components in one place and have proper tracking in one place to monitor the performance of the models.

This level of architecture, as can be seen in *Figure 2.3*, adds some more components into the mix, such as:

- Automated deployment services
- Shared model registry
- Uniform feature store
- Metadata store
- Machine learning pipeline orchestrator

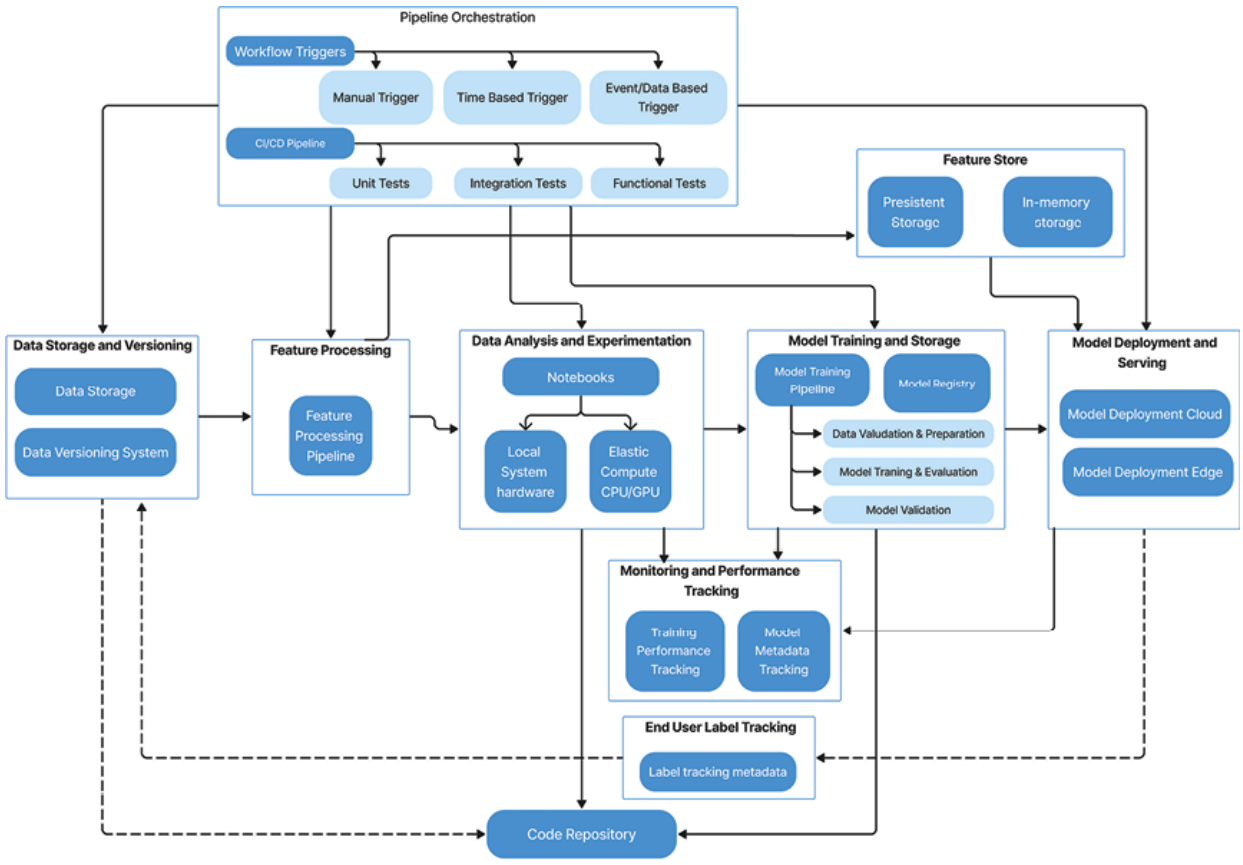

Figure 2.3: *Architecture Level 3: Enterprise grade MLOps*

Characteristics of level 3 architecture are defined as follows:

- **Development and experimentation**: The machine learning team regularly tries new algorithms and modeling where the experiment steps are orchestrated. These experiments result in new or improved code of the machine learning pipeline, which is then committed to a source code repository.
- **Pipeline continuous integration**: Continuous integration pipeline builds the source code and runs various tests before deploying. The output of the continuous integration pipeline generally is to run all the tests and formatters and make sure that the code is ready to be deployed to production.
- **Pipeline continuous delivery**: Continuous delivery is the pipeline to deploy the artifacts produced by the continuous integration stage to the desired environment and to make sure that this process is repeatable.

- **Automated triggering**: The pipeline for CI/CD can be set up to be automatically executed in production based on either the arrival of a new set of fresh data or on a specific time cadence. The output of this stage is a newly trained model that is pushed to the model registry.
- **Model continuous delivery**: This is the stage where we can serve the trained model as a prediction service automatically using the pipelines. The output of this stage is a deployed model which can be used for predictions.
- **Monitoring**: This is an important characteristic of this architecture level, where different statistics on model performance are collected based on the data going through the model. The output of this stage is not any artifact, so to speak, but a trigger to execute the pipeline or to execute a new experiment cycle based on monitoring the performance of the model or data drift.

Note: The preliminary data analysis is always a manual process for machine learning engineers before the pipeline starts a new iteration of the experiment. The model analysis step is another manual process.

Building a durable and scalable MLOps architecture and infrastructure is a prerequisite to transform the product or service offering with machine learning. Setting up the right level of MLOps architecture, depending on the stage the organization is in their use of machine learning, will ensure that the engineering, data science, and operations teams have a proper environment to enable the model deployments. Whether an organization is at the minimum viable architecture level, production or enterprise grade MLOps architecture level, the infrastructure components selected for the user will contribute greatly to the likelihood of success.

The semantics of dev, staging, and production

As we know, machine learning workflows include the following key assets:

- Code,
- Model, and
- Data

These assets need to be developed (dev environment), tested (staging) and deployed (production environment). For each stage, we also need to operate within an execution environment. Thus, all the preceding elements – execution environment, code, models and data need to be integrated into the dev, staging and production environments. They also need to work well with our MLOps architecture.

These divisions can best be understood in terms of quality guarantees and access control. Assets and services in production are always considered business critical which means it needs to have the highest guarantee of quality and strict control on who can modify them. Similarly, assets and services in the staging environment need to have a high level of access control similar to production to ensure a reliable and accurate representation of the execution environment. Conversely, dev assets are more widely accessible to people but offer no guarantee of quality.

For example, many data scientists will work together in a dev environment, freely producing dev model prototypes. Any flaws in these models are relatively low risk for the business, as they are separate from the live product. In contrast, the staging environment replicates the execution environment of production. The staging environment acts as a gateway for code to reach production, and accordingly, fewer people are given access to staging. Code promoted to production is considered a live product. In the production environment, human error can pose the greatest risk to business continuity, and so the least number of people have permission to modify production models.

One might be tempted to say that code, models and data each share a one-to-one correspondence with the execution environment; for example: all dev code, models and data are in the dev environment. That is often close to true but is rarely correct. Therefore, we will next discuss the precise semantics of dev, staging and production for execution environments, code, models and data. We will also discuss mechanisms for restricting access to each.

Note: There can be variations in these three environments, such as splitting staging into separate "test" and "QA" substages. However, the principles remain the same. We will continue to stick to a dev, staging and production environment in this chapter.

Execution environment

An execution environment is a place where models and data are created or consumed by code. Each execution environment consists of compute instances, their runtimes and libraries, and automated jobs. An organization could create distinct environments across multiple cloud accounts, multiple workspaces in the same cloud account, hybrid setup with dev in internal hardware, staging and production cloud environment and so on.

Code

Machine Learning project code is often stored in a version control repository, with most organizations using branches corresponding to the lifecycle phases of development, staging or production. There are a few common patterns. Some use only development branches (dev) and one main branch (staging/production). Others use main and development branches (dev), branches cut for testing potential releases (staging) and branches cut for final releases (production). Regardless of which convention is used, separation is enforced through Git repository branches.

Models

While models are usually marked as dev, staging or production according to their lifecycle phase, it is important to note that model and code lifecycle phases often operate asynchronously. That is, we may want to push a new model version before we push a code change and vice versa. Consider the following scenarios:

- The team develops a machine learning pipeline to detect fraudulent transactions that retrain the model weekly or bi-weekly. Deploying the code can be a relatively infrequent process, but each week a new model undergoes its own lifecycle of being generated, tested and marked as *production* to predict the most recent transactions. In this case, the code lifecycle is slower than the model lifecycle.
- To classify documents using large deep neural networks, training and deploying the model is often a one-time process due to cost. Updates to the serving and monitoring code in the project may be deployed more frequently than a new version of the model. In this case, the model lifecycle is slower than the code.

Since model lifecycles do not correspond one-to-one with code lifecycles, it makes sense for model management to have its own model registry to manage model artifacts directly. The loose coupling of model artifacts and code provides flexibility to update production models without code changes, streamlining the deployment process in many cases.

Data

Some organizations label data as either dev, staging or production, depending on which environment it originated in. For example, all production data is generated in the production environment, but dev and staging environments may have read-only access to them. Making data this way also indicates a guarantee of data quality - dev data may be temporary or not meant for wider use, whereas production environment data may offer a stronger guarantee of reliability and freshness. Access to data in each environment can be controlled with table access controls or Cloud storage permissions.

Table 2.1 summarizes the **Semantics**, which indicates that when it comes to MLOps, we will always have an operational separation between dev, staging and production. **Asset**s in dev will have the least restrictive access controls and quality guarantee, while those in production will be the highest quality and tightly controlled.

Asset	Semantics	Separated by
Execution environments	Labeled according to where development, testing and connections with production systems take place	Cloud provider or workspace access controls
Models	Labeled according to the model's lifecycle phase	Cloud storage permissions or model registry permissions
Data	Labeled according to its origin in dev, staging or production execution environments	Table access controls or cloud storage permissions
Code	Labelled according to the software development lifecycle phase	Git repository branches

Table 2.1: *Summary of different semantics and assets in MLOps*

Machine learning deployment patterns

As we looked at in the previous section, models and code can be managed and tagged separately with regard to the environment it belongs to. It results in multiple possible patterns for getting machine learning models from the staging environment and eventually into the production environment. There are two major patterns which can be used to manage model deployment. We will look at those patterns in the following.

These two patterns differ in terms of whether the packaged model is promoted through different environments, finally to production or the training code which can produce the model is promoted toward the production environment.

Deploy models

In this deployment pattern, a packaged model artifact is created by the use of training code in the development environment only. It is then used in all other environments. This model artifact, after being generated in the development environment, is then tested in staging on representative data for compliance and performance before it gets deployed to the production environment. This is the easiest pattern for data scientists to use. It also comes in handy in cases where model training is prohibitively expensive; training the model once and managing that artifact may be preferable. Level 1 architecture is something which can make use of this deploy model pattern. However, this simpler architecture comes with limitations. To train the data properly in the development environment, we need access to data in the dev environment which is a proper representation of the data we receive in the production environment. If this representative data cannot be made available in the development environment either for security reasons or any other compliance, this architecture may not be viable. This architecture does not naturally support automated model retraining. While we could think of setting it up in a way where we automatically retrain the model in the development environment, we would then be treating *dev* training code as production ready, which may not be a good idea. This option hides the fact that featurization, inference and monitoring code also needs to be deployed to production, requiring a separate code deployment path.

Deploy code

In this deployment pattern, the code is shared between the different environments. The code which is used to train the models is developed and finalized in the development environment, and this code is moved to staging and then eventually to production. Models will be trained in each environment before deployment using the data from that specific environment only. If an organization restricts the data team's access to production data from dev or staging environments, deploying code allows training on production data while respecting access controls. Since training code goes through code review and testing, it is safer to set up automated retraining. Code for featurization, inference and monitoring follow the same deployment path as model training code, which means all this code passes through integration tests in staging together. However, the pattern comes with a kind of learning curve for handling code and sharing it between different environments. Therefore, it is a good idea to use opinionated project templates and strict workflows.

In general, the **deploy model** approach works well for level 1 architectures. However, when moving to level 2 or 3 architecture, organizations should consider switching to the **deploy code** approach as that better respects the security and regulatory compliance of the organization. This approach also provides separation of responsibility between teams, which provides separation of access. For example, the machine learning team might not have access to the production data and thus are not able to train on that data, but in the code deploy approach, the training happens in the environment from the deployed code, so the model will be trained on the production data. In the deploy model approach, we need to create the exact representative data in the dev environment as per the data we received in production to make sure the final model works well in the production environment. Whereas the deploy code approach allows more flexibility as the model will be retrained in each environment based on the data available in that particular environment. So we need not worry about making sure the data in dev is the exact representation of production data and that it is properly segmented.

Nevertheless, there is no perfect process that covers every scenario. The preceding options outlined are not mutually exclusive. Within a single organization, we may find some use cases for deploying training code and other deploying model artifacts. The choice of the process will also depend on the business use case, maturity of the machine learning infrastructure,

compliance and security guidelines, resources available and what is most likely to succeed for that particular use case.

Bringing the architectural components together

So far, we have looked at all the components of the machine learning architecture, different levels of architectural maturity, and different aspects of execution environments and artifacts, as well as the deployment approaches. We are now ready to see how to put it all together and take a look at the general reference architecture for implementing MLOps.

Note: This is intended to cover most of the use cases and ML techniques, but it is not comprehensive.

We begin with an overview of the system end-to-end, followed by a more detailed view of the process in development, staging and production environments. The figures show the system as it operates in a steady state, with the finer details in the following figures.

This structure is summarized as follows:

As indicated in *Figure 2.4(a),* the code once ready and deployed in the development environment moves to the staging environment where it is deployed, tested and approved for the production deployment. The code is then finally merged into the production environment for deployment into the live production systems.

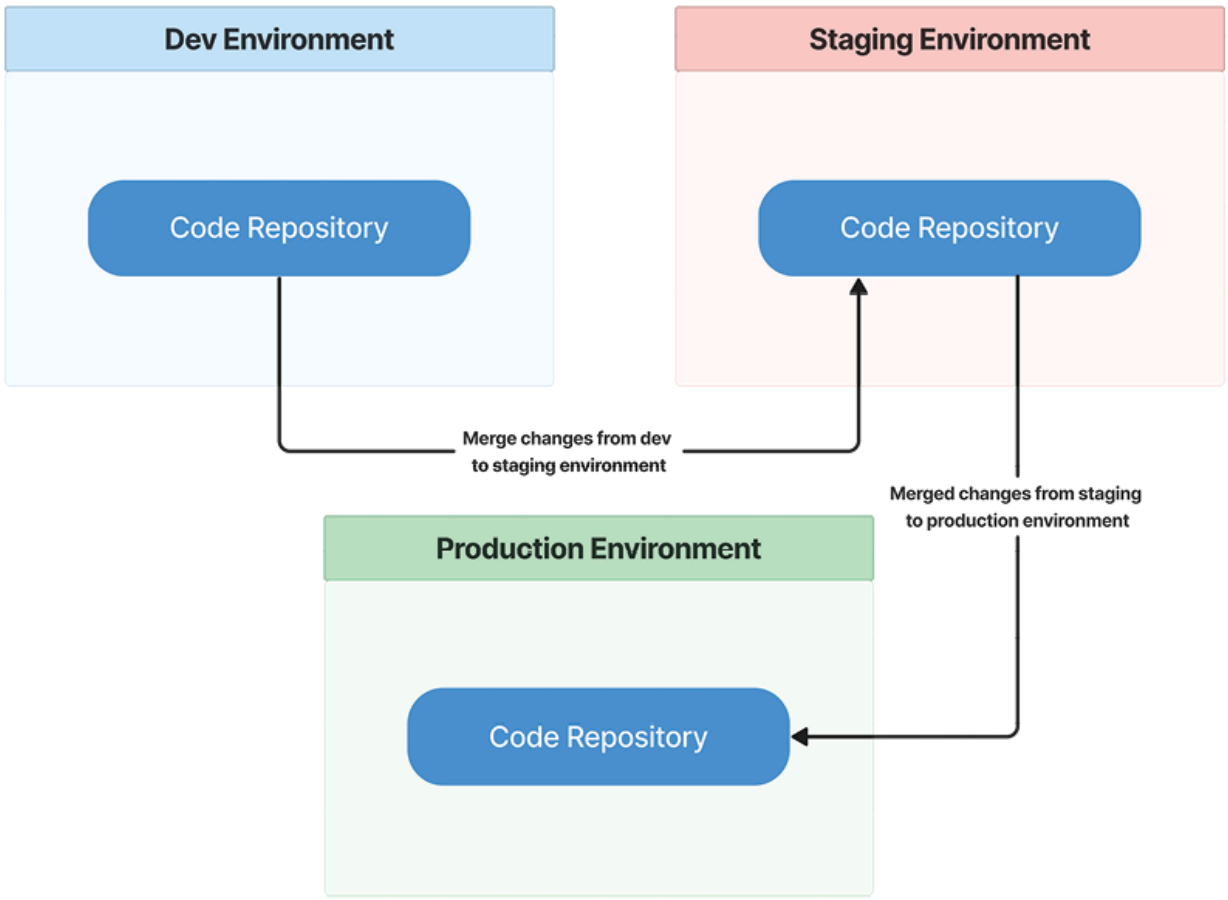

Figure 2.4(a): *MLOps reference architecture: Code movement in different environment semantics*

Now, let us combine the MLOps architecture in different environments with the deploy code approach and see how we can get everything to work together. *Figure 2.4(b), Figure 2.4(c) and Figure 2.4(d)* shows how the architecture will look when it is put together and up and running in different environments.

In the architecture, we illustrate an end-to-end process, from development data sources and initial data analysis all the way up to deployment and serving the model in the production environment. Following are the different steps which take us in this journey:

Development environment

Following are the steps that we generally perform in the development environment while working on the initial steps of the project:

1. Once we have the machine learning project planned, with the problem statement and objective defined, we can start the project.
2. Dev data sources are used, which are cleaned, versioned, and processed for initial data exploration and analysis.
3. Feature processing and experimentation are done in the **Dev Environment**, followed by **Model Training and Storage**.
4. This model is deployed in the **Dev Environment** and goes through Model Testing and Evaluation.
5. All this code goes to the **Code Repository** in the dev branch. This branch then gets merged into the staging branch.

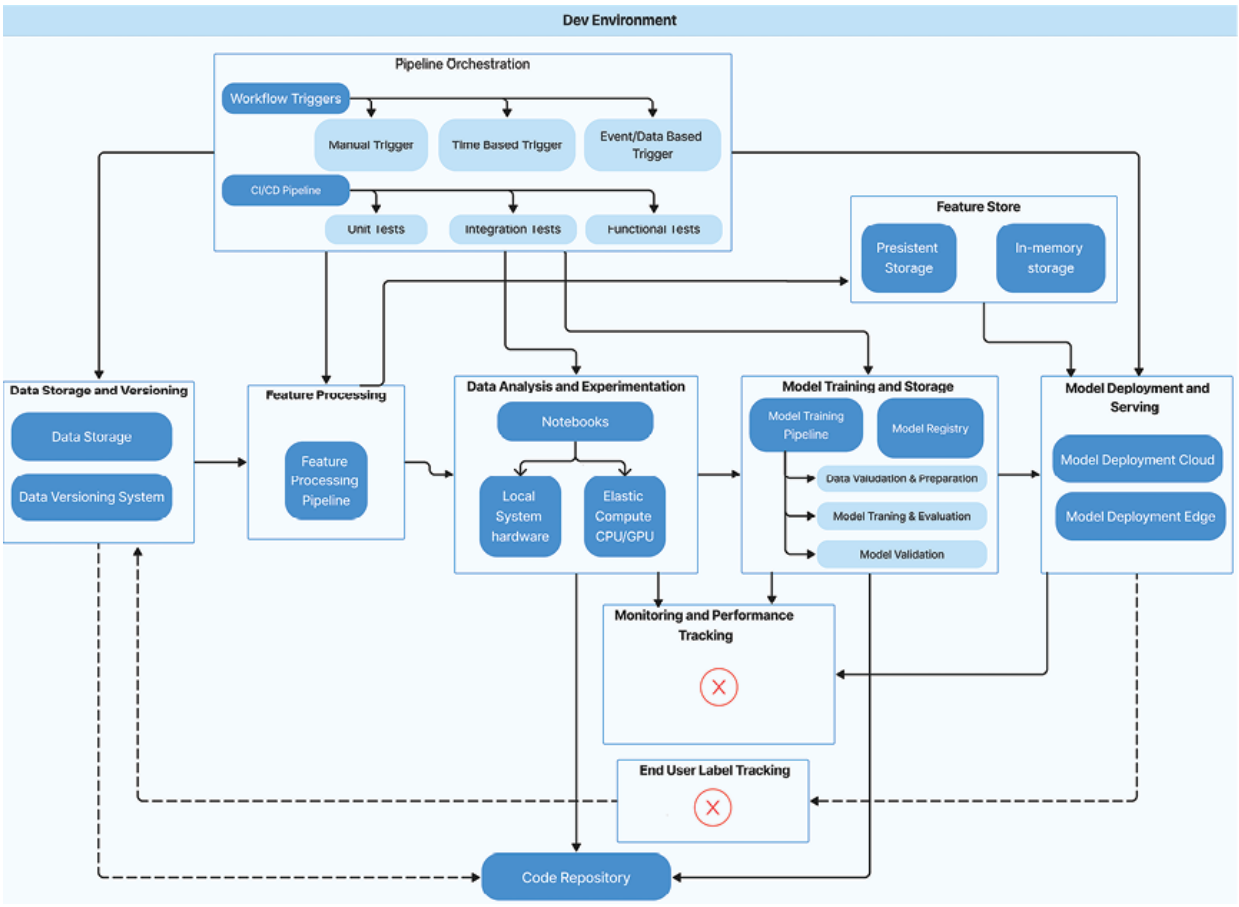

Figure 2.4(b): *MLOps reference architecture for development environment*

Staging environment

Once the code is merged into the staging branch, the following steps are performed in the staging environment:

1. Upon merging from the development branch to the staging branch, it triggers the CI/CD pipeline, which takes data from the staging environment storage, and processes the data, features and the model trained and stored on this data. This model is then deployed in the staging environment.
2. Ideally staging environment is the replica of the production, so it is a good place to test and QA the model.
3. Once we are satisfied and ready to deploy the model to the Production Environment, the code is merged from the staging branch to the main/production branch. Some of these merges may require extra approval depending upon the organization's policies of production code merges.

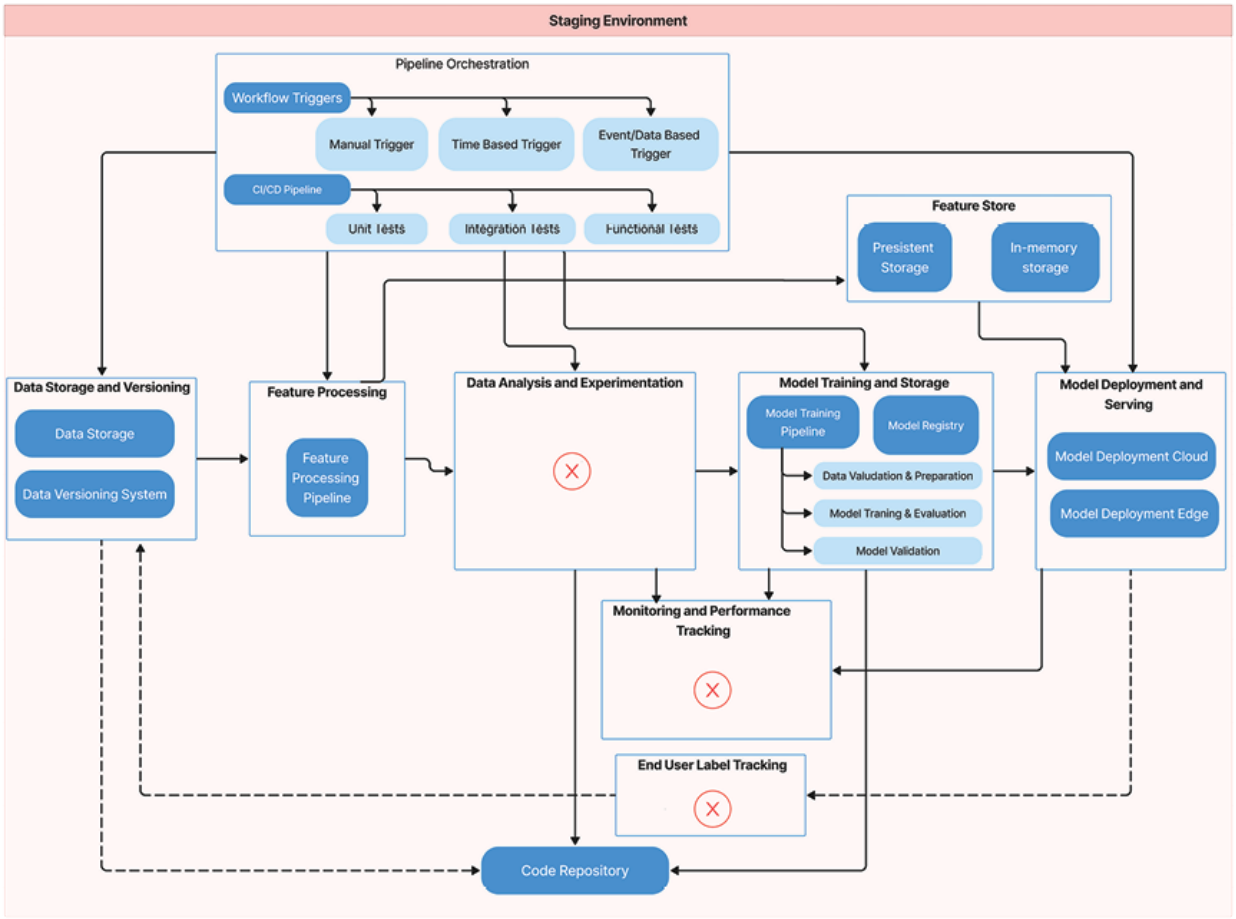

Figure 2.4(c): *MLOps reference architecture for staging environment*

Production environment

Finally, the following are the steps that are to be performed in the production environment once the mode moves from the staging to the production branch in the code:

1. Merging from the staging branch to the main/production branch triggers the CI/CD workflow in the Production Environment.
2. After data preparation, featurization, model training, evaluation, and validation based on the data in the production environment, the model is then deployed to the Production Environment to serve the end users.
3. Retraining of this model is done using the orchestration workflows depending upon the use case. It can be triggered manually, based on regular time cadence, or event-based triggers, which retrain the model based on new data arrival.
4. Monitoring is also an important component in the production environment. We need proper monitoring and observation setup in the production environment to keep us informed about the system performance in the live production environment.

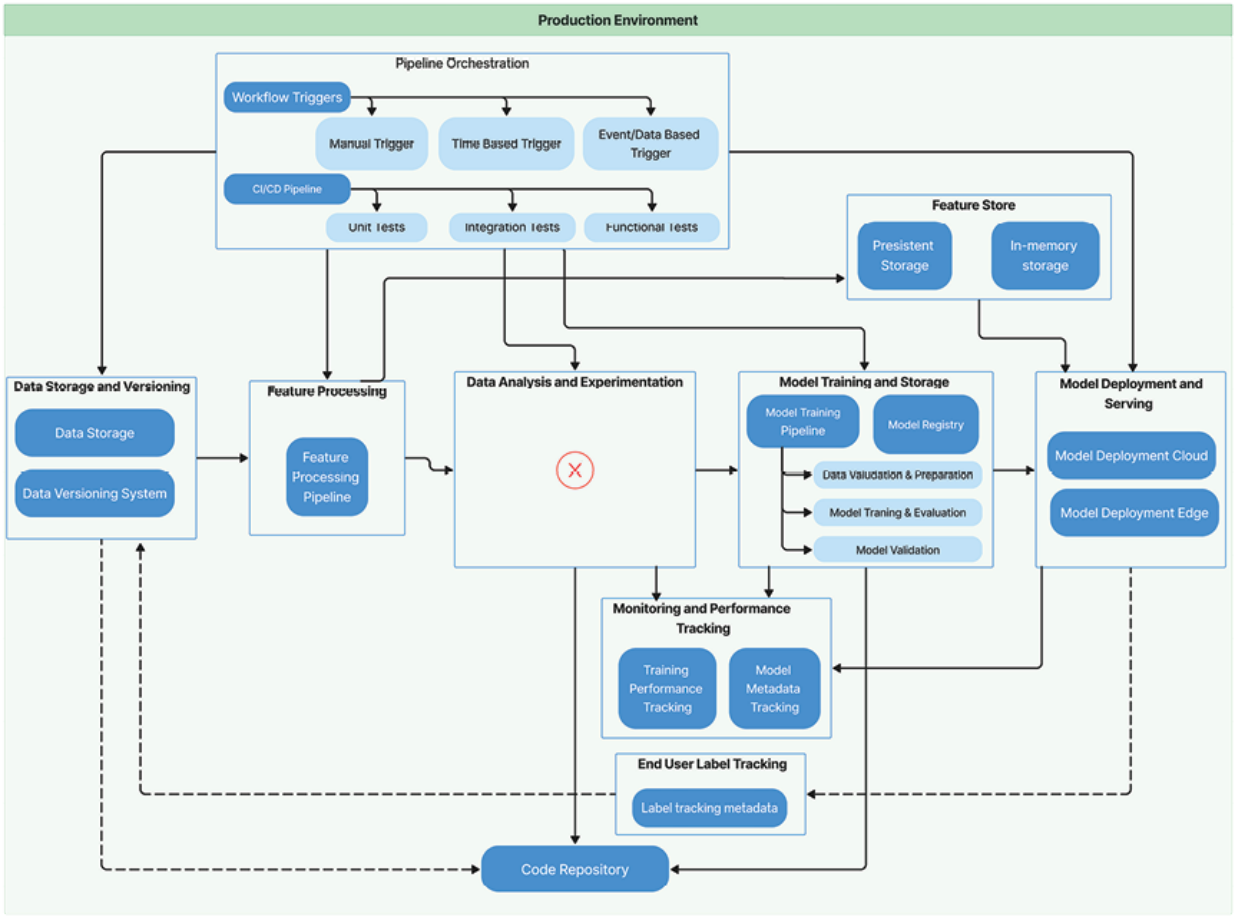

Figure 2.4(d): *MLOps reference architecture for production environment*

In *Figure 2.4(b), Figure 2.4(c) and Figure 2.4(d)*, you will notice some components are crossed in some specific environments. These are the components we either do not need in those specific environments or are optional:

- In the dev environment, monitoring and performance tracking, as well as end-user label tracking, are option components. We can skip those in the dev environment infrastructure.
- Similarly, in the staging environment, we can also skip monitoring and label tracking components. Another component not required in the staging environment is *Data analysis and experimentation,* as we do all the initial data analysis and experimentation in the dev environment. As a result, we come up with the scripts to train and deploy models. We do not need to perform the same experimentation in the staging environment.

- In the production environment as well, we do not require an infrastructure component for *Data analysis and experimentation* as we will never perform direct manual analysis and experimentation in the **Production Environment.**

Conclusion

This chapter covers a lot of moving parts of the MLOps architecture. We looked at all the components of the architecture individually. We learned that there are different levels of architecture based on the project requirements and infrastructure maturity of the organization. The semantics of different environments (dev, staging, and production), as well as deployment patterns, are also discussed. Ultimately, we looked at the bird's eye view of how MLOps architecture can look once we put all these pieces together and how the architecture will work with the deploy code pattern combined with the dev, staging, and production environment.

In the next chapter, we will look at machine learning systems, and how we can design machine learning-based systems that can be used as stand-alone systems or as part of a bigger enterprise ecosystem powered by machine learning systems. We will also go over the end-to-end workflow of designing machine learning systems and some of the industry standard best practices.

Points to remember

- MLOps aims to enable collaboration, automation, scalability, and reproducibility of machine learning workflows. It helps manage and operationalize ML models in production.
- Major components of MLOps architecture include data sourcing/versioning, data analysis, code repositories, pipeline orchestration, model training/storage, deployment, monitoring, and feature processing.
- There are three levels of MLOps architecture maturity: Level 1 for proof of concepts with manual processes, level 2 for production-grade with automated pipelines, and level 3 for enterprise-grade with shared infrastructure.

- Execution environments, code, models and data have different semantics in dev, staging and production environments. They are separated by different levels of access control to the team.
- Two main model deployment patterns are deploying packaged models versus deploying training code through environments. Deploy code is better for automation, security, and compliance.
- The end-to-end architecture integrates all components across dev, staging and production environments. CI/CD pipelines automate deployment upon branch merges.
- Retraining pipelines can be triggered manually, on schedules or by events like new data arrival. Monitoring helps track model and data drift.

Key terms

- **MLOps architecture**: MLOps architecture aims to enable seamless collaboration, automation, scalability, and reproducibility of machine learning workflows, ensuring efficient management and operationalization of ML models in the production environment.
- **Data warehouse**: A data warehouse is a central repository of information that can be analyzed to make more informed decisions. Data flows into the data warehouse from transactional systems, relational databases, and other sources, typically on a regular cadence. This data then serves as the central repository for downstream systems like ML or BI systems.
- **Data lake**: A data lake is a central repository of unstructured or unprocessed data that allows us to store all kinds of structured and unstructured data at any scale. Data can be stored, without having first to structure the data.
- **Workflow orchestration**: Workflow orchestration is the process of automating the steps of a workflow in a way that respects the orchestration rules and business logic.

- **Directed Acyclic Graphs** (**DAG**): They are graphs without any directed cycles. These graphs represent the execution order and artifact usage of single workflow steps. In a directed acyclic graph, it is impossible to start at one point in the graph and traverse the entire graph. Each edge is directed from an earlier edge to a later edge.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 3
MLOps Infrastructure and Tools

Introduction

Machine Learning (**ML**) systems are intricate, with numerous interconnected components, so they can benefit from a well-structured infrastructure. When designed correctly, this infrastructure can streamline processes, minimize the reliance on specialized expertise, and save quite a bit of engineering time. As a result, ML applications can be developed and deployed more rapidly, the likelihood of bugs is reduced, and enable new use cases. Conversely, a poorly designed infrastructure can be challenging and costly to modify or replace.

In this chapter, we will delve into how to setup a good infrastructure for ML systems. Before we start, it is crucial to acknowledge that each organization's infrastructure needs will vary according to its use cases, existing IT infrastructure, and application dependencies. As we discussed in the previous chapter, an organization's maturity level and other requirements influence its infrastructure needs as well. While we will discuss the generic and common components of a well-designed ML infrastructure, it is important to remember that most of these components can be adapted, replaced, or adjusted to suit any organization's unique needs.

By prioritizing data storage and management, compute resources, model training and experimentation, model deployment and serving, monitoring, and automation, we can create a robust and adaptable infrastructure.

Structure

In this chapter, we will discuss the following topics:

- Getting started with the infrastructure
- Storage
  - Extract, transform, load/extract, load, transform
  - Batch processing and stream processing
- Compute
  - Public cloud vendors versus private data centers
  - Development environments
- Containers
- Orchestration/ workflow management
  - Airflow installation
- Machine learning platforms
  - Model deployment
  - Model registry
  - Feature store
  - Installing MLflow
- Build versus buy

Objectives

By the end of this chapter, we will have an understanding of what tools and infrastructure setup is required to run MLOps efficiently, and how to setup different environments to move the models throughout the project more efficiently from development to production environment. We will also learn about setting up an orchestration platform to control all the pipelines and transitions. And, finally, when it makes sense to buy versus building the infrastructure in-house.

Getting started with infrastructure

To set up the right infrastructure for our use case, it is important to understand what infrastructure means and what it consists of.

In machine learning, infrastructure is the set of fundamental facilities that support the development and maintenance of ML systems. It is important to consider that the *fundamental facilities* vary greatly from company to company, as discussed earlier in this chapter. In this section, we will examine the following layers of the infrastructure:

- Storage
- Compute
- Containers
- Development environment
- Orchestration/Workflow management
- ML platform
- Model deployment
- Model registry
- Feature store

These various layers of infrastructure range from the least abstract to the most abstract from the perspective of the machine learning engineer or data scientist. As depicted in *Figure 3.1*, these layers fall on different points of the abstraction versus importance spectrum. For instance, storage and computing are the most abstract for engineers, as they do not need to directly interact with these layers once they are operational. In contrast, the development environment is the least abstract and most crucial for engineers, as they use it daily. Refer to the following figure:

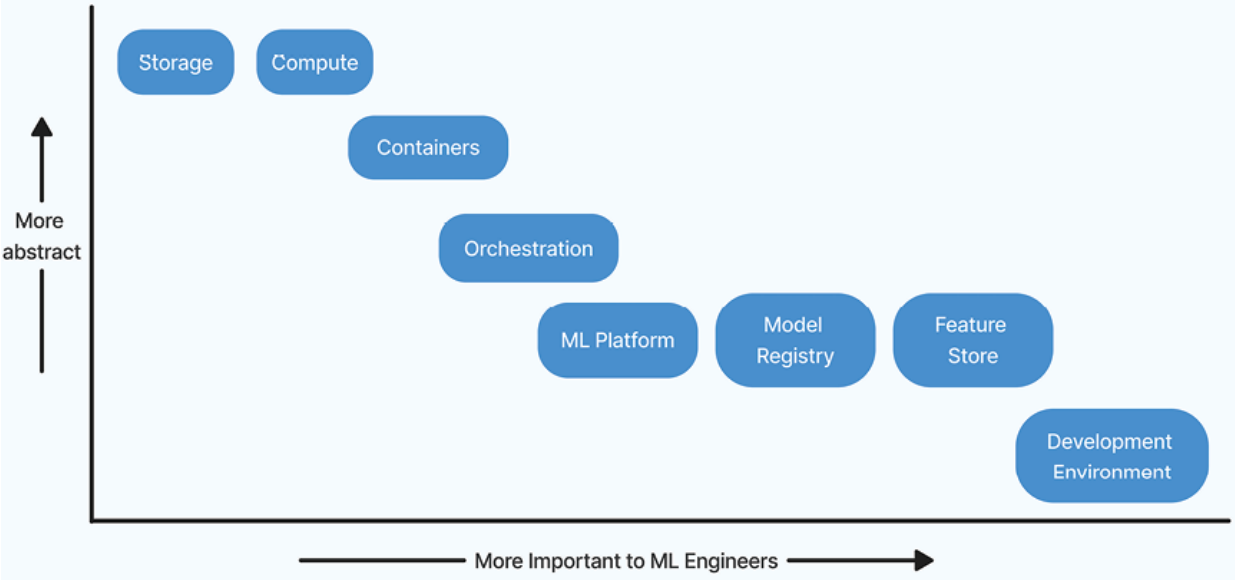

Figure 3.1: *Different layers of infrastructure on the scale of abstraction and important*

The infrastructure can differ significantly among organizations and use cases, depending on the choices made during implementation, as well as the technology stack selected for each category. It also depends upon whether we decide to buy or build in-house for each of these components.

