

**UNIVERSIDAD POLITÉCNICA DE MADRID**

**ESCUELA TÉCNICA SUPERIOR  
DE INGENIEROS DE TELECOMUNICACIÓN**



**MÁSTER UNIVERSITARIO EN  
INGENIERÍA DE TELECOMUNICACIÓN**

**TRABAJO FIN DE MASTER**

**Design and Development of Automatization Pipelines for  
Machine Learning Projects based on the MLOps  
Frameworks**

**Aitor Martín-Romo González  
2023**



## TRABAJO DE FIN DE MASTER

**Título:** Design and Development of Automatization Pipelines for Machine Learning Projects based on the MLOps Frameworks

**Título (inglés):** Design and Development of Automatization Pipelines for Machine Learning Projects based on the MLOps Frameworks

**Autor:** Aitor Martín-Romo González

**Tutor:** Carlos Ángel Iglesias Fernández

**Ponente:**

**Departamento:** Departamento de Ingeniería de Sistemas Telemáticos

## MIEMBROS DEL TRIBUNAL CALIFICADOR

**Presidente:**

**Vocal:**

**Secretario:**

**Suplente:**

**FECHA DE LECTURA:**

**CALIFICACIÓN:**



**UNIVERSIDAD POLITÉCNICA DE MADRID**

ESCUELA TÉCNICA SUPERIOR DE  
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos  
Grupo de Sistemas Inteligentes



**TRABAJO DE FIN DE MASTER**

Design and Development of Automatization Pipelines for  
Machine Learning Projects based on the MLOps  
Frameworks

**Aitor Martín-Romo González**

Febrero 2023



# Agradecimientos

---

A mis padres, Esmeralda y José María, por su esfuerzo incansable. Por hacer que sólo tuviera que centrarme en mi camino.

A mis abuelos, Pilar y Diego, por enseñarme lo que significa la familia y el estar siempre cuando los buenos momentos no acompañan.

A mi hermano Alex, porque pueda acompañarte siempre en tu vida.

Gracias.





# Resumen

---

Este Trabajo Fin de Máster (TFM) se enfoca en la evaluación, selección y despliegue de entornos MLOps, destacando el análisis, diseño, implementación y validación de sistemas para la gestión de proyectos de Machine Learning. Este estudio compara múltiples plataformas y herramientas MLOps disponibles en el mercado, considerando criterios como la automatización del flujo de trabajo, gestión de modelos, escalabilidad e integración continua. Tras un análisis exhaustivo, se elige la opción más adecuada y se implementa en un caso de estudio real.

Además de abordar los objetivos iniciales, este TFM se propone delinear las conclusiones alcanzadas y los objetivos logrados. Concebido para solucionar los desafíos existentes en el despliegue de modelos de aprendizaje automático en producción, el proyecto arroja resultados sustanciales y transformadores. Inspirado en las observaciones de Deborah Leff sobre la alarmante tasa de fracaso de los proyectos de ciencia de datos en llegar a producción, los objetivos generales se diseñan meticulosamente para enfrentar estos problemas y mejorar la tasa de éxito de los despliegues de aprendizaje automático.

A lo largo del ciclo de vida del proyecto, se lleva a cabo un análisis de requisitos meticuloso, resultando en una arquitectura personalizada que no solo satisface las necesidades organizativas, sino que también demuestra escalabilidad, seguridad y cumplimiento. El proceso automatizado de integración y despliegue continuo fomenta la agilidad y confiabilidad en las actualizaciones de modelos, garantizando operaciones eficientes.

La culminación del estudio de caso proporciona evidencia tangible de la eficacia del marco MLOps, ejemplificando un despliegue exitoso de modelos, valiosas lecciones aprendidas y métricas clave de rendimiento. Al abordar los desafíos pasados en el despliegue, integrar medidas de seguridad sólidas y medir el valor empresarial a través de indicadores clave de rendimiento, el proyecto muestra un enfoque integral.

**Palabras clave:** MLOps, aprendizaje automático, entornos de producción, integración continua, entrega continua, gestión de modelos, MLFlow, Grafana, Prometheus, Docker, API, orquestación de flujos.



# Abstract

---

This Master Thesis (TFM) focuses on the evaluation, selection and deployment of MLOps environments, highlighting the analysis, design, implementation and validation of systems for the management of Machine Learning projects. This study compares multiple MLOps platforms and tools available in the market, considering criteria such as workflow automation, model management, scalability and continuous integration. After a thorough analysis, the most suitable option is chosen and implemented in a real case study.

In addition to addressing the initial objectives, this TFM aims to outline the conclusions reached and the objectives achieved. Designed to solve existing challenges in deploying machine learning models in production, the project yields substantial and transformative results. Inspired by Deborah Leff's observations on the alarming failure rate of data science projects in reaching production, the overall objectives are meticulously designed to address these problems and improve the success rate of machine learning deployments.

Throughout the project lifecycle, meticulous requirements analysis is conducted, resulting in a customised architecture that not only meets organisational needs, but also demonstrates scalability, security and compliance. The automated process of continuous integration and deployment fosters agility and reliability in model updates, ensuring efficient operations.

The culmination of the case study provides tangible evidence of the effectiveness of the MLOps framework, exemplifying successful model deployment, valuable lessons learned and key performance metrics. By addressing past deployment challenges, integrating robust security measures and measuring business value through key performance indicators, the project demonstrates a holistic approach.

**Keywords:** MLOps, Machine Learning, production environments, continuous integration, continuous delivery, model management, MLFlow, Grafana, Prometheus, Docker, API, flow orchestration.



# Contents

---

<b>Agradecimientos</b>	<b>VII</b>
<b>Resumen</b>	<b>IX</b>
<b>Abstract</b>	<b>XI</b>
<b>Contents</b>	<b>XIII</b>
<b>List of Figures</b>	<b>XIX</b>
<b>List of Tables</b>	<b>XX</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Context . . . . .	2
1.2 Project goals . . . . .	2
1.3 Structure of this document . . . . .	4
<b>2 State of Art</b>	<b>5</b>
2.1 First software development: structured programming . . . . .	6
2.2 Waterfall methodology . . . . .	6
2.3 Agile Methodologies . . . . .	7
2.4 Foundations of DevOps . . . . .	8
2.5 MLOps . . . . .	9
2.5.1 Definitions and challenges . . . . .	9
2.5.2 Principles and roles of the machine learning . . . . .	11
2.5.3 Life cycle of data . . . . .	13
2.5.4 Maturity levels of machine learning . . . . .	15
<b>3 Components of MLOps</b>	<b>17</b>
3.1 Introduction . . . . .	18

3.2	End-to-end MLOps platforms . . . . .	18
3.2.1	MLFlow . . . . .	20
3.2.2	KubeFlow . . . . .	21
3.2.3	Azure ML . . . . .	22
3.2.4	Vertex AI (Google) . . . . .	23
3.2.5	Amazon SageMaker . . . . .	24
3.2.6	Domino . . . . .	25
3.2.7	H2O MLOps . . . . .	26
3.2.8	Platform selection . . . . .	27
3.3	MLOps Frameworks . . . . .	28
3.3.1	Scikit-learn . . . . .	28
3.3.2	TensorFlow . . . . .	29
3.3.3	Tempo Framework . . . . .	30
3.3.4	PyTorch . . . . .	31
3.3.5	Framework selection for machine learning . . . . .	32
<b>4</b>	<b>Requirement Analysis</b>	<b>35</b>
4.1	Introduction . . . . .	36
4.2	Actors dictionary . . . . .	36
4.3	Use case description . . . . .	37
<b>5</b>	<b>Architecture</b>	<b>43</b>
5.1	Architecture . . . . .	44
5.1.1	Source Code Repository . . . . .	46
5.1.2	CI/CD Component . . . . .	46
5.1.3	Workflow Orchestration Component for ML . . . . .	47
5.1.4	Feature Store System . . . . .	47
5.1.5	Model Training Infrastructure . . . . .	47
5.1.6	Model Registry . . . . .	48
5.1.7	ML Metadata Stores . . . . .	48
5.1.8	Model Serving Component . . . . .	48
5.1.9	Monitoring Component . . . . .	48
5.2	Analysis of tools . . . . .	49
5.2.1	Code version control . . . . .	49

5.2.1.1	Git . . . . .	49
5.2.1.2	Apache Subversion . . . . .	49
5.2.2	Version control of datasets . . . . .	50
5.2.2.1	DVC . . . . .	50
5.2.2.2	Git LFS . . . . .	50
5.2.2.3	MLFlow . . . . .	50
5.2.3	Model deployment tools as CI: Build and package pipeline components . . . . .	50
5.2.3.1	Jenkins . . . . .	51
5.2.4	Model deployment tools as CD . . . . .	51
5.2.4.1	Docker and kubernetes . . . . .	51
5.2.5	Workflow orchestration component for ML . . . . .	52
5.2.6	Recording of artifacts, experiments, metrics and hyperparameters .	52
5.2.7	APIs as Model Serving Component . . . . .	53
5.2.7.1	Flask . . . . .	53
5.2.7.2	FastApi . . . . .	53
5.2.8	Monitoring . . . . .	53
5.2.8.1	Prometheus . . . . .	53
5.2.8.2	Grafana . . . . .	54
5.2.8.3	ELK stack . . . . .	54
5.3	Selection of tools . . . . .	54
5.3.1	GitHub as Source Repository . . . . .	55
5.3.2	MLFlow as Model Registry and version control of the datasets . .	55
5.3.3	PostgreSQL as ML metadata store . . . . .	55
5.3.4	Jenkins for pipeline deployment . . . . .	56
5.3.5	Docker for build and package pipeline components . . . . .	56
5.3.6	Flask for model serving . . . . .	56
5.3.7	FastAPI for prediction service . . . . .	57
5.3.8	Prometheus y Grafana for monitoring . . . . .	57
<b>6</b>	<b>Case study</b>	<b>59</b>
6.1	Introduction of the case study . . . . .	60
6.2	Configuration of the project . . . . .	61
6.2.1	GitHub . . . . .	62