Machine learning infrastructure is a critical component in the development and maintenance of ML systems, with various layers that serve different purposes. Understanding the intricacies of these layers and how they interact is vital for machine learning engineers and data scientists, as well as organizations, to ensure the efficient operation and management of ML systems.

Storage

As we know, the machine learning system often works with many datasets from various sources. These datasets can be quite large and need to be stored properly to ensure easy accessibility, effective management and sharing. This is where the storage layer of the ML infrastructure plays a crucial role in addressing these needs. Storage layer can be implemented in several ways, including:

- **Local hard drives in individual systems**: It must be noted that this provides limited scalability.

- **Centralized Cloud-based object storage services such as AWS S3**, **Google Storage**, **Azure Blob Storage**: These provide virtually unlimited scalability and are commonly used for backup, archival, and intermediary storage. Object storage is ideal for storing large data as objects or files.
- **On-premises servers or private data centers**: This provides more control but requires infrastructure management.
- **Cloud-based data warehouses like Snowflake**, **Databricks**, **BigQuery**, **Redshift, etc:** These are optimized for analytical workloads and provide SQL querying over structured/semi-structured data.
- **Cloud-based data lake solutions like AWS Lake Formation and Azure Data Lake Storage**: These can handle structured, semi-structured, and unstructured data in native formats.

In addition to selecting appropriate storage engines, we need mechanisms for data versioning and lineage tracking. This allows us to maintain data quality and trace the evolution of datasets from source to destination. Popular versioning approaches include assigning a unique ID to each dataset and maintaining a log of dataset changes.

Datasets used in machine learning can have different characteristics, be used for different purposes, and may require different processing methods. For example, timeseries data from sensors, image data, natural language text, genomic sequences, etc. They often come in different formats, which can make storage challenging. In fact, they will be in different formats; thus, storing this data is not always straightforward.

It is also essential to consider how the data will be used in the future and make reasonable assumptions to ensure that the chosen storage format and patterns are appropriate. In many cases, data must be transferred from these various sources into a centralized storage, such as object storage, data warehouse, **Relational Database Management Systems** (**RDBMS**), or a data lake capable of handling structures, semi-structured or unstructured data.

Data storage can involve serializing and de-serializing the data object files and storing them in object storage engines like AWS S3, Google Storage,

and Azure Blob storage. This storage is often used as archival, backup, or intermediatory storage when the models are trained, or data is transferred between systems. The other common storage is storing data in storage engines like databases. Typically, there are two types of workloads the databases are optimized for, transactional processing and analytical processing, with a significant difference between them.

Online Transaction Processing (**OLTP**) databases handle transactions generated by actions like posting on social media, ordering food online, or sending emails. These transactions are inserted into the database as they occur and are occasionally updated as needed. OLTP databases are typically row-major, meaning that each transaction is processed as a separate unit. OLTP systems are optimized for fast queries on individual records.

On the other hand, **Online Analytical Processing** (**OLAP**) databases are designed for aggregating and grouping data to draw conclusions or perform analyses. These databases allow users to query data from multiple perspectives and are optimized for processing large volumes of data in columns across multiple rows.

Overall, the storage aspect of the MLOps stack must provide scalable, efficient and convenient access to a variety of datasets while also maintaining data integrity, security and traceability. This is crucial for operationalizing machine learning applications successfully.

Extract, transform, load/extract, load, transform

Extract, transform, load (**ETL**) and **extract, load, transform** (**ELT**) are critical processes in preparing data for machine learning model development and deployment. They play an integral role within the MLOps workflow.

Before data can be used to train machine learning models, it often must be extracted from various sources; most of the time, it then needs to be transformed from one format to another and loaded into destination storage. This process is called **ETL** or **ELT**, based on which sequence the transform and load components are planned. ETL/ELT provides a way to pull this disparate data into a unified location and transform it into a cleaned, structured, and analysis-ready form.

In an ETL approach, data is first extracted from diverse sources. Then, data transformations and cleansing occur before the data is loaded into the **target**

database or **data warehouse**. With ELT, the order is reversed. Data gets extracted from sources and immediately loaded into the target system before transformations occur.

Whether ETL or ELT is used, the goals are the same: prepare quality, consistent data that can feed into the machine learning model building and training process. The outputs from the ETL/ELT pipeline serve as the inputs for ML engineers and data scientists to leverage when developing, training, and retraining models.

These processes are not exclusive to machine learning and are typically handled by data engineering teams, but they remain critical for the success of any ML project. However, it is a critical dependency for MLOps. Without properly curated and prepared data, machine learning models will suffer from garbage-in-garbage-out and fail to deliver robust, accurate predictions. A broken ETL/ELT pipeline will block new model iterations since fresh data will be unavailable.

By carefully designing ETL/ELT architecture to align with the specifications of ML teams, data engineers provide the critical data infrastructure needed for successful MLOps model development, deployment, monitoring, and updating. ETL/ELT is thus an integral component within the end-to-end MLOps workflow.

Batch processing and stream processing

Batch processing and stream processing are fundamental concepts in big data pipelines that have significant implications for MLOps. The choice between batch and streaming data ingestion and processing impacts the design of the overall machine learning architecture and the tools used in the MLOps stack.

As the name indicates, batch and stream represent how incoming data is processed. Batch processing involves processing data in batches at regular intervals (for example, hourly, daily, weekly, monthly, and so on). It is well-suited for workloads not requiring real-time responses, such as periodic retraining of ML models on new data. Batch processing enables robust data pipelining and orchestration using workflow schedulers like **Apache Airflow** and allows for efficient resource utilization through job

parallelization. For model training, batch processing is preferred as it allows for complete passes over large datasets.

In contrast, stream processing involves transferring data from the source to the destination in real-time or near real-time as it becomes available. Stream processing is essential for computing features that change rapidly and require low-latency predictions or decisions in machine learning. For example, the number of players playing the game, the number of tournaments running at the moment, which needs players to join in, fraud detection, IoT data analysis, and so on. For most modern machine learning systems, we do not only need batch features but stream features as well to make optimal decisions in time.

The batch versus streaming design choice has cascading effects on the MLOps stack. Batch workloads lend themselves to containerized microservices that handle discrete jobs. Streaming needs fast, stateful, and flexible data processing which is enabled by serverless frameworks like **Knative**. Data storage architectures also vary, with batch processing learning towards data warehouses, while stream processing requires message bused and pub-sub systems. Monitoring, testing, and CI/CD processes must also account for these architectural differences.

For ML systems that leverage streaming features, the streaming computation is rarely simple. These machine learning applications, like fraud detection, language models, and so on, can have features not just in hundreds but thousands and possibly even more. Extracting sophisticated features from data streams frequently calls for implementing high-performance stream processing engines. Tools like **Apache Flink**, **KSQL**, and **Spark Streaming** are commonly used for this purpose, with Apache Flink and KSQL being more widely recognized and offering a user-friendly SQL abstraction for data scientists.

In summary, batch and stream processing underpin many important machine learning use cases. By considering their implications early on, MLOps architecture can be tailored to optimally support the required data processing patterns for a given ML application. The batch versus streaming design choices impact nearly all aspects of the MLOps stack.

Compute

The compute layer is the powerhouse for executing all tasks required in a machine learning project. It encompasses all the compute resources an organization can access and the methods employed to utilize these resources for the projects. The availability of compute resources heavily influences crucial decisions in machine learning projects, such as the choice of models to use, the extent of processing, the requisite parallelization, the scalability of the solution, and so on.

For example, the amount of compute resources available can dictate the size and complexity of models that can be practically trained and deployed. Resource-intensive models like large transformer networks may be infeasible without access to distributed GPU clusters or cloud-based compute. The possible scale of data processing and feature engineering is also tied to compute availability. With greater resources, more data can be processed in parallel during pre-processing pipelines.

Furthermore, the ease of scaling compute up and down affects the efficiency of the overall MLOps lifecycle. If training compute can be provisioned and decommissioned dynamically based on workload, it avoids situations where large allocated resources sit idle. Similarly, efficient autoscaling of deployment compute resources to match real-time inference demand improves cost-effectiveness. Having flexible compute infrastructure is key to maximizing utilization.

In its simplest form, the compute layer could consist of a single system or a server, a single CPU or GPU. Conversely, it could entail clusters of cloud compute resources such as **AWS ECS** managed by the cloud providers or **Kubernetes** (**K8s**) cluster infrastructure running numerous CPU and GPU cores delegated based on the requirements. While cores, memory, and storage are the fundamental units of compute resources, more mature architecture often employ additional abstractions. For instance, computation engines like **Apache Spark**, Ray uses the term *job* as a unit, while Kubernetes uses *pod* as a unit. To execute any job, the necessary data must be led from the storage layer into the compute unit's memory, followed by executing the required operations on the data.

Public Cloud vendors versus private data centers

Like the data storage layer, the compute layer has become largely commoditized. Instead of establishing their data centers for storage and

compute, organizations can pay cloud service providers such as AWS, Azure, GCP, or other specialized cloud vendors for the precise amount of compute power needed. This approach offers the advantage of building projects without worrying about the compute layer. It is particularly beneficial for organizations with fluctuating workloads depending on the projects they are working on.

For instance, if an organization typically requires 50 CPU cores but a specific project needs 200 extra CPU units for a week of experimentation, cloud providers make it easy to set up the necessary resources and discontinue them once the project is completed. We will only pay for the resources we use and for the duration of use. In contrast, private data centers would be a more expensive undertaking, as we would need to pay upfront for 200 CPU cores, as well as set them up for the project. This flexibility is helpful in machine learning as these workloads tend to be dynamic in terms of resource needs, with resource consumption being significantly higher during the development phase when numerous experiments are conducted, compared to a more consistent workload during the production phase.

However, the choice between public cloud and private data centers is not always binary. Many organizations adopt a hybrid cloud approach, maintaining some on-premises infrastructure while utilizing public cloud resources. This provides the ability to keep sensitive data and core services on-premises while leveraging the flexibility and scalability of the public cloud for temporary resource needs. Managing a hybrid MLOps infrastructure brings additional complexity in coordinating across environments and moving data and models between them. Organizations must weigh factors such as data gravity, latency, security, and compliance requirements when designing a hybrid cloud architecture. With careful planning, a hybrid approach can enable organizations to optimize for both cost efficiency and business requirements.

Development environments

Development (**Dev**) environment, as we are familiar with, is the environment where we write code, conduct experiments and interact with the production environment, where champion models are deployed, and challenger models are evaluated. The development environment typically

consists of **Integrated Development Environment** (**IDE**), **Version control systems** (**Github** and so on), and **CI/CD pipelines**.

Development environment setup

The development environment should be set up with all the necessary tools to make it easier for engineers to do their job. Additionally, it should also consist of tools for versioning code as well as data.

In *Chapter 4, What are Machine Learning Systems?* we will examine the standardized cookie cutter project structure for the data science projects, which provides us with a guiding rail as to how we can structure the projects in our dev environment so that it works well with the code versioning systems.

In *Chapter 5, Data Preparation and Model Development*, we will discuss other essential aspects of the development environment. We will delve into the best practices for code repositories, particularly in the context of machine learning and MLOps projects. Additionally, we will explore the use of pre-commit hooks and data versioning in more detail.

A well-configured development environment should also include a CI/CD and test suite to assess the code before it is pushed to the repository or other higher environments, such as **staging** and **production**.

Integrated development environments

IDEs, as we know, are the code editors where we write our code. Many machine learning practitioners not only use IDEs like **VS Code** and **PyCharm** but also prefer notebooks like **Jupyter Notebooks** or **Google Colab**, especially for exploratory data analysis.

Notebooks have a good property that other native IDEs like VS Code miss: statefulness. This means that notebooks can retain states between multiple runs. If a script fails halfway through execution, we can resume and rerun from the failed step onwards instead of starting from the beginning. This feature is particularly useful when dealing with large datasets that might take a long time to load. Notebooks also facilitate iterative development by allowing code execution in any order.

However, it is crucial to remember that statefulness can be a double-edged sword, as it permits the execution of code in the cells in any order. So, if the

notebooks are not properly structured to run the code or the order of the cell execution is not well documented, they can become disorganized and challenging to manage.

To avoid these pitfalls, it is crucial to adopt the following good practices when using notebooks:

- Thoroughly document the purpose and dependencies of each cell.
- Logically order cells to reflect workflow steps.
- Restart kernels and run all cells in order before committing changes.
- Modularize reusable logic into functions/classes.
- Use version control and regular code reviews.

Following these principles makes notebooks more robust and production-ready. The benefits of rapid iteration can be harnessed without sacrificing code quality or maintainability.

With proper structure and documentation, notebooks can be an integral part of a scalable MLOps infrastructure. IDEs remain essential for production models and pipeline code. Using both tools effectively is key to productive machine learning development.

Containers

While any machine learning project is in the development phase, we typically work on a single system or perhaps a few individual systems. We have those configured according to the requirements, including the necessary dependencies, packages, and libraries. When the project is ready for deployment and going into the staging or production environment, we may initially set up a server based on all those dependencies once, and it will. However, this approach falls short when addressing workloads that must scale according to the demand. That means if we have a couple of server instances running, processing the data, and serving a model, and suddenly, during some events, the workload on the application increases ten times, we need to scale our server instances to handle the processing requirements. All modern infrastructure systems can manage this scaling out and scaling in instances based on the demand. However, to make it possible, we need a

method to recreate the same project setup in these on-demand containers as we may have done in the initial server instances.

This is where container technology, such as **Docker**, becomes invaluable. There are multiple container technologies available, but Docker is the most popular one. It provides a platform for creating a Docker file that includes a detailed, step-by-step guide on reconstructing the environment necessary for the efficient execution of a machine learning model. This process involves setting up environment variables, navigating to specific directories, and performing other essential tasks to establish an optimal operational setting. These instructions enable newly spun-up instances to be prepared and run the code efficiently.

Two key concepts in Docker are the **image** and **container**. Executing all the instructions in a Docker file results in a Docker image. When we run this Docker image, we obtain a Docker container. We can think of a Docker file as the recipe to construct a mold, which is the Docker image. From this mold, we can create multiple running instances; each is a Docker container.

Similar to how we have Git, a container registry serves a similar purpose where we can share a Docker image or find an image created and shared by others. Registries can be public or private. Most organizations that use Docker registries use private registries to store and share the propriety images within the organizations. Docker hub is an example of one such registry; a private repository can also be set up in AWS **ECR** (**Elastic Container Registry**) for organizational use.

A Docker file is the code file that defines how a Docker container can be used as a base for our projects. Following is the example of a simple Docker file:

```
FROM pytorch/pytorch:latest


RUN git clone http://github.com/dummy-
organization/new-ml-project.git


WORKDIR /new-ml-project
```

```
RUN git clone
https://github.com/huggingface/transformers.git &&
\

    cd transformers && \

    python3 -m pip install –no-cache-dir.
```

The previous code is an example of a Docker file which runs the following instructions:

1. Download the latest **PyTorch** base image from the public registry.
2. Clone the **`new-ml-project`** from the git account of **`dummy-organization`** (these are dummy names and the repository and project does not actually exist).
3. Set **`new-ml-project`** as the working directory.
4. Clone transformers from Hugging Face's GitHub repository into the working directory, navigate to the transformers folder and install transformers.

This works great for setting up one kind of container, but in reality, most real-life projects may have multiple containers. Perhaps each step of the pipeline runs in its container, model is server in its separate container, and so on. Now let us assume we have numerous microservices running, managing Docker files for each of those containers will be tricky. This is where **container orchestration** comes in handy. **Docker Compose** is a lightweight orchestration tool that can handle managing multiple containers easily. Docker Compose works well on a single host. However, if we need to run these containers on multiple different hosts, it is not possible with Docker Compose. This is the requirement that leads most organizations to the use of Kubernetes. It is a tool that enables us to run containers on multiple different hosts as a cluster; it creates a network for containers to communicate and share resources on a cluster. This adaptability allows for the seamless deployment of additional containers during peak demand while simultaneously deactivating containers when they are no longer necessary. Furthermore, these capabilities contribute to maintaining the system's high availability, ensuring optimal performance at all times. However, Kubernetes

is not the most data scientist-friendly tool, as it needs significant DevOps and infrastructure work to be made available and managed.

Orchestration/workflow management

Orchestration is a crucial layer in the overall resource management of any modern data platform. As previously discussed, storage and compute resources are elastic for every modern data platform, meaning they can be dynamically adjusted based on the demand or the workload of the project. To manage these resources effectively, we require schedulers or orchestration platforms.

In our world of data, the terms *orchestrators* and *schedulers* are often used interchangeably, as we typically run orchestration platforms to manage our resources and schedules of the workflows. However, there is a noteworthy distinction between these two concepts. Orchestrations, specifically, are concerned with managing the resources and how and where to get those resources and their infrastructure. Kubernetes is a prime example of a widely-used infrastructure orchestrator. On the other hand, schedulers focus on determining when to execute jobs and the resources needed for their completion.

In the context of machine learning, *we generally are not directly concerned with infrastructure orchestrators, and when we say orchestrators, we are referring to data-specific orchestrators like Apache Airflow, Argo, Prefect and Dagster, Kubeflow, Metaflow and so on*. These platforms are capable of orchestrating resources for the jobs and have their schedulers as well to manage those jobs. They handle the scheduling and coordination of ML workflows, while replying on infrastructure orchestrators to provide the underlying compute resources. In some literature and also in organizational teams, these tools and platforms are also referred to as workflow management tools.

Two primary attributes of ML workflows that impact their resource management are:

- Iterative nature
- Dependencies

As we have established, building ML systems is an iterative and repetitive process. For example, training and evaluating multiple models with different hyperparameters involves iterative experimentation. While the development process may involve numerous iterations, training, and re-training must also be periodically carried out, such as on a weekly or monthly basis or dynamically when model performance declines and a model drift is observed. Workflow management tools like Airflow allow configuring pipelines to execute iterative workflows and periodic re-training in a reliable and reproducible manner.

Furthermore, the comprehensive ML pipeline typically has quite a few dependencies. There could be jobs in the workflow that should only be executed if the preceding job was successful, creating a dependency. For example, an initial step might involve fetching the data from data warehouse followed by its feature extraction. Now executing the second job only makes sense if the first one is completed successfully and the data is available. Often, new models are trained, compared with the existing production model, and updated only if the performance of the new model surpasses the performance of the previous one. This type of dependency, known as a conditional dependency, determines the next step based on not only the success of the previous step but also on its outcome. Again, workflow management tools like Airflow allow encoding these dependencies to orchestrate conditional execution of pipeline steps.

As depicted in *Figure 3.2*, a machine learning engineer creates a **Workflow** that consists of task definitions. These definitions spin up the individual tasks, and a **Scheduler** manages their scheduling which then requests the required resources from an **Orchestrator**. This orchestrator then allocates resources from the **cluster instance pool** to execute each task. Please refer to the following image:

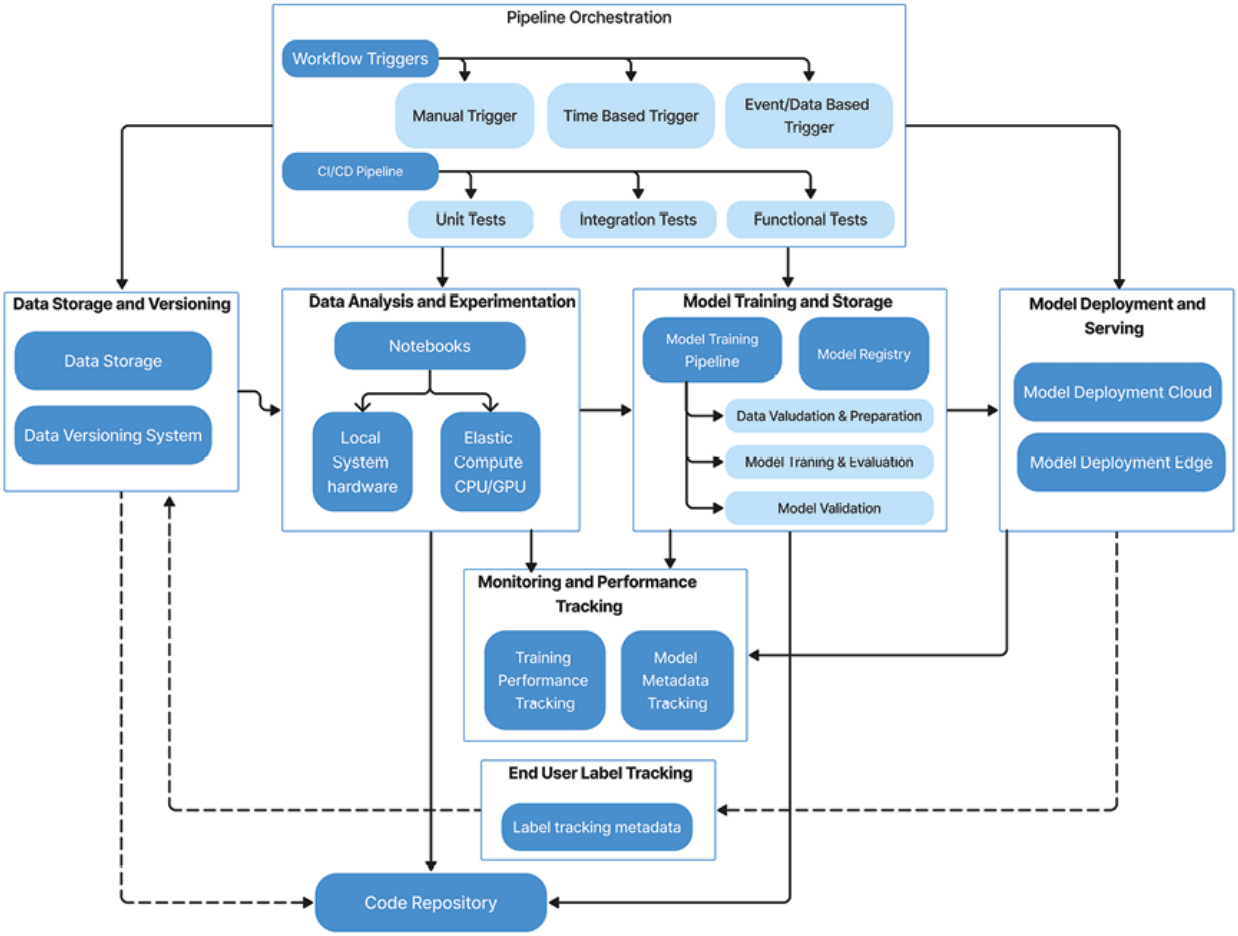

Figure 3.2: *Data science workflow steps*

One of the most widely used workflow management and orchestration tool is Apache Airflow. Initially created at **Airbnb** and launched in 2014, Airflow is a remarkable task scheduler equipped with a comprehensive library of operators. This feature simplifies the integration of Airflow with a wide array of cloud services, database systems, storage solutions, and more. In the context of MLOps, Apache Airflow can automate, orchestrate, and monitor ML pipelines including data ingestion, preprocessing, feature engineering, model training and evaluation, comparing model experiments, model deployment, and monitoring model drift. Its scheduling capabilities handle dependencies between pipeline steps and iterative workflows, while integration with infrastructure orchestrators like Kubernetes allows it to tun ML pipeline tasks at scale.

Airflow installation

Let us look at how we can install Airflow and run it for our orchestration and scheduling needs. Numerous methods are available for configuring and installing Airflow, depending on the what kind of environment we want to run it on. On local systems, it is fine to run airflow with python `pip` installation for development purposes, or using Docker if you are familiar with the Docker and containers. For testing environments also, it makes sense to run airflow using Docker. For production, the preferred method of running Apache Airflow is Kubernetes cluster using Helm Charts.

Installing using PyPi

Assuming we already have Python and `pip` installed, installing Airflow becomes easier. Following is the command to install Airflow:

```
pip install "apache-airflow[celery]==2.5.3" -constraint
"https://raw.githubusercontent.com/apache/airflow/constraints-
2.5.3/constraints-3.7.txt"
```

Airflow is both a Python library and an application, so sometimes it becomes tricky to manage dependencies. Libraries usually keep the dependencies open, and applications usually pin them. Airflow community decided to keep dependencies as open as possible so that we can install different versions of the libraries, if required, or keep the default if that works for us. That is why, if you look at the installation command, we define the constraints file for the version, which is specific for all the dependencies and their version. We can at any time, use our own modified dependency file, if needed.

Upon successfully installing Airflow, we can execute the following command:

```
# The standalone command will initialize the database, make a user,
and start all

# components for us.

airflow standalone
```

Once it is up and running, the terminal will display the login user and password we can use to login to the UI. Visit `localhost:8000` in the browser and login with the provided credentials.

Out of the box, Airflow uses **SQLite** database, which is fine for the local development or experimentation but if we outgrow that we should look into using **PostgreSQL** or **MySQL** as a backend database.

Following is an example of an Airflow **Directed Acyclic Graph** (**DAG**) created for Airflow v2.0+ using TaskFlow APIs. Taskflow APIs were launched in v2.0, before that the structure of writing the DAGs was completely different. It is therefore important to make sure we are installing the latest version in our infrastructure.

Installing in Docker

This setup assumes that we are familiar with Docker and its functioning. Moreover, we must have Docker and docker-compose installed in our system.

First of all, we will need to fetch the docker-compose file. We can either download the file using `curl` or manually by visiting Airflow documentation. The command to download the file is:

```
curl -Lf0 'https://airflow.apache.org/docs/apache-airflow/2.5.3/docker-compose.yaml'
```

Looking at this docker-compose file, we will notice that there are multiple containers Airflow will need to run mentioned as follows:

- **Airflow scheduler**: It monitors all the tasks and DAGs, in addition to managing when and how task instances are triggered.
- **Airflow webserver**: The webserver is available for us to interact with Airflow instance.
- **Airflow worker**: The component responsible for carrying out the tasks scheduled by the Airflow scheduler.
- **Airflow trigger**: The mechanism that initiates an event loop specifically designed for tasks that can be deferred.
- **Airflow init**: It is the command which initializes the Airflow service.
- **Postgres**: Postgres is the standard databased used for Airflow which stores all its metadata as well as the database to run Airflow.

- **Redis**: It is the broker that manages the communication between the scheduler and worker.

This Docker installation also mounts some directories from our local system to Docker containers:

- `./dags`: Directory where we will need to store all our dags.
- `./logs`: Output logs from executors and scheduler are stored here.
- `./plugins`: Director which stores all the plugins if we install any.

Mounting these directories allows us to share DAGs, store logs, and manage plugins between our local system and Docker containers, enhancing the flexibility and convenience of Airflow usage.

Once we are ready to initialize airflow, we need to follow the commands below:

```
# This command first spins up the airflow-init docker container
which sets up the

# prerequisites for airflow.

docker compose up airflow-init



# After initialization is complete, we will see a message like this

airflow-init_1       | Upgrades done

airflow-init_1       | Admin user airflow created

airflow-init_1       | 2.5.3

start_airflow-init_1 exited with code 0



# This indicates that admin account is ready and we can use the
credentials

# airflow:airflow for webserver login. Now it is time to spin up all
other containers
```

```
docker compose up
```

Once the containers are ready, we can login to the web interface and begin experimenting with DAGs. The web server will be available on the development machine at **http:localhost:8000**. The default account has the login airflow and password airflow.

Airflow in production

To deploy our DAG in a production setting, it is essential to ensure that Apache Airflow is production-ready. This involves several key aspects, as described:

- **Database backend configuration**: By default, Airflow is equipped with an SQLite backend, enabling users to run it without an external database. However, this setup is designed primarily for testing purposes and is unsuitable for production use due to the risk of data loss. Configure the backend to use an external database like PostgreSQL or MySQL to establish a robust production environment.
- **Multi-node cluster setup**: Airflow's default `SequentialExecutor` is limited to executing a single task at a time and pauses the scheduler during task execution. This is not ideal for a production environment. Instead, we need to use the `LocalExecutor` for single-machine setups and the **Kubernetes Executor** or the **Celery Executor** for multi-node configurations.
- **Logging configuration**: For clusters with disposable nodes, we need to set up log storage using a **distributed file system (DFS)** like S3 or GCS, or utilize external services such as **Stackdriver Logging**, **Elasticsearch**, or **Amazon CloudWatch**. This approach ensures log availability even if a node is removed or replaced.
- **Production container images or Helm Charts**: The Apache Airflow community provides a Docker image (OCI) for use in containerized environments as well as Helm Charts to be used for the Kubernetes deployment. We should use these images or help charts to ensure consistent software performance regardless of the deployment location.

Note: We can also choose to use any of the managed Airflow services provided by the cloud providers and third-party companies like AWS or Astronomer. This document in Airflow provides the overview of the Airflow Ecosystem including all the managed airflow services https://airflow.apache.org/ecosystem/

Example: Airflow Direct Acyclic Graphs

Following is an example of Airflow Direct Acyclic Graphs (DAG) using Taskflow API and **`DockerOperators`**. Each of these tasks will run using a Docker image:

```
import json

import pendulum


from airflow.decorators import dag, task

@dag(

    schedule=None,

    start_date=pendulum.datetime(2021, 1, 1,
tz="UTC"),

    catchup=False,

    tags=["example"],

)

def example_dag():

    """

    ### TaskFlow API Example DAG

    This is a simple data pipeline example which
demonstrates the use of

    the TaskFlow API using three simple tasks for
Extract, Transform, and Load.
```

```
    """
    @task.docker(image="python:3.9-slim-bullseye")
    def extract():
        """
        #### Extract task
        A simple Extract task to get data ready for the rest of the data
        pipeline. In this case, getting data is simulated by reading from a
        hardcoded JSON string.
        """
        data_string = '{"1001": 301.27, "1002": 433.21, "1003": 502.22}'

        order_data_dict = json.loads(data_string)
        return order_data_dict

    @task.docker(image="python:3.9-slim-bullseye", multiple_outputs=True)
    def transform(order_data_dict: dict):
        """
        #### Transform task
        A simple Transform task which takes in the collection of order data and
        computes the total order value.
```

```
        """
        total_order_value = 0

        for value in order_data_dict.values():
            total_order_value += value

        return {"total_order_value":
total_order_value}

    @task.docker(image="python:3.9-slim-bullseye")
    def load(total_order_value: float):
        """
        #### Load task
        A simple Load task which takes in the
result of the Transform task and
        instead of saving it to end user review,
just prints it out.
        """

 print(f"Total order value is:
{total_order_value:.2f}")

    # These 3 lines below define the task
dependencies and the data movement.
    order_data = extract()
```

```
    order_summary = transform(order_data)

    load(order_summary["total_order_value"])


example_dag()
```

It must be noted in this DAG file is that the Python script is interpreted by Airflow and is a configuration file for the data pipeline. The first thing we do is instantiate a DAG with the name **`example_dag()`**, which is a collection of tasks and defines the dependencies between those tasks. The definition is created using the **`@dag`** decorator. We also define three tasks by the name **`extract()`**, **`transform()`** and **`load()`** to simulate an ETL pipeline. Each of those is also decorated with **`@task`** decorator and uses the Docker task operator. And fina

lly, we define the dependencies in each task and the data movement. This is a simple task definition to provide an example of the structure of the Airflow DAGs. However, in real life, we will have a few more configurations in the DAG, like retries, notifications, maybe somewhat complex scheduling, and so on.

Besides Airflow, there are other workflow management tools that can be used based on the requirements and how they fit with the overall infrastructure of the organization. Some of those are:

- **Prefect**: Prefect was created by a team who was the core contributor of Airflow. They wanted to address some drawbacks of Airflow and build tools to address features like passing parameters to the DAGs, having dynamic DAGs that can create tasks in runtime, and so on.
- **Argo**: Argo considers containers as the first class citizen of the tool and every step in an Argo workflow is executed in its container. However, Argo, workflows are defined in YAML. One main drawback of Argo is that it can run only on Kubernetes clusters. For the local development, we will need to set up and run **`minikube`** to simulate a Kubernetes cluster in our local machine.
- **Kubeflow**: The primary objective of Kubeflow is to facilitate the seamless execution of workflows in both development and production

environments by eliminating the need for infrastructure boilerplate code, typically required for running platforms like Airflow or Argo. However, Kubeflow often depends on an external scheduler to manage the execution schedules effectively. Since Kubeflow pipelines are built on top of Argo, it shares the limitation of being exclusively runnable on Kubernetes clusters.

- **Metaflow**: Metaflow is similar to Kubeflow, where it abstracts the development and production boilerplate code and also generally requires an external scheduler. However, one difference is that Metaflow can be used with AWS Batch as well as Kubernetes cluster.

Machine learning platforms

As the organizations go through the maturity of the ML infrastructure and architecture, they uncover an increasing number of use cases for ML across various applications. This growth in use cases leads to the emergence of platforms and applications that can be utilized and repurposed for multiple projects. Organizations can maximize the benefits of their ML deployments by leveraging a common set of tools for each application. This shared set of tools for ML deployment forms the foundation of a machine learning platform.

As machine learning tools and platforms evolve and progress through the hype and growth curve, we see the always-changing definitions of what ML platforms consist of and what components are part of the platform. New tools are constantly being introduced that fit within the platform, adding value to the overall MLOps infrastructure. We will look into some of those common components which are almost always required in the ML platform. These are:

- Model deployment
- Model registry
- Feature store

Model deployment

Model deployment is the final and essential concluding step in any machine learning project. In *Chapter 6: Model Deployment and Serving*, we will delve into the intricacies of the deployment and model serving process. The most straightforward method of deploying a model involved pushing the model and its dependencies to a location accessible in the production (or dev and staging) environment, exposing the model as an endpoint for users. For online predictions, this endpoint must trigger the model to generate a prediction and return the data; for batch predictions, the ending endpoint retrieves a precomputed prediction and returns the data.

A deployment service or component can facilitate this process by enabling the seamless pushing of models and dependencies to an accessible location and serving the model through endpoints. A multitude of tools are available for this purpose, including those offered by major cloud providers such as tools like **AWS Sagemaker**, **GCP Vertex AI**, **Azure ML**, **Alibaba ML Studio.** Moreover, there are niche companies which offer proprietary and open-source tools for model deployment management like MLflow models, **Seldon**, **Cortex**, **Ray Serve**, and several others.

When evaluating and selecting deployment tools to incorporate into the ML platform, it is crucial to consider how well they integrate with the team's and organization's overall infrastructure and how much they simplify the deployment process. Many organizations rely on different tools for handling batch predictions and online predictions. For instance, an organization may choose to use Seldon for serving online predictions while employing Databricks for batch predictions.

Model registry

Model registry or model stores have become essential components of any machine learning platform, particularly when a team or organization manages multiple models. It can be challenging to manage numerous models, track different versions, store all relevant artifacts, and maintain metadata for various models and their versions. For maybe a single or a few models, they can be stored directly in object storage like AWS S3. However, scaling and managing becomes increasingly difficult as the number of use cases and models in a production environment grows.

Especially to help with troubleshooting, debugging, or auditing the models, it is crucial to track as much information as possible associated with a

model. Some of the artifacts and metadata to keep track of include:

- **Model code**: This consists of the code used to generate the model whenever required, encompassing details of the framework used, training and testing process, hyperparameters, and so on. It also includes code and functions for creating features and generating predictions from input data. These functions are usually wrapped in the endpoints and exposed to the end users.
- **Experiment artifacts**: During model development, various experiments are conducted to identify the best model. Storing and managing these experiment artifacts is a good practice, as it allows revisiting data and models to support decisions and for auditing purposes.
- **Dependencies**: For all the dependencies, we may require to run the model in any environment.
- **Data**: It refers to the data used to train the model and its various versions. It can be the copy of the dataset or the pointer to the location of the data stored in data warehouses or databases. It can be a query to fetch the data required to train the model and so on. If we use tools like DVC to version the data we use during the model development, then git commit can be used to capture the version of the data and model we are working on.
- **Model definition and parameters**: It is the metadata and information needed to create the model, as well as the parameters and their values which are used to generate a specific version of the model. For example, what loss function to use, how many vector dimensions to use and so on.
- **Tags**: Tags can be any set of metadata values that will help us in model discovery, tracking, and filtering. These tags can be any sort of metadata value and can be different for different organizations and teams. Some of the tags can be the owner of the model, the team responsible for this model, environment tags, version tags, and so on.