6.2.2	Jenkins . . . . .	62
6.2.3	Machine Learning . . . . .	64
6.2.4	MLFlow server . . . . .	67
6.2.5	Postgres database . . . . .	69
6.2.6	Exporter to Prometheus FlaskAPI . . . . .	69
6.2.7	Prediction FastAPI . . . . .	70
6.2.8	Prometheus . . . . .	71
6.2.9	Grafana . . . . .	72
6.3	Results . . . . .	72
<b>7</b>	<b>Conclusions</b>	<b>83</b>
7.1	Introduction . . . . .	84
7.2	Achieved Goals . . . . .	84
7.3	Conclusion . . . . .	85
<b>A</b>	<b>Code</b>	<b>87</b>
A.1	docker-compose.yaml . . . . .	88
A.2	Dockerfile MLFlow . . . . .	90
A.3	Dockerfile Flask . . . . .	91
A.4	Dockerfile Fast . . . . .	91
A.5	Dockerfile Postgres . . . . .	92
A.6	init-user-db.sql . . . . .	92
A.7	mlflow_exporter.py . . . . .	92
A.8	api.py . . . . .	93
A.9	html . . . . .	96
A.10	train_model.py . . . . .	100
A.11	prometheus.yaml . . . . .	102
A.12	dashboard.json . . . . .	103
A.13	dashboard.yml . . . . .	117
A.14	datasource.yml . . . . .	118
A.15	grafana.ini . . . . .	119
<b>B</b>	<b>Aspectos éticos, económicos, sociales y ambientales</b>	<b>121</b>
B.1	Introducción . . . . .	122



B.2	Descripción de impactos relevantes . . . . .	122
B.3	Análisis detallado de los principales impactos . . . . .	122
B.3.1	Aspectos Éticos . . . . .	122
B.3.2	Aspectos Legales . . . . .	123
B.3.3	Aspectos Económicos . . . . .	123
B.3.4	Aspectos Sociales . . . . .	123
B.3.5	Aspectos Sociales . . . . .	123
B.4	Conclusiones . . . . .	124
<b>C</b>	<b>Presupuesto económico</b>	<b>125</b>
C.1	PRESUPUESTO ECONÓMICO . . . . .	126
	<b>Bibliography</b>	<b>127</b>



# List of Figures

---

2.1	Comparison of waterfall and agile methodologies [31]	8
2.2	MLOps birth [69]	10
2.3	Comparison of DevOps and MLOps [69]	11
2.4	Roles involved in MLOps [67]	13
2.5	Google maturity levels [21]	16
2.6	Microsoft maturity levels [38]	16
3.1	General schema of tempo framework [65]	31
4.1	Use cases diagram	38
5.1	Architecture	44
5.2	MLOps principles for an Architecture [31]	46
5.3	Architecture with the tools	58
6.1	Architecture with the tools	61
6.2	Jenkins Credentials	63
6.3	Correlation Matrix	66
6.4	MFlow experiments	68
6.5	MFlow models table and chart	68
6.6	Jenkins parameters	73
6.7	Jenkins Pipeline	74
6.8	GitHub repository	74
6.9	MFlow experiments	75
6.10	MFlow models table and chart	75
6.11	Prediction	76
6.12	Metrics in the API	77
6.13	Prometheus targets	77

6.14 Prometheus metrics . . . . .	78
6.15 Grafana grafics from Prometheus . . . . .	79
6.16 Grafana grafics from Machine Learning . . . . .	80
C.1 PRESUPUESTO TOTAL . . . . .	126

## List of Tables

---

3.1	Comparing of end-to-end platforms . . . . .	28
4.1	Actors List . . . . .	37
4.2	UC1: MLflow: Registration and Management of Models . . . . .	39
4.3	UC2: APIs for prediction and service . . . . .	40
4.4	UC3: MLFlow Model Registry: Implementation and maintenance . . . . .	40
4.5	UC4: Workflow Orchestration . . . . .	41
4.6	UC5: Docker containers: services orchestrator . . . . .	41
6.1	Description of the features . . . . .	65



# CHAPTER 1

## Introduction

---

*This chapter introduce the context of the project, including a brief overview of all the different parts that are discussed in the project. It also breaks down a series of objectives that are achieved during the realization of the project. In addition, it introduces the structure of the document with an overview of each chapter.*

### 1.1 Context

In recent years, the number of projects that use machine learning has increased exponentially, as has the amount of investment companies are making in this technology. This growth has brought about a number of problems since the first models until now, such as the incorporation of machine learning models into production. Until 2022, up to Deborah Leff, former Chief technology officer (CTO) for data science and Artificial intelligence (AI) at IBM, 87% of data science projects never make it to production [71] and among the 90% of companies that have made some investment in AI, fewer than 2 out of 5 report business gains from AI, improving this number to 3 out of 5 when we include companies that have made significant investments in AI [52]. These are the reasons why only a small percentage of the Machine Learning (ML) projects manage to reach production. It is essential to find out what the problems are since such an extraordinary inversion from the companies should never be wasted.

### 1.2 Project goals

In response to the challenges faced by organizations in the deployment of machine learning models in production, this project focuses on establishing a robust MLOps framework. Leveraging the MLFlow Model Registry for versioning and management, Grafana and Prometheus for real-time monitoring, and Application programming interfaces (APIs) for prediction and model export, the project aims to address the high failure rates of machine learning projects in reaching production.

The project encompasses key goals such as seamless model deployment, versioning and management, real-time monitoring, performance optimization, database integration, and scalable infrastructure. Additionally, it emphasizes the development of Predictive and Export APIs, ensuring user-friendly access to model predictions and enabling secure model sharing.

A critical aspect of the project involves the implementation of Continuous integration/Continuous delivery (CI/CD) pipelines for automated testing and deployment, facilitating agility and reliability in model updates. Documentation and knowledge transfer initiatives ensure a shared understanding of the MLOps pipeline and its components.

By tackling the challenges identified in previous deployments, emphasizing business value measurement and prioritizing security and compliance, the project aims to sig-



nificantly improve the success rate of deploying machine learning models. Ultimately, this initiative seeks to maximize the returns on companies' substantial investments in AI technologies, addressing issues highlighted by industry experts and fostering a culture of successful machine learning integration into production systems.