MLflow is a popular open-source platform that offers this functionality, enabling users to store artifacts, models, and metadata, track experiments,

and establish a proper model lineage. Other tools that provide similar capabilities include **Neptune.ai**, **Comet**, **Guild.ai**, and **Sacred**.

Feature store

Feature stores, as the name suggests, are centralized stores for managing features utilized in various machine learning projects. There are three primary components of a feature store are:

- **Feature management**: Organizations often have multiple ML projects running simultaneously, each generating a set of features for their respective models. Often, features used for one model can be useful for another, so it is logical to centralize their management and enable sharing among other teams. A feature store can help teams share and discover features and manage roles and sharing settings for each feature. Some of the examples of feature stores that provide this catalog are Amundsen which was developed by Lyft and DataHub developed by LinkedIn. In addition, **feature versioning** and **lineage tracking** are crucial as well. Feature versioning ensures models use the same feature sets during training and inference. Lineage tracking helps trace the origin and transformation history of features, enabling auditing and debugging.
- **Feature computation**: Feature extraction logic after it is defined will need to be computed. For example, a feature extraction logic can be use the average time a player spends on the game over the last week. The computation involves looking into the data and computing this average value to be used by the model, depending upon if the feature is computation intensive or not can dictate when it is computed. If the feature is easy and light to compute, then each model using the feature can compute it on the runtime, but in case it needs heavy computation or takes time to compute, then it might make sense to compute it and store it to be used by all the models.
- **Feature consistency**: As discussed earlier, a model might have separate pipelines for batch and streaming online data. A batch pipeline and a streaming online pipeline depend upon the use case and how historic data is coming for the training part. If a feature is defined

and executed at two different places, there is always a possibility of having inconsistencies in the data or if there is a change made in one pipeline but not in other that will also lead to inconsistencies. Feature stores solve this problem by providing a unified place to store the logic and values of the feature, which can be shared among different pipelines.

The most popular open-source feature store as of now is **Feast**. It needs to be deployed and managed by us on-prem or cloud servers. **Tecton** is another feature store which is commercial and is a fully managed feature store where we do not need to be concerned about the infrastructure setup and deployment ourselves. If you would like to look at the in-depth comparison of Feast and Tecton and see which one best works for your use cases and infrastructure, take a look at this resource **https://resources.tecton.ai/choosing-the-right-feature-store-feast-or-tecton**. Platforms like SageMaker and Databricks which provide comprehensive ML platforms, also offer their interpretations of feature stores.

Installing MLflow

We can install MLflow by running:

```
# Install MLflow

pip install mlflow



# Install MLflow with extra ML libraries and third party tools

pip install mlflow[extras]
```

Once we have the MLflow installed, the first thing to do to start using it is to integrate the tracking API into our model development code. The MLflow Tracking API lets us log metrics and artifacts from the machine learning project and see a history of the runs. The following code demonstrates the use of the tracking API:

```
import os
```

```
from random import random, randint
from mlflow import log_metric, log_param,
log_artifacts


if __name__ == "__main__":
    # Log a parameter (key-value pair)
    log_param("param1", randint(0,100))


    # Log a metric; metrics can be updated
throughout the run
    log_metric("foo", random())
    log_metric("foo", random() + 1)
 log_metric("foo", random() + 2)


    # Log an artifact (output file)
    if not os.path.exists("outputs"):
        os.makedirs("output")
    with open("outputs/test.txt", "w") as f:
        f.write("hello world!")
    log_artifacts("outputs")
```

Once we have the parameters and metrics logged, we can look at this run and see all the logged parameters in the MLflow UI. To run the UI, use the following command:

```
# Run the MLflow UI and view it at http://localhost:5000
mlflow ui
```

Having an understanding of these components and layers prepares us to dive into in-depth MLOps topics, which we will delve into in the coming chapters.

Build versus buy

As it always comes down to, it all depends upon the use cases and the types of machine learning projects an organization is running and is interested in. The maturity of the infrastructure is also a driving factor in this decision.

The investment in MLOps infrastructure also depends on where we want to invest. Do we want to invest more in hiring team members or in the infrastructure licensing itself? For example, if we want to use fully managed Databricks clusters, we probably need only one engineer to manage them. However, if we want to host your own Kubernetes or Spark clusters to run all the projects, we might need five or more people to set up and manage that infrastructure. It is the same with data movement platforms, we can manage our own and set up open-source platforms like Apache Airflow or buy platform licensing like **Fivetran** where the setup is minimal, and we do not need a full team to manage it. Most of the companies fall somewhere in between build versus buy; we might end up having the compute infrastructure built in-house but have Fivetran and Snowflake as data warehouse, which is managed or vice-versa.

Some other deciding factors other than the team and the financial investment are:

- **Maturity level of the company**: We might want to leverage vendor solutions to get started as quickly as possible so that you can focus on our limited resources on the core offerings of the product. As our use cases grow, however, vendor costs might become exorbitant, and it might be cheaper to invest in an in-house solution.
- **Focus of the company**: If it is something we want to be good at and focus on as a company to make it our competitive advantage, it makes sense to build it in-house rather than buying. If it is a support system and not the focus of the competitive advantage then it is better to incline towards buying it.

- **The maturity of the available tools**: This generally matters in the early days of a vertical not being mature enough or not having enough commoditized products in the space. In the early stages of machine learning development, this was predominantly the scenario. For example, if we want to set up a feature store for our projects and platforms and we want to buy it, but if there is no offering in the market that satisfies our requirements or use cases, it might make sense to build an inhouse on top of some open-source system or even from scratch.

The build versus buy decisions are complex, highly context-dependent, and likely what heads to infrastructure spend much time mulling over. Making the right decision here is important because once we have a platform and infrastructure in place, it cannot be changed easily, and we will need to use and depend upon that infrastructure for years to come.

Conclusion

We started the chapter by discussing what infrastructure and tools make sense for an organization and how we can consider going about it. Bringing machine learning models to production and making them available for use is not only a data science or machine learning problem but as much an infrastructure problem. Possessing the appropriate tools and infrastructure for successful machine-learning projects is essential.

In this chapter, we have explored the various layers of infrastructure required for machine learning systems, beginning with the storage and compute layers. This layer offers essential resources for any ML engineering project that demands intensive data and computational power. The storage and computer layer has become highly commoditized, leading many organizations to pay cloud service providers for the amount of storage and compute resources being used, rather than setting up private data centers. While cloud providers make it easy for an organization to get started, costs can become prohibitive as the usage expands.

Next, we discussed the development environment, where data scientists write code and interact with the production environment. The development environment is crucial, as it is where engineers spend most of their time. Enhancements in this area directly impact productivity. One of the initial

steps a team can take to improve the development environment is to standardize it for data scientists and machine learning engineers. In this chapter, we have looked at the reasons behind the recommendation of the standardized development environment and the methods to achieve this standardization.

Furthermore, we delved into an infrastructure-related topic that has recently been the subject of much debate: **Resource management**. Resource management is a critical aspect of data science workflows, but the question remains whether data scientists should be responsible for managing it. In the next chapter, we will look into the process of data preparation and model development which forms the basis of any machine learning system and workflow.

Points to remember

- Machine learning infrastructure consists of various layers such as storage, compute, containers, orchestration/workflow management, and ML platforms. Each layer serves a different purpose in developing and maintaining ML systems.
- For storage, common options include local hard drives, cloud storage services, and on-premises servers/data centers. Key considerations are ETL/ELT, batch versus stream processing, OLTP versus OLAP databases.
- For compute, organizations can use public cloud vendors or private data centers. The development environment is crucial and should be standardized. Containers like Docker help recreate environments efficiently.
- Orchestration platforms like Apache Airflow schedule workflows and manage resources. Key factors are the iterative nature of ML and task dependencies.
- ML platforms provide capabilities like model deployment, model registries, and feature stores. These enable reusability and simplify management as use cases grow.

- On build versus buy decisions, factors to consider include maturity level, company focus, tool maturity, cost, and team skills. There are tradeoffs between investing in licensing versus engineering.
- Overall, having the right infrastructure to develop, deploy and manage ML systems efficiently is critical, though the optimal setup depends on the organization's specific needs and context.

Key terms

- **Extract Transform Load (ETL)**: In computing, extract, transform, load is a three-phase process where data is extracted from the source, transformed and loaded into the destination system which can be data warehouse or data lake and so on.
- **Extract Load Transform (ELT)**: ELT is similar to ETL and has three steps but how differs in sequence. In ELT, data is extracted from source, loaded to the destination system in the raw form and then transformed on the destination system from raw to the format required by the downstream systems.
- **Batch processing**: Batch processing is the method used to periodically compute high-volume, repetitive data jobs.
- **Stream processing**: Stream processing is the method of processing, analyzing and ingesting data in real-time or near real time as it arrives in the stream.
- **Online Transaction Processing (OLTP)**: Online transaction processing is a type of database system used in transaction-oriented applications where the application needs to return the result to the user in a short period of time. Some examples are online banking, shopping, sending text messages and so on.
- **Online Analytical Processing (OLAP)**: Online analytical processing is a type of database system used to analyze business data from different points of view. It organizes large business databases and support complex analysis which can take some time to process and are not instantaneous.

- **Directed Acyclic Graphs (DAG)**: **DAGs** are graphs without any directed cycles. These graphs represent execution order and artifact usage of single steps of the workflow. In a directed acyclic graph, it is impossible to start at one point in the graph and traverse the entire graph. Each edge is directed from an earlier edge to a later edge.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 4
What are Machine Learning Systems?

Introduction

Machine learning is becoming the norm in the industry. More and more organizations are finding ways to address the requirements of their systems by integrating machine learning into their offering or creating completely new machine learning-based offerings. This use of machine learning capabilities and consequent dependency is ever increasing. In the last decade, machine learning entered almost every aspect of our lives ranging from how we consume information, communicate, get help over the internet, to how we live and work.

Even with these advancements, the majority of us consider machine learning and data science to be mere algorithms. Though algorithms are an important part, a production-ready machine learning project is more than just an algorithm. In fact, the algorithm is only a small part of it. In this chapter, we will revisit what machine learning systems are, some of their use cases. Most importantly, we will learn how we should approach designing the machine learning systems so that it fits well with MLOps frameworks.

We covered the introduction to MLOps and understanding its architecture in part one of the book; in this second part, we will go over the machine learning systems and how MLOps can be integrated. In the next part, we will put all this together and see how infrastructure can work together with the

well-designed machine learning system to give us a complete end-to-end MLOps workflow. If you have been working on developing machine learning systems/applications in the production environment, you might be familiar with the discussion in this chapter. If you have only had experience with machine learning in an academic setting, this chapter will provide a good starting point in understanding the difference between research and production-grade machine learning systems.

Structure

In this chapter, we will discuss the following topics:

- What is a machine learning system
- Understanding machine learning systems
- An implementation roadmap of MLOps-based machine learning systems
- Machine learning development: Cookiecutter data science project structure

Objectives

By the end of this chapter, you will understand what machine learning systems are and how they are different from machine learning research. Readers will explore what it takes to get a machine learning system deployed to a production environment and a roadmap to follow.

What is a machine learning system

The increasing importance of machine learning in real-world applications brought awareness of a new field focused on machine learning in production called **machine learning systems** (also called **MLOps** as a generic name). Machine learning systems are the intersection of machine learning combined with software engineering and DevOps practices. The major parts of a machine learning system are algorithms, data, the infrastructure to develop and run the system, and the hardware required for the computing. It also involves a focus on making the system reliable, maintainable, and scalable.

Machine learning system includes a variety of different verticals and layers including understanding the business requirements, the initial experimentation and the corresponding infrastructure, the production infrastructure to deliver and update models, the interface for users to interact with the system, the data stores, application code including both frontend and backend, monitoring and tracking capabilities and so on.

Figure 4.1 shows the technical layers which build a complete **Machine Learning System**. For the applications outside this system, the interface is the way to access the information from this system. **Data** and **ML Algorithm** plays an important part in the whole system which is heavily dependent upon the **Infrastructure** and the **Hardware** on which the system is running. Please refer to the following figure:

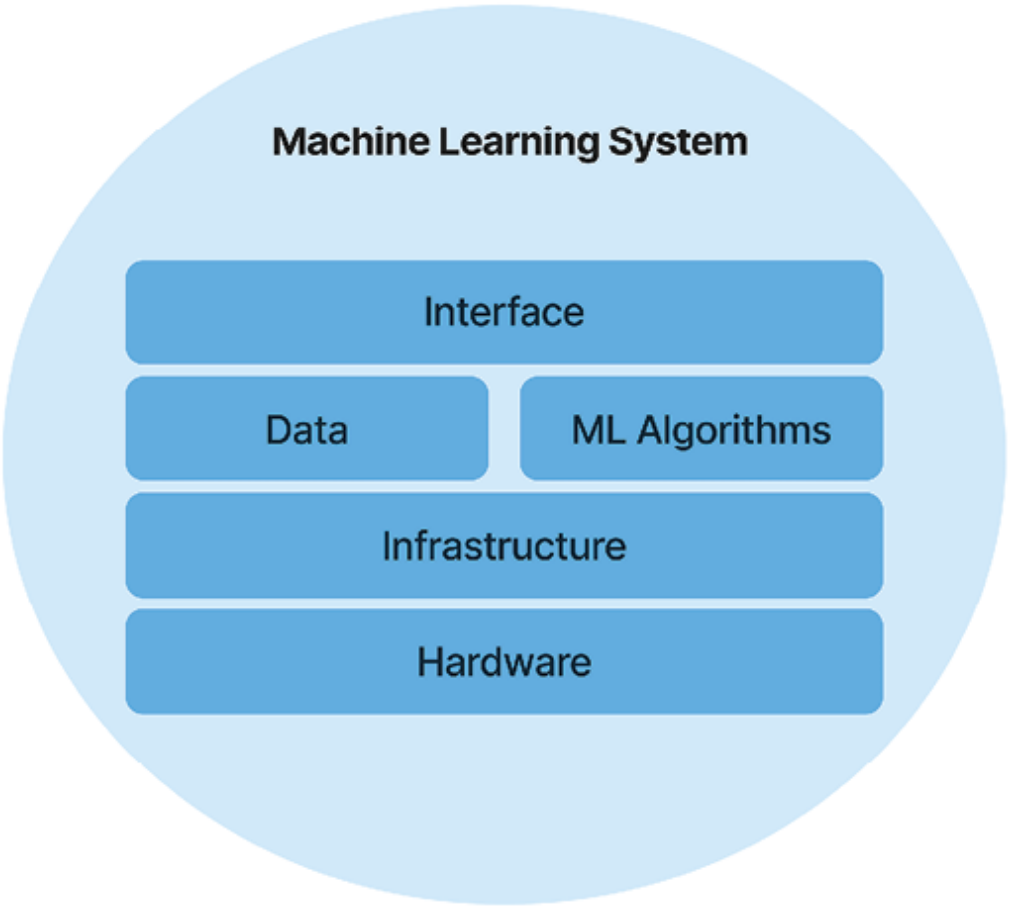

Figure 4.1: *Machine learning system*

Another major part of the **Machine Learning System** building is approaching the ability of all these verticals to work together in sync. We

need to take a complete systems approach to MLOps where we ensure that all the verticals, components and stakeholders can work together and achieve the objectives of the system and the project. For instance, Infrastructure team collaborating on the hardware and the infrastructure component, which can be on-prem or cloud based. Data science and machine learning engineering teams working with software engineering as well as DevOps to work on building the user facing components, setting up CI/CD for deploying and making sure the systems are working as expected and providing an interface which is useful for the users. Software engineering, ML and DevOps also need to collaborate on monitoring and observability of the overall system.

Machine learning systems use cases

With the explosion in the availability of data, information, and services, we have also seen a great usage and adoption of machine learning applications in both B2B and B2C domains. Be it the search engine, we use every day, or the recommendation systems integrated into the e-commerce websites and streaming platforms, if not for machine learning it would have been extremely challenging to find what we want. For example, when we use streaming platforms like *Netflix* or *YouTube*, we start seeing recommendations based on our watch history, if we want to tweak those recommendations, what we can do is search for specific items and watch. Over time, those new items will also power the recommendations we will get.

Smart consumer gadgets are another area where we are assisted by machine learning continuously. Smart personal assistants such as *Apple Home*, *Alexa* or *Google Home* use machine learning to process user interactions and respond to them. Things which might seem as small as typing on a phone are easier with **predictive typing**, a machine-learning system which suggests what you might want to say next.

Even though the demand and use cases of machine learning applications targeting consumer applications are on the rise, still a chunk of machine learning use cases are still in the enterprise world. Enterprise machine learning systems are completely different than what we see in consumer applications, they tend to have different requirements.

Talking about enterprise applications, in financial organizations, one of the main use cases of machine learning systems is **fraud detection**. Anomalies

in the historic fraud transactional data can be used for training to build a machine learning system to predict if any of the live transactions are fraudulent.

Price optimization is another machine learning-based system widely used in the enterprise, especially in e-commerce organizations. It is the process of estimating the best price for a product at a given time based on the demand, the company's margin, revenue and so on.

A set of machine learning systems use cases that has generated much excitement in the last decade is healthcare both face-to-face and telehealth. This technology can help us a lot in providing assistance in medical diagnosis, be it skin cancer detection, lung cancer or detecting diabetes, we have come a long way in creating systems to assist medical professionals.

Understanding machine learning systems

Understanding machine learning systems will be helpful in designing, developing and integrating them with MLOps principles and framework. In this section, we will understand the difference between machine learning projects in research and in production systems as part of an organization.

Machine learning in research versus production

A right to passage for machine learning usage in the technical industry is generally by gaining expertise through academia that is by taking courses and lessons, doing academic research, and via academic papers. Even though most of the learning and experience from academia is in-depth but one thing it generally lacks is understanding the challenges of deploying, running, and maintaining these machine learning systems in production environments. These challenges of taking the research to production environment and the overwhelming set of possible solutions which can be adopted to do so, makes the overall process quite complex.

Machine learning in an industrial production environment is different from machine learning in research. *Table 4.1* shows five of the major differences:

Properties	Research	Industry
Objectives and requirements	Use benchmark datasets to produce state-of-the-art model	Objectives and requirements differ based on the stakeholders involved

	performance	
Computational priority	Fast training, high throughput	Fast inference, low latency
Data	Generally static	Constantly shifting
Fairness	Often not be a focus	Must be considered
Interpretability	Often not be a focus	Must be considered

Table 4.1: *Overview of differences in machine learning approach in research versus production*

Let us look at each of these differences in detail.

Objectives and requirements

Generally, there is a single stakeholder and a single objective in most research projects, which is model performance. The end goal is to develop a model to achieve state-of-the-art results on available benchmark datasets. Achieving that performance and results is the only objective in the research projects, irrespective of how scalable or production-ready the system is. On the other hand, in industry, there are generally multiple stakeholders for the system. Everyone has their own requirements and objectives which sometimes makes it difficult to design and develop a system that addresses all the requirements.

Industry project generally involves multiple teams and departments and is truly a cross-team collaboration. There are machine learning engineers who want to deliver the best possible results and are ready to build more complex systems to achieve that performance. Then there is the sales team who wants to focus on the system which brings in the highest amount of money. The product team wants the overall performance of the algorithm in line with the overall system performance requirements set forth in the project. The infrastructure team wants the system to be scalable and easily manageable; then there is the leadership team who generally wants to maximize the margin and positive ROI from the system. It is important to understand the common objective, and non-negotiable requirements and assess which ones are good to have and depending upon that design a system. For example, if latency is a must-have requirement, then the system and the algorithm should focus on fulfilling it.

We all have heard the common statement that most machine learning projects never make it to production. It is especially true for projects where a

research approach is taken and the main focus is just the algorithms but other requirements of productionising the system are ignored or missed. For example, the machine learning technique of ensemble models is extensively used in research projects to generate the best possible results but is not a preferred way to approach a system in production environments for the majority of the projects due to the complexity it adds to the system and the ROI it provides.

Computational priorities

The computational priorities of the project mostly differ between research and industry. In research, the computation priority and a major chunk of the effort is in focusing on the model development and training part. During training, we might run multiple models, hyperparameter tuning, and versions to come up with a model that performs well. At this point, the priority is on the model development part and that is where all the efforts and the computational power should go. Whereas in production, once the model is developed and is being served, the computation priority is the inference and ability to provide predictions in a scalable way.

Data

During the research and experimentation phase, the datasets are often clean and well-formatted. Even if that is not the case in the real world, we go through the initial step of cleaning the data. In research, commonly used datasets are static in nature and are consistent so that they can be used for benchmarking new architectures, new models, and systems.

In industry, even if data is available beforehand, it is a lot messier and constantly shifting. Frequent changes in project and business requirements may necessitate regular data updates. Then there are other requirements that come with live industry data like privacy and regulatory concerns.

In research, most of the work is done with historical data. In industry, we generally work with data that is constantly being generated live by the users, systems, or third-party data.

Fairness

During the research phase, fairness of the model is in most cases an afterthought and not the main goal unless the research is specifically targeted

at fairness studies. With the increasing awareness about machine learning and its use cases, fairness is important to be considered as an important objective of any machine learning system in the industry. It is especially vital if the system has any direct impact on the user actions like ML systems for approval of mortgage applications or systems for recruitment and hiring.

This is especially true when the decisions made with the help of machine learning can have an impact on someone's life. For example, people had their loan applications rejected because the algorithm made some decision based on their date of birth, race, or address. Resumes of job applicants might be rejected or ranked lower if machine learning systems are used that are biased or are able to make any negative correlation with even a small thing on your resume.

We must always keep in mind that machine learning systems do not predict the future but learn from past data and correlate that learning with the new things they process. Now, this comes with an inherent risk that if the data contains biases, then the resulting system might also learn that bias and incorporate it into its decisions and predictions.

With the increasing awareness about bias in machine learning systems, a lot of organizations are focusing on it with research studies are being conducted. However, we are a long way from being able to mitigate risks to equity and fairness, such as algorithmic bias and discrimination. Nevertheless, this is changing rapidly.

Interpretability

Machine learning advancements are not always celebrated, there is a negative connotation with the world of machine learning and AI as well. People tend to think that AI will somehow attempt to rule the world, like in movies. This makes people susceptive to the capabilities of machine learning and thus they want to understand the working of the system properly before trusting it.

Since most machine learning research does not take that into consideration, the researchers are not incentivized to work on model interpretability. They instead focus on the main objective of a particular research project, unless the project's main objective is model interpretability itself. However, interpretability is a core objective of any machine learning use case and

implementation in the industry, to make us agile in the use of existing knowledge and systems in future development.

Now why is it important to understand what machine learning systems are and the difference between academia and industry with respect to machine learning? There are a lot of companies and organizations that conduct pure research in machine learning outside academia, but the number is small as most companies cannot afford to put their limited resources into research-driven objectives. For them, all the machine learning work needs to generate short-term business applications.

An implementation roadmap for MLOps-based machine learning systems

Starting a machine learning project with a well-thought methodology and a battle-tested framework ensures that we are setting up the project for success. In this section, we will go over an implementation roadmap as summarized in *Figure 4.2*; that can streamline the development of any machine learning project and facilitate MLOps approach for proper productionisation of the system. This roadmap is not a strict approach but is built with the goal of providing generic direction and guidance on all the moving parts of the system which need to work in sync to make the machine-learning system successful. Please refer to the following figure:

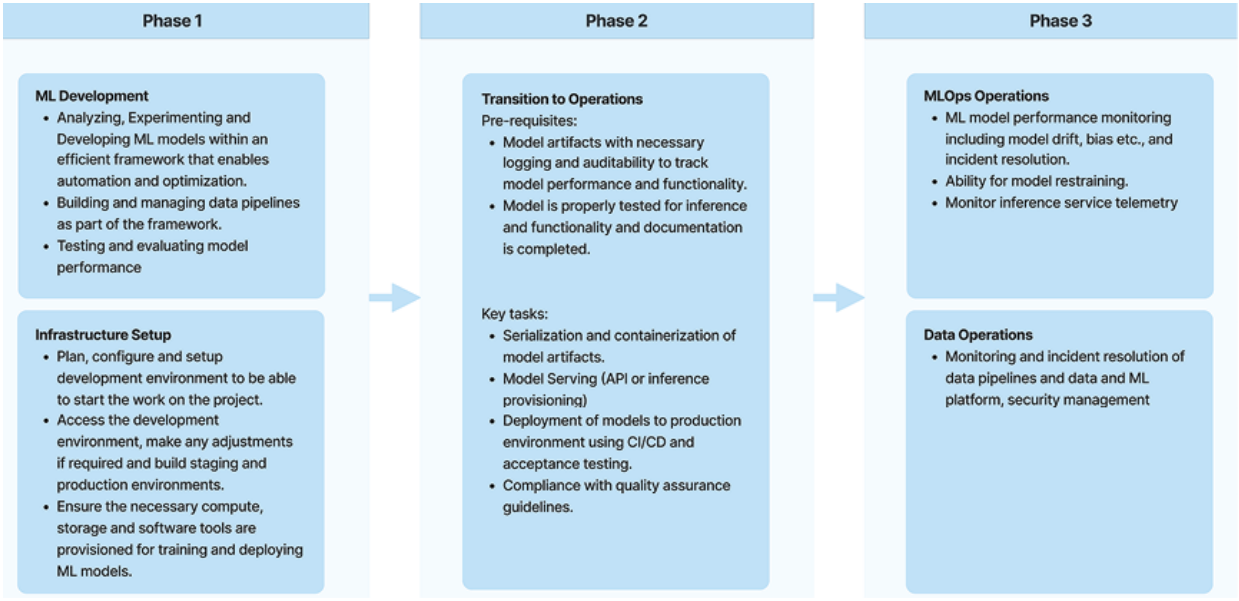

Figure 4.2: *Implementation roadmap for an MLOps-based machine learning system*

Using the preceding roadmap, we can transition any project from the machine learning development phase to MLOps operations with ease. Let us understand all three phases in detail.

Phase 1: Initial development

The first phase consists of two components, ML development and infrastructure setup. This is the stepping stone to implementing the MLOps framework for the project. Before starting on the analysis and experimentation phase to implement the requirements, the problem and solution must be clear and vivid. In this phase, for the first part, we want to set up our project structure with an efficient framework that standardizes usability, automation, and optimization for us. Having a component-driven code and repository structure in place helps us to build and manage data pipelines quickly and easily. Not only that, but the framework also allows us to think about testing and evaluating the model from the start.

In part two, we consider the system requirements to design and implement a robust and scalable MLOps framework. We begin by selecting the right tools and infrastructure needed for storage, computing, inference and monitoring to implement the MLOps.

When the infrastructure is set up, we should be provisioned with the necessary development, test, and production environments to execute training and testing tasks. The best approach for managing different environments is to train the models in the development environment, test the model in the staging environment, and finally deploy the models in the production environment. When infrastructure is set up and the first machine learning model is trained, tested, serialized, and packaged, phase 1 of the MLOps framework is set up and validated for robustness. Next, we move to implement phase 2.

Phase 2: Transition to operations

Phase 2 concerns transitioning to operations. It involves serializing and containerizing of the trained models and getting them ready for deployment to different environments. The models can be served in multiple different forms; as APIs, data or event triggered interfaces, or independent artifacts

which can be used for batch inference. Fruitful monitoring infrastructure is also important for the continuous surveillance of the incoming data as well as the performance of the model.

There are some prerequisites for this phase as well. The model which is developed in phase 1 should have all necessary logging, monitoring and auditability capabilities in the code. By the end of phase 2, you will have package models served and deployed in the production environment performing inference in real-time.

Phase 3: Operations

Phase 3 is the core part of the deployed models. In this phase, we monitor the deployment of model performance in terms of model drift, bias, or other metrics. Based on the model's performance, we can enable continual learning via periodic model retraining and enable alerts and actions. With the successful implementation of this phase, we can monitor the deployed models and retrain them in a robust, scalable, and secure manner.

For any successful machine learning project, all three phases are suggested to be implemented and managed. There are some cases though, where phases 1 and 2 are enough, for example, when the machine learning model is designed to make batch inferences it might only need phases 1 and 2 there. By achieving these milestones and implementing all three phases, we have set up a robust and scalable ML lifecycle for our applications systematically and sustainably.

In the next section, we will discuss the ML development part of phase 1 and in subsequent sections, we will look at the other parts and phases as well.

Machine learning development: Cookiecutter data science project structure

Adopting agile practices in machine learning and data science is complicated and challenging to implement but beneficial in the long run. When we work with any machine learning project which in its essence is exploratory in nature, using tools like **Jupyter notebooks** is extremely helpful in the exploratory phase of the projects but they tend to be messy and unorganized. It is problematic because it is hard to convert the code to reproducible scripts

and deploy the project properly. `Cookiecutter` data science provides a standardized but adaptable repository structure for working on data science and machine learning projects geared towards production deployment. It provides structure to the projects repositories but still is flexible enough that it can be adapted for different use cases. The project template can be accessed in the `cookiecutter` **GitHub** repository. (**https://github.com/drivendata/cookiecutter-data-science**)

What is cookiecutter

If you are not already aware, `cookiecutter` is a Python package, which can be installed with `pip` or other package managers, that enables us to create and use templates for various types of software projects. It is a command line tool for setting up project structures. (**https://cookiecutter.readthedocs.io/en/1.7.2/README.html**)

Why cookiecutter

It is not a secret that good experimental analysis is not a linear process, a good analysis is the result of scattershot and serendipitous explorations and multiple different experiments that may or may not work out to give us a good outcome. But after these initial data exploration and experiments comes the part where we want to develop and deploy this project and at that point, the structure becomes important.

If a well-defined, standard project structure like `cookiecutter` is adopted for projects, it makes it easy to understand the project. For example, other team members you start the project with or colleagues who might join the project later on, can easily jump in and understand the project structure and where to find specific things in the project without digging into extensive documentation.

Note: It is important to note that the cookiecutter project/repository structure is not binding. If the team or the organization has a slightly different way to handle projects, if there is a different standard to handling naming conventions or if the project is a little nonstandard and does not fit with the current structure and cookiecutter is intended to provide a standardized starting point for the projects, feel free to adapt it to your particular needs. If you do not want to use cookiecutter library but like the structure of the project template; it is not compulsory to use cookiecutter; we can decide to create and follow the same structure of the code repository manually as well. What cookiecutter provides is easy access to the template and makes it easily repeatable for multiple data science and machine learning projects.

Getting started with cookiecutter data science

Getting started with the data science project template is really easy if we use the `cookiecutter` Python package. We have to make sure that all the prerequisites are installed before starting a new project, which are:

- Python 2.7 or 3.5
- `Cookiecutter Python package >= 1.4.0:` This can be installed using the standard `pip` package manger. The command we will need to use to install `cookiecutter` package is:

  ```
  pip install cookiecutter
  ```

  For more details on the cookiecutter package and alternate ways to install, please refer the documentation at **https://cookiecutter.readthedocs.io/en/latest/installation.html**

Once we have the pre-requisites installed and ready, we are good to start working on a new project which is as easy as running the following command. It will setup the complete structure of the project with all the required files and folders:

```
cookiecutter https://github.com/drivendata/cookiecutter-data-science
```

Repository structure

Following is the repository structure from `cookiecutter`. It divides the components we are required to deploy and scale the project into independent segments. The structure is easy to follow and understand, we will look at and discuss a few important components as follows:

- **Makefile**: Makefile is for the commands to process data, train the model, evaluate and so on. Depending upon what you want to use as the orchestration tools for the pipelines, we can stick with Makefile or change it with another configuration file for example, if we use Airflow for orchestration, we can have the airflow DAG defined here instead of Makefile.
- **Data**: Data directory is generally divided into multiple sub-directories for external data, raw data, interim, and processed data. If the raw or external data is not required to be stored in the directory in cases like

where it is coming from the database or data warehouse directly, we can skip those sub-directories and store the interim or processed data only.

- **Model**: It is the directory to store the trained and serialized models or their metadata.
- **Notebooks**: This is where all the exploratory and experimental notebooks are to be stored.
- **References**: As the name indicates, any reference material or scientific papers for the project can be stored here.
- **Reports**: Any reports, graphics and figures which can be used for reporting on the project.
- **src**: The most important directory in the whole repository is `src`. It contains the actual source code of the project. This is where we store the scripts to be used by the orchestration platforms for the pipelines, for data pre-processing, featurization, model training, evaluation and so on. The preferred structure of the sub-directories is:
  - **Data**: To store all the scripts to process data from the raw format to the final format required by the project.
  - **Features**: Scripts for featurization and turning the data in the required features.
  - **Models**: Scripts to train the model, generate predictions and evaluation.
  - **Visualization**: Scripts which can be used to generate visualizations for reporting purposes.

```
├── LICENSE

├── Makefile           <-Makefile with commands like
`make data` or `make train`

├── README.md          <-The top-level README for
developers using this project.

├── data
```

```
|   ├── external    <-Data from third-party
sources.
|   ├── interim     <-Intermediate data that has
been transformed.
|   ├── processed   <-The final, canonical data
sets for modelling.
|   └── raw         <-The original, immutable data
dump.
|
├── docs            <-A default Sphinx project;
sphinx-doc.org for details
|
├── models          <-Trained & serialized models,
model predictions, or
|                       model summaries
|
├── notebooks       <-Jupyter notebooks. Naming
convention is a number (for
|                       ordering), the creator's
initials, and a short `-`
|                       delimited description, e.g.
|                      `1.0-jqp-initial-data-
exploration`.
|
├── references      <-Data dictionaries, manuals,
and all other explanatory
|                         materials.
```

```
│
├── reports         <-Generated analysis as HTML,
PDF, LaTeX, etc.
│   └── figures     <-Generated graphics and
figures to be used in reporting
│
├── requirements.txt    <-The requirements file for
reproducing the analysis
│                                environment
e.g.with `pip freeze > requirements.txt`
│
├── setup.py        <-Make this project pip
installable with `pip install -e`
├── src             <-Source code for use in this
project.
│   ├── __init__.py     <-Makes src a Python module
│   │
│   ├── data            <-Scripts to download or
generate data
│   │   └── make_dataset.py
│   │
│   ├── features        <-Scripts to turn raw data
into features for modeling
│   │   └── build_features.py
│   │
```

```
|   ├── models         <-Scripts to train models
and then use trained models
|   |   |              to make predictions
|   |   ├── predict_model.py
|   |   └── train_model.py
|   |
|   └── visualization   <-Scripts to create
exploratory and results oriented
|                            visualizations
|       └── visualize.py
|
└── tox.ini         <-tox file with settings for
running tox;
                           see tox.readthedocs.io
```

Source: **https://drivendata.github.io/cookiecutter-data-science/**

As you can notice from the repository structure above, it is an opinionated structure in the sense that this structure is based on certain opinions that have been reached through experience. It helps in maintaining the reliability of the project and improves collaboration. Some of these opinions are:

- **Data is immutable**: Do not edit raw data manually using Excel and so on outside the project. Raw data should be in the exact format that we will receive from the source system. One should treat the data as immutable. All the data processing should be done via the code. It can be done in the notebooks in the beginning but once we have the final processing structure and code, it should be moved to the data preparation scripts to be reused when required. This helps in processing any new raw data through the same pipeline using the scripts. Having raw data and a proper pipeline to process and convert it into the final format removed the dependency on storing any interim

format of data any of the team members can generate the required data using the code in the scripts and the raw data. We can altogether skip the raw data folder in the `gitignore` file. On the other hand, if the data we have is small and rarely changes, we may want to include that in the repository if that makes things easier. In short, whether to store the data in the git repository or fetch every time from the database is up to our use case and how we want to store data and make it accessible for the project.

- **Notebooks should be used for exploration and reporting only**: Notebooks or other literate programming tools are very effective for exploratory data analysis. However, these tools are not made for team collaboration and are challenging for the source control repositories to handle. There are two recommended steps for using notebooks effectively:
  - Follow a naming convention that includes the owner of the notebook and the order of the notebook. Use the format `` `<step>-<user>-<description>.ipynb ``
  - Refactor the good parts and convert them to scripts instead of writing the same code in different notebooks. These refactored scripts then can be put into the src folder and imported into the notebooks for use.
- **An analysis is a Directed Acyclic Graph (DAG)**: Often in an analysis, we have long-running steps to pre-process the data or train the models. A lot of time, we would like to avoid rerunning these compute and time-intensive steps. One way to avoid that is treating the analysis as a DAG. Each step should be idempotent and store the output somewhere for example, in data/interim to be used by the next step. That way, if we decide to rerun the code, we can skip running the data pre-processing step again, saving time and effort.
- **Keep secrets and configurations out of version control**: This is common knowledge but important to reiterate. Please always make sure no secret key, username or passwords are checked into Github. Always store the secrets and config variables separately either in the environment file which is never checked into the source code

repository or in secret management services like AWS secret manager which can store the configurations and secrets and load them into the environment when needed.

Starting with this repository structure and adapting it to the organization's and team's needs will set us up for success down the line when we want to take the project to production deployment.

Conclusion

We started the chapter by understanding what machine learning systems are, and some real-world use cases of machine learning systems in production. The chapter also takes us through the difference between machine learning in research and in production.

Machine learning systems are complex, lengthy and consists of many different components. Data practitioners working with machine learning systems in production will likely find that focusing only on the algorithms part is far from enough. It is important to know about other aspects of the system, starting with the objectives and requirements of different stakeholders, source data implication and what data stack to use, development, evaluation, deployment, monitoring and retraining of the models. Not to forget infrastructure which is also a big part of the system. This chapter is the first step in our journey to move from machine learning algorithms to taking a systems approach to machine learning systems and MLOps, which means we will consider all components of a system and how to make them work together. We started in this chapter with an understanding of the implementation roadmap of building a machine learning system and now we can build the system in different phases.

We also looked at the `cookiecutter` data science repository structure and how it can be a stepping stone for us. In the coming chapters, we will go further into understanding how we can work on different phases of a machine learning project using this template and how it can integrate with the MLOps infrastructure in order to build a reliable and scalable production system.

Points to remember

- Machine learning systems involve more than just algorithms - they require data, infrastructure, interfaces, monitoring, and so on.
- Machine learning in research versus production have different goals, priorities, and constraints. Research focuses on state-of-the-art performance while production focuses on scalability, reliability, and business impact.
- An implementation roadmap for ML systems involves three phases: initial development, transition to operations, and operations. This allows for an iterative process from experimentation to production deployment.
- Using a standardized project structure like the `cookiecutter` data science template facilitates collaboration, reproducibility, and eventual productionization.
- Treating data as immutable, converting notebooks to scripts, treating analysis as DAGs, and avoiding putting secrets in version control are best practices for reliable and maintainable ML systems.
- Understanding the differences between ML research and production systems, and taking a full systems approach, is key to developing successful real-world ML applications.

Key terms

- **Machine learning systems**: Machine learning system includes a variety of different verticals and layers, including understanding the business requirements, the initial experimentation and the corresponding infrastructure, the production infrastructure to deliver and update models, the interface for users to interact with the system, the data stores, application code including both frontend and backend, monitoring and tracking capabilities and so on.
- **Cookiecutter**: A command line utility that creates projects from cookiecutters (project templates), for example, creating a Python package project from a Python package project template or a data science project from a data science cookiecutter template.

- **Data warehouse**: A data warehouse is a central repository of information that can be analyzed to make more informed decisions. Data flows into the data warehouse from transactional systems, relational databases, and other sources, typically on a regular cadence. This data then serves as the central repository for downstream systems like ML or BI systems.

CHAPTER 5
Data Preparation and Model Development

Introduction

In this chapter, we will continue our exploration of the `cookiecutter` data science template we discussed in the previous chapter, and most importantly, we will look into how different components of a machine learning system fits into the template. We will also look at how this template can be a building block of the MLOps platform, which will help us automate most of the manual work we need to perform in productionizing a machine learning project. We will begin with some of the pre-requisites, we should add to every machine learning project, which will help us in structuring the project properly and improving the code quality.

The focus of this section will be to discuss the practical tips and principles that will help elevate the project structure and how to implement them in the project repository to improve the overall MLOps structure and prevent a situation where the code for the machine learning project repositories is unstructured and hard to understand.

As an experienced machine learning engineer or data scientist, if you are already familiar with analysis and experimentation with data, and building a machine learning model, this chapter will still be useful as we will make use of the `cookiecutter` data science template to design the work and develop a model.

Structure

In this chapter, we will discuss the following topics:

- MLOps code repository best practices
- Data sourcing
- Exploratory data analysis
- Data preparation
- Model development
- Model evaluation
- Model versioning

Objectives

By the end of this chapter, you will understand how to write the code and scripts for each component of the data preparation stage and model development stage of the machine learning lifecycle and how all those fit into the `cookiecutter` template we set up in *Chapter 4: What are machine learning systems*

MLOps code repository best practices

Working on a machine learning model, from ideation to deployment and monitoring throughout the process, is a time-intensive and complex process. As the scale of the projects grows, their complexity increases as well. MLOps best practices can help standardize the machine learning projects and the code repositories so that we become a cross-team collaboration. Moreover, switching between different projects can be streamlined and easy to understand. Some of the best practices when it comes to standardized code repositories for machine learning projects are:

- **Use pre-commit hooks**: When we work with code, over time, we tend to write code that might not be structurally sound or have some small issues. There are many easy-to-use libraries available that can make sure we are not committing a sub-optimal code in the repository. Having poorly unstructured or dirty code can hide issues and bugs in

the code that can become problematic to debug over time; adding hooks helps discover those easy-to-detect issues with the code as soon as possible. This best practice also aligns with the broader goals of MLOps to ensure reproducibility and minimizing debugging efforts. We will look at some of the examples of `pre-commit` hooks in the next section.

- **Attach a Git hash to each trained model**: For better tracking, monitoring, and debugging purposes, ensure we attach the Git hash to each trained model. This will also make sure that the models are only trained with the code that is committed to the repository. Making sure we have a code commit to track every model version is critical as it also allows us to look at the changes that went into the different model versions. Not only that, but this best practice also helps create a clear lineage between code and model versions, which is essential for ensuring transparency and accountability in machine learning projects. As we will see in the coming chapters, once we merge a new code and train a model on that code either using **Github Actions** or **Airflow** or any other **CI/CD platform**, it is always good to add the **Git commit hash** as the metadata and tag the code with the version.
- **Use a monorepo**: Having a monorepo for a single machine learning project means having a single project repository that encompasses the end-to-end pipeline, including the code for data preparation, data processing, training, evaluation, and deployment. This greatly reduces the complexity between project components by offering everything in the same place. It also makes code sharing easier and simplifies dependency management as well.
- **Data versioning**: As we know, data is the major component of any machine learning source code and should be treated as such. Reproducing a model is critical and the state of the data used for that specific run is important. We can use tools like **Data Version Control** (**DVC**) to achieve that and version our data used for training. Details about DVC can be found on the website **https://dvc.org**. DVC also integrates seamlessly with Git, making it a practical choice for managing data versioning in machine learning projects.

- **Data quality**: This is perhaps the least intuitive point. If we are using DVC and data is now part of the code in the repository, it might have bugs in data generation or storage as well. We can automate things as much as possible, but they are not airtight. Having a human in the loop to sanity check the data or quality check code which checks the quality of the data after preparation and before model training, can save a lot of trouble later in the process. However, dedicating a small percentage of the time to either manually check the data or writing the code to do so will pay off greatly later down the line.
- **Monitor the models**: Many times, teams thinking about monitoring fall into two categories. Either the team decides not to invest any time into making it work due to complexity or bandwidth; or decides to go all in and look for an end-to-end platform. It is always possible to take a middle ground based on the maturity of the project and the team. Even if, at the beginning of the monitoring journey, we manually label a few predictions and update a metric to track, it can also provide a great way to get reasonable scale monitoring.
- **Retrain models on a regular cadence**: As we know regular retraining helps models adapt to changing data distributions and maintain their performance over time, which is crucial for ensuring the reliability of machine learning applications. A common first approach is to develop a metric threshold to trigger retraining, but that can be quite complex and may need custom systems built to identify metric changes. A simpler and more effective alternative is to set a regular time interval (for example, once a month) to retrain. This is easier to set up and maintain. The exact cadence will depend on the type of the model and freshness requirement, which will vary from model to model. There are other metrics as well which can help determine the appropriate retraining frequency, such as data drift, concept drift or changes in problem domain. We will go deep into this topic later in the book.
- **Continuous Integration (CI) for ML projects**: Sometimes, with projects in experimental fields like machine learning, when we move from initial exploration to production-ready code, we either forget to or skip writing any code coverage for the project with test cases. This may cost us a lot of time and effort to fix the mistakes down the line.

We should, at minimum, start with writing unit tests for the code. This testing should be run on every commit and build similar to how it is done in software engineering using CI tools. A good way to start is with end-to-end tests, followed by more specific tests that cover the project code implementation properly. Continuous testing and integration help in catching the issues early in the development process, promote collaboration among team members, and ensure the new code changes do not negatively impact model performance.

These best practices will help us write and structure the machine learning project in a way that is scalable, reproducible, deployable, re-trainable, and extendable as per the requirement.

In the next section, we start by looking at how we can add `pre-commit` hooks to our existing code template and use them to enforce the best practices not just while starting the project but throughout the code lifecycle of the project.

pre-commit hooks

`pre-commit` hooks are a set of scripts that gets executed before every Git commit and can enforce certain behaviors or standards on the code being committed. This helps us to implement or follow the standards and rules even if, over time, we forget to follow the standards. We get a reminder and a nudge to fix it. This helps in maintaining code consistency, improving code quality, and preventing common coding mistakes, contributing to the overall success of the project.

To implement `pre-commit` hooks, we can choose the `pre-commit` framework, which is an open-source framework. It provides `pre-commit` hook recipes for many programming languages. More details and the instructions to implement `pre-commit` hooks can be found on the website **https://pre-commit.com**

Some of the most commonly recommended `pre-commit` hooks to be added to every machine learning project to make sure everything is formatted in a standard way are:

- `check-yaml`: It verifies the syntax of all YAML files. If you use the YAML in the project, make sure to add this hook to the project to

make sure incorrect YAML syntax is not being committed to the repository.

- **`end-of-file-fixed`**: It ensures that files terminate only with a newline. This is one of those best practices that may seem small but is always a good standard.
- **`check-ast`**: It determines if files parse as correct Python before they can be committed to the repository.
- **`Trailing-whitespace`**: It trims trailing whitespace by default, but it is flexible enough that it can be customized to trim any specific characters if we need to.
- **`black`**: It is One of the best and industry standard Python code formatter, which should be present in every Python-based project.
- **`mypy`**: It is a **`pre-commit`** hook, which enforces static type checking for Python.
- **`isort`**: It offers to utility to sort the Python imports in every Python file before it gets committed to the repository.

Implementing and using these **`pre-commit`** hook libraries is pretty straightforward. It requires installing **`pre-commit`** library followed by creating a file name **`.pre-commit-config.yaml`** in the repository, which details the repository from which the hook needs to be downloaded, revision/version of the hook to use, and details about the hook. Following are the steps to setup **`pre-commit`** hooks in the project:

Start by installing the **`pre-commit`** library:

```
pip install pre-commit
```

Once we have it installed, we create a file named **`.pre-commit-config.yaml`** and add the following content to the file:

```
repos:
  - repo: https://github.com/nbQA-dev/nbQA
 rev: 1.5.3
```

```
    hooks:
      - id: nbqa-black
      - id: nbqa-pyupgrade
      - id: nbqa-isort
  - repo: https://github.com/timothycrosley/isort
    rev: 5.10.1
    hooks:
      - id: isort
        args: [ --profile, black ]
  - repo: https://github.com/psf/black
    rev: 22.10.0
    hooks:
      - id: black
  - repo: https://github.com/pre-commit/mirrors-
mypy
    rev: v0.991
    hooks:
      - id: mypy
        args: [ --ignore-missing-imports, --pretty,
--show-error-codes ]
 - repo: https://github.com/asottile/pyupgrade
 rev: v3.3.0
    hooks:
      - id: pyupgrade
```

```
  - repo: https://github.com/pre-commit/pre-commit-
hooks
    rev: v4.3.0
    hooks:
      - id: check-ast
      - id: check-case-conflict
      - id: check-docstring-first
      - id: check-json
      - id: check-merge-conflict
      - id: debug-statements
      - id: check-symlinks
      - id: check-yaml
        args: [ --unsafe, --allow-multiple-
documents ]
      - id: detect-private-key
      - id: end-of-file-fixer
      - id: requirements-txt-fixer
      - id: trailing-whitespace
        args: [ --markdown-linebreak-ext=md ]
```

Once we have the file ready, to make sure the hooks are installed correctly, run:

```
pre-commit-install
```

Once this is done, everything we commit from that time onwards, these **`pre-commit`** helper libraries, will ensure we do not miss anything important.

Now that we have a well-defined code structure and the `pre-commit` hooks in place, we will make sure we adhere to the best practices when it comes to writing and committing structured code. We can start to look at the next stages of the project, which is handling data, analysis, and experiments for model preparation.

Data sourcing

In this and the following sections, we will look at each component of the machine learning system in terms of MLOps and see how we should design each component so that they fit well together to make an end-to-end system.

Figure 5.1 represents the MLOps component flow from *Chapter 1: Getting Started with MLOps,* and the very first component of this workflow is **Data Sourcing**, which we will start with here. Please refer to the following figure:

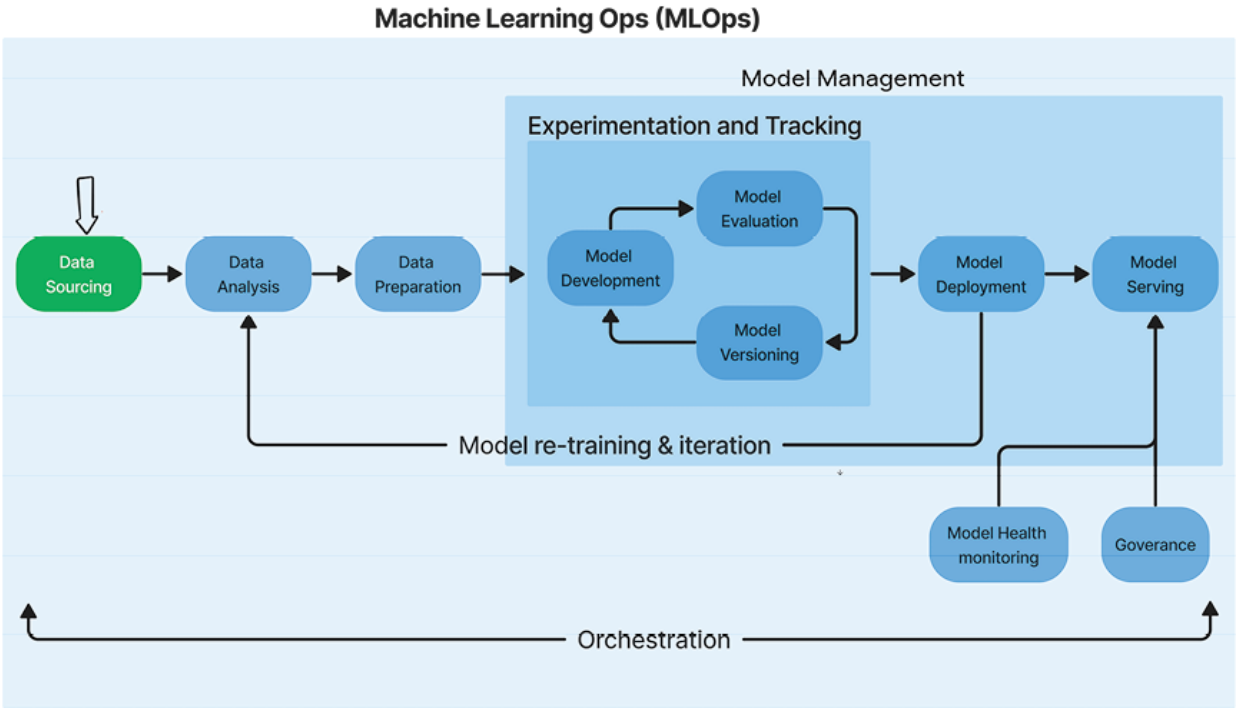

Figure 5.1: *Data sourcing stage of MLOps-based ML project lifecycle*

Having a well-defined set of data for the experiments is extremely important. To make sure we have it before diving into the analysis, we need to make sure that the data sources are well-defined, we have access to the required data, and understand the format of the data.

If this is third-party data, or if the data is somehow protected by copyright or other legal standards like **Personally Identifiable Information** (**PII**) or **General Data Protection Regulation** (**GDPR**), we might also want to consider how that data is collected to be used. Do we have the proper permission to use the data, or do we need to negotiate a new licensing agreement with the data collector or provider to use the data legally?

Once all these business and legal questions are answered, the follow-up technical question which we should also answer before starting on the project is whether there is enough data available to be used for training, testing, and evaluation of the models.

Data quality is another major factor affecting the model performance. The dataset we plan to use for the experiments may not be up to the standards in terms of the quality of the data. If we trust the dataset blindly, we might realize that no matter how hard we try to improve the quality of the model, its performance on new data is low.

Before we prepare the data for the model experiments from the data sources, there are five aspects to consider: **accessibility**, **size**, **useable**, **comprehensibility,** and **reliability** of data.

A good dataset for the experiments is one that contains good quality information that can be used for modeling, has good coverage and representation of what we want to do with the project, and reflects real data that we might receive once the mode is in production. Other important features of the datasets are that they should have unbiased data, have consistent labels, and be large enough to allow generalization.

In this first component data sourcing, we will look into two subcomponents:

- Data sources
- Data versioning

Data sources

Data can be stored in different data formats and on several data storage levels and platforms.

Common ways data is available to data science teams is either in data stores like **Relational DataBase Management Systems** (**RDBMS**), data

warehouses, data lakes, and so on, where the data from different sources from within the organization or third party are bought together, pre-processed using either **extract, transform, load** (**ETL**) or **extract, load, transform** (**ELT**) and stored to be used for data analysis, data experimentation, reporting and so on.

Note: ETL or ELT is the process of data engineering that allows us to extract the data from internal or external resources, transform/process the data, load into the data store and make it available for use to different teams and stakeholders.

Another source of data is non-relational data which can be as simple as a spreadsheet dataset, or document store databases; data lakes can also store non-relational data.

Once we have the data sourcing component of the project ready, the next critical element of the machine learning system is data versioning, especially once the data is fetched from the data source and processed as part of the data pipeline of the MLOps project. We will look into data versioning in the next section.

Data versioning

The easiest way to handle data versioning is to treat data as part of the source code as well and version the data similarly as we version the source code, that is, using Git. Now we know the limitation of Git for storing the amount of data we might have. That is where **Data Version Control** (**DVC**) comes in; it is an open-source tool to precisely accomplish this.

DVC provides data versioning out of the box. We can build DVC pipelines to process and version the data. This specific version of the data then can be used for training and associated with the version of the trained model. For example, let us assume we build a DVC pipeline to process the data, and the processed data is then tagged as v1. We can use this version of the data to train a model and tag it as m1. Now, we can associate v1 of the data to the m1 model. This will make it easy to track what version of the data is used to train which version of the model. That way, we will maintain the consistency and lineage of the data being used and the models being trained. It also provides us with a way to audit the models and the corresponding data whenever we want.

Assuming DVC is already installed in the system, it can be added to the cookie cutter project template by running the following command inside the `cookiecutter` Git project:

```
dvc init
```

This will create the following internal files, which should be committed to the Git repository:

- `.dev/.gitignore`
- `.dev/config`

Once we have the files committed to the Git repository the next step is the ability to prepare the data to be tracked with DVC. The code to process the data from raw to interim and to the final stage needs to be under the `src/data/make_dataset.py` folder. The actual data needs to be versioned and stored under the data folder. The script should be able to fetch the data from the source, be it raw data or a data store, process it, and create a dataset usable in the code by the model.

Once the data is stored under the data folder, we can start tracking the file using `dvc add` command. Assuming the filename is `data.xml`, we can use the following command:

```
dvc add data/data.xml
```

This will instruct DVC to create a new text file under the data folder with the name `data.xml.dvc`; this file does not store the actual data but the metadata of the actual data file. It can be easily versioned in Git. What happens behind the scenes when we run the `dvc add` command is that it moves the actual data to the project's cache `.dev/cache` and links it back to the workspace using the hash value of the data file. This hash value is stored in the `data.xml.dvc` file.

The actual data can and should also be pushed to a remote backend. DVC supports many remote storage types, with **AWS S3**, **Microsoft Azure Blob Storage**, **Google Cloud Storage (GCP)** being the most widely used ones. Remote storage can be added, data can be pushed and pulled from remote storage using the following commands:

```
dvc remote add -d storage s3://mybucket/dvcstore
```

```
dvc push

dvc pull
```

In this structure, we assume that we will have the dataset available in files that can be stored in raw, interim, and processed folders but that is not always the case. Often, the initial data or even the data in the final format is stored in the database, data warehouse, or data mesh. In those cases, the recommendation will be to query the data from the database, process it if required, and store the final processed data in the data folder. On the other hand, one can read the data from the database, store it in the data/raw folder, and then process it to bring it into the data/processed format. There are multiple ways to reach there. As a result, the final version of the processed data is checked in DVC and Git to be used in the model training to map the model and data one-to-one for maintaining the lineage.

Exploratory data analysis

Once data sources are streamlined, the next step, as indicated in the MLOps workflow in *Figure 5.2* is data analysis. Data analysis starts with an exploration of data to make sense of it. Within this component of the ML process, data scientists and machine learning engineers analyze the statistical properties of the data available. The aim is to determine if the data will address the business question. This is where we must be in close contact and iteration with business stakeholders.

In the context of MLOps, it is an iterative process of exploring, sharing, and prepping data for the ML lifecycle. It requires creating reproducible, editable, and shareable datasets, tables and visualizations. These explorations generally start with notebooks either locally like **Jupyter notebooks** or cloud-based like **Google collab** and so on. In the `cookiecutter` project template, we have a provision to store and manage these notebooks under the `folder /notebooks/.`

These notebooks should strictly be used for data analysis and are not supposed to be part of the machine learning pipeline for automation. Once we have a good chunk of code from the initial analysis that can be used for the pipeline, it is recommended to move that code out of notebooks into proper scripts under the intended folder structure so that we maintain

consistency and these scripts can be used in the pipeline. Please refer the following figure:

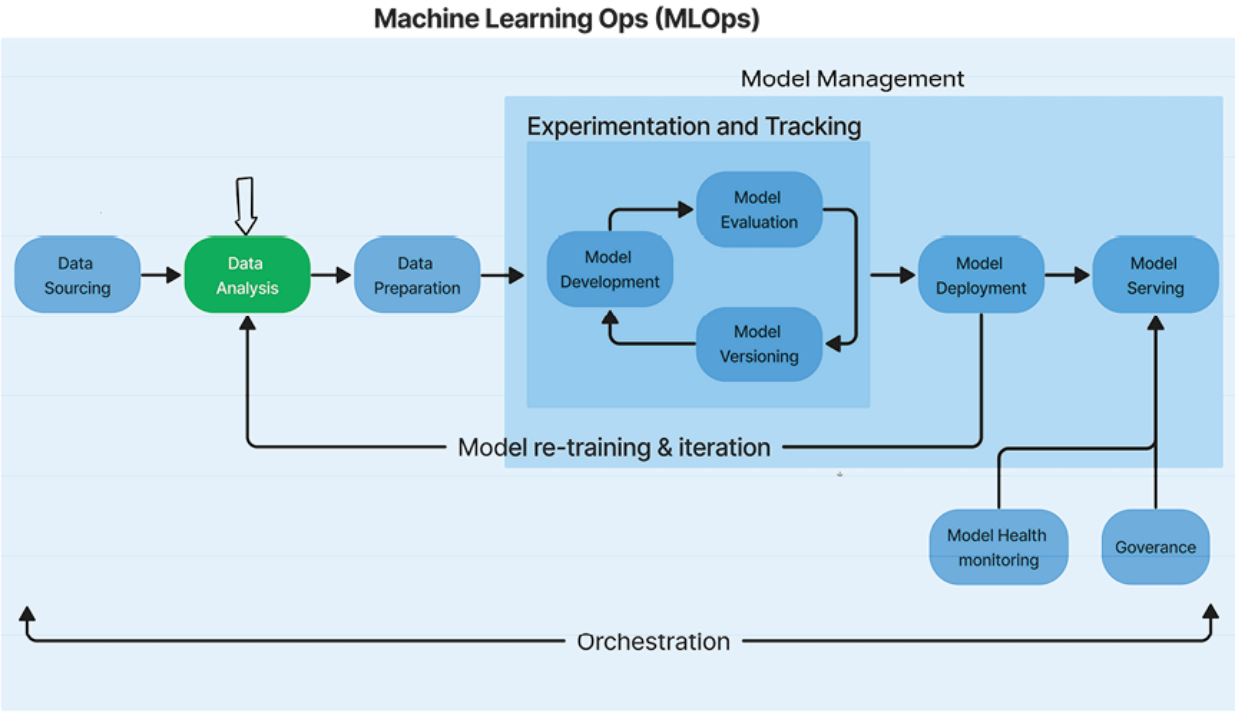

Figure 5.2: *Exploratory data analysis stage of MLOps-based ML project lifecycle*

A general outcome of this exploratory data analysis is to come up with some of the common prerequisites which tell us if the data is fresh enough or in good shape to be used for training and retraining of the models. Moreover, those analyses can be used in the pipeline during the continuous automatic training runs to make sure we are running the training pipeline if the analysis tells us that the data is available and is usable for next round of training.

Data preparation

Data preparation, as the name suggests (shown in *Figure 5.3)*, is the third component where we prepare the derived data required for our experiments or model development runs. Data preparation may consist of two sub-components:

- **Derived data preparation**: These are the code snippets or scripts which will convert our data from the initial format in the dataset to the

vectors or embeddings required for the model development or feature engineering.

- **Feature engineering**: Feature engineering is the process of transforming a raw data point into a feature vector. It consists of conceptualizing and deciding upon a feature, possibly during the exploratory data analysis, and then creating the code script to transform the entire dataset into a set of features to be used in model development. Feature engineering is a creative process where data scientists or ML engineers apply their imagination, intuition, and domain expertise to create a set of features to be used in model training.

Feature engineering is a vast topic and is beyond the scope of this chapter and book. In the context of MLOps, feature engineering is another step which contains scripts that need to be executed to convert the raw data to data used for model development. Please refer the following figure:

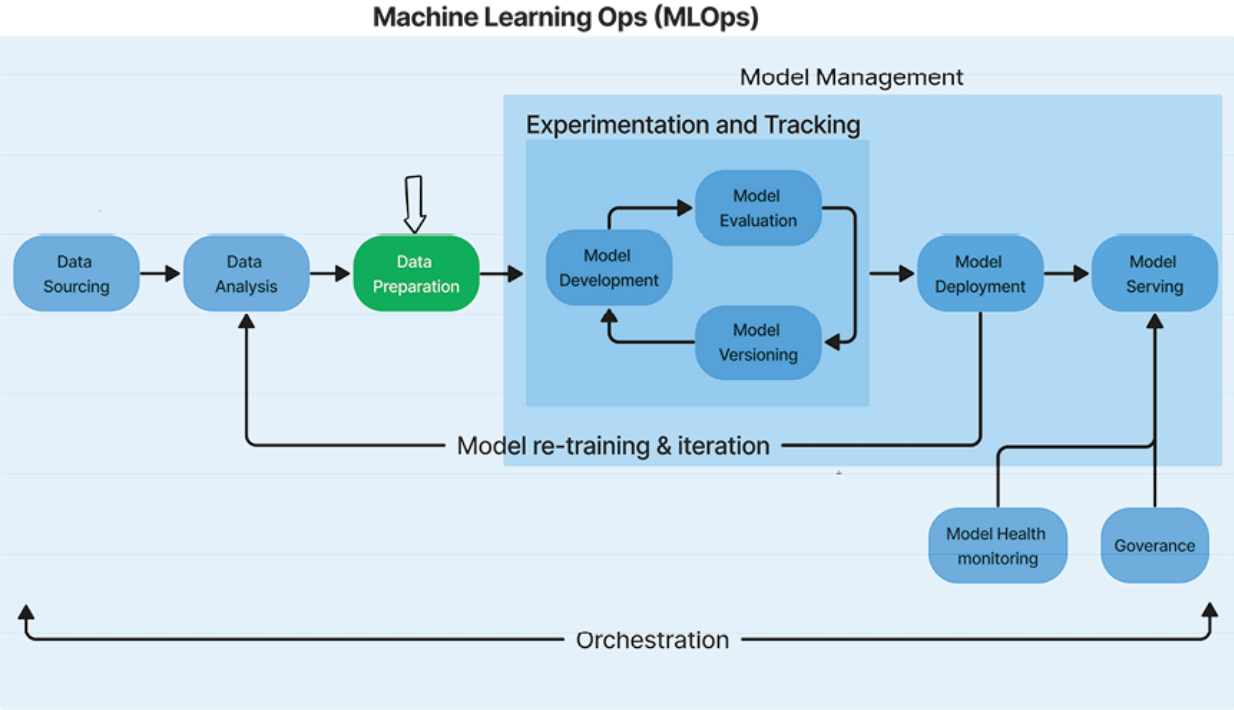

Figure 5.3: *Data preparation stage of MLOps-based ML project lifecycle*

The code to process the data and create a feature set for model development needs to be under the `src/features/build_features.py` folder. Actual

feature data can be stored and versioned in DVC using the same steps as we did in the **Data Versioning** section or stored these features in a feature store.

Note: The decision to have a feature store or not differs significantly across projects and teams. It also depends on the maturity level of the infrastructure, as feature store usage and maintenance introduce significant infrastructure complexity.

Some of the best practices for data preparation are:

- **Keep code, model, and data in sync:** The version of the feature extraction code must be in sync with the model's version and the data used to build it. These three have to be deployed or rolled back at the same time. Each time the model is loaded in production, it is useful to check that the three elements are in sync (their versions are the same).
- **Isolate feature extraction code:** The feature extraction code must be independent of the remaining code that supports the model. It should be possible to update the code responsible for each feature without affecting other features, the data processing pipeline, or the way the model is created. The only exception is when many features are generated in bulk, like in one-shot encoding and bag-of-words.
- **Log the values of features:** Log the feature values extracted while running the experiments. MLflow can be used for that, and values for each experiment run can be logged. We will look at how to use MLflow in the code in the next section.

Model development

Model development is the next step in developing any machine learning system as shown in *Figure 5.4*. It falls under the experimentation and tracking component of MLOps lifecycle and can consist of multiple rounds of development, evaluation and versioning.

When it comes to model training, data scientists explore multiple algorithms and hyperparameter configurations using the prepared data. The goal is to determine and come up with the best-performing model that meets the required standards and can perform better than the set threshold metrics.

In MLOps, this component taps into popular open-source libraries for training and improving model performance. When we start with the model

development part, we are assuming that the development strategy, target expected performance level, performance metric to track, and other ML decisions are already made. We are purely starting on building the scripts for model development/training and tracking those metrics in MLflow or similar experimentation platforms. Please refer the following figure:

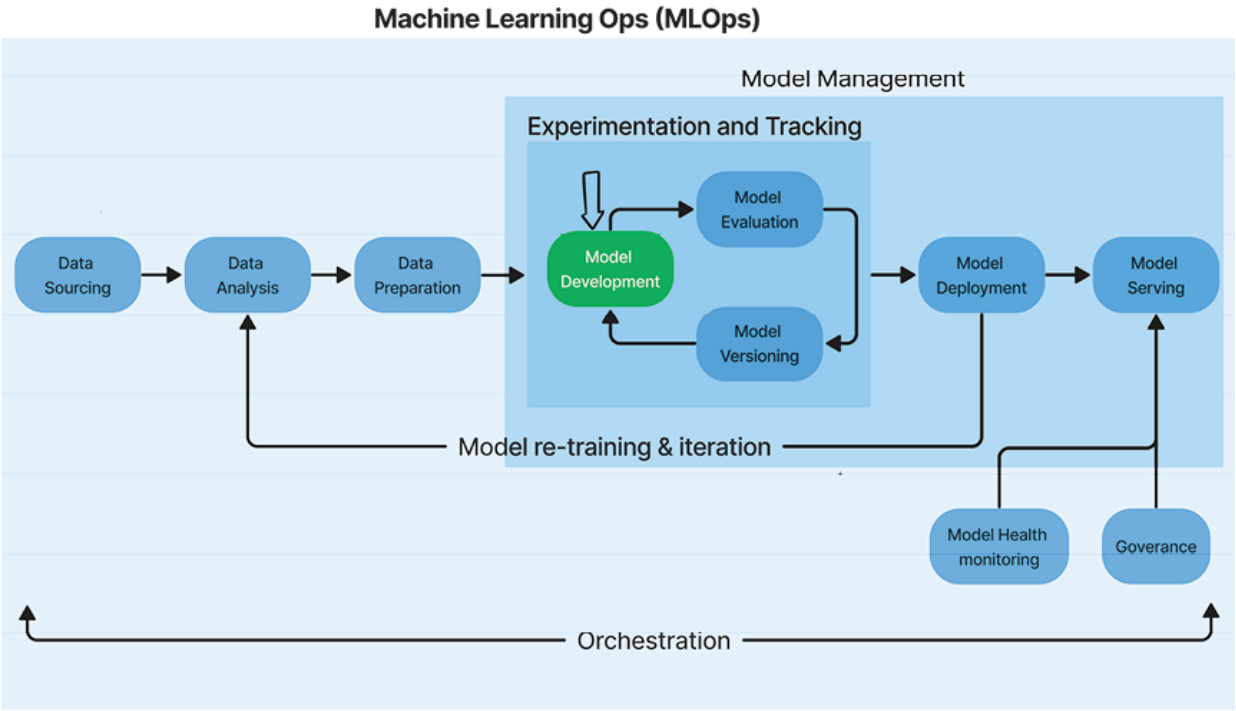

Figure 5.4: *Model development stage of MLOps-based ML project lifecycle*

The code to train the model needs to be under the **`src/model/train_model.py`** folder. We will not be covering the code to train the model with the assumption that it is already known and done by the data scientist or machine learning engineers. We will cover how to add MLflow client to the training code which will start tracking the changes. Let us assume we are using the **LightGBM** algorithm for our model:

```
# Other imports for the script

import mlflow

Import mlflow.lightgbm
```

```
# Enable auto logging
mlflow.set_tracking_uri('http://yourdomain.com/mlflow')
mlflow.lightgbm.autolog()

# Load training data

mlflow.set_experiment('LightGBMClassifier')

def main():
        with mlflow.start_run() as run:
                # Train model

                # Let's log some custom parameter to the MLflow run
                mlflow.log_params({
"model_name": "LGBMClassifier",
"max_depth": 5
})

                # Let's also log custom metrics which is outside the auto logging
                mlflow.log_metrics({
        key1: value1,
```

```
        key2: value2
})


print("Run ID:", run.info.run_id)


if __name__ == "__main__":
        main()
```

Once this code is executed manually or as part of the auto orchestration for re-training, it will log the run to the MLflow server. We can visit the server to see the experiment and download the model file from the browser. The above code is self-explanatory, but the important parts are:

- **Line 2-3**: Code starts by importing MLflow and LightGBM.
- **Line 5**: This is where we set up the MLflow server's tracking URI at which the server is running and should be used for tracking and logging.
- **Line 6**: This is where we initiate the autologger. Autologger by default logs the models as well so we do not need to specifically log the model to MLflow.
- **Line 8**: Set the experiment in MLflow under which everything needs to be tracked.
- **Line 10**: This is the first line for every experiment that is tracked by MLflow. It is how we indicate the start of an MLflow run.
- **Line 13**: Shows us how we can log a custom parameter to MLflow.
- **Line 18**: Shows us how we can log a custom metric. This comes in handy when we get to logging and continuous monitoring. We can use metric logging to log random metrics we want to track.

Deep dive in MLflow workflow

As we have seen in the example above, MLflow gives us the option to log parameters or metrics during the run manually but if we need to track everything manually, it will become tedious very soon to extract all the parameters and metrics on our own. That is why Mlflow also exposes auto-logging for many popular ML frameworks, including:

- `sklearn.autolog`
- `tensorflow.autolog`
- `keras.autolog`
- `xgboost.autolog`
- `lightgbm.autolog`
- `pytorch.autolog`

The autologging classes automatically record a model's parameters, its training metrics, and, whenever available, fit params like early stopping, epochs, and similar. In the case of `sklearn`, the `autolog` class also records the results of hyperparameter tuning trials with grid search or results of pipeline objects. Auto-logging class also logs the models by default which allows us to skip manually logging the model using `log_model()`.

At the end of this component, we should have the model trained and ready to be deployed for serving, but before that, there are a couple of other components we need to address. The next one is the model evaluation to evaluate whether the resultant model is performing above the threshold metrics and can be used in the production environment. That is what we will look into next.

Model evaluation

Before a model is versioned and deployed, as *Figure 5.5* below indicates, it goes through the evaluation and validation step to be sure it performs above the estimated baseline level of performance. Any models which are running in the production environment must be carefully and continuously evaluated. Please refer to the following figure:

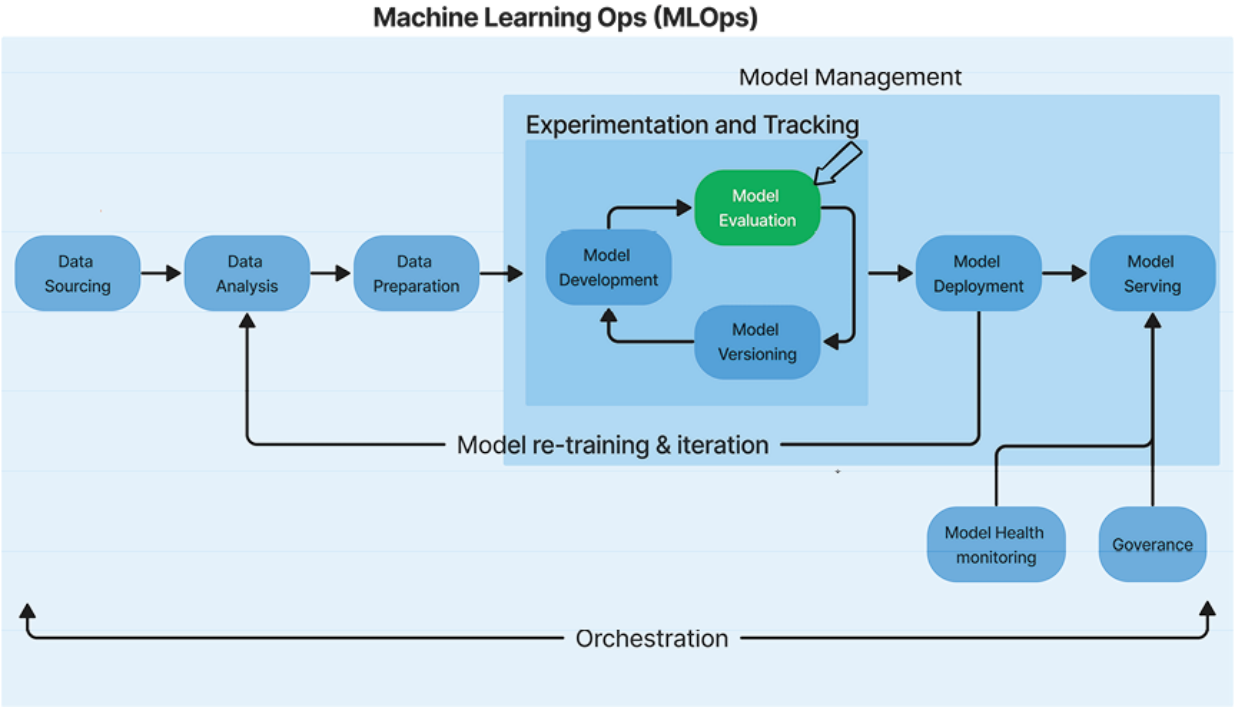

Figure 5.5: *Model evaluation stage of MLOps-based ML project lifecycle*

Depending on the model's applicative domain and the organization's business goals and constraints, model evaluation may include the following tasks:

- Log and compare the main properties of distributions of the training data versus the production data. Comparing the statistical distribution of examples, features, and labels, in both training and production data, is how distribution shift is detected. A significant difference between the two indicates a need to update the training data, and retrain the model.
- Evaluate the performance of the model. Before the model is deployed in production, its predictive performance must be evaluated on the external data, that is, data not used for training. The external data must include both historical and online examples from the production environment. The context of evaluation on the real-time, online data should closely resemble the production environment.