- **Seamless Model Deployment:** Ensure the smooth deployment of machine learning models from development to production. Implement a robust MLOps pipeline that facilitates the easy transition of models to operational environments.
- **Model Versioning and Management:** Using the Model Registry to version and manage machine learning models. Implement a systematic approach to track changes, dependencies, and performance metrics of models throughout their lifecycle.
- **Monitoring and Alerting:** Integrate applications for real-time monitoring of model performance, resource utilization, and system health. Establish alert mechanisms to respond promptly to deviations from expected behavior in the deployed models.
- **Performance Optimization:** Continuously monitor and analyze model performance metrics to identify opportunities for optimization. Implement strategies for automatic or manual retraining of models based on performance degradation or changing data distributions.
- **Database Integration:** Connect the Machine Learning Operations (MLOps) pipeline to a reliable and scalable database to store and retrieve model-related metadata, configurations, and performance metrics. Ensure data consistency and integrity between the machine learning pipeline and the associated database.
- **Scalable Infrastructure:** Design the MLOps architecture to be scalable, accommodating varying workloads and growing data volumes. Utilize containerization and orchestration technologies for efficient resource allocation and management.
- **Predictive API:** Develop a user-friendly API for model predictions, allowing seamless integration with other applications and systems. Ensure high availability, low latency, and scalability of the prediction API to meet various operational requirements.
- **Continuous Integration and Continuous Deployment (CI/CD):** Establish a CI/CD pipeline to automate the testing, validation, and deployment of machine learning

models. Enable rapid and reliable updates to deployed models in response to changes in the underlying data or business requirements.

### 1.3 Structure of this document

The remaining of this master thesis is structured as follows.

**Chapter 2: State of art** In this chapter, the current landscape of machine learning deployment and operations is explored, providing an overview of industry trends, challenges, and best practices. Insights from existing literature and real-world case studies inform the foundation for the subsequent chapters.

**Chapter 3: Components of MLOps** This chapter delves into the fundamental components of MLOps, detailing the role of MLFlow Model Registry, Grafana, Prometheus, and APIs in creating a comprehensive machine learning operations framework. The functionalities and contributions of each component to the overall system are elucidated.

**Chapter 4: Requirement Analysis** An in-depth analysis of project requirements is conducted in this chapter, encompassing the needs and constraints of the implementation of machine learning models using the identified MLOps components. This phase establishes the foundation for designing a solution that aligns with organizational goals and operational demands.

**Chapter 5: Architecture** The architectural design of the MLOps system is presented in this chapter, incorporating insights from requirement analysis. The chapter outlines the technical specifications, integrations, and interactions between different components, ensuring a scalable, maintainable, and efficient machine learning deployment infrastructure.

**Chapter 6: Case Study** A practical application of the MLOps framework is illustrated through a detailed case study, showcasing the successful deployment of machine learning models using the proposed architecture. Lessons learned, challenges overcome, and performance metrics are highlighted, providing valuable insight for future implementations.

**Chapter 7: Conclusions** The final chapter consolidates the findings, summarizes key takeaways, and offers reflections on the project outcomes. Recommendations for future improvements and considerations for broader industry implications are discussed, concluding the document with a comprehensive overview of the MLOps initiative.

## CHAPTER 2

# State of Art

---

*In order to understand the current situation of the software industry, we must understand the previous steps of the software path.*

## 2.1 First software development: structured programming

The first software developments occurred in the 1940s and 1950s, when the first computer machines came out. Such was the rise of these new technologies that software's developers had to give an extra twist to make them useful. So, they started developing software using Structured Programming.

Structured programming [72] is a trend that was born to simplify the lives of developers. It was not until 1966, when Böhm and Jacopini launched the structured program theorem, which says that any program could be written using just three instructions, that its consolidation began. In 1968, Edsger Dijkstra published a well-known article, *Go To Statement Considered Harmful* [12], claiming the use of this new concept and the banishment of the Goto sentence. Ending the consolidation of the structured programming.

As has been said before, it is based on the three basic structures: sequence, selection, conditional, and iteration. These three basic features made code understanding easier, with a clearer structure, better testing and debugging optimization, and maintenance expenses were reduced [72].

## 2.2 Waterfall methodology

At the beginning of the 1970s the projects were complex enough having a long list of requirements, also needing a lot of documentation, alongside a proper design's planning of the solution. With the aim of solving these new challenges, the waterfall methodology [70] was born.

Based on a correct requirement's definition, due to that they must be unchanged during all the process, it is a linear process of project management. Each step in the procedure cannot begin unless the previous phase is completed, and once completed, it is completed, as waterfall management does not allow you to return to a previous phase. Normally, waterfall methodology varies somewhat depending on the source, but they generally include:

- Requirements gathering and documentation.
- System design
- Implementation

- Testing
- Delivery/Deployment
- Maintenance

## 2.3 Agile Methodologies

Due to the slowness and delays caused by the traditional way of working, which used waterfall methodologies, the software industry designed agile methodologies. The previous projects were based on fixed requirements, not being able to change them once the process started, and the big efforts that were meant to suppose if a change was made drove to not-as-high-as-expected quality projects.