The code to evaluate the model can be included as part of the training script itself or a complete separate script. If it is part of the training script then it can be added to the `src/model/train_model.py` otherwise if the evaluation

script is kept separate, the file containing the evaluation code can be created under `src/model/evaluate_model.py` folder. The output or any other significant metric can be logged to MLflow using the custom metric logging as we discussed in the previous section, these metrics can be compared between different versions of the model to keep track of the performance of the models over time.

Model versioning

Model versioning is the next important component that helps us in maintaining the traceability and lineage of the current models running in different environments, especially if we are re-training the model automatically. It is part of the **Experimentation and Tracking** sub-system of the MLOps workflow, as indicated in *Figure 5.6*:

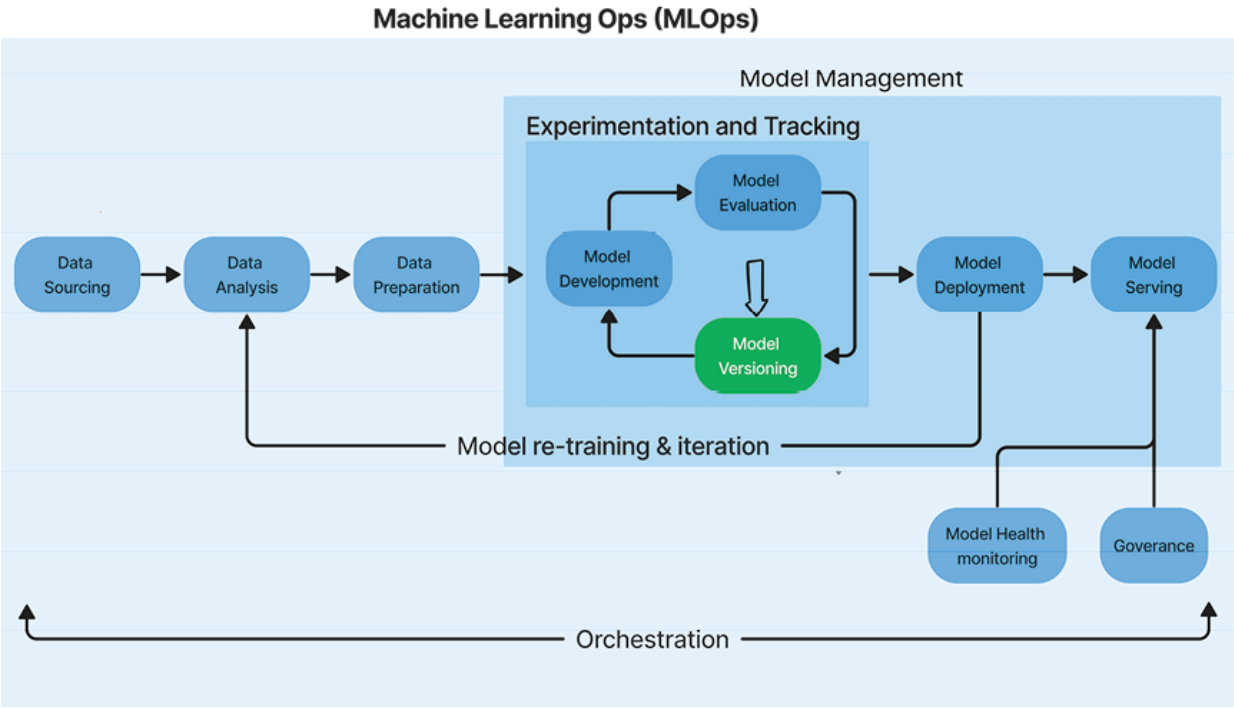

Figure 5.6: *Model versioning stage of MLOps-based ML project lifecycle*

MLflow's model registry component makes it possible for us to store versioned models. There are three ways a model can be registered and versioned in MLflow:

- With each run of MLflow experiment, if we are using `autolog()`, it will automatically store the models as artifacts, we can also pass `registered_model_name` parameter to the function to register the model and auto-increment the version of the model if it already exists or create a new one with version 0.
- If we are not using `autolog()`, we can use the `log_model()` function to register a model by passing the same `registered_model_name` parameter.
- The third option is to not log the model using the API but register the model from the UI after manually verifying, evaluating the run and choosing the best model.

This means that this model will be versioned and will be available in the **Models** section in MLflow UI.

Based on the different requirements and maturity level of the project, any one of the options to register a model can be used. For our workflow, we are discussing a recommended method to adapt is:

- Not to have the models registered in autolog. Models will still be stored as artifacts in each run.
- Execute the model evaluation scripts based on the output if the model is acceptable and is performing well.
- Register the model using `log_model()` function.

This approach will allow us to keep the registry clean and not have hundreds of model versions that may or may not end up in production if they are not performing well, especially if we have an auto re-training pipeline in place which may retrain the model even multiple times a day.

If keeping the registry clean is not a consideration or if we are retraining the model on a longer cadence then we can decide to use autolog and just register and version every run of the experiment.

The decision to use autolog or manually registering models may also depend on other factors such as the need for model tracking, collaboration or reproducibility requirements of the project, as well as auditing requirements.

As we know, MLflow is a very flexible platform and can be adopted as per our use case and project requirements.

Deep dive in MLflow models

In the **Models** section, we will find all models that are registered. By selecting one of them, we will see all the existing versions of that model. Here, a given version can be assigned to a given stage.

Stages enable us to assign a label to the existing models and have visibility of the state of the model that has been versioned, that is, a registry with all past registered models that have been put into production in the past and which are currently in production.

Conclusion

We started the chapter by continuing the discussion on `cookiecutter` data science template repository, which we looked at in the last chapter, and looking at the best practices for managing code repositories for MLOps-based projects. We understood the use of `pre-commit` hooks for better code management and to enforce proper structure to the repository before diving into components of the MLOps project lifecycle, including data sourcing, versioning, exploratory data analysis, data preparation, feature engineering, model development and evaluation. Most importantly, we looked at how these components fit into the `cookiecutter` data science project template and how they can integrate with tools like DVC and MLflow to provide better management of the projects.

In the next chapter, we will continue our exploration of the MLOps project lifecycle components. We will see how the models can be deployed for servicing and how it all fits into MLOps project scope and structure.

Points to remember

- Use `pre-commit` hooks like `Black`, `Flake8`, and so on. to enforce code quality and formatting standards in the ML project repository. This makes the code more maintainable.

- Version control the data using DVC. This allows us to track the data used for each model version.
- Do exploratory data analysis in notebooks to understand the data. Then, move any reusable code to scripts.
- Feature engineering is important to convert raw data into useful features for ML models.
- Use MLflow to track experiments, log parameters and metrics, and register model versions. This provides model lineage and auditability.
- Evaluate models on test data before deployment to production. Remember to log key metrics for tracking purposes.
- Model versioning with MLflow staging allows tracking models from development to production.
- Follow MLOps principles like CI/CD and infrastructure-as-code to automate the ML lifecycle.

Key terms

- **pre-commit hooks**: `pre-commit` hooks are libraries or scripts to automatically check and clean the code before it is committed to the code repository. They are executed when committing changes to Git.
- **Data Version Control** (**DVC**): DVC is a free and open-source, platform-agnostic version system for data, machine learning models and experiments. It is designed to make machine learning models sharable, experiments reproducible, and to track versions of data and pipelines. DVC works on top of Git repositories and cloud storage.
- **Personally Identifiable Information** (**PII**): PII is information or data that, when used alone or with other relevant data, can identify an individual such as name, social insurance number, date of birth and so on.
- **General Data Protection Regulation** (**GDPR**): GDPR is the European Union regulation on information privacy in the **European**

Union (**EU**) and the **European Economic Area** (**EEA**). It also governs the transfer of personal data outside the EU and EEA.

- **RDBMS**: A **Relational Database Management System** (**RDBMS**) is a system for databases which are based on the relational model of data.
- **Feature engineering**: Feature engineering is the process of extracting features from raw data
- **MLflow**: MLflow is used to manage the machine learning lifecycle from initial model development through deployment and beyond to sunsetting

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 6
Model Deployment and Serving

Introduction

In this chapter, we will look into the next steps of the machine learning project lifecycle: the deployment of the model we prepared and how to serve them for consumption by the API. The main focus will be how this deployment and serving components fit with MLOps.

We have come a long way with exploring the principles and structure of projects for improved machine learning and MLOps practices. We will look at how the machine learning projects can be deployed, the options we have for different principles of deployment and serving, as well as the mode of deployment.

Structure

In this chapter, we will discuss the following topics:

- Model deployment
  - Static deployment
  - Dynamic deployment on edge devices
  - Dynamic deployment on a server
- Deployment strategies

    - Single deployment
    - Silent deployment
    - Canary deployment
    - Multi-armed bandits
- Model inference and serving
    - Modes of model serving
    - Model serving in real life

Objectives

By the end of this chapter, you will understand what different deployment strategies can be adopted and how models can be served for consumption by end users, applications, or machines.

Model deployment

The model prepared, evaluated, and versioned in *Chapter 5: Data Preparation and Model Development*, is now ready to be deployed. Model deployment is the process of making a model available to be used by end-users or downstream systems. The model can be deployed by batch, streaming, or online serving. The approach we choose from these three depends on the particular use case and its requirements. Once models are deployed, then comes the serving of these models. In model serving/inference, the system accepts the query from the end-users, which transforms into a feature vector. The model then uses these vectors for prediction and scoring. The results of this prediction and scoring are then returned to the end user.

Model deployment, as shown below in *Figure 6.1*, is the seventh stage in the MLOps process of a machine learning project. Please refer to the following figure:

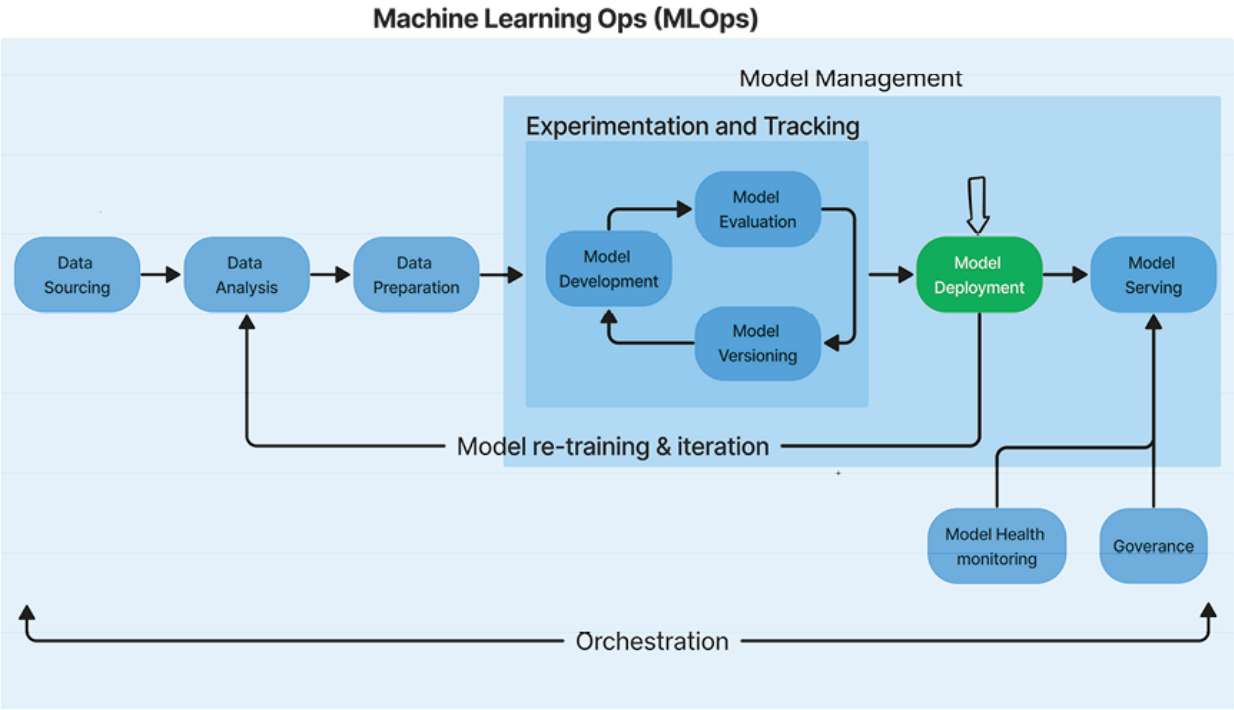

Figure 6.1: *Model deployment stage of MLOps-based ML project lifecycle*

The MLOps process can automate the testing of new models against current production models, deploying new models to serving clusters and REST API model endpoints, and monitoring the predictions after deployment. Ultimately, we will need to keep tabs on the availability of the model, its predictions and performance on live data, and the performance of the ML system from a computational perspective.

A machine learning model can be deployed for consumption by utilizing various approaches:

- Static deployment, integrated within an installable software package.
- Dynamic deployment on the user's device
- Dynamic deployment on a remote server, or
- Through model streaming techniques

Static deployment

The process of statically deploying a machine learning model is similar to the conventional software deployment approach. In this case, an installable binary of the entire software is prepared, with the model being packaged as a

resource that can be accessed during runtime. There are numerous advantages to static deployment, including:

- The software has direct access to the model, resulting in faster execution times for users.
- There is no need for the system to upload any data to an outside system or server, which then runs the prediction.
- There is no dependency on the external system, which means the end-users, or the system need not be online to process any data.
- The software vendor may not need to worry about maintaining the model operations, as this responsibility falls on the user. The vendor does need to provide updates and support for the models, but the ultimate responsibility for the operations or updates is in the hands of the end-user.

However, static deployment also has certain drawbacks. One significant concern is the unclear separation between the machine learning code and application code, as most of these systems end up becoming tightly coupled with the application code. This makes static deployment complex and challenging because if we want to upgrade or change the ML model, oftentimes, it means that we need to upgrade the complete application running on the user device. Additionally, the process becomes further complicated if the model has specific computational requirements for predictions, like GPU or a specialized infrastructure, as we will then need to make sure those requirements are satisfied by the user devices. This, in turn, can lead to potential limitations and complications in the deployment process.

Having said that, there are certain scenarios where a static deployment strategy is helpful and should be preferred:

- **Limited resources**: Training machine learning models can be computationally intensive and require substantial resources. In situations where there are constraints on computational power, static deployment might be the only feasible option.
- **Regulatory and compliance reasons**: In some regulated industries, models must be certified and validated before deployment. If changing

the model frequently requires recertification, it might be easier and more practical to deploy a static model, especially if the model performance is acceptable.

- **Latency constraints**: In some real-time applications where low latency is critical (such as some aspects of autonomous vehicles or certain industrial automation tasks), static deployment might be necessary to ensure immediate responses without the latency introduced by depending upon external systems.

Dynamic deployment on edge device

Dynamic deployment on the edge devices overlaps with the static deployment process. These are the edge devices controlled by the users. We have limited control over those, and the model runs on those edge devices. However, the key difference lies in the fact that, in dynamic deployment, the model is not incorporated within the application's binary code. This results in better separation of concerns, as model updates can be pushed without updating the entire application running on the user's device. It also allows us to incorporate logic in the application where it was selected and download the appropriate model based on the infrastructure and computer resources available on the users' edge devices.

Following are some of the techniques with which dynamic deployment can be planned as part of the application:

- **Model parameter deployment**: In this technique, only the finalized parameters are included in the model file, while an environment running the model is installed on the end user's device. Lightweight versions of machine learning packages like **TensorFlow** run on the user's device and can make this approach possible.
- **Serialized object deployment**: In this technique, the trained machine learning model is packaged into a serialized format file that comprises a serialized object that the runtime application can deserialize. This technique eliminates the need for a runtime environment for inference on the user's device, as all required dependencies are deserialized along with the model object, and the application running the model can deserialize and use the model. However, this method can lead to

substantial updates every time we want to release an update, which may be problematic when handling a large number of users simultaneously. There is another limitation in terms of file sizes; serialized files can end up being quite large, and that can be a concern for devices with limited storage capacity. Some common serialization formats are **Pickle** or **Open Neural Network Exchange** (**ONNX**).

- **Browser-based deployment**: Today, almost every device we have, be it a desktop, mobile, or even most of the integrated edge devices, can access a browser. Frameworks like `TensorFlow.js` allow us to run JavaScript-compatible versions of the pipeline, which can be used to train and execute models within the browser using the JavaScript runtime environment. It can also detect and take advantage of the resources available on the device. If a desktop is equipped with a GPU, `TensorFlow.js` can also utilize that to accelerate the processing.

The primary advantage of dynamic deployment on users' devices is the speed of calls to the model for the user. This also eliminates the need to run and maintain a server or cluster infrastructure.

Dynamic deployment has some drawbacks as well, particularly during model updates. Serialized objects can be large, and users may be offline during updates or may disable future updates. This can result in users employing different model versions, complicating server-side application upgrades and maintenance. Further, deploying models on users' devices exposes them to third-party analyses and attempts at reverse engineering the model. Another disadvantage is similar to static deployment, that is, deploying models to users' devices makes it challenging to monitor model performance due to possible limited connectivity of the edge devices to send monitoring and observability metrics. Monitoring also requires some resources, which can be challenging on edge-devices with limited storage and processing capabilities. There are ways to overcome some of these challenges, which require a well-thought-out approach that balances the need for real-time insights with the constraints of resource and connectivity limits. If dynamic deployment on edge devices is the approach we want to take, some of the possible solutions can be, local logging and monitoring along with periodic reporting so that we are not sending everything to the server but only high-sensitivity metrics and alerts to the server. We will also need to use

lightweight monitoring tools that work well with the edge-device processing and storage capabilities.

Dynamic deployment on a server

Due to the complexities mentioned earlier and challenges in performance monitoring, the most common deployment approach involves deploying the model on a server (or clusters of servers) and making it accessible through the **Representational State Transfer Application Programming Interface** (**REST API**) in the form of a web service or **Google's Remote Procedure Call** (**gRPC**) service. There are multiple approaches regarding what infrastructure we can use to deploy models on a server, which we will explore in this section.

Virtual machine deployment

As we mentioned above, there are two options to serve the output: REST APIs and gRPC. This can be made possible by deploying the model in the cloud environment. This setup is similar to any typical web server deployment, which runs on a virtual machine in a cloud environment. Once deployed, it will accept input requests from the end-users containing the request details. The web service then verifies the request followed by passing it to the machine learning system to be processed. Once processing is complete, the web server takes the output of the machine learning system, transforms it into the expected output and returns to the end-user. To handle high loads, multiple identical virtual machines can be configured to run in parallel along with a load balancer in front on them to provide the required scalability.

Suppose we plan to use multiple virtual machines. In that case, a load balancer is a must-have, as it is responsible for receiving incoming requests and distributing these requests to a particular virtual machine based on its availability and the load. Depending on the requirement and load, we can add or remove the virtual machines either manually or by defining and using an autoscaling group that is responsible for launching and terminating the virtual machines based on the request load. *Figure 6.2* illustrates this deployment pattern, with multiple virtual machines that are controlled by the **Auto Scaling Group**, and requests from the users are handled by the load balancer. Please refer to the following figure:

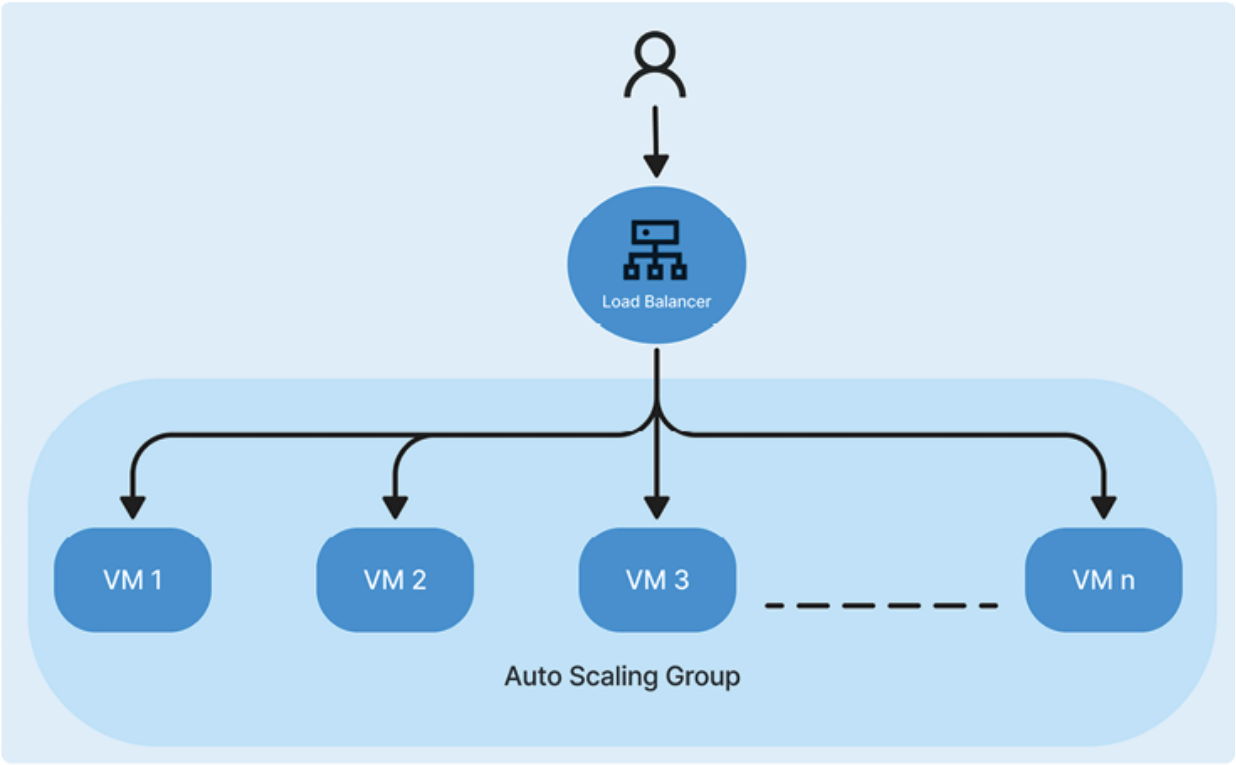

Figure 6.2: *Virtual machine deployment setup with load balancer and auto scaling group*

In this approach, each instance of the virtual machine needs to:

- Have the complete code, not just the model execution code but also the code for web server,
- Also need to run any required transformations,
- Running the input through the model,
- Preparing the output and formatting the output in an acceptable format.

Deploying on a virtual machine offers several advantages, such as straightforward architecture and minimum complex infrastructure requirements. However, this deployment method also has some downsides, like maintaining servers, whether physical or virtual. Network latency can also be a concern, depending on the speed at which we may need to process scoring results. Virtual machines can be less flexible and slower to scale compared to container based or serverless solutions. Deploying the applications and models on virtual machine based environments generally has higher costs as well in comparison to the container based or serverless

deployment. In the next sections, let us look at these two approaches as well and see how similar and different they are.

Container deployment

As we all know, in recent years, container-based deployment has emerged as a more efficient and flexible alternative to virtual-machine-based deployment. It is similar to a virtual machine-based environment in that it isolates the resources and environment the code is being run on. A major improvement in container-based deployment is that containers run on the same physical machine and share the OS Kernel and libraries while having separate user spaces. This is in contrast to virtual machine deployments, where virtual machines are completely isolated and operate their instance of the operating system. This difference makes container-based deployment better than VM deployment.

The deployment process for container-based systems involves installing the machine learning system and the web service within a container, which is typically a Docker container. However, alternatives exist, followed by using an orchestration system to run and maintain these containers in a cluster. Kubernetes is a widely used choice for container-orchestration systems, whether running on-premises or on a cloud platform. All major cloud platforms support Kubernetes orchestration systems as well as provide proprietary orchestration systems like **AWS Fargate**, **Google Kubernetes Engine**, **Azure containers**, and so on.

As illustrated in *Figure 6.3*, a cluster is built in an environment handed over to the container orchestration system. This orchestration system then manages all the resources and autoscaling needs of the system. When deploying a project in a cloud environment, a cluster autoscaler can launch or terminate containers based on the cluster's requirement, usage, effectively adding or removing them from the cluster as needed.

As with every other deployment method, there are several advantages to deploying in a container compared to using a virtual machine. Containers are more resource-efficient and can automatically scale with prediction requests from users facilitated by the orchestration systems (for example: **Kubernetes**, **AWS ECS** and so on) based on the defined rules and triggers. This scaling capability extends to the concept of scale-to-zero, which allows a container to be reduced to zero replicas when idle and brought back up

when needed to serve a request. Consequently, resource consumption is lower than that of always-running services, which leads to reduced power usage and lower costs for cloud resources. However, there is a notable drawback to containerized deployment: it is generally perceived as more complicated and requires specialized expertise. Despite this challenge, container-based deployment remains an attractive option for its resource efficiency and flexibility in scaling to meet varying demands. Please refer to the following figure:

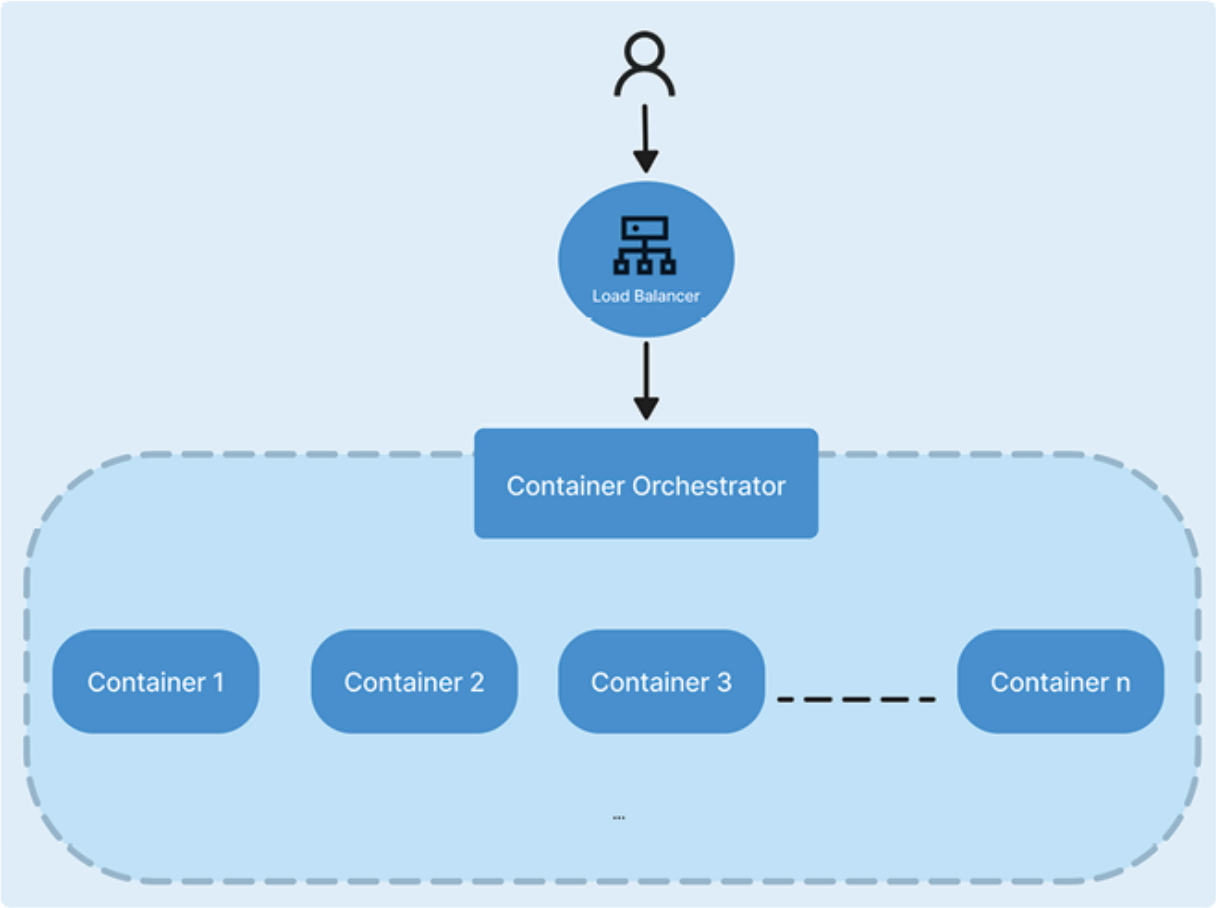

Figure 6.3: *Container-based deployment cluster setup with load balancer and container orchestrator*

Serverless deployment

All major cloud infrastructure providers, AWS, Google, and Microsoft, provide serverless computing services. These services are Lambda-functions on Amazon Web Services, functions on Microsoft Azure, and functions on Google Cloud Platform.

Serverless deployment involves preparing a zip archive containing all the necessary code to run the machine learning system, including the model, feature extractor, and prediction code. Along with the core code, the zip archive should also include a specific file or a function definition as per the requirement of the cloud provider, which acts as an entry point for running the code instance. Finally, this code is zipped into a single component and is uploaded and deployed.

The cloud providers then generally offer an API for submitting inputs which are then diverted to the zip archive's serverless function. This API specifies the function's name, provides the payload, and returns the output. Once we have defined the deployment configuration, defined triggers, and specified resource allocations, the cloud provider's service handles everything behind the scenes, including deployment of the code in the zip archive, making it accessible and available to process requests and sending back the output to the user. We do not need to care about the servers or any other resources, thus the serverless deployment. Keep in mind even though serverless deployment is sort of a hands-off approach, there are still optimizations and tasks we might have to do to make it production-ready like optimizing the resource allocations, storage and so on to make sure everything is running smoothly.

Having said that, with this ease of use, there comes a set of limitations as well which can become critical in various real-world scenarios, like the time function can take for execution, zip archive size, and the amount of RAM and other resources available during runtime. Out of these, the zip file size is the biggest challenge for serverless deployment. A typical machine learning model requires multiple heavyweight dependencies, such as Python's libraries like `Numpy`, `SciPy`, and `scikit-learn`, for proper execution, which can sometimes be a challenge to fit within the size limits of the zip file for the serverless environment. This limitation becomes critical for use cases like **real-time image recognition** or **natural language processing** where large model files and the need for GPU or a large amount of CPU availability are crucial.

Despite these constraints, there are numerous advantages to serverless deployment as well. Most notably, there is no need to provision resources like **servers** or **virtual machines** which is abstracted and handled by the service providers, install dependencies, or maintain and upgrade the system

which can save organizations significant time and resources. So in scenarios where workload can fluctuate greatly, serverless systems shine due to their high scalability potential. They offer high scalability, supporting thousands of requests per second effortlessly. Additionally, serverless functions can operate in both **synchronous** and **asynchronous modes**. In synchronous mode, functions are executed immediately in response to a specific event or request and are called **waits for the response**. This is a useful mode for scenarios where real-time or immediate response is required. Whereas in asynchronous mode, serverless functions are triggered in response to an event or request, but the caller does not necessarily wait for the function to complete its execution; it continues its operations and the serverless function operates independently.

The use of serverless deployment also helps carry out the canary deployment easily. **Canarying** is a software engineering strategy where updated code is pushed to a small group of end-users, usually unaware of the change. This method allows for quick reversals if the new code contains bugs. It is easy to set up multiple versions of serverless functions in production and send low-volume traffic to one version for testing without affecting many users. We will look at canary deployment in detail in the next section on deployment strategies.

There are some drawbacks to serverless deployment, such as the zip archive size limit and the available RAM and other resources during runtime. One major drawback of serverless deployment is the unavailability of GPU access, which makes it impossible to deploy some models and applications that depend on the GPU to process the data.

Streaming model deployment

Streaming model deployment is a system that eliminates the need to have REST API interfaces for the models. In streaming models, the requests are sent to an input queue, where they wait to be picked up for processing by the system. Once the output is ready, those outputs are put into the output queue, which the end user can subscribe to receive updates.

The advantage of streaming systems is more visible in complex systems where multiple models are in play. Consider an input consisting of a social media feed. One model might predict the sentiment of each post in the feed, another could extract named entities, and a third might generate a popularity

score of the post, among other tasks. With the REST API deployment pattern, we would need one REST API for each model, and we would need to call these REST APIs one after the other to make it work.

Streaming operates differently. Instead of having multiple REST APIs in this example system, all the models are arranged into a topology based on how they need to run and the input-output dependencies. Topology here refers to the high-level structure or configuration of a stream processing application. These topologies are generally run on **Stream Processing Engines** (**SPE**) like **Apache Spark**, **Flink** or **Apache Storm**, and so on. The input and output queues are generally handled by streaming queue frameworks like **Apache Kafka**, **Akka Streams**, **AWS SQS**, etc. While a detailed discussion of these SPEs and SPLs is beyond the scope of this book, they all share a common property that distinguishes them from REST API-based applications. In this kind of streaming deployment, data can be continuously sent to the queues and processed by the system as it comes.

Deployment strategies

There are multiple deployment strategies that can be used to achieve the type of model deployment discussed in the previous section:

- Single deployment
- Silent deployment
- Canary deployment
- Multi-armed bandit

Single deployment

Single deployment is the most straightforward method of updating machine learning models. It involves replacing the old model with the new one and updating the feature extractor if necessary. This method can be applied in various environments, such as cloud-based servers, physical servers, or user devices. Although single deployment is simple, it carries risks since bugs in the new model or feature extractor may impact all users.

In a Cloud environment, we would start by preparing a new virtual machine or container that runs the updated version of the model. Once ready, the

virtual machine image or **container image** will be replaced. The old machines or containers are then gradually shut down, allowing the autoscaler to replace them with new ones. This process ensures a smooth transition to the updated model without any downtime for users.

For deployment on a physical server, we would need to upload the new model file and the updated feature extraction object to the server. Once these files are uploaded, and the code is updated to use the new versions, the web server needs to be restarted to start using the new models. This method allows the updated model to be implemented on the server without additional hardware or software changes.

To deploy on a user's device, we would push the new model file and any required feature extraction object updates to the device. Once the updates are in place, the software must be restarted to incorporate the changes. This method ensures each user has the most up-to-date model running on their device.

Single deployment has advantages and risks. The primary advantage of single deployment is its simplicity, making it an easy option for updating machine learning models. However, this simplicity comes with risks. If the new model or feature extractor contains a bug, it could affect all users, potentially causing widespread issues. Therefore, it is crucial to thoroughly test and validate the new model and feature extractor before deploying them using this method.

Silent deployment

A variation of the single deployment approach is known as **silent deployment**. In silent deployment, the new versions of the model and feature extractor are deployed in parallel to the existing deployment. They operate concurrently, but the users do not interact with the new version until an eventual switch is made. The predictions generated by the new version are logged but not shown to the users. These predictions are then audited and analyzed to make sure there are no issues or bugs with the new version. Once verified, the new version is launched, and the old one is decommissioned.

The advantage of silent deployment is providing ample time to confirm that the new model operates as intended without negatively impacting the users'

experience. However, this approach does have some drawbacks. It requires running double the number of models, which increases resource consumption and the associated costs.

By following this method, developers can ensure a smooth transition between model versions while minimizing the risk of unexpected issues affecting the users. However, it is essential to consider the resource implications and the feasibility of evaluating the new model without user exposure when deciding to implement silent deployment.

Canary deployment

Canary deployment, also known as **canarying**, is a strategy that involves deploying a new version of a machine learning model and its code to a small percentage of users. In contrast, most users continue to interact with the previous version. This approach allows data scientists and engineers to test and validate the performance of the new model, as well as assess the impact of its predictions, before fully deploying it to all users.

The primary advantage of canary deployment is that it minimizes the potential negative impact of bugs and issues on users. It helps in detecting common or critical issues early in the deployment process. If there are any issues with the new model, only a small fraction of users will be affected, making it easier to address and fix the problem before rolling out the model to the entire user base.

However, implementing canary deployment comes with challenges. One of these hurdles is the increased complexity of managing multiple versions of the model simultaneously. This requires additional resources and effort from the development team to ensure that each version is maintained, updated, and monitored.