In 2001, the Agile Manifest [7] was created by the principal Chief executive officers (CEOs) of the software industry in Utah . The Agile manifest was based in four keys:

- Iterations and individuals above processes and tools.
- Functional products above exhaustive documentations.
- Partnership above the deal negotiations.
- Change with the problem above and follow the strict plan.
- Delivery/Deployment
- Maintenance

The main advantages of agile methodologies that make them the base of a large number of Development Operations (DevOps) processes are (Fig. 2.1):

- Ease and reduction of process overload: Being able to adapt at every stage of the development process makes it easier to achieve what is expected at each milestone, without extra effort, in the end.
- Better quality of the product: at every iteration at least a minimum functionality is required, but in the end, an improvement from the previous ones. Moreover, the customer is involved at every moment in the development process, able to request changes depending on the market realities.

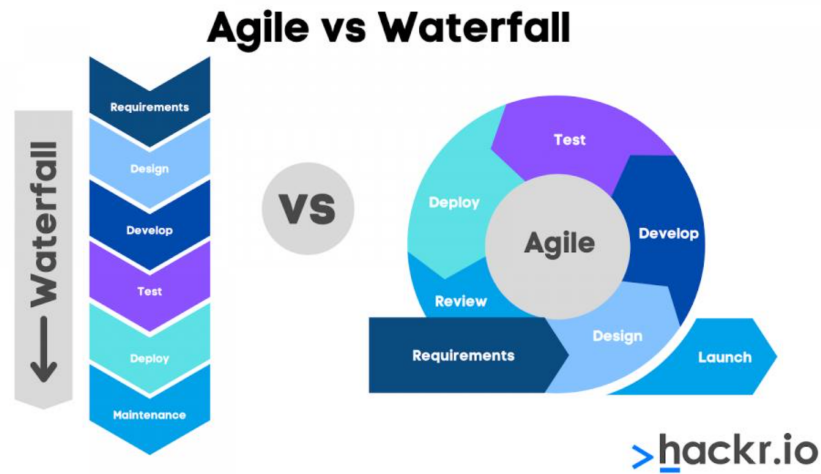


Figure 2.1: Comparison of waterfall and agile methodologies [31]

- Improvement in the foresees through better management of the risk: Agile methodologies were designed to respond to possible changes and problems that arise during the project life cycle.
- The customer is involved in every stage of development, so he could give feedback at every point, creating a stronger relationship between developers and the customer and improving satisfaction.

## 2.4 Foundations of DevOps

Both previous methodologies have the same objectives: to deliver production-ready software products. The problems came with the gap between the development teams and the operations teams, reaching even a point of isolation between them, competing against each other. Both teams had different KPIs, different squad leaders, and different objectives. So, a concept called DevOps emerged in the years 2007-2009 [37] trying to solve all the differences. DevOps is more than a pure methodology and represents a paradigm that addresses social and technical issues in organizations involved in software development [31].

During the product life cycle, DevOps uses agile methodologies by definition. In addition, it is based on the automation of processes; it is managed through continuous

integration, continuous delivery, and continuous deployment (CI/CD). It also allows fast and reliable solutions to be delivered. Moreover, it is designed to facilitate continuous testing and quality assurance, and, thanks to other applications and tools, to monitor, log, and feedback loops. The most important assets of the DevOps methodology are [31]:

- Workflows and repositories, also called source code management: Tools such as GitHub, BitBucket, or GitLab.
- Monitoring and Logging (e.g., Prometheus, Logstash).
- Build process (e.g., Maven).
- Continuous integration (e.g., Jenkins, GitLab CI)
- Deployment automation (e.g., Kubernetes, Docker, Ansible)

Today, the main cloud providers have already built solutions for DevOps tooling, reducing the time needed to start giving value to a project.

## 2.5 MLOps

### 2.5.1 Definitions and challenges

Therefore, summarizing everything said before and in the gross mode, MLOps is the application of DevOps methodologies alongside the use of machine learning in a production environment. In a more technical way, MLOps is the standardization and streamlining of the management of the machine learning life cycle [69]. For most organizations and companies, the machine learning process is relatively new, and the number of projects in productions is not big enough yet to accommodate them to an automation environment, which is where it becomes more critical. For those who have done so, the main challenges they face during the life cycle of the data are as follows.

- Changing environment: data and the business needs shift constantly, is required to be sure that the model in productions aligns with the expectations and if it satisfies the original problem and goals.
- Misleading communication: Despite being in the same company or sharing the same goals, not everyone shares the same tools, procedures, or skills.

With the facts gathered explained before, it was easy to come up with a solution truly related with the DevOps models, MLOps, which is quite similar but not identical since the software development used in DevOps is almost static, almost because some companies can iterate changes during the software life cycle, against continuously changing data of Machine Learning. Machine learning models are always learning and responding to new environments in which they are in, also including both data and code.

So, the final MLOps model was born from the development and operations of DevOps and data engineering (Fig. 2.2).