A notable limitation of canary deployment is its inability to detect rare errors effectively. For example, if the new model version is deployed to 5% of users and a particular bug impacts 2% of those users, there is only a 0.1% chance that the development team will discover the issue. This is because the small percentage of users exposed to the new version dilutes the bug's impact.

To summarize, canary deployment is a valuable technique for testing and validating new versions of machine learning models in a controlled

environment, minimizing the potential negative impact on users. While it does introduce additional complexity and may not be effective at detecting rare errors, the benefits of early detection and mitigation of issues often outweigh these drawbacks. By carefully monitoring the performance and impact of the new version on the selected user group, engineers and data scientists can make informed decisions about when to fully deploy the updated model.

Multi-armed bandits

We introduced the concept of **multi-armed bandits** (**MABs**) as a method for comparing and selecting the best-performing model among multiple versions in a production environment. MABs are named after the one-armed bandit, a nickname for slot machines found in casinos, but in this case, we are dealing with multiple arms, which represent multiple versions of the models.

One advantage of the multi-armed bandit approach is that once the system matures and the algorithm has sufficient performance data to make an informed decision, it converges to selecting the best model(arm) most of the time. This means after the MAB algorithm converges, most users will be directed to the software version running the most effective model.

The major disadvantage of the multi-armed bandit approach is that making it work needs significant work and infrastructure in place. Potential complexities and challenges involved in implementing MAB algorithms require a mature process and infrastructure for the project. MABs require careful configuration, monitoring, and tuning, and their effectiveness depends on factors like the choice of reward metrics we use and the frequency of updates.

The multi-armed bandit strategy solves two main issues with deployment:

- Online model evaluation
- Model deployment

Online model evaluation

The MAB algorithm continuously updates its understanding of each model's performance, allowing it to make informed decisions about which model is

most effective in real time. This is particularly useful in situations where the environment is dynamic, and the performance of models may change over time.

Model deployment

The MAB algorithm not only evaluates the performance of different models but also makes decisions on which model to deploy in the production environment, ensuring that users are always interacting with the most effective model. This dynamic model deployment process can lead to improved overall performance for the system.

Model inference and serving

Model inference and serving is the component of MLOps which is all about making predictions for end-users and applications. The inference and service system may provide predictions to downstream systems via published tables of predictions, streams of predictions, or REST APIs. Some applications will benefit from batch or streaming inference, which is high-throughput and low-cost. Other applications will require low latency predictions, which can be provided via online serving.

In this section, we will delve into the best practices of model inference and serving in production. As indicated in *Figure 6.4*, this essential component encompasses the eighth stage in the machine learning project lifecycle with MLOps. This is the stage that provides the interface for the end-user to interact. Please refer to the following figure:

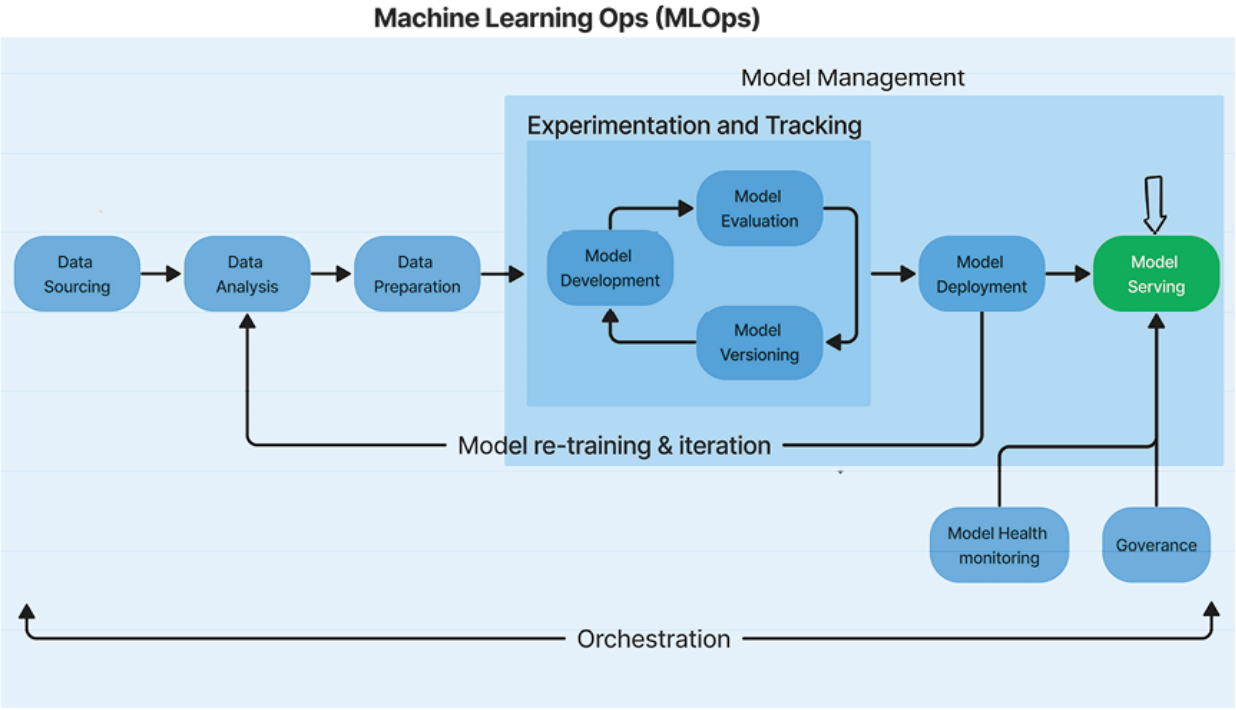

Figure 6.4: *Model serving stage of MLOps-based ML project lifecycle*

In the next section, we will explore various modes of **Model Serving**, such as batch processing and on-demand processing, followed by some of the challenges every machine learning system faces while running in the real world.

Modes of model serving

There are two primary modes of model serving:

- Batch processing
- On-demand processing

Each mode has its own set of advantages and use cases, which makes it suitable for different types of applications and scenarios.

Batch processing

Batch processing-based model serving is best suited for scenarios where we need to process huge amounts of input data through the system at once. This approach is widely used when processing extensive datasets, such as information from all users of a product or service, or when systematically

analyzing incoming events, like tweets or comments on online publications. The advantage of batch processing is that it is generally more resource-efficient than on-demand processing. Resources can spin up only when required compared to on-demand, where resources always need to be up and running as we do not know when we will receive a request. But batch processing is only suitable when a certain degree of latency can be tolerated. If the low latency is critical and the resources need to be readily available for the specific project, then batch processing may not be a suitable approach. In such a case, we should consider on-demand processing.

To determine the optimal batch size for speed, it is essential to experiment with different sizes. The outputs generated from the batch processing are typically saved to a database rather than sent directly to specific consumers. Batch mode can be used for various purposes, like:

- Generating a daily list of users who should be sent discount coupons based on their recent activity on the website.
- Classifying the flow of incoming comments on online news articles and blog posts as spam or not spam.
- Extracting named entities from documents indexed by a search engine, and so on.

In conclusion, serving models in batch mode is a powerful method for handling large-scale data processing tasks. It allows for efficient utilization of resources and can accommodate a certain level of latency when dealing with massive datasets. By experimenting with various batch sizes, optimal performance can be achieved, leading to better results and more precise insights.

On-demand processing: Human as end-user

The process of serving a machine learning model's prediction to a user involves six key steps:

1. Validating the request.
2. Gathering the context.
3. Transforming the context into model input.
4. Applying the model to the input and obtaining output.

5. Ensuring that the output is meaningful.
6. Presenting the output to the user.

Before running a model in a production environment for a user request, it may be necessary to verify whether the user has the appropriate permissions for the model.

Context is the metadata sent by the user, which tells the system about the state the user is in. This information is used by the feature extractor to generate the features used as inputs to the model. This can be built as a separate object or as part of the machine learning pipeline. Once the results are ready by the model, they may not be in a format understandable for the end user. They may need to be converted into a more user-friendly format. Another step that is quite common as part of preparing the output is to measure the prediction confidence score as well to make sure the results are relevant, and the information is clear enough for the end user. This gives the end user an idea of how to interpret the results and how much trust to put in the model's outcomes for that particular instance.

Logging every action by the model can also be useful if we need to debug or audit the system. We should be logging the request we received, the context that was provided, the final transformed context, the output of the system, confidence score, and so on, along with all the metadata like timestamps and so on.

On-demand processing: To machines as end users

In many cases, building a REST API is suitable for serving machine learning models. But there are also use cases where streaming is the better fit for the overall system. One such case is if a model is providing some information and predictions to a machine downstream based on which the next systems/machines in the topology need to take some action. Machines typically have standard and predetermined data requirements for inputs and outputs, which makes it easy to use streaming for those use cases.

One of the challenges of serving on-demand models is the varying demands, for example, demand can be high during the day and low during the night. To address this, one straightforward solution is to have autoscaling in place, which can bring up new instances or remove instances based on the demand. It serves the purpose well, but there are situations where autoscaling may not

be agile enough, for example, handling unexpected spikes in demand. To address such situations, on-demand architectures incorporate a message broker, like **RabbitMQ** or **Apache Kafka**. Message brokers use queues to pass along the data between two systems where one system can write data in the queue, and the other system can subscribe to the queue's output and read the data. This approach not only allows us to cope with demand spikes but also leads to more efficient resource utilization.

In summary, serving machine learning models to machines can be achieved through various methods, including REST APIs, streaming applications, and message brokers. By selecting the right approach for a given scenario, machine learning practitioners can ensure the efficient use of resources and the ability to handle fluctuating demands.

Model serving in real life

In real-world scenarios, when individuals interact with a software system, deploying machine learning models can become complex. Predicting every user action and reaction is generally not feasible. Therefore, a well-designed software system architecture must be prepared to handle three critical phenomena:

- Errors
- Change
- Human nature

Errors

Errors are an inevitable part of any software system. They can arise from various sources, such as incorrect data input, bugs in the code, or unexpected user behavior. To build a robust machine learning model, it is essential to account for these errors and have mechanisms to handle them efficiently. This could involve implementing error detection and correction techniques, monitoring system performance, and having fallback options when the model fails to provide accurate predictions.

Change

The world is constantly changing, and a software system must be adaptable to evolving circumstances. This can include changes in user behavior, updates to the underlying data, or technological advances. If a machine learning project is well-designed, it should be able to handle these changes. This may involve continuously updating the model with new data, using transfer learning to adapt to new tasks, or incorporating modular components that can be easily updated or replaced.

Human nature

Human behavior is complex and often unpredictable, making it challenging to create a software system that can cater to all users' needs effectively. To address this, a machine-learning model must be designed to consider the human element. This can involve understanding user preferences, incorporating feedback loops for continuous improvement, and ensuring the system is user-friendly and accessible.

In conclusion, when designing a machine learning model for real-world applications, it is crucial to account for errors, change, and human nature. By considering these three phenomena, developers can build more robust and adaptable software systems that can better serve their users and stand the test of time.

Conclusion

We started the chapter by continuing to discuss how a model can be deployed. This chapter provided an overview of various aspects of model deployment and serving within the MLOps lifecycle. We explored various deployment strategies, such as static, dynamic, and streaming, each with its own strengths and limitations based on factors like latency, compliance, infrastructure, and cost... We compared how the models can be deployed on end-user devices or on servers using virtual machines, containers, serverless technologies, highlighting their unique benefits in terms of flexibility, resource efficiency, and ease of management.

We also explored serving models to end users, whether humans or other machines/applications. We looked into how a model can serve predictions in batch mode or on-demand, including APIs or streaming. And how these deployments and inference can be achieved using single, silent, canary, and

multi-armed bandit deployments. These allow controlled rollout of new models while monitoring their performance.

In the next chapter, we will continue our exploration of the MLOps project lifecycle components. We will see how the model's health monitoring and governance are an important part of the MLOps.

Points to remember

- Model deployment is the process of making a trained machine-learning model available for use by end-users or downstream systems. The main deployment options are static deployment, dynamic deployment on edge devices, dynamic deployment on servers, and streaming deployment.
- Static deployment involves packaging the model in the installable binary of a software application. Dynamic deployment allows the model to be updated independently of the application code.
- For server deployment, virtual machines provide isolation but can be resource-intensive. Containers enable efficient scaling and resource sharing. Serverless simplifies deployment but has constraints like file size limits.
- Deployment strategies include single deployment, silent deployment, canary deployment, and multi-armed bandits. Each has pros and cons in terms of simplicity, resource requirements, and ability to detect bugs.
- Model serving refers to making predictions available via batch processing, APIs, or streaming. Batch processing is efficient for large volumes but has latency. APIs enable real-time low-latency predictions.
- When serving models, it is important to handle errors, adapt to changes, and account for unpredictability in human behavior. Continuous monitoring and updating helps maintain robust model performance.

Key terms

- **Static deployment**: Static deployment is the process of statically deploying a machine learning model as part of the installable binary of the entire software, with the model being packaged as a resource that can be accessed during runtime.
- **Dynamic deployment**: In dynamic deployment, the model is trained online or offline. Data continuously enters the system and is incorporated into the model through continuous updates. In this deployment, predictions are made on-demand using a server. The model is deployed using a web framework like **FastAPI** or **Flask** and is offered as an API endpoint that responds to user requests, whenever they occur.
- **Batch processing**: Batch processing for model serving is the process where a large chunk of data is processed as a batch at once, and the predictions are stored in a data store. These predictions are then served whenever required. Batch processing is more resource-efficient as compared to on-demand processing.
- **On-demand processing**: On-demand processing is when a model is serving requests in real-time on demand. The model is deployed on a server with an API endpoint that can receive processing requests, validate the request, process, generate the prediction, and respond to the user requests with that prediction. It is much more resource-intensive than batch processing as it needs to always have the servers up and running to respond to user requests in real time.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 7
Continuous Delivery of Machine Learning Models

Introduction

In this chapter, we will explore methods for effectively implementing and automating **continuous integration** (**CI**), **continuous training** (**CT**), and **continuous delivery** (**CD**) in machine learning systems. These practices and components are essential in continuously delivering machine learning models to production environments.

As we know practical MLOps means aiming to achieve automation and monitoring throughout all stages of the machine learning pipeline including building, integration, testing, deploying, serving and infrastructure management phases. The process of training and preparing ML models is a core but relatively small part of the overall machine learning system especially once we have the algorithm and the training process selected and streamlined. In this regard, available tools and processes have already reached a high level of maturity. Nevertheless, the real challenge does not lie solely in creating machine learning models. The primary concern is to build a fully integrated machine-learning system that can be continuously operated and updated in a production setting. This aspect is closely related to the principles of DevOps and necessitates a proper understanding of how these principles can be effectively intertwined with machine learning systems.

By thoroughly examining the techniques for implementing CI, CT, and CD in the context of machine learning, we will lay the groundwork for an efficient machine-learning pipeline. This will empower ML engineers and data scientists to manage ML infrastructure and deploy models with ease, resulting in improved system performance and timely updates.

Structure

In this chapter, we will discuss the following topics:

- Traditional continuous integration/continuous deployment pipelines
- Pipelines for machine learning/artificial intelligence
  - Architecture level 1
  - Architecture level 2
  - Architecture level 3
- Continuous integration
  - GitHub Actions
- Continuous training
  - Continuous training strategy framework
  - When to retrain
  - What data should be used
  - What should we retrain
- Continuous delivery

Objectives

By the end of this chapter, we will understand what it takes to enable continuous delivery of machine learning models into the production environment. We will understand the ML/AI pipelines and different maturity and architectural levels of the pipeline and how continuous integration and training and the prerequisites of continuous delivery to the production environment.

Traditional continuous integration/continuous deployment pipelines

CI/CD are a set of best practices implemented in the application development pipelines. DevOps teams primarily adopt these practices to facilitate software developers in swiftly and dependably introducing updates to production applications.

Note: In CI/CD pipelines, continuous deployment and continuous delivery are often used interchangeably, but there are differences. Continuous delivery is a software practice when the code changes are ready to release. Continuous deployment aims at continuously and automatically releasing the code changes into the production environment as long as they pass all the check.

Some of the fundamental advantages of an efficient CI/CD pipeline encompass:

- Reliability
- Reproducibility
- Agility
- Secure deployment
- Version control

As shown in *Figure 7.1*, at their core, these tools attempt to formalize a well-known workflow: **Build** | **Test** | **Deploy**:

Figure 7.1: *CI/CD in traditional application*

Although CI/CD are frequently mentioned and used together, they signify two distinct (yet connected) concepts. CI mainly focuses on testing the code as it is pushed to the code repository, ensuring that new application features are automatically tested using unit tests. On the other hand, CD pertains to the actual delivery or release of this tested code. For instance, a CD system might outline how various features or release branches are deployed or even how new features are selectively rolled out to new users (such as testing feature A on 20% of the customer base to make sure it is working well and is

facing no issues in adoption before it is released to the whole customer base).

These crucial concepts have only recently gained significant traction in MLOps and modern machine learning pipelines. Many organizations are adopting CI/CD principles for ML model deployment to improve reproducibility, reliability, and agility.

Pipelines for machine learning/artificial intelligence

The optimal system for machine learning presents several key distinctions compared to traditional pipelines. As illustrated in *Figure 7.2*, there are striking similarities between machine learning and traditional software systems. These resemblances include the continuous integration of code, testing (including unit testing, integration, or component testing), and continuous delivery:

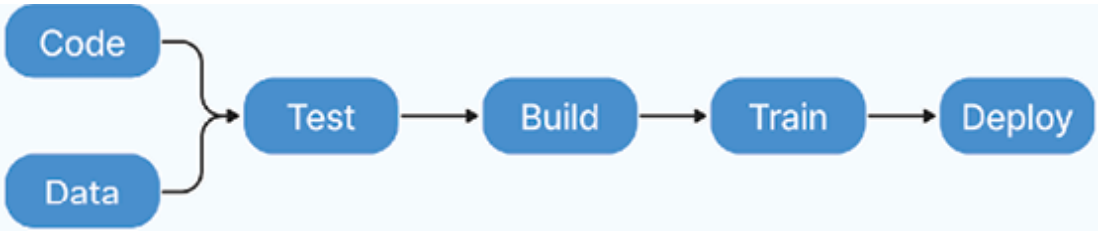

Figure 7.2: *CI/CT/CD for machine learning and AI applications*

Nevertheless, in machine learning, there are a few noteworthy differences as well:

- CI goes beyond simply testing and validating code and components, extending to the examination and validation of data, data schemas, and models.
- CD has evolved beyond the scope of the singular software package or service. Instead, it now encompasses the entirety of a system that is responsible for the automated deployment of all machine learning services, including the model prediction service.
- CT is a completely new component of the CI/CD/CT pipeline, which is unique to data science and machine learning systems that focus on automatically retraining the models.

The level of continuous delivery and automation in the deployment of the models also depends upon the architecture level and maturity of the infrastructure of the organization. To better understand the relationship, let us revisit the architecture levels we discussed in *Chapter 2: MLOps Architecture and Components* and examine how CI/CT/CD capabilities align with the infrastructure's level of maturity.

Architecture level 1

In this preliminary stage of maturity, teams are focused on building state-of-the-art models. However, developing and deploying **machine learning** (**ML**) models remains manual or reliant on scripts. This implies that every step, such as data analysis, data preparation, model training, and validation, required manual intervention for execution and transition between steps.

At this level, organizations struggle with implementing CI, CD, and CT, primarily due to inadequate infrastructure. At this stage, fewer implementation changes are assumed; therefore, there is no need for a CI infrastructure setup. Similarly, infrequent model deployments and the lack of continuous training can contribute to insufficient **Return On Investment** (**ROI**) to spend resources on CT and CD.

Adopting MLOps practices for CI/CD and CT can be instrumental in addressing the challenges associated with this manual process. By deploying an ML training pipeline in an orchestration system like **Apache Airflow**, we can facilitate CT. In addition, for CI/CD systems that need rapid testing, building, and deployment of new implementations of the ML pipeline, organizations can opt for either Airflow or another CI/CD platform suited to their needs.

By refining and optimizing the architecture at the initial stage, organizations can set a strong foundation for the successful integration and management of ML models as they progress to more advanced stages of maturity. This, in turn, will lead to more efficient processes, high accuracy, and improved overall performance.

Architecture level 2

At maturity level 2, the primary goal revolves around enabling continuous training of the models by automating the ML pipelines. This level facilitates

the continuous delivery of model prediction services by implementing automated training processes using new data.

To achieve this, it is necessary to incorporate steps that will validate the model and data automatically within the pipeline, in addition to pipeline triggers and metadata management. The orchestration of ML experiment steps takes place at this level, allowing for seamless transition and automation when transitioning between the processes. This rapid iteration of experiments contributes to enhanced preparedness for transitioning the entire pipeline into a production environment.

This level ensures the use of up-to-date data while training the model in production by enabling automated training of models with live pipeline triggers. However, to construct machine learning pipelines, components must be reusable. Although the **Exploratory Data Analysis** (**EDA**) code can still live within notebooks, the source code for components needs to be modularized and, ideally, containerized. This allows for a more efficient and effective production ML pipeline to deliver continuous prediction services using freshly trained models.

Compared to level 1, where trained models are deployed as prediction services in production, level 2 advances this approach by deploying an entire training pipeline. This enables automated and recurrent execution of the training process, consistently providing updated and validated models for use within the prediction service.

Architecture level 3

When it comes to continuous delivery of machine learning models, to ensure rapid and reliable updates of production pipelines, a robust automated CI/CT/CD system is crucial. This level of automation enables the rapid experimentation of new possibilities for improvement in the machine learning system. If the outcomes of these experiments are useful, the implementation can be automatically created, tested, and deployed to different environments.

The pipelines consist of the following components:

- **Pipeline continuous integration**: This involves building source code and running various tests. The outcome of the continuous integration

stage is not just the running of the code tests but also the artifacts which can be used to train and deployed in the stages which follow.

- **Pipeline continuous training**: At this stage, the data is validated, the model is trained, evaluated, and the model artifact is stored in different environments. The outcome of this stage is the artifact which can be used by the delivery pipeline to deploy in the environment.
- **Automated triggering**: Even though not an independent component, automated triggering is a crucial part of all these automated processes. We need to have a component in place that can automatically trigger the execution of the pipelines either based on a specific schedule or certain events.
- **Model continuous delivery**: At this stage, the machine learning project is updated with the latest model and prepared for inference. The outcome of this stage is the inference and serving pipeline with the latest version of the model.

As seen in *Figure 7.3*, by implementing this robust automated CI/CT/CD system, teams can efficiently apply new ideas and adapt to changes while ensuring their pipelines remain accurate and up-to-date. Please refer to the following image:

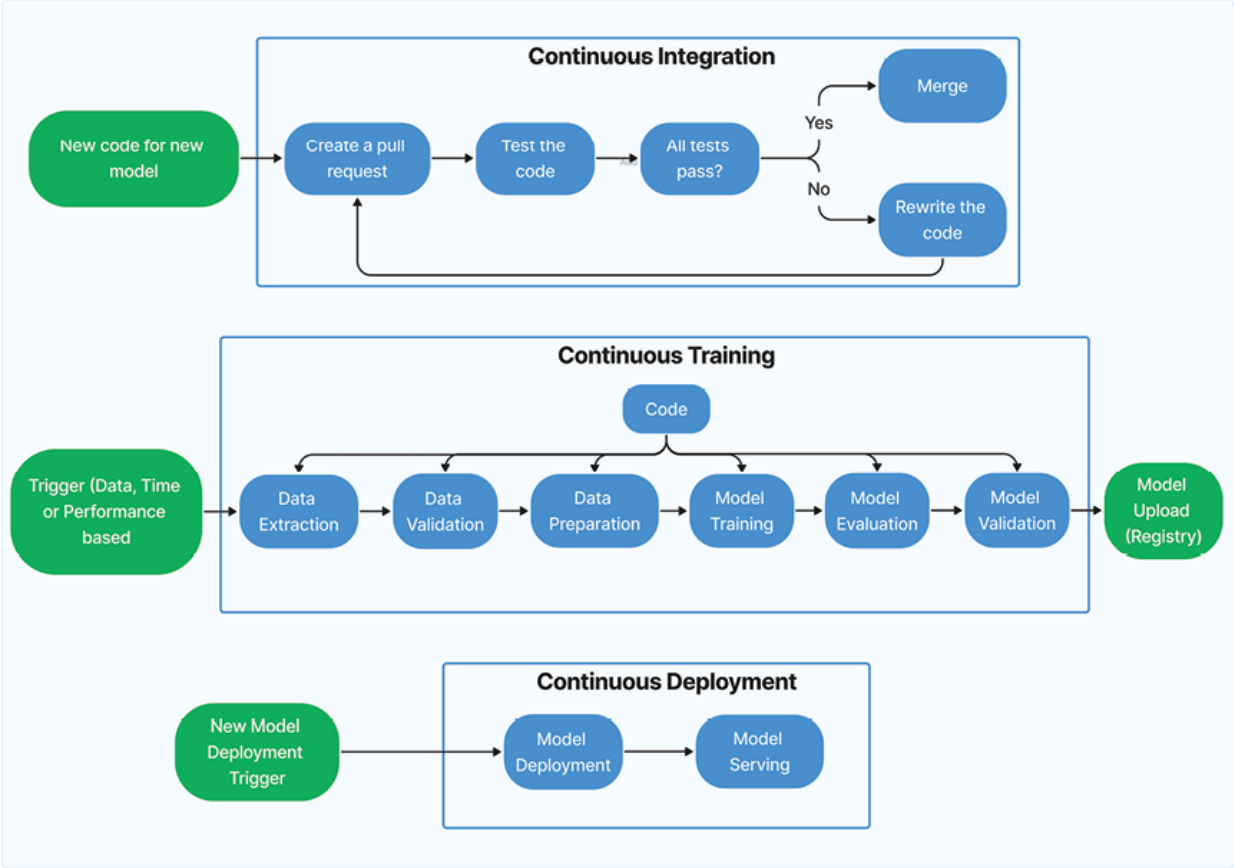

Figure 7.3: *Automated continuous integration, continuous training, and continuous delivery system*

This approach offers a flexible and adaptable environment for testing and deploying machine learning models, accelerating the overall development process.

Continuous integration

Continuous integration (**CI**) is the stage where the pipeline, including all the components of the system, is tested, built if required, and packaged to be used in the later stages whenever there is a new code committed and pushed to the code repository. Besides building packages, container images, and executables, the CI process can also encompass the following tests based on the use case and requirements of the project:

- Unit testing on the logic used for feature engineering.
- Unit tests on various methods implemented in the model. For instance, the model may contain a function that receives a categorical data

column and encodes it as a one-hot feature.

- Testing the model training coverage ensures that the loss decreases with successive iterations and that the model overfits only a few sample records. It also ensures that we have tested for proper data coverage as well as feature coverage.
- Testing that each unit of the code in the pipeline works as expected and creates the correct artifacts.
- Testing integration between pipeline components.

By enhancing the CI process, the development cycle remains efficient, facilitating quicker identification and resolution of issues. Consequently, this allows for the creation of continuously improving and dependable machine learning models.

There are many platforms and tools that can be used for handling CI components similar to the traditional software CI/CD; some of them are:

- CircleCI,
- Travis CI,
- GitHub Actions,
- Jenkins, and so on

Let us take GitHub Actions for our example and see how we can setup continuous integration component for unit, integration and functional testing of every new code commit. The workflow pipeline will contain the same steps as indicated in the continuous integration workflow component in *Figure 7.3* previously.

GitHub Actions

GitHub Actions allow us to automate the workflow for when a code is pushed to the GitHub repositories, making it faster to build, test and deploy the code.

Most of the GitHub Actions workflow pipelines look as follows:

```
name: Pipeline Name
```

```
on: push                    # trigger for the pipeline
i.e. push or pull etc

jobs:

    job_1:                  #ID of the job

    name: Job no. 1

    runs-on: ubuntu-latest

    steps:

            . . . . .


    job_2:


    name: Job no. 2

    runs-on: ubuntu-latest

    steps:

            . . . . .
```

There are some important components of this workflow pipeline, discussed as follows:

- When an event like a **`push`** or **`pull`** request occurs in the GitHub code repository, a workflow pipeline can be triggered.
- Jobs in the pipeline are not dependent on each other for the execution environments. They run in independent containers.
- These jobs consist of steps that are executed sequentially inside these independent containers. The execution of these steps depends on each other, though, with one step depending upon the success or failure of the previous step.

- Contrary to jobs, steps are dependent on execution, and as they are sequential, one step depends on the success or failure of the previous step.

To make sure that the new code that is being written and pushed to the repository will not cause any issues in the ML system, we generally use a pipeline that runs all the relevant tests as part of the continuous integration pipeline. Only if the code passes all the tests it is allowed to be merged into the environment branch (master, staging, or development branch).

The workflow will need to be inside a YAML file under **`.github/workflow/test_code.yaml`**.

A full CI code for testing the code can look as follows:

```
name: Testing CI pipeline

on:

    pull_request:

paths:

            - config/**

            - training/**

            - application/**

            - .github/workflows/test_code.yaml

jobs:

    test_model:

        name: Test new model

        runs_on: ubuntu-latest

        steps:

            -name: Checkout

             id: checkout
```

```
            uses: actions/checkout@v2

          -name: Environment setup
           uses: actions/setup-python@v2
           with:
               python-version: 3.8
                cache: pip

          -name: Cache
           uses: actions/cache@v2
           with:
               path: ~/.cache/pip
               key: ${{ runner.os }}-pip-${{
hashFiles('**/dev-requirements.txt')}}
               restore-keys: ${{ runner.os }}-
pip-

    -name: Install packages
            run: pip install -r dev-
requirements.txt

          -name: Pull data
           run: . . . .              # Code to
pull the required data.
```

```
    -name: Run training tests
     run: pytest training/tests


    -name: Save model
     run: python application/src/save_model.py


    -name: Serve the app locally and run app tests
              run: |
                 bentoml serve
./application/src/create_service.py:service & sleep
10
                 pytest application/tests
                 kill -9 `lsof -i:3000 -t`
```

Continuous training

As we know, machine learning models tend to become outdated over time, leading to a decline in performance. This occurs because the data changes in real-time due to evolving trends or changes in the behavior of the users and so on, a phenomenon referred to as **data drift**. Moreover, given sufficient time, even the relationships between inputs and outputs will likely experience alterations, known as **concept drift**. To ensure the relevance of our machine learning models, it is crucial to keep them up-to-date with the data, which is done with continuous training. We can continuously train a model in multiple ways:

- **Incremental training**: This method involves training a model using newly available data (over the existing model) as soon as it arrives.

- **Batch training**: In this approach, once a significantly large amount of data is accumulated for training, the model training pipeline is triggered to train over the existing model.
- **Retraining**: This approach entails retraining the entire new model by combining old and new data collected.

These approaches have advantages and disadvantages, making them appropriate for distinct scenarios. Nonetheless, all these approaches necessitate an automated workload management process, as manual execution can be labor-intensive and inefficient. By adopting such methodologies, machine learning models can be consistently updated and optimized for better performance in an ever-changing data landscape.

The continuous training process encompasses six distinct stages:

- Data extraction
- Data validation
- Data preparation
- Model training
- Model evaluation
- Model validation

These stages make the workflow pipeline of continuous training as displayed in the continuous training workflow in *Figure 7.3*.

Following is the sample code structure of the training pipeline using Apache Airflow, which is set to trigger daily and train the model:

```
from datetime import datetime

from airflow.decorators import dag, task

from airflow.models import Variable


@dag(

    dag_id="training_pipeline",
```

```
    schedule="@daily",
    start_date=datetime(2023, 6, 15),
    catchup=False,
    tags=["model-training", "model-verification"],
    max_active_runs=1,
)
def training_pipeline():
    @task.virtualenv(
        task_id="data_extraction",
        python_version="3.9",
 multiple_outputs=True,
        system_site_packages=True,
    )
    def data_extraction():
        """
        Run the data extration task.
        """
        # code for data extraction from the source
system


    @task.virtualenv(
        task_id="data_validation",
        python_version="3.9",
```

```
        multiple_outputs=True,
        system_site_packages=False,
    )
    def data_validation():
        """
        This function validated the data fetched in
the previous task.
        """
        # code for data validation


    @task.virtualenv(
        task_id="data_preparation",
        python_version="3.9",
        multiple_outputs=True,
        system_site_packages=False,
    )
    def data_preparation():
        """
 This function runs the data preparation steps
before that data can
        be used for training.
        """
        # code for data preparation
```

```
    @task.virtualenv(
        task_id="model_training",
        python_version="3.9",
        multiple_outputs=False,
        system_site_packages=False,
    )
    def model_training():
        """
        Model training task. This is the main function which takes
        the data and trains the model
        """
        # code for model training


    @task.virtualenv(
        task_id="model_evaluation",
        python_version="3.9",
        multiple_outputs=True,
        system_site_packages=False,
    )
    def model_evaluation():
        """
        Evulated the model trained in the previous step.
```

```
        """
        # code for model evaluation

 @task.virtualenv(
        task_id="model_validation",
        python_version="3.9",
        system_site_packages=False,
    )
    def model_validation():
        """
        Model validation task before it can be
uploaded and registered.
        """
        # code for model validation goes here.


    @task.virtualenv(
        task_id="model_upload",
        python_version="3.9",
        system_site_packages=False,
    )
    def model_upload():
        """
        This is the function that uploads and
registers the model
```

```
        which is ready for production deployment
        """
        # code for model upload goes here.


    # Define Airflow variables.
    days_delay =
int(Variable.get("training_pipeline_days_delay",
default_var=15))
    days_export =
int(Variable.get("training_pipeline_days_export",
default_var=30))


 # Training pipeline
    data_extraction = data_extraction()
    data_validation = data_validation()
    data_preparation = data_preparation()
    model_training = model_training()
    model_evaluation = model_evaluation()
    model_validation = model_validation()
    model_upload = model_upload()


    # Define DAG structure.
    (
        data_extraction
```

```
        >> data_validation
        >> data_preparation
        >> model_training
        >> model_evaluation
        >> model_validation
        >> model_upload
    )

training_pipeline()
```

Continuous training strategy framework

Each machine learning use case presents unique data environments that may result in concept and data drift. But regardless, to develop an effective continuous training strategy, it is crucial to address three primary questions:

- **When should the model be retrained?** The primary goal is to maintain a highly relevant and optimally performing model at any given time. Consequently, it is essential to determine the ideal frequency for model retraining.
- **What data should be used?** Generally, teams assume that the recent data is the most relevant and can be used for the training. But is that true, or does it depend on a case-to-case basis? It raises several questions:
    - Should new data replace older data, or should they be combined?
    - If the data should be combined, what should be the case of new and old data in the training dataset?
    - How recent must data qualify as new, and when does it transition from new to old?

- **What should we retrain?** Is it sufficient to create a new training dataset and retrain the existing model, keeping everything else the same? Or should we adopt an intermediate approach such as transfer learning or fine-tuning? Alternatively, does it make more sense to take a more comprehensive approach and start by reanalyzing the process in light of new data, updating the complete pipeline, and rebuilding the model from scratch?

These major questions should be answered first to decide upon the best approach to retraining for each use case. While exploring these questions, numerous factors need to be considered. By evaluating these considerations and selecting appropriate methods to address these questions, a successful continuous training strategy can be developed, ensuring the ongoing relevance and high performance of machine learning models.

When to retrain

There are five primary strategies to consider for retraining a machine learning model, which includes:

- Adhoc/manual retraining
- Periodic time-based retraining
- Periodic data volume-driven retraining
- Performance-based retraining
- Data changes-based retraining

Adhoc/manual retraining

Adhoc or manual retraining is the most straightforward approach, which does not require much maturity on the architectural level. This process encompasses developer-initiated retraining where retraining is started manually on an adhoc basis whenever required.

Periodic time-based retraining

Implementing a periodic retraining schedule is the most simplistic and direct method for updating machine learning models. Typically, this schedule is based on time, such as retraining every week, month, or quarter.

The advantages of regular retraining cycles are equally clear-cut. This method is easily manageable, highly intuitive, and easy to implement because it allows easy determination of the timing for the next retraining iteration.

However, this method often turns out to be poorly suited or inefficient for certain scenarios. The main challenge lies in determining the ideal retraining frequency. We can decide to retrain the model daily, weekly, monthly or annually. While it is desirable to update the model as frequently as possible to maintain its effectiveness, retraining too often without a legitimate reason, like the absence of concept drift, can be expensive and unnecessary. Retraining requires substantial resources, both in terms of computational power and the involvement of data science teams. Conversely, retraining at lengthy intervals might undermine the purpose of continuous model adaption and fail to address data fluctuations, not to mention the risks associated with retraining on noisy data.

Ultimately, a periodic retraining schedule can be effective if the chosen frequency corresponds to the dynamics of the domain. Failing to do so may expose the model to potential risks and result in updated versions that are less useful than their predecessors.

Periodic data volume-driven retraining

Like periodic time-based retraining, the periodic data volume-driven retraining approach is a straightforward way to update the machine learning models. This method is typically based on the volume of data, for instance, retraining after every 100K new data points, and so on.

This approach is both easy to manage and implement, although it presents similar challenges in determining the optimal volume of new data that should trigger the retraining of the model. Retraining can happen irregularly depending upon the volume of data ingestion or spikes in data from time to time.

As with time-based retraining, significant resources are required for training, so finding a balance in appropriate new data volume is essential. Retraining too early can lead to resource wastage, as using a small amount of new data for retraining may provide little to no benefit and could even negatively impact the model's performance. On the other hand, waiting for a substantial

volume of data before retraining may undermine the process, as critical data fluctuations might not be accounted for.

To maximize the benefit of data-driven retraining, it is necessary to carefully consider the amount of new data that warrants model updating. By balancing resource expenditure and model performance, data-driven retraining can help develop increasingly effective and adaptable machine learning models.

Performance-driven retraining

Among the various methods for determining when to retrain a machine learning model, performance-based retraining triggers are often employed to refresh the model once an observable decline in the performance of the existing model is detected.

This approach, which is more empirical than its counterparts, presupposes that a continuous evaluation of the model's performance in production is available. One primary challenge in exclusively relying on performance is the time taken to acquire ground truth, assuming it is even obtainable. In instances like user conversion predictions, it can take anywhere from 30 to 60 days to receive ground truth, while in situations involving transaction fraud detection or **Lifetime Value** (**LTV**), the wait could extend to six months or longer. This prolonged delay in acquiring comprehensive feedback could result in retraining the model too late after the business has already experienced consequences. In such scenarios, either we use some of the other retraining options or consider a few approaches that can be taken to mitigate these risks like using proxy metrics or surrogate measures that can provide quicker feedback which can be used as a proxy for ground truth to expedite training decisions.

Another crucial question that warrants consideration is - how do we define **performance degradation**? In this scenario, we may rely on the sensitivity of the predetermined thresholds and the precision of the calibration, potentially causing the model to be retrained too frequently or insufficiently.

In summary, employing performance-driven retraining triggers is most effective in use cases where there is rapid feedback and a substantial volume of data, such as in real-time bidding. This allows for the performance to be measured as closely as possible in almost near-real time and ensures a high degree of confidence supported by the substantial volume of data.

Data changes-based retraining

This method is the most dynamic and inherently triggers retraining due to the ever-changing nature of the domain. By monitoring variations in the input data, specifically alterations in the feature distribution utilized by the model, we can identify data drifts that signal the model might be outdated and require retraining with new data.

This approach is an alternative to performance-based methods in cases where we lack rapid feedback or cannot evaluate the model's performance in production in real-time. Furthermore, it is also beneficial to combine and employ this method even when fast feedback is available, as it could signify suboptimal performance without evident degradation. Recognizing data shifts to initiate a retraining process is extremely valuable, particularly in dynamic settings.

So, when is it appropriate to retrain? This decision relies on essential factors such as the availability of feedback, data volume, and the model's performance visibility. In general, there is no universal solution for determining the optimal time to retrain a model. Instead, depending on available resources, the objective should be to align as closely as possible with production-driven goals.

What data should be used

As we have learned, timing plays a crucial role in MLOps (for instance, determining when to retrain models). Now, let us delve into the crux of MLOps pipelines: the data selection. When retraining a model, how can we choose the most suitable data to incorporate? Choosing the right data to use during the retraining process ensures that the model delivers reliable results. The following considerations may help determine which data should be used for retraining:

- **Relevance to the problem**: Ensure that the selected data is directly related to the problem your model aims to solve. The data should be representative of real-world scenarios, allowing the model to work effectively in practical applications.
- **Data quality**: The age, source and reliability of the data are the most important factors. Using outdated or low-quality data might hinder the

learning process of the model, potentially leading to poor performance. High-quality, recent data is more likely to yield better results.

- **Diversity of data**: Including a diverse range of data can improve a model's ability to generalize and perform well on various tasks. Gathering data from different sources, periods, and demographics can reduce potential biases in the model.
- **Data pre-processing**: Before using the data to retrain models, it should be cleaned, pre-processed, and transformed into a suitable format for the model to digest. This process ensures the model can effectively learn from the input data patterns during retraining.
- **Data balance**: Ensuring that the data used for retraining is balanced helps prevent overfitting or underfitting and supports a more effective learning process. Maintaining an appropriate balance of data classes can lead to a well-rounded model that performs consistently across different tasks.
- **Feedback incorporation**: Use the feedback from previous deployments of the model to identify any issues or areas for improvement. This information can guide the data selection process, helping to address specific limitations or shortcomings in the model's performance.
- **Data ethics and privacy:** Ensure that the selected data for retraining respects all the privacy regulations and ethical guidelines of the domain. For example, if we are working on any kind of medical data, we need to follow **Health Insurance Portability and Accountability Act (HIPAA)** compliance as well as avoid using any kind of **Personal Identifiable Information** (**PII**).

By carefully considering the factors mentioned above, there are multiple approaches to data selection we could use for retraining, as discussed as follows.

Fixed window size

One of the most elementary methods involves utilizing a fixed window size as the training set. For instance, incorporating data from the previous X months. The main advantage of this approach lies in its simplicity and ease of implementation.

However, this approach presents several drawbacks stemming from the challenge of determining the ideal window size. If the window size is too broad, it may overlook emerging trends, while a window that is too narrow will result in overfitting and increased noise. The selection of the window size should also factor in the frequency of retraining: it is inefficient to retrain the model every three months if only the past month of data is utilized in the training set. Moreover, the data can be significantly different from the actual data in the production environment. This discrepancy can occur immediately after a change point (in case of rapid or abrupt fluctuations) or during irregular events (for example, holidays or special days such as elections), as well as data issues and other anomalies.

In conclusion, the fixed window size approach, like other `static` methods, offers a straightforward heuristic that provides satisfactory results in certain scenarios. However, it may not adequately capture the highly dynamic nature of environments where the change rate is versatile and irregular events, such as holidays or technical disruptions, are frequent.

Dynamic window size

The dynamic window size approach aims to address some of the shortcomings of the fixed window size. It is done by determining the extent of historical data to be included in a more data-driven manner and considering the window size as an additional hyperparameter that can be optimized during the retraining process.

This method is particularly suitable for cases where quick feedback is required (for example, real-time bidding or food delivery). The test set can leverage the most pertinent data, allowing for optimization of the window size similar to other model hyperparameters.

The primary advantage of the dynamic window size approach is its data-driven nature, which relies on model performance, making it more resilient in highly time-sensitive environments. However, this technique demands a more intricate training pipeline to assess various window sizes and choose

the optimal one, resulting in increased computational intensity. Additionally, similar to the fixed window size approach, there is an assumption that the latest data is the most significant, which may not always hold.

Dynamic data selection

The third method of selecting data for training models aims to fulfill the primary goal of any retraining strategy, which is the foundational assumption of all machine learning models: utilizing training data that closely resembles production data. This approach, while complex, demands a high level of visibility into the production data.

To achieve this, it is necessary to conduct a comprehensive analysis of production data's evolution to comprehend if and how it alters over time. One way to accomplish this is by calculating the input data drift between pairs of days, that is, accessing the degree of difference between data from one day to another. Following this, a heatmap can be generated, revealing the fluctuations in data patterns over time.

In short, selecting appropriate data for retraining models necessitates an in-depth understanding of the production data behavior. Although employing fixed or dynamic window sizes may aid in addressing the issue, it often remains a rough estimation that could potentially end up becoming more costly than efficient. The dynamic data selection approach focuses on data similarity and relevance, which proves to be critical for maintaining optimal model performance in a continually changing environment. As these data selection strategies have their pros and cons when used individually, there are scenarios where machine learning practitioners might also use a hybrid approach that combines fixed and dynamic approaches for example, we might use fix windows approach for stability and dynamic window to capture recent changes. All these strategies and approaches are flexible enough for use to adapt for our unique use cases.

What should we retrain

After addressing the timing and data selection for retraining, let us now explore the next question, which components of the model should undergo retraining? Options include solely retraining the model instance using new data, retraining some or all of the data preparation steps, or adopting a more

comprehensive approach by running the entire pipeline to simulate the research process.

The underlying principle of retraining is based on the recognition that the model, which was originally trained during the research phase, will become outdated over time due to concept drift or data drift, thereby requiring the need for retraining. In most scenarios, however, the model instance represents the final stage of an extensive pipeline crafted during the research phase, including data preparation steps. The crucial question is: what is the magnitude of the drift and its consequences? Or, in relation to retraining, which elements of the model pipeline should be examined and considered for retraining?

During the research phase, various elements within the model pipeline are examined and evaluated to optimize the model. Broadly, these elements can be categorized into two main components: data preparation and model training. Data preparation encompasses the process of readying the information for consumption by the model and covers methods such as feature engineering, scaling, imputation, dimensionality reduction, and more. On the contrary, model training involves choosing the most suitable model and its corresponding hyperparameters. The result is a pipeline of sequential steps that process data, prepare it for the model, and generate predictions for the target outcome using the said model.

Retraining only the model, which constitutes the final step in the pipeline, using new relevant data is the most straightforward approach. This methodology may be sufficient in preventing performance deterioration. However, there are cases where this model-centric approach proves inadequate, calling for a more extensive retraining strategy. Expanding the scope of retraining can be executed along two dimensions:

- **What to retrain?** Which elements of the pipeline should undergo retraining using the fresh data?
- **Retraining extent**: Are we solely refitting the model or any other steps using the new data? Are we optimizing hyperparameters? Or are we scrutinizing the model selection itself, testing other model types?

Another way to look at this is that the retraining essentially aims to automate and eliminate the labour-intensive model optimization research typically performed by a data scientist or machine learning engineer in the absence of

an automatic retraining process. Consequently, the level of automation desired should be determined to effectively mimic the manual research process.

Adopting automation for the initial stages of retraining is more complex. With time and progress, the process becomes more resilient and adaptable as we build more automation in the pipeline. However, this also introduces additional complexity as more edge cases and considerations must be accommodated.

Continuous delivery

In this phase, the system consistently deploys updated pipeline implementations to the target environment, providing prediction services utilizing the most recent trained models. To ensure rapid and dependable continuous delivery of pipelines and models, we should consider the following aspects:

- Confirming the model's compatibility with the target infrastructure before deployment. This involves aspects like making sure all the required packages are available, and the environment configuration and resources are proper as the model requires.
- Testing the service by invoking the service API with predefined inputs and verifying that the response aligns with the expected response. Such tests typically capture issues that may arise when updating the model version, and it expects a different input parameter.
- Assessing the performance of the prediction service entails conducting load tests to measure metrics like queries per second and model latency.
- Validating the data utilized for either retraining or batch prediction.
- Ensuring that models meet the specified predictive performance targets before deployment.
- Automatic deployment to a testing environment, for instance, through code pushes to the development branch that triggers the deployment process.

- Semi-automated deployment to a staging environment, such as a deployment that is initiated by merging code into the staging (or main) branch after the changes have been approved by reviewers.
- Manual deployment to a production environment after all the new changes were executed successfully in the staging or pre-production environments.
- In case, there are issues or bugs in the code that are identified in staging or pre-production environments, that will cause those deployments to fail and be marked as unsuccessful thus stopping production deployment which then needs to be worked upon and the whole process needs to go through another iteration to make sure everything is successful before production deployment can be performed.

By addressing these considerations, we can establish an efficient system for continuously delivering and updating pipelines and models, enabling the provision of accurate and reliable predictive services.

Conclusion

We started the chapter by continuing the discussion on the three core components of the automation of any machine learning system: **CI, CD and CT**. These three components are crucial for any machine learning pipeline.

Continuous integration is the stage responsible for ensuring the code and the changes are ready to be integrated with the system. It performs all the required tests and stores the outcome of the test runs to ensure we can integrate only the code that passes all the tests and checks.

Continuous training is the component specifically for machine learning projects as it deals with preparing the data, training the new versions of the model, and evaluating and verifying the model artifacts before they can be handed over to the next stage in the pipeline for deployment. As part of the continuous training component, we also discussed the continuous training strategy framework, where we addressed the three main questions, when to retrain, what data to use for retraining, and what to retrain. We discussed the strategies which will come in handy while making these decisions.

Continuous delivery is the component responsible for delivering the latest model to the target environment and making sure the latest approved model is used for inference and serving the predictions. It also prepares/upgrades the environment for supporting libraries or resources needed to serve the model.

Points to remember

- **Continuous integration** (**CI**), **continuous training** (**CT**), and **continuous delivery** (**CD**) are crucial for automating machine learning pipelines.
- CI tests and validates code changes before integration. CT handles data preparation, model training, evaluation, and verification for updated models. CD deploys the latest validated model into production.
- There are 3 levels of ML pipeline architecture maturity: manual process (level 1), automated training (level 2), full CI/CT/CD automation (level 3).
- Considerations for when to retrain include: performance degradation, data drift, periodic time or data volume triggers.
- Data selection considerations include: relevance, quality, diversity, preprocessing, balance, and incorporating feedback.
- Continuous delivery involves testing compatibility, performance, validating data/models before deployment.
- Effective CI/CT/CD enables rapid iteration and adaptation of ML systems while ensuring models stay accurate and up-to-date.

Key terms

- **Exploratory Data Analysis** (**EDA**): EDA is the critical process of performing initial investigations on data so as to discover patterns, to spot anomalies, to test hypothesis and to check assumptions with the help of summary statistics and graphical representations.

- **GitHub Actions**: GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows us to automate the built, test and deploy pipelines.
- **Data drift**: Data drift is unexpected and undocumented changes to data structure, semantics, or infrastructure which results in the distribution of the input data change for the machine learning models thus negatively impacting the predictions of the model over time.
- **Concept drift**: Concept drift occurs when the task that the model was designed to perform changes over time. It can be understood as changes in the relationship between the input and target output.
- **Incremental training**: This method involves training a model using newly available data (over the existing model) as soon as it arrives.
- **Batch training**: In this approach, once a significantly large amount of data is accumulated which can be used for training, the model training pipeline is triggered to train over the existing model.
- **Retraining**: This approach entails retraining the entire new model by combining old and new data collected.

CHAPTER 8
Continual Learning

Introduction

In this chapter, we will dive into the world of **continual learning** in **machine learning** (**ML**) solutions and its crucial importance in MLOps. The essence of machine intelligence is predicated upon its capacity for adaptation. In other words, the system's effectiveness is directly proportional to its ability to adapt. For an ML system to achieve its maximum potential, enabling continual learning is indispensable. Continual learning is especially valuable in scenarios where adaptation to an ever-changing external environment is vital. By constantly learning and updating their knowledge, ML systems can achieve superior performance and deliver maximum value to users.

We will explore the significance of continual learning in machine learning solutions and systems. Reflect on the fundamental reasons behind the need for continual learning in machine learning. We will examine how a system's ability to adapt to changing circumstances and new data is essential for long-term success. By highlighting the benefits of continual learning, we will delve into the ways in which it can revolutionize the capabilities of machine learning systems.

We will also explore the critical aspect of model retraining and the maintenance of **continuous integration and continuous deployment** (**CI/CD**) pipelines. As we embark on this journey, we will examine the intricate details of continual learning and its applications in MLOps.

Through a comprehensive exploration of these topics, we aim to equip readers with the knowledge and tools necessary to harness the power of continual learning and drive innovation in machine learning.

Structure

In this chapter, we will discuss the following topics:

- Understanding the need for continual learning
  - Continual learning
  - The need for continual learning
- Principles of continual learning: Stateless retraining and stateful training
- Challenges with continual learning
  - Obtaining fresh data
  - Data quality and preprocessing
  - Evaluating model performance
  - Optimized algorithms
- Continual learning in MLOps
  - Triggering the retraining of models for continual learning

Objectives

By the end of this chapter, we will understand the concept of continual learning and why it is important for machine learning systems to continually adapt and improve over time. We will cover the principles of stateless retraining versus stateful training.

You will also learn about major challenges with implementing continual learning. We will discuss hot to enable continual learning in an MLOps context. By tracking model and data lineage, establishing triggers for retraining, validating retraining decisions, and maintaining control over the automated retraining process.

You will understand the importance of robust monitoring and observability infrastructure to support triggers for continual learning and evaluate model updates before deployment. Overall, by the end of this chapter, you will gain the knowledge to make incremental retraining and updating of model as easy and automated as possible while maintaining transparency, safety, and human oversight over automated continual learning systems.

Understanding the need for continual learning

We all know about the challenges faced by organizations in adopting AI technologies. One significant obstacle is the absence of continual learning in ML systems. In this section, we will address this challenge and explore how to enable the capabilities of continual learning to overcome these limitations.

Continual learning

Underpinning the concept of continual learning is the principle of lifelong learning drawn from various sources, including data streams, human expertise, and external contexts. Essentially, continual learning allows machine learning systems to learn perpetually, fostering a culture of adaptation and improvement within organizations. By promoting ongoing learning and evaluation, continual learning establishes norms where experimentation, growth, and openness to change become integral to the organizational culture. This culture empowers teams to constantly refine workflows, policies, and objectives to align with emerging needs and opportunities.

In addition, continual learning enables systems to accrue intelligence incrementally, thereby perfecting their responsivity to the tasks at their disposal. For instance, a continual learning-based visual inspection system in a manufacturing plant can progressively enhance its defect detection capability by assimilating real-time feedback from quality experts on the production line. Similarly, a continual learning algorithm powering a chatbot can improve its conversational ability based on interactions with human users. Through this knowledge assimilation insights from the human experts working on the system and its environment, the system's effectiveness and responsiveness to assigned tasks improve perpetually.

Integrating continual learning into any ML system can substantially amplify its effectiveness, enabling users to harness the maximum capability of an AI system progressively. Thus, the implementation of continual learning is highly recommended.

Many people associate the term continual learning with the online training paradigm, where a model updates itself with every incoming sample and data point. However, that is not always the case. This approach is only suitable for select use cases where new data arrives rapidly, such as weather monitoring or user-activity tracking. Training on every new data point can not only become expensive but can also create more issues than it solves. For example, in the case of neural networks, learning with every new data point can make the model susceptible to **catastrophic forgetting**. It is the issue where a neural network can completely and abruptly forget previously learned information upon learning new information.

Most situations where we employ continual learning will benefit from periodic model updates using micro-batches of new data instead. Strategies like incremental learning update the model every few hours or days using batches of new data to prevent catastrophic forgetting. Other techniques like replay learning intermix new data with previously seen data to consolidate knowledge. Choosing the right continual learning approach depends on the use case, data velocity, and model constraints.

There are numerous advantages of an ML model that incorporates continual learning compared to the traditional one that heavily relies on human processes. For instance, let us consider the qualitative coding process for a survey where we need to discover topics being mentioned in the survey. Followed by organizing these topics and making sense of the data before submitting the results, illustrated in *Figure 8.1* are two separate processes. The first one outlines the process managed solely by human experts, resembling a typical traditional qualitative coding process by analysts. Meanwhile, the second figure defines a process augmented or automated using an ML system to perform the tasks of topic discovery and organizing these topics into groups which are then reviewed by human experts to either accept, decline or modify the results to correct them. This process is then followed by the continual learning loop, where the ML model learns from the decision of the human expert. The traditional process takes weeks to complete, whereas the ML system, working synergistically with human

expertise, completes the same tasks within a few hours. Please refer to the following figure:

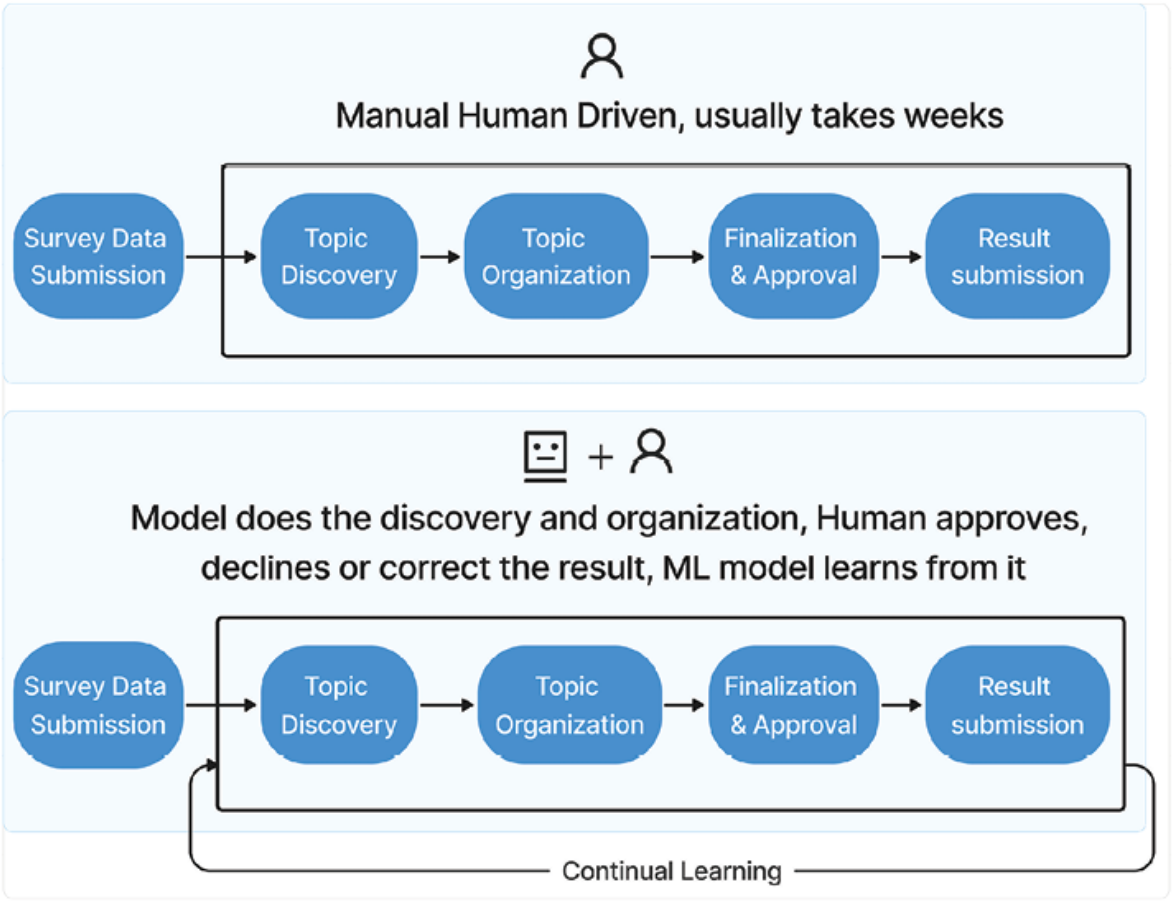

Figure 8.1: *Fully manual process versus ML augmented process with continual learning*

Undoubtedly, the ML system, capable of continuously improving its knowledge through learning, provides a faster and more robust mechanism for organizations. It holds an enormous advantage over systems entirely reliant on humans. One significant aspect of this advantage is the indefinite term of ML systems. For human employees, their term of employment and the associated expertise tend to end when they leave the organization. This departure leaves a knowledge gap that is often costly when training or onboarding a new employee. On the other hand, an ML model that continually learns and improves with the help of human experts preserves knowledge over time. This knowledge retention offers a unique advantage where the institutional knowledge can be preserved indefinitely, unlike in traditional approaches where human employees, and their expertise,

constantly change over time. In essence, continual learning can unlock immense possibilities for any ML system while proving to be a valuable investment for the business in the long run.

However, implementing continual learning effectively poses some notable challenges. First, the ML model must have access to quality, representative data over time to support ongoing learning. If the data is biased or fails to capture key changes in the program space, the model's learning and adaptations will be flawed. Additionally, continual learning systems require careful monitoring and governance to ensure the model does not drift to develop harmful biases over time. Ongoing human oversight is critical but labor-intensive. Furthermore, while the model's knowledge persists, integrating and acting upon those learnings still depends on the capabilities of the human developers and domain experts involved.

Consequently, while continual learning holds considerable long-term value for both an ML system and the business, realizing its full potential requires mitigating key challenges around data, monitoring, and integration. With careful implementation, continual learning can enable organizations to not only optimize processes and achieve faster decision-making but also ensure sustainability by mitigating the risks associated with employee turnover. However, the limitations must be addressed to reap the full benefits over time while minimizing risks. In this way, a balanced understanding of both the advantages and challenges of continual learning empowers organizations to plan effective strategies as well as adapt and thrive in an ever-evolving technological landscape.

The need for continual learning

To understand the need for continual learning, let us go through some of the advantages of continual learning and its impact on machine learning systems:

Adaptability

In many rudimentary machine learning applications, the underlying data distribution may remain relatively constant over time as more data continues to stream into the system. However, there exist numerous real-world applications whereby the data drift or concept drift occurs and the input data undergoes dynamic transformations, the statistical properties of the data, or

the relationships between variables change over time. These include but are not limited to, recommendation engines and anomaly detection systems that continuously ingest fresh data sequences, and the distribution of this data is likely to shift as well. Additionally, the surrounding environments and operating conditions of these systems often evolve dynamically. These dynamic scenarios with data drift and concept drift underscore the importance of continual learning in ensuring that the models can adapt, ensuring accurate predictions.

Scalability

The scale with which we as humankind are producing the data, if we continue on the same trajectory of data production, we might face a situation where we will lack the infrastructure to store all the data being produced. In such scenarios, it is imperative that we devise mechanisms to process data on the fly, as failure to do so will result in valuable information being lost as storage infrastructure fails to keep pace with the rate data will be produced. The trick in such scenarios will be to process all the incoming data in real-time as it comes and store only the essential information we need and get rid of the rest.

Relevance

The pertinence of predictions emanating from ML systems cannot be overemphasized. They need to acclimatize to the shifting contexts and maintain their value proposition. This demands a commitment to continual learning to ensure that ML systems stay relevant and effectively tailor their strategies to navigate ever-changing landscapes.

Performance

Commitment to continual learning facilitates heightened performance within the ML system. The ability to consistently adapt the machine processing logic to cater. It powers machine learning systems to stay relevant by adapting to changing data and the environment. In other words, models staying more relevant will improve the performance of the ML system in areas such as predictive accuracy and other benchmark metrics. This translates to more meaningful and valuable outputs.

Given these reasons, it becomes abundantly clear that the projects which run in dynamic environments can take advantage of continual learning. Moreover, its absence within the ML system setup can introduce a significant risk of project failure, drastically limiting the value realization from ML investments. In stark contrast, a careful setup and continual learning process for a model may be the key to succeeding in these machine learning projects

Principles of continual learning: Stateless retraining and stateful training

The term continual learning sometimes makes people think of the ability to update models frequently within minutes or hours. There is a negative connotation attached to the topic as most experts believe that the majority of organizations do not need to update the models frequently as they do not have enough new data collected in that time frame to make a difference.

The continual learning process is not all about retraining frequency; this learning process is incremental and adaptive, enabling the model to update its knowledge without forgetting the previously learned information. This can also be explained using the terms **stateless retraining** and **stateful training**. In stateless retraining, the model is training from scratch each time, so if we retrain the model on the last 1 month of data and decide to train the models every 15 days, the data being used will be the latest one month's data as a sliding window. Whereas, in stateful training, the model continues training on new data only in an incremental fashion without forgetting what it learned in previous iterations.

In case of MLOps, it is important to make this decision as it impacts the resource requirements for the training. Stateless retraining of a model from scratch every time required a lot more data storage and processing than stateful training, which allows us to update the model with less data. For example, in stateless retraining, we might need to use all the data from the last month. However, if we fine-tune the model from yesterday's checkpoint in stateful training, we only need to use new data from the last one day.

Note: Stateful training does not mean any training from scratch. The ideal strategy combines retraining from scratch and stateful training in an incremental format. The most successful use of stateful training also needs occasional retraining of the model from scratch on a large amount of data to calibrate it.

Once we have the infrastructure set up to allow both stateless retraining and stateful training, the training frequency is just a knob to twist. We can update the models once an hour, once a day, or whenever a distribution shift is detected. Continual learning is about setting up infrastructure in a way that allows the data scientist or machine learning engineer to update the models whenever it is needed, whether from scratch or fine-tuning and to deploy this update quickly.

Continual learning as any other concept also faces some challenges in how to achieve the end goal. MLOps tooling for continual learning is maturing continually which will make it easy to overcome these challenges and make adapting the continual learning as easy as batch learning. In the next section, we will learn about some of these challenges.

Challenges with continual learning

Continual learning is promising for numerous use cases, and many companies have already implemented it to good effect. However, several key challenges remain. This section explores three major obstacles that continue to impede the more widespread adoption of continual learning:

- Obtaining fresh data
- Evaluating model performance
- Optimized algorithms

While much progress has been made, overcoming these barriers will be crucial to unlocking the full potential of continual learning going forward.

Obtaining fresh data

The first challenge with continual learning is obtaining new data continuously. To update a model hourly, we need fresh data every hour. Many companies currently pull new training data from data warehouses. But there is a dependency. Fetching new data from the data warehouse depends on how fast data is ingested into the warehouse, which can be slow if data comes from multiple sources. A solution for the issue is to pull data directly from real-time streams like **Kafka** or **Kinesis**, which are generally used to transport data from the source to the warehouses even before the data reaches warehouses.

Quite often, merely getting new data is not enough. If a model needs labeled data for training, that data must be labeled too. It becomes an issue at times as most of the labeling is done manually to ensure top-notch quality. Labeling speed limits how fast the data for training and available, and the model can be trained. A solution for this is to use the continual learning framework for use cases that have natural labels stock forecasting, ad click prediction, ETA prediction and so on.

Building an efficient streaming infrastructure for fresh data and fast labeling requires extensive engineering. The engineering efforts to enable continual learning go beyond MLOps and need a streaming-first infrastructure and architecture in place for accessing fresh data and extracting fast labels from real-time transports, which is extremely complex and often costly.

Data quality and preprocessing

As we discussed above, one of the main challenges of continual learning is obtaining fresh data, however, simply obtaining more data is not enough for effective continual learning. The quality and preprocessing of the data are also critical factors.

Incoming data will likely contain noise, missing values, outliers, and other inconsistencies that must be handled. Poor quality data can lead to skewed model updates. Preprocessing steps like cleaning, normalization, feature engineering, and data augmentation may need to be performed on streaming data before training. The infrastructure must support near real-time data quality checks and preprocessing at scale. Data quality and monitoring should be made an integral part of the continual learning pipeline.

The entire data pipeline from ingestion to model update should be designed carefully to support incremental learning from dynamic data streams. A holistic approach covering data and model management is required for robust and reliable continual learning in production.

Evaluating model performance

The biggest challenge with continual learning is not writing code to update models. The code is easy. The real challenge is ensuring each update is good and safe before deploying it. As we have seen over time, flawed ML can

cause major harm, from unfair loan denials to fatal crashes. Risks grow with continual learning. More frequent updates mean more chances for failure.

Also, online learning makes models vulnerable to manipulation and attacks. Real-world data opens the door for users to trick models into learning the wrong things. In 2016, Microsoft's chatbot **Tay** started spouting racist and offensive tweets after trolls bombarded it. Microsoft shut Tay down after merely 16 hours.

Each update must be thoroughly tested before going live to prevent similar failures. Evaluation takes time, which can slow updates. Continual learning demands rigor in testing and evaluation to ensure each update is sound. Speed matters, but safety comes first. Tools and techniques to evaluate quickly without compromising robustness remain an active area of research.

Optimized algorithms

Compared to getting data and evaluating models, the algorithm challenge only affects certain models updated frequently, like hourly. To understand why, compare a neural network and a matrix-based model like collaborative filtering. The neural network can update with any size data batch or sample because of its inherent flexibility and adaptability. Neural networks are universal function approximators that can continuously modify their weights and biases to fit new data. Architectures like multilayer perceptrons and convolutional neural networks are especially suited for incremental and continual learning scenarios.

In contrast, a matrix model like collaborative filtering must build the full user-item matrix before reducing dimensions. Doing that full process frequently is too slow and computationally expensive. Matrix factorization techniques like **Singular Value Decomposition (SVD)** need the full dataset for decomposition. This makes frequent incremental updates prohibitive.

Neural networks adapt easily to continual learning due to their ability to modify weights and biases incrementally. However, matrix and tree models do not have the same flexibility. Some algorithms help, like the **Hoeffding Tree** algorithm which constructs decision trees incrementally. However, their use is limited to certain model types. Not only the core algorithm but also the feature engineering must work with partial data.

Features are often scaled using statistics like min, max, and variance. Those require a full pass over the dataset. With small subsets, the stats vary wildly between batches. So models may not generalize across batches. To keep stats stable, compute them incrementally as data streams. Running averages approximate final stats.

Some frameworks like `sklearn` offer `partial_fit` for running stats. But built-in options are limited. Alternative libraries like **Spark Streaming**, **Kafka Streams**, and **Samza** offer more comprehensive streaming and incremental processing. These include tools for managing data streams, computing incremental statistics, and model updates.

In summary, some algorithms adapt to continual learning better than others. Neural networks are the most flexible given their incremental learning capabilities. Matrix and tree models need special handling or alternative online learning algorithms. Feature engineering also needs rethinking to work with partial data. Frameworks help but have gaps compared to streaming libraries. Algorithms and tooling will improve over time to better support continual learning.

Continual learning in MLOps

So far, we have discussed continual learning, its importance, and the challenges. Next, we will discuss how to overcome these challenges when implementing a MLOps architecture and infrastructure for the model development.

For majority of the organizations, starting their MLOps implementation with continual learning may not be the best initial strategy. For such use cases, it is best to start with the process of training and deploying as a linear process. However, certain use cases and industries, such as real-time recommendation engines or anomaly detection systems, may require continual learning capabilities from the outset due to their specific use cases. The decision to adopt continual learning from the start or start as a linear process and then move to continual learning should depend on the specific use case and business requirements.

In the start, the focus of most machine learning teams is not continual learning but developing models that can solve the required business problem at that point in time. At this point, preparing the models which are updated

regularly is not the priority. Most of the time, the models are trained on a set of available data, tested and deployed. While this linear process of training and deployment works for some organizations initially, it evolves into a more sophisticated continual learning approach over time as needs change.

For most organizations, the thought of having continual learning becomes important once we have solved the specific problem we are trying to solve with this model. Now it is the time to make the decision about stateless retraining or stateful training.

In the case of stateless retraining, the MLOps pipeline is as straightforward as with any training pipeline. Fetch the data for the last X amount of time, clean, prepare, feature extraction, train, and verify the model. Once ready, bump the version, and if the model passes the test and verification for deployment, then deploy it to the inference environment.

In case we choose to go ahead with stateful training, the pipeline steps need to be a bit different, especially the initial data preparation steps and the pipeline needs to be reconfigured. The most important thing we need for this is a mindset shift. Currently, stateless retraining from scratch is such a norm that organizations are used to ML teams handing off a new model version to the engineering team for deployment once it is ready. The infrastructure in those cases is not at all set to enable stateful training which needs to be tackled first before we can implement stateful training.

For enabling stateful training, the first thing we need is a way to track the data and model lineage. We need checkpoints on the data to mark which model version is training on the data and up to what points and which base model version is used to train the new version. MLflow is a good place to store these checkpoints in terms of model versions along with the metadata specifying the base model version, point in time on data that is used for training the model, and so on. Once we decide to train the new model version, we need to fetch this metadata and make sure we can locate these previous checkpoints and load the model into memory before continuing training with the data from the checkpoint onwards.

Now we will need to establish the mechanism to trigger the mode updates. As discussed in the previous chapter, the triggers can be **time-based**, **performance-based**, or **drift-based**. If we use performance or drift-based triggers, we will need a robust monitoring solution to continually gather the

data, detect the signals and actuate the retraining pipelines accordingly. We will cover monitoring and observability in the next chapter.

Transitioning from routinely retraining models from scratch to continuously updating them can involve significant technical and process-related complexities. It is important to invest upfront in building organizational alignment and capabilities for this transition.

Finally, version control using tools like **Git** or **DVC** is essential for experiment tracking, reproducibility, and model lineage in MLOps. This is especially critical for stateful training, where new model versions build off previous versions incrementally. By incorporating version control for machine learning artifacts early on, organizations can set themselves up for more controlled scaling of continual learning capabilities.

Triggering the retraining of models for continual learning

The final but one of the most important steps in a continual learning pipeline is to trigger the retraining of the model. ML pipeline is prepared for continual learning where the following tasks are being handled:

- We have the data.
- Experiments are running the training.
- Predictions are being collected and monitored.
- The new data is being cleaned.

Next, we must decide the triggers for retraining your model. There are various effective methods to accomplish this. One common approach is periodic retraining, which involves updating the model at regular intervals. For instance, teams working on recommender systems or ads often opt for retraining every 30 minutes or an hour to keep the model up-to-date with the latest trends.

Alternatively, we can consider a more targeted approach by retraining the model only on new incoming data. This method lets us focus on the most relevant and recent information, saving computational resources and time.

Besides this, keeping an eye on model decay, model bias, low confidence predictions, or any other alerts in the production environment can also serve as triggers for retraining. By actively and continuously monitoring these

metrics, we can ensure that the model performs optimally and remains unbiased.

Regardless of our retraining method, automating the machine learning pipeline can be a powerful advantage in continual learning. It allows models to be retrained and updated seamlessly as new data becomes available. However, as with any automated system, there are risks and challenges that require human oversight. A key risk is losing visibility into when and why retraining is triggered, without proper tracking of retraining decisions and model versions, it is easy to end up with inaccurate predictions or degraded performance in production environments.

To mitigate these risks, it is vital to keep track of the triggers and validate them. Losing sight of when the model was last retrained can lead to significant problems and result in inaccurate predictions or performance degradation.

Maintaining control over the retraining process, even in automated scenarios, is indispensable. Robust tracking of retraining triggers, model evaluation metrics, and deployment records is essential. This level of oversight allows us to understand and debug the model in the production environment effectively. By understanding the retraining decisions, we can make informed adjustments and improvements, ensuring the model's reliability and success in real-world applications.

Overall, the benefits of automating continual learning are clear, faster experimentation and reduced maintenance burdens being chief among them. However, human oversight remains indispensable to ensure model transparency, evaluate retraining decisions, and prevent uncontrolled feedback loops or concept drift. A balanced approach recognizes the power of automation while retraining human-in-the-loop validation. With thoughtful process control and monitoring, we can utilize automated continual learning to improve model accuracy over time while also maintaining reliability in production.