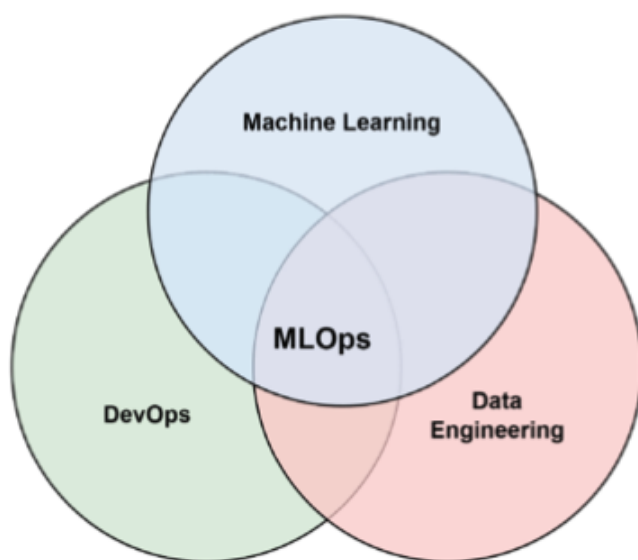


Figure 2.2: MLOps birth [69]

The greatest similarity between MLOps and DevOps methodologies lies in the concepts of CI/CD, which allow software deliveries to occur with a high frequency with reliable results. In most companies that use Machine Learning, they develop the different models and put them into production manually, without incurring MLOps. This also brings with it the problem that machine learning has no return on investment until it can be used. Therefore, all efforts must focus on the steps that follow the development of the model itself, specifically the interfaces between the ML response and the infrastructure where it is implemented [69] (Fig. 2.3).



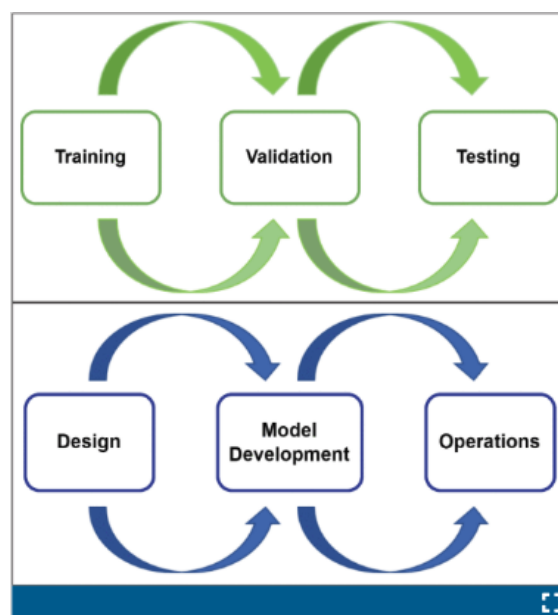


Figure 2.3: Comparison of DevOps and MLOps [69]

### 2.5.2 Principles and roles of the machine learning

As with any methodology, MLOps are also governed by a set of principles. These are not fixed, but any process that aims to end up with a machine learning model in production have to be governed by certain fundamentals such as the following:

- **CI/CD automation:** this process carries out the building, testing, shipping and deployment of software through continuous integration, continuous delivery and continuous deployment. As mentioned above, this allows for early identification of process failures, continuous product value, and increased productivity.
- **Reproducibility:** the ability to replicate the same processes and obtain the same results in different environments for the same inputs.
- **Workflow orchestration:** coordination of the different machine learning tasks in a pipeline according to acyclic graph directives.
- **Versioning:** different versions of data, models, and code allow for greater control over regulatory compliance and audits.
- **Continuous monitoring:** By continuously monitoring the data, the model, the code

or the infrastructure itself, errors can be detected and/or changes can be made according to previous results in a more efficient and less costly way.

- Feedback loops: this type of loop is necessary to manage the relationship between the different stages of quality assessment and the engineering or development processes.
- Continuous retraining of the different models: As each retraining is based on new data, it is a consequence of previous processes of continuous monitoring, feedback loops, and workflow orchestration. One aspect to be managed is the expenditure to be allocated to this stage, as a higher frequency of retraining entails a much higher cost.

Following the roles needed for an MLOps process, these are based on agile methodologies. As in any software development process, a good definition of the participants in the process is fundamental to design, manage, automate, and operate any machine learning system [67] (Fig. 2.4).

- Business stakeholder: in charge of defining the business objectives is also in charge of taking care of the communication between the team and the client.
- Solution architect: define the architecture and technologies that are used.
- Data scientist: converts business problems and requirements into machine learning problems. At a technical level, he/she oversees model engineering.
- Data engineer: designs and implements data pipelines and engineering features.
- Software engineer: converts the ML problem given by the data scientist into a well-engineered product by applying software framework design and best practices.
- DevOps Engineer: strives to link development and management processes under CI/CD methodology, ML workflow orchestration, model deployment to production and monitoring.
- combination of all the above roles, cross-functionally. Manages the ML infrastructure and ML automation flows.

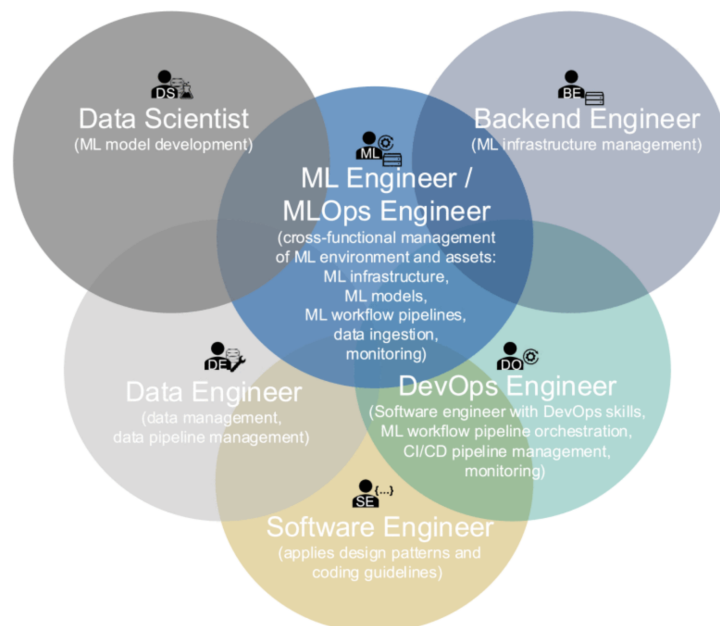


Figure 2.4: Roles involved in MLOps [67]