Conclusion

We started the chapter by continuing the discussion on the core concept of continual learning. The need to move and shift the model development paradigm to a continual learning-based paradigm.

The challenges we might face the successfully enable a continual learning-based approach in our machine learning pipelines and MLOps infrastructure and how we can overcome those challenges to take advantage of the process.

We learned what changes we might have to adapt in our approach to model management and training practices to set up the infrastructure and processes in place to achieve the goals of continual learning.

We ended the discussion on the importance of having a solid monitoring and observability infrastructure in place before we can even think of adapting continual learning and continuous delivery of the new model versions. In the next chapter, we will go over the monitoring and observability requirements, how to set those up, and use those metrics to make decisions about when and how to train new model versions and trigger training pipelines.

Points to remember

- Continual learning enables machine learning systems to learn perpetually and adapt to changing data and environments. This helps improve performance and stay relevant over time.
- Key principles of continual learning are stateless retraining (training from scratch on sliding windows of data) and stateful training (incremental training on new data without forgetting past learning).
- Major challenges with continual learning include getting fresh data continuously, thoroughly evaluating each model update before deployment, and using algorithms optimized for incremental learning.
- To enable continual learning in MLOps, track model and data lineage/checkpoints, establish triggers to retrain like time or performance drift, validate and understand each retraining decision, and maintain control over the automated retraining process.
- Solid monitoring and observability infrastructure are crucial before adapting continual learning to understand when to trigger retraining and evaluate model updates.
- The end goal is to make retraining and deploying updated models as easy and automated as possible while maintaining safety, transparency, and control.

Key terms

- **Continual learning**: Continual learning (also known as incremental learning, life-long learning) is a concept to learn a model for a large number of tasks sequentially without forgetting knowledge obtained from the preceding tasks, where the data in the old tasks are not available anymore during training new ones.
- **Catastrophic forgetting**: Catastrophic forgetting is the issue where a neural network can completely and abruptly forget previously learned information upon learning new information.
- **Stateless retraining**: In stateless retraining, the model is training from scratch each time, so if we are retraining the model on the last 1 month of data and decide to train the models every 15 days, the data being used will be the latest one month's data as a sliding window.
- **Stateful training**: In stateful training, the model continues training on new data only in an incremental fashion without forgetting what it learned in previous iterations.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 9
Continuous Monitoring, Logging, and Maintenance

Introduction

In this chapter, we will delve into the fundamental principles for monitoring machine learning models across all environments. The central goal of monitoring machine learning systems is to foster trust, transparency, clarity, and explainability of the **Machine Learning** (**ML**) models, all of which are pivotal in augmenting the business impact of machine learning.

Once the machine learning model is deployed into the production environment, the potential degradation in quality can happen fast and without warning. Sometimes by the time it is detected is too late. This is why we need to log everything the models are doing and continuously monitor the systems. Moreover, this is the reason model monitoring is a significant step in the machine learning model lifecycle and an integral part of MLOps. The deployment of a model or an ML system would lack trust and merit if not accompanied by an extensive monitoring system. Monitoring performance is one of the most vital parts of an ML system. It empowers us to meticulously analyze and map out the business impact an ML system can provide to stakeholders in a qualitative and quantitative manner.

Achieving maximum business impact hinges upon serving users of ML systems in a most convenient manner which enables them to consume the

machine learning system in the way it was intended and generate tangible value.

Structure

In this chapter, we will discuss the following topics:

- Key principles of monitoring in machine learning
- Why model monitoring matters
- Monitoring in the MLOps workflows
- Frameworks for model monitoring

Objectives

The objective of this chapter is to understand the principles of monitoring and logging. The importance of continuously monitoring a machine learning system, its impact, and how it fits in the MLOps workflow.

Key principles of monitoring in machine learning

Building trust in **Artificial Intelligence** (**AI**) and ML systems is important with a rising demand for products to be data-driven and to adapt to the changing environment and regulatory frameworks. One of the major factors for the failure of a lot of machine learning projects in delivering value to businesses is the lack of trust and transparency in their decision-making process and the consistency of model performance over time. To be able to build this trust and provide transparency, we need to keep track of each and every action of the model and its actions. To do so we need a robust monitoring infrastructure in place.

In the following subsections, we will go over some of the key principles we should follow and keep track of when it comes to monitoring machine learning models.

Model drift

We live in a dynamic changing world. Due to this, the environments and the data in which a machine learning model is deployed to make predictions are

continuously evolving. It is essential to keep track of these changes and how it impacts our models. Drift is related to changes in the environment and refers to the degradation of the predictive ML model's performance and the relationship between the variables degrading.

Following are the four types of model changes with regard to models and data:

Data drift

This occurs when the characteristics of the factors we are studying change. For instance, data might change because of new behaviors of the users or new products being introduced to meet consumer demands. To detect data drift, statistical tests like **Kolmogorov-Smirnov** can be used to compare the distribution of new data to the training data. Data monitoring tools like **Evidently** can also automatically monitor for data drift. Strategies to mitigate data drift include retraining models on new data or using techniques like continuous retraining.

Feature drift

This happens when the properties of the features themselves change over time. For example, the temperature changes with the different seasons. It is cooler in winter compared to summer or autumn. So if there is a model that predicts energy consumption, the temperature fluctuations and changes may change the features in use. Feature drift can be detected by monitoring feature distributions over time. Adding automated tests to validate feature values can also help. To mitigate, features may need to be transformed or normalized before model input.

Model drift

Model drift is where the properties of dependent variables change. For instance, in a fraud detection ML system, this would be when the classification of fraud detection changes. Model drift is often measured by a deterioration in model performance metrics like accuracy, precision or AUC. New validation datasets can be used to test for model drift. Retraining on new data is the main way to address model drift.

Upstream data changes

This is the case when the data pipeline undergoes data changes, like when a certain feature is no longer being collected, leading to missing information and thus impacting the behavior of the model and the final output. Monitoring data pipelines for schema changes can detect upstream drift. Adding validations for required features can also help. Mitigations include updating data pipelines, inputting missing values or retraining models.

Model transparency

AI is non-deterministic in nature. Machine learning in particular continually evolves, updates, and retrains over its life cycle. Machine learning and AI are impacting almost all industries and areas. With this increasing adoption of machine learning in everyday processes and important decisions being made using machine learning systems, it is crucial to establish the same level of trust that we have in deterministic systems.

Model transparency is all about working towards building trust in machine learning systems. The purpose of this is to ensure fairness, to get rid of bias as much as possible, and keep the system accountable by auditing the end-to-end process of how the system makes decisions and explaining why the model makes certain predictions and system decisions.

There are several concrete strategies and techniques that can be used to increase transparency in machine learning models:

- **Explainable AI** (**XAI**): XAI techniques like **LIME** and **SHAP** help explain individual predictions made by black box models like deep neural networks. They identify the key input features that led to a particular prediction.
- **Interpretable models**: Using inherently interpretable models like linear regression, decision trees, or rule-based systems instead of black box models. Their working can be explained in terms comprehensive for humans.
- **Model introspection**: Techniques like testing with edge cases, saliency maps, and adversarial examples that analyze the model itself to identify flaws.
- **Quantifying uncertainty**: Adding confidence intervals and uncertainty estimates to predictions to know when the model is

unsure.

- **Data provenance**: Detailed logging and documentation of the data collection and pre-processing steps make the origination and preparation of data transparent.
- **Auditability**: It includes allowing independent auditing of the entire machine learning pipelines, including data, model training code, model optimization steps, and final model evaluation.
- **Explainability infrastructure**: Open-source libraries like MLflow that capture model explainability data for consistent model monitoring and auditing.

Adopting these best practices for explainable and transparent AI will be crucial as machine learning becomes more widespread. They help build user trust, reduce bias, and provide recourse when questionable model behavior is identified.

Model bias

Model bias is a type of error due to certain features of the dataset (used for model training) being more heavily represented and/or weighted than others. A misrepresenting, unbalanced, or biased dataset can result in skewed predictions generated by the model, and accuracy might be low. In other words, it is the error resulting from incorrect assumptions made by the ML model. High bias can result in predictions being inaccurate and can cause a model to miss relevant relationships between the features and the target variable being predicted.

Model compliance

Ensuring the models are compliant has become crucial in recent years as governments and regulatory bodies impose regulations on AI/ML systems. If the models are non-compliant, it can cause fines from the governments as well as harm society. Some key regulations that machine learning systems may need to comply with include:

- **General Data Protection Regulation** (**GDPR**): It regulates the use of personal data and provides rights to EU citizens related to their data

privacy. MLOps can help ensure GDPR compliance by properly managing the securing training data.

- **Health Insurance Portability and Accountability Act (HIPAA)**: It regulates protected health information. MLOps can assist with HIPAA compliance through proper de-identification and anonymization of patient data used in model training.
- **Sarbanes-Oxley Act (SOX)**: It regulates financial reporting for public companies. MLOps can enable audit trails showing how models were developed, tested and validated to comply with SOX.

Operationalizing regulatory compliance is becoming more and more important to avoid unnecessary fines and damage to society. End-to-end auditing of ML systems is essential for monitoring compliance. MLOps can play a vital role in facilitating auditing and retracing the operations and business decisions that are made by machine learning systems. It can facilitate this by keeping track of all the ML models' inventory, configuration, inputs, outputs, and performance metrics. This model lineage provides visibility into how models work and how they are governed. With proper MLOps implementations, organizations can operationalize compliance by setting up continuous validation checks, monitoring for model drift, and generating detailed reports for auditors. Overall, MLOps is key to scaling compliant ML deployments and avoiding regulatory pitfalls.

Why model monitoring matters

In the world of MLOps, it is crucial to closely track model iterations and continuously monitor the performance of the models as well as the overall machine learning systems. However, it is equally important to recognize that the monitoring requirements differ for different organizational stakeholders.

Let us look into some of the priorities of different stakeholders.

For DevOps or infrastructure teams

For the DevOps or the Infrastructure team, the main concerns are the infrastructure requirements and performance of the system and include questions like:

- Is the model running properly on the hardware and getting the job done quickly enough?
- Is the amount of memory and compute capacity the system is using is below the threshold?

These monitoring metrics are standard for any system's performance monitoring and DevOps/infrastructure teams can monitor this well if they are not already doing it. The infrastructure resources (both storage and compute) used for machine learning systems are not significantly different from any other technical system. Therefore, the implementation of metrics for monitoring these resources has been well studied and mature. But this is not all; a machine learning system can meet these requirements and still be ineffective due to many other reasons or dependencies.

For data science or machine learning teams

For machine learning engineers and data scientists, the interest in monitoring is to detect and track model drifts and degradation and monitor the performance of the model to figure out if they are degrading. There are two approaches most commonly used for this, ground truth-based process or input drift-based detection. Understanding each concept at a high level is important to facilitating conversations and detecting the issues based on monitoring metrics.

Ground truth

We know the predictions made by the model, but in addition to that, if we can know the ground truth of all of those predictions as well, it will help us to compare and calculate the performance of the model. Sometimes we can find out the ground truth quickly after a prediction. For example, in recommendation systems, if a recommendation is shown to a user on a web page, and the user clicks on the recommendation and ends up buying, then we will know the final result and ground truth immediately.

But in lots of cases, getting the ground truth takes a lot more time, which makes it difficult to monitor the performance of the system on a continuous basis in a timely manner. It might be too late to figure out and alert on the performance degradation of the model by the time we have the ground truth available for comparison.

If we need quick feedback, it might be better to look at input drift as an approach to model performance drifts.

Input drift

Input drift monitoring and performance calculation are based on the idea that a model will only be able to provide good predictions if the training data is an accurate reflection of the type of data we will see in the production environment. So, if we compare some of the recent requests or inputs to the model against the data it was trained upon, and if they are quite different, there is a strong chance that the model performance will not be good and that the model might degrade.

This is why we check for input drift. All the data we need to make this calculation is already available within an acceptable time frame. We can monitor the system in a continuous fashion without having to wait for the ground truth data to be made available, which might be delayed.

For business stakeholders

Business stakeholders are not concerned with what kind of resource the model is using or what kind of input data it is receiving. Their approach is more holistic in nature when it comes to monitoring, and they might have questions like:

- Is the model delivering the promised value to the organization?
- What is the **Return On Investment** (**ROI**)? Does the use of the model and the advantage it provides outweigh the cost of developing and running the model?

Generally, business stakeholders prepare a set of KPIs to quantify the success of any project. Those KPIs are also part of this monitoring and reporting. It is best if these numbers are tracked and monitored automatically, but that is not always easy to implement. The key consideration is, at some point, the machine learning projects being used would not work well anymore, and model retraining becomes inevitable. Now when that happens, it is outside the control of the model and the system itself in most cases. Moreover, it depends upon how quickly the input data and the overall environment change. Another consideration is the effort required to retrain the model and the ROI of that improvement. If it is easy

to arrange the training data, retrain on that data, evaluate, and deploy the model, then we can retrain more frequently than when it requires a lot more efforts.

For legal and compliance teams

Legal and compliance teams have a strong interest in monitoring model performance to ensure models meet regulatory requirements around fairness, explainability, and data privacy. Key monitoring needs for these teams include:

- Monitoring model outputs for signs of unfair bias or discrimination against protected groups.
- Assessing model explainability by monitoring how well humans can understand model outputs and decisions.
- Tracking what data is used to train models and ensuring continued compliance with relevant privacy regulations.

Capturing metrics in these areas and alerting on potential issues is critical for maintaining organizational compliance and managing legal risk with machine learning systems.

Monitoring in the MLOps workflow

As we learned about the MLOps workflow, **Model health monitoring** and **Governance** are an integral part of any MLOps workflow as indicated by *Figure 9.1*. **Governance** and auditing are, in a way, dependent upon monitoring and proper logging. Whenever a new version of the model is deployed to the production environment and is serving, we have to keep track if it is performing better or worse. Not only do we have to know it, but also the information has to be easy to access and straightforward to log. Automation brings reliability into these processes and can effortlessly create information availability which can then be tracked for performance metrics and auditing purposes.

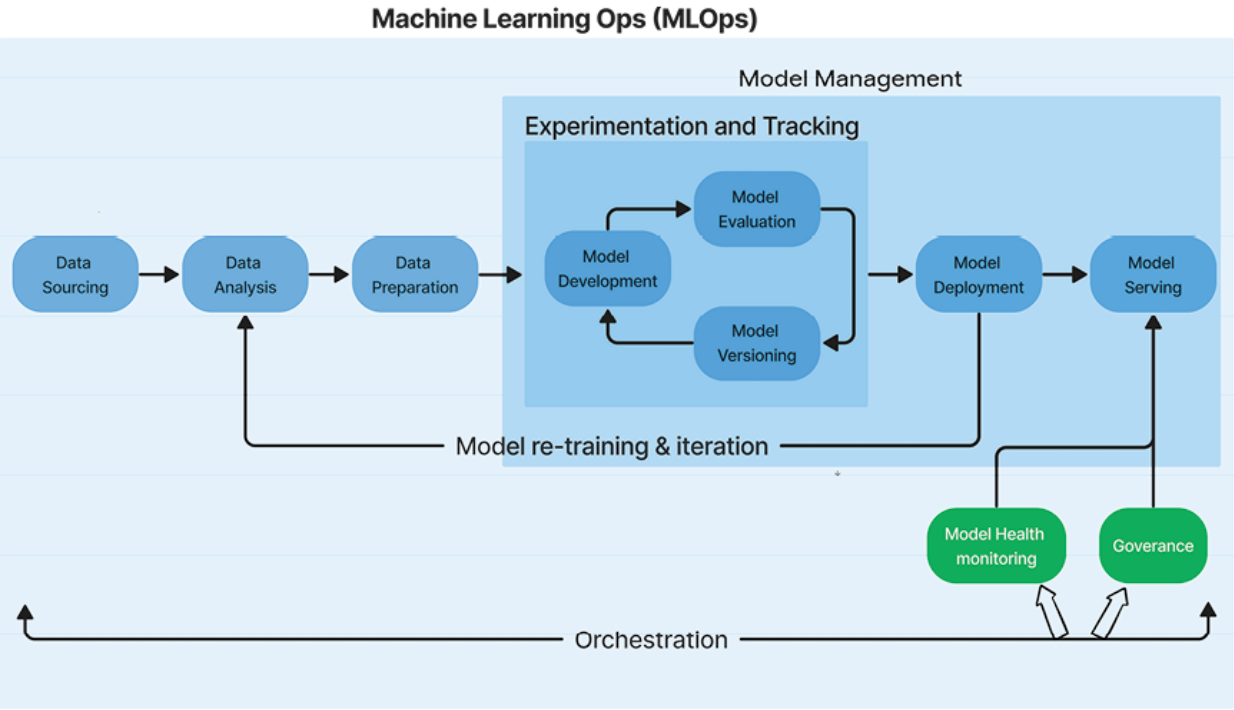

Figure 9.1: *Monitoring and governance components of an MLOps workflow*

Monitoring in MLOps incorporates everything involved in serving a model in production; from capturing information about the systems and services to the performance of the model itself. Monitoring and logging are similar to knowing the data before we decide which features and algorithms to use to train a model. The more we log and know about the data, the better decisions we can make about improving the models and overall systems.

Likewise, the better we understand the steps in getting a model ready and into production, the better choices we can make about what to track and when to get alerts. The basic principles we discussed earlier in the chapter set a good base to build our monitoring components. These principles help us clarify what data to collect, how often to collect, how to alert best and visualize this data, and which stakeholders need what metrics. Depending upon what step of the MLOps lifecycle we are on, the metrics we want to collect and monitor also differ. For example, when collecting data and cleaning it up, it might be best to detect the number of empty values per column and perhaps the time it takes to complete every step.

There are some specific types of metrics common in a lot of systems and something we can use in a machine learning system as well. These are:

- **Counter**: This metric can be useful when counting any type of item. It helps in iterating over items. For example, this could be useful for counting empty cell values per column.
- **Timer**: A timer is excellent when determining the processing duration. This is crucial for performance monitoring, as its core responsibility is to measure time spent during an action. One example of how a timer can be used is if we have a hosted model, a timer would help capture how long the model took to produce a prediction over HTTP.
- **Value**: When counters and timers do not fit the metrics to capture, values come in handy. For example, the size of the database or the size of the file stored in S3, which is produced and will be used for training the model.

Machine learning models, in general, need to be monitored at two levels:

- At the infrastructure resource level, which includes making sure the model is running properly in the main production environment. Some of the key questions, as also discussed above under DevOps metrics, include:
  - Is the system alive? This can be monitored by using a heartbeat metric where we send an API call at a certain interval to see if we still get a proper HTTP 200 response.
  - Other infrastructure metrics like CPU, RAM, network usage, and disk space are under the threshold and as expected from a system like this.
  - Are requests being processed at the expected rate, and is the latency of the response under the threshold?
- At the model performance level, which is monitoring the performance of the system over some time frame to make sure everything is as expected. Some of the important questions to ask for this set of monitoring metrics are:
  - Is the model still receiving the input data for decision-making, which is in line with the type of data used for training, and there is no model drift yet?

    - Is the model performance still above the threshold and baselines used during the development of the model, or is there a negative change in that performance?

As we know, the first level is the domain of infrastructure or DevOps teams. They can lead the implementation of those metrics and checks. We will not delve into details on those topics. The second level is of importance for data scientists and machine learning engineers. Continuous model performance monitoring for these metrics helps in tracking model degradation, and, if the performance falls below the acceptable threshold, the system will also trigger the retraining pipeline of the model with more latest training data available.

Logging

Monitoring any system, not just machine learning systems means collecting, aggregating, and visualizing as much data as possible about its states. With increasingly complex machine learning systems being used in production environments for decision-making, the infrastructure requirements are also complex for these systems. Therefore, a robust and detailed logging system is important. To effectively analyze and monitor the log data, it is essential to centralize this data from all the environments where the model is running. Ideally, this should happen automatically, but if not, then at least manually whenever required. This is an essential component to achieve continuous improvement in the machine learning system.

A log is a record with some relevant information to tell us about the system's state, along with a timestamp to tie it to a specific timeframe. This is true for any information system and not just specific to machine learning systems. Information that this log event needs to include so that we can understand the system may contain elements like:

- **Model metadata**: It includes the identification of the model, the environment it is deployed in, and the version of the current implementation.
- **Model inputs**: The input values to the model for that specific event. This allows us to decipher whether the values match the types the model expects and is trained upon or if the data values change. This

also helps to understand how the model's performance is tied to the input data and allows us to detect data drift.

- **Model outputs**: Output values, as evident from the name, are the predictions made by the model. These values, when compared with the ground truth, also allow us to reach a conclusion about the performance of the model in this environment and are an important data point in deciding if we should opt to retrain the model or wait for now.
- **System action**: System action is the data point that tells us what action the system took with the prediction provided by the model or how the prediction impacted the system's state. For instance, in an email spam detection system, if the model gives a high probability that an incoming email is spam, then the system takes action to mark the email as spam and move it to the spam folder instead of the inbox which is not directly visible to the user unless they specifically check the spam folder.
- **Model explanation**: Model explanability is becoming quite important in today's environment with many important and high-risk decisions being taken or influenced by machine learning models, especially in domains like healthcare or finance. It becomes important to be able to figure out and audit how the system made a specific decision, and which features had the most influence on the prediction.

In addition to the above information, it is important to log enough information so we can figure out what is going wrong. If the model is used in a platform where it directly interacts with the user or influences the actions of the user, it is a good idea to log the user's context as well along with other logging events, that is what actions the user was performing and how they were interacting with the model. The context here means details like the state of the application and what the user did immediately after the model prediction was presented to the user as well as the web session details of the user.

Some of the best practices for setting up an effective logging system include:

- Using structured logging formats like JSON that allow for easier searching, filtering and analysis compared to unstructured text logs.

JSON logs can capture metadata, timestamps, severity levels, and other fields in a consistent machine-readable format.

- Organizing logs into a hierarchy based on factors like logger name, application name, environment, and so on. This provides the context needed to pinpoint issues.
- Centralizing logs from all environments and applications into a log aggregation tool like **ElasticSearch**, **Splunk**, **Datadog**, and so on. for easy access and analysis. These tools allow for search, visualization, alerts and more.
- Implementing log rotation and retention policies to avoid filling up storage. Older logs can be archived or aggregated.
- Anonymizing any personally identifiable data in logs before storage and analysis to protect user privacy. Techniques like hashing, truncation, randomization, and aggregation can be used. Providing clear notifications to users on what data is logged and how it is used. Give them an opt-out if possible.
- Consulting local regulations like GDPR when implementing data retention policies. Only keep data as long as necessary.
- Using data masking techniques when collecting data to comply with privacy regulations. This includes partial masking and pseudonymization of personal data fields.

Now on paper, it seems considerable that we should log and track all these events and the related data points. Still, in large complex systems with millions of users, where the model is serving all these users quite frequently, it becomes almost impossible to log every event and every detail. In such scenarios, it makes more sense to follow stratified sampling to collect the representative data points and still be able to make sense of the system's state. We first need to decide which group of events we want to log, and then we log only a certain percentage of events in each group. Now here, the groups can be represented as a subset of users or a subset of sessions, and so on.

Another important aspect of collecting all this user activity data and interaction data with the model is the governance of this data. The users

should be made aware of what kind of data is being logged and user, how it is used, and for how long it will be stored, along with the possibility for them to opt out of this data collection if they prefer. Some of the governance policies also require us to anonymize or aggregate the user data points, so it is impossible to pinpoint a specific user. Governance controls also require us to manage who and how this data is accessed and who has access to the data for what period of time. Different regions also have data retention policies such as the **General Data Protection Regulation** (**GDPR**) in Europe, the **California Consumer Privacy Act** (**CCPA**) in the USA, and so on.

Model evaluation

Once the infrastructure for the logging and the robust logging system is ready, we will fetch data from the production environment regularly, keep track of how it is performing, and have alerts and visualizations in place to help monitor. Everything has gone well for some time now, and then a one-day data or model drift alert is triggered. For instance, we might be alerted of the input data distribution drifting away from the training data distribution or the model prediction performance falling below the acceptable threshold. We know that the model performance is degrading and needs our attention.

The model goes into review; we review the available monitoring data and decide that the model needs to be retrained. The data collection, preparation, and retraining pipelines get triggered which prepares a new version of the model which performs well during the evaluation on the test dataset as well. Now we have two available options based on the maturity of the overall machine-learning infrastructure:

- We can update the model in the production environment and continuously monitor how it is performing.
- We move to an online evaluation setup using one of the following processes:
    - Champion/challenger testing method
    - A/B testing setup

Whether we decide to evaluate offline before deployment or online evaluation using one of the above-mentioned processes, evaluating the model is important to maintain the performance. Model evaluation serves as

a structure that allows us to centralize the data we use across the model comparison life cycle, which allows us to compare different models or model versions.

Steps and decisions for the monitoring workflow

In MLOps workflow, we have to make some decisions for evaluation, testing, and monitoring of the models to properly use these components to make good decisions about model performance and deployment to production environments.

Before the model evaluation, testing, and monitoring

We should have a clear KPI or metric that we want to optimize the model performance against. This metric needs to be quantifiable such as recommendation acceptance rate or click-through rate.

Next, we need to finalize a cohort that we will use to quantify this performance metric. We can choose a segment of users from the total user base making sure the segment is balanced and representative. This may be just a random split, or we can apply more complex segmentation to make sure there is no bias in the selected subset.

Now as we have the performance metrics we want to use and the cohort of users we want to use for this calculation, we need a statistical protocol that will be used to compare the resulting metrics, and the null hypothesis is either rejected or the model needs to be retrained.

To reach a robust conclusion, we need to define the sample size and the cohort beforehand, we also need to fix a test duration so that an informed decision can be made based on these observations.

During the evaluation and testing

During the evaluation process, we need to make sure the tests are running for the whole duration of the timeframe selected for testing. It is the only way to ensure we have statistically acceptable tests that we can reliably use. Sometimes, the statistical tests start to return a significant metric difference, and we decide to stop the tests midway. This practice may generate unreliable and probably biased results as we are cherry-picking the desired outcome.

After the evaluation and testing

Only once the test duration is over we move to the final stage of evaluation; we can now check the resultant data and make sure the quality is acceptable to make good decisions. Once done, we can run the statistical tests, which we decided beforehand. If the metric difference is statistically significant in favor of the new model (candidate model), we will go ahead and replace the current version of the model with the new version. If the difference is significantly in favor of the old version, then the candidate model fails the challenge and should not be deployed to the production environment and should be discarded. This brings us to the point where we need to re-evaluate and see how we want to improve upon the previous model and the circle continuous.

Frameworks for model monitoring

There are many libraries and frameworks available in the market, both open source and proprietary, which can be used for model performance monitoring. We will glance over some of the open-source libraries and frameworks in this section and also into how these can be integrated into MLOps tools for training, testing and deployment.

Frameworks

Frameworks in general sense are the set of concepts, practices or tools that provide the foundation for designing, building and solving specific problems in any particular field. In the field of model monitoring, there are various frameworks and libraries that helps data scientists and machine learning track the performance, reliability, and behavior of machine learning models. **Whylogs**, **Evidently** and **Alibi Detect** are some of the popular frameworks which we will look into the following sections.

Whylogs

Whylogs (**https://whylabs.ai**) is an open-source library for data logging for machine learning models and data pipelines. It provides visibility into data quality and model performance over time. Whylogs generate summaries of the dataset called **whylogs profiles** which provide us visibility into the data. Some of the features are whylogs framework as listed below:

- Open source, and high-performance statistical data logging library
- Can run in Python as well as Apache Spark environments
- Ability to capture model metrics and monitor data quality
- Statistical metrics about structured or unstructured data
- The library is platform-agnostic thus it is extremely easy to integrate with any platform.
- There is no alert mechanism available directly with the library, so we have to rely on other alerting tools to achieve that.

Evidently

Evidently (**https://www.evidentlyai.com**) is an open-source Python library for data scientists and machine learning engineers to help evaluate, test, and monitor data and ML models from validation to production. Evidently can work very well with tabular text data as well as embeddings. Moreover, *Evidently AI* is the company behind the framework and open-source library, which also provides a Cloud service to streamline the process but is optional to use if we do not want to or have our infrastructure for monitoring. Key features of Evidently are:

- The core of evidently framework is open source.
- It can help us generate interactive reports on model and data performance.
- It can capture data drift, target drift.
- It is easy to integrate with any platform as it is platform-agnostic.
- It can log metrics directly of MLflow as well which can streamline the platform where all the metadata is available.
- One drawback is that it does not provide any alert mechanism. So we have to have our own alerting system in place.

Alibi Detect

Alibi Detect is another open-source Python library focused on outlier, adversarial and drift detection. It provides both online and offline detectors

for tabular data, text, images and time series. Some key points of the library are:

- It is an open-source Python library focused on outlier and drift detection.
- It has the ability to capture drift for tabular data, text, images and time-series data.
- It supports both **Tensorflow** and **Pytorch** backends for drift detection.
- It can integrate with MLflow and log metrics in the platform which makes it easy for us to aggregate all the metadata in the same platform.
- Like all other libraries and frameworks, there is no integrated alert mechanism.

Integrating with tools

Integrating these monitoring frameworks such as Whylogs, Evidently, and Alibi Detect with MLOps pipeline tools like Apache Airflow provides us with an ability to automate the monitoring and keep track of all the model versions for comparison purposes. This integration ability not only streamlines the deployment and monitoring of machine learning models but also empowers data scientists, machine learning engineers and operations teams to collaboratively ensure the robustness and transparency of the models and the pipelines. These integrations might work slightly different in training and testing phase versus the production phase. In the following section, we will see what possible approaches we can take for testing/training pipelines and for production systems.

In training and testing pipelines

For testing and training pipelines, the example tool we used so far in our discussion is Apache Airflow. An Airflow pipeline is a **Directed Acyclic Graph** (**DAG**). It means that every execution will have a task which it starts from, and it will always execute tasks in the forward direction, without loops. In machine learning pipelines, a successful pipeline run is not enough; successful pipelines might represent wrongful results. For example, an ML

job that was completed successfully might have led to drifted data or the model's performance below the threshold.

One way to handle this is to integrate any of the above-mentioned frameworks into DAGs and monitor whatever metrics we need. These metrics can also be logged into our MLflow run so that all the data we need to make decisions is available in the MLflow dashboard already associated with the specific run.

In production systems

Once the models are deployed in production systems, MLfLow might not be a good place to track the metrics for each interaction of the metrics. This is where production-grade monitoring and observability platforms come in handy. One such industry standard platform is **New Relic**.

New Relic also offers monitoring for machine learning models, which makes it easy for machine learning engineers and data scientists to log the model metadata, metrics, and outputs to the platform which most likely is already used by the other software teams in the organization. New Relic supports automatic and custom dashboards on the data, alerts and correlated system events.

New Relic have built and open sources its own Python library (**https://github.com/newrelic-experimental/ml-performance-monitoring**). This set of tools will allow us to monitor various ML outputs directly from the code.

The library can be installed using the following command:

```
pip install git+https://github.com/newrelic-experimental/ml-performance-monitoring.git
```

Once we have it properly installed, in order to use it in our code, we need to get a license key from New Relic dashboard and set it as an environment variable `NEW_RELIC_LICENSE_KEY`. Provided we have a New Relic account, if not, we will need to setup and procure a New Relic account first.

We can send some model-based data by creating a dictionary with a set of metadata key values and prepare the data that we need to send. We can use `wrap_model()` function to wrap the model or pipeline. This wrapper will send the inference data and data metrics automatically to New Relic. If we want

to control and send the inference and metric data manually, then we can use **`MLPerformanceMonitoring`** instead.

```
from ml_performance_monitoring.monitor import
wrap_model


metadata = {"environment": "aws", "dataset":
"Example Dataset". "version": "1.0"}

features_columns, labels columns =
(LIST_OF_FETURES_COLUMNS, LABELS_COLUMN)


# This returns the instance of
MLPerformanceMonitoring

ml_performance_monitor_model = wrap_model(

insert_key=NEW_RELIC_KEY,

model=MODEL_BEING_USED,

model_name=NAME_OF_THE_MODEL,

metadata=metadata,

send_data_metrics=True,

features_columns=features_columns,

labels_columns="categorical"

)


# Calling the predict on the wrap function will run
the predict function followed by automatically
recording the inference data
```

```
Y_pred =
ml_performance_monitor_model.predict(X=X_test,)


rmse = round(np.sqrt(mean_squared_error(y_test,
y_pred)), 3)

metrics = {"RMSE": rmse,}


# Send the model metrics as a dictionary to
NewRelic

ml_performance_monitor_model.record_metrics(metrics
=metrics)
```

All this data, once logged, will be available in New Relic dashboard under the ML Model monitoring view.

This is one example of how existing production-grade platforms can be used for continuous monitoring of machine learning systems in production environments and can be reported upon. There are other similar tools that can be used instead of New Relic based on what infrastructure is already available and is used in the organization. Other alternatives to New Relic can be:

- **Elasticsearch, Logstach, Kibana (ELK) stack**: Open source logging and visualization stack that can handle high volume ML monitoring data.
- **Prometheus and Grafana**: Provides another combination that is frequently used to log and report on metrics.
- **Datadog**: Monitoring and analytics platform designed for cloud application and containers and can be used for MLOps and model inference tracking as well.
- **Amazon CloudWatch**: Fully managed monitoring service that integrates very well with other AWS services.

The choice of which monitoring solution to use during development and evaluation and what to use in the production environment depends on factors like existing tooling, data volumes, infrastructure, visualization needs and ease of interfacing with the ML deployment environment. Having clear requirements around metrics, logs and visibility needed for the ML systems can help guide the selection of the right monitoring and observability platform.

Conclusion

We started the chapter by continuing the discussion on the core principles of model monitoring and logging. The basic difference between software systems and ML systems is that softwares are built to satisfy specifications and its ability to do so does not change over time, whereas ML models are build to satisfy certain statistical objectives and as a result, its performance can degrade over time. especially when the statistical properties of the incoming data changes.

This results in the need to continuously monitor the performance drift of the models so that early actions can be taken to improve the performance. Among possible drift mitigation measures, the workhorse is retraining on new data, while model modification remains an option. Once a new model is ready to be deployed, its improved performance can be validated using A/B testing.

We also looked at some of the frameworks and libraries available for facilitating the monitoring of the models. It is advised to try these frameworks and choose the one that best fits the overall use case and infrastructure. Besides these frameworks and libraries, the specific platforms to use for observability also depend upon the overall infrastructure maturity of the organization. If there is already infrastructure setup and present to be used for logging and observability for other systems, it is good practice to use the same systems for machine learning observability as well. Some of those platforms that are most frequently used are New Relic, ElasticSearch and Prometheus-Grafana.

Irrespective of which framework or platform is used, all the principles and metrics we have discussed in this chapter are of high importance and

platform agnostic. They can be continuously tracked and monitored using any of the observability platforms.

Points to remember

- Model drift refers to the degradation of a model's predictive performance over time due to changes in the data or environment. It is important to monitor for different types of drift like data drift, feature drift, and model drift.
- Model transparency and explainability are crucial for building trust and ensuring fairness. Monitoring can help track model bias and compliance issues.
- Different stakeholders like data scientists, DevOps engineers, and business leaders care about different performance metrics. It is important to monitor metrics relevant to each group.
- Logging model metadata, inputs, outputs, system actions, and explanations are essential for monitoring model performance. Stratified sampling can help manage log volume.
- Frameworks like whylogs, Evidently, and Alibi Detect can facilitate model monitoring. Tools like New Relic also have ML monitoring capabilities.
- Model evaluation methods like champion/challenger and A/B testing allow controlled experiments to validate new models before deployment.
- Monitoring should occur continuously through the ML lifecycle - during training, testing, and in production. This allows early detection of drift and performance issues.
- Governance controls like data privacy and retention policies must be considered when logging and monitoring model performance data.

Key terms

- **Model drift**: Model drift is the decay of models' predictive power as a result of the changes in real-world environments. It is caused by a variety of reasons including changes in the digital environment and ensuing changes in the relationship between variables.
- **Ground truth**: Ground truth is the information that is known to be real or true, provided by direct observation and measurement (That is, empirical evidence) as opposed to information provided by inference.
- **Return On Investment** (**ROI**): It is a ratio between net income and investment. A high ROI means the investment's gains compare favorably to its cost. As a performance measure, ROI is used to evaluate the efficiency of an investment or to compare the efficiencies of several different investments.
- **Key Performance Indicator** (**KPI**): It is a type of performance measurement. KPIs evaluate the success of an organization or of a particular activity (such as projects, programs, products and other initiatives) in which it engages.
- **Explainable AI** (**XAI**): It refers to the ability of AI systems to provide understandable explanations for their decisions. It aims to make complex AI models transparent and interpretable, offering insights into their reasoning through features like feature importance, rule-based explanations, visualization, natural language, and more. XAI is crucial for ensuring trust, accountability, and ethical use of AI in critical decision-making contexts.
- **General Data Protection Regulation** (**GDPR**): It is the European Union regulation on information privacy in the **European Union** (**EU**) and the **European Economic Area** (**EEA**). It also governs the transfer of personal data outside the EU and EEA.
- **California Consumer Privacy Act** (**CCPA**): The act is a state statue intended to enhance privacy rights and consumer protection for residents of the state of California in the United States.
- **Sarbanes-Oxley Act** (**SOX**): The act is intended to increase transparency and accountability in publicly traded companies. It imposes strict rules for financial reporting, mandates independent

boards and audit committees, ensures auditor independence, requires strong internal controls, protects whistleblowers, and imposes harsh penalties for misconduct.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

V

W