### 2.5.3 Life cycle of data

In general, all AI projects, whether machine learning or deep learning, share common phases that range from business understanding to model deployment. They also include stages such as data transformation and explanatory analysis. It is worth noting that this process is not linear, but circular or iterative, as once the model is deployed, it needs to be monitored and revised to continuously adapt to the needs of the business. This process is known as Cross Industry Standard Process for Data Mining (CRIPS-DM) [60] and is the most widely used analytical model. Both the Google and Microsoft representations are intended to illustrate that the process is iterative and that at all times it is essential to maintain a clear understanding of the desired outcomes and objectives of each phase in order to carry out the project as efficiently as possible:

1. Understanding the business and the data: The first step before development begins involves understanding the use case, the sector you are working in, and the specific problems you intend to address. It is also essential to know all the details about the data required, including its location, format, and how it can be used effectively. This understanding is essential to focus on the project objectives and

to ensure that you have the data you need to meet those objectives. In addition, knowing the format and location of the data is crucial for its efficient extraction and exploitation.

2. **Data ingestion:** The data ingestion phase involves obtaining the data, and this varies significantly depending on the sources and frequency of data collection. In this phase, as in the previous one, a careful assessment of the amount of data and the type of temporal architecture is vital to avoid problems related to system capacity as the volume of data increases. Accurate volume estimation is crucial, as many data science projects face challenges in moving into production due to underestimating the volume of data or getting too little data compared to the amount generated.
3. **Data preparation and cleaning:** Although this phase is related to the previous ones, it stands out because it is often one of the most complex stages and, in most projects, the most extensive. Data quality is essential for AI models to work properly, so data preparation and data cleaning are essential. Although automation and standardization of this task is often sought, human supervision is preferable, as data quality is crucial for better results in the modeling phase.
4. **Exploratory data analysis:** In this stage, also known as data visualization, clean and prepared data is interpreted and visualized to make informed decisions and understand it in the best possible way. Visualization and representation of the value of the data through reporting tools, graphs, and charts are essential in a data project and allow critical questions to be addressed to advance development.
5. **Modeling and Model Evaluation:** Modeling is considered the core of data analysis. A model takes the prepared data as input and provides the desired data as output. At this stage, important decisions are made, such as the choice of the appropriate model and the selection of hyperparameters to optimize its performance. Performance is measured by various metrics, which vary from model to model, in order to evaluate the model and obtain the best possible result.
6. **Deployment of the model:** Deployment is one of the critical parts of development and is closely related to DevOps. From the first stage of implementation, it is necessary to continuously monitor to detect problems and adapt to changing business

needs. Despite the importance of MLOps techniques in previous stages, at this point they become of significant value, as they allow problems to be detected efficiently, corrected, and new models deployed quickly, achieving an effective system with the best possible results and long-term sustainability.

#### 2.5.4 Madurity levels of machine learning

The level of automation within a MLOps system can be categorized into the corresponding levels. In spite of the fact that there is no accepted maturity model by the community, the two biggest hyperscaler companies, Microsoft and Google, have raised two proposals, one each. The Google model [21] consists of three levels (Fig. 2.5):

- MLOps Level 0: No automation at all; everything is done manually.
- MLOps Level 1: Automation of ML pipelines.
- MLOps level 2: Automation of CI/CD pipelines.machine learning

The Microsofts' model [38] has 5 levels instead (Fig. 2.6):

- Level 0: No MLOps.
- Level 1: DevOps without MLOps.
- Level 2: Automated training.
- Level 3: Automated Model Deployment.
- Level 4: Fully automated MLOps operations.



Figure 2.5: Google maturity levels [21]

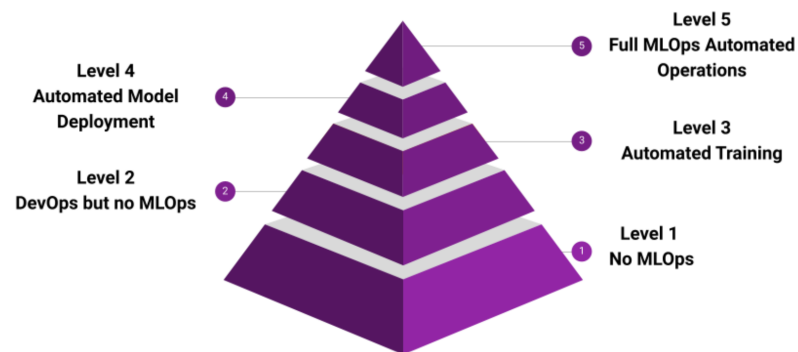


Figure 2.6: Microsoft maturity levels [38]

## Components of MLOps

---

*This chapter offers a brief review of the main technologies that have made this project possible, as well as some of the related published works.*

## 3.1 Introduction

When we talk about Machine Learning Operations (MLOps), we mean a structured orchestration to obtain the results of the model with the least human involvement in the process, but what is strictly necessary. Throughout this chapter, we develop and explain the components that are necessary to carry out this process and that are different from the DevOps process. First, the end-to-end platform is in charge of the orchestration of the different models and their respective metrics. Then, we use the main types of frameworks that are used today when implementing machine learning solutions: we base ourselves on their functionalities, ease of use, and costs when choosing which one is chosen in this process.

## 3.2 End-to-end MLOps platforms

An end-to-end MLOps platform can be defined as a conceptual and technological structure, i.e. a set of tools, processes, and best practices designed to manage and automate the lifecycle of machine learning projects, from model development and training to implementation and maintenance in production environments.

In the next point, different MLOps end-to-end platforms are evaluated; for this, the criteria for selecting the right technology is open / closed source, level of MLOps they reach, where they are deployed (remote or on-premise), and the learning curve required for their correct use. Also, depending on where the technology is going to be applied, there are several factors to take into account that may contribute to its choice. Those factors are more related with being a much more productive business environment, so in this project are not taken into account, but are worth knowing [47]:

- Technological strategy: the choice of the tool has to be made to be consistent with the rest of the technological stack, and it supports the languages and frameworks used during the process.
- Available budget: when evaluating the possible technologies, the costs for a scalable and always available environment must be assessed and budgeted in order to avoid failures in the deployments and that these fall within our objectives.
- Employee knowledge: It is useless to have the best possible tools if no one in our staff knows how to use them; therefore, we must choose a tool about which they



know or whose training does not involve a too exaggerated performance. Training is a very important part of the aforementioned budget.

- Use cases and application trajectory: correctly identifying the problems and requirements to be solved makes the choice of the tool one way or the other.
- User support: in order to offer a correct service to customers, it is necessary that the provider's support be fast and concise, including documentation, tutorials, forums, customer service, etc.

The main features of an MLOps platform are described below [31, 69, 47]:

- Version management: Track changes to code, data, and models.
- Workflow Automation: Automate tasks such as data collection and preprocessing, model training, and production deployment.
- Environment Management: To ensure that development, test, and production environments are consistent.
- Monitoring and logging: Track the performance of models in production and ensure their proper operation.
- Testing and Validation: Verify the quality and performance of the models before deploying them in production.
- Deployment and Orchestration: Implement models in production environments in an efficient and scalable way.
- Model management: Track the version of the models, manage their lifecycle, and enable the update and deployment of new versions.
- Security and compliance: To ensure the security of the data and the model and to comply with regulations and privacy policies.
- Collaboration and communication: To foster collaboration between the data science, development, and operations teams.

Examples of popular end-to-end MLOps platforms include MLFlow [43], Kube-flow [32], Azure ML [39], Vertex AI (Google) [23], Amazon SageMaker [6], Domino [14],

H2O MLOps [25], etc. These platforms provide a structure that facilitates the implementation of MLOps practices in machine learning projects, helping to ensure reliability, scalability, and efficiency in model management in production. Some examples are explained below.

### 3.2.1 MLFlow

MLFlow [43] is an open-source platform developed by Databricks to manage the lifecycle of machine learning projects. It allows developers and data scientists teams to manage experiments, keep track of models, collaborate, and track implementations efficiently. Here are some of the key features of MLFlow [43]:

- **Experiment Management:** MLflow allows users to keep track of machine learning experiments. You can organize and compare different model runs to evaluate their performance and make data-driven decisions.
- **Model Registration:** MLflow allows you to register and version trained models along with their metadata and parameters. This facilitates model reuse and deployment in different environments.
- **Version control:** MLflow provides a version control system for models and experiments, which helps to maintain a history of all changes made to code and models.
- **Model packaging:** Models registered in MLflow can be packaged in standard formats (e.g., in Docker containers) for easy deployment and deployment in different environments, such as the cloud or embedded devices.
- **Parameter and Metric Tracking:** MLflow enables tracking and visualization of metrics, parameters, and artifacts associated with each run of an experiment, making it easier to understand how a model behaves in different configurations.
- **Integration with Machine Learning Libraries:** MLFlow integrates seamlessly with popular machine learning libraries such as scikit-learn, TensorFlow, PyTorch, XGBoost, and others, making it easy to record models and training results.
- **Model Collaboration and Sharing:** Teams can collaborate effectively on machine learning projects and share registered models with other team members.

- **Model Deployment in the Cloud:** MLflow makes it easy to deploy models on popular cloud services such as Azure ML, AWS SageMaker, Google AI Platform and others.
- **APIs for Different Languages:** MLflow offers APIs for Python, R, and REST, providing flexibility in the choice of programming languages to work with MLflow.

It is also important to point out that MLFlow allos any type of coding language, some of them have support and the remaining ones no. For those languages with support, there are extra functionalities such as allowing the development of ML and CI/CD pipelines. It is open source, therefore free, but can be used with paid applications and frameworks.

On the contrary, it is a tool that pretends to be the server to reach the minimum level of automation, however, it is possible to increase this level with several external applications. There is also no record of knowing the accuracy of an algorithm without the help of third parties.

#### 3.2.2 KubeFlow

Kubeflow [32] is an open source platform specifically designed to facilitate the development, deployment, and management of machine learning applications in Kubernetes environments. Kubernetes is a widely used container orchestration system, and Kube-flow extends its capabilities to meet the needs of teams working in machine learning. Here are some of the key features of Kubeflow [32]:

- **Model Orchestration:** Kubeflow enables the orchestration of machine learning workflows, making it easy to create, train, and deploy models in a scalable and reproducible manner.
- **Unified Development Environment:** Provides a unified environment for developers and data scientists, facilitating machine learning project collaboration and resource management.
- **Data Management:** Provides tools for data management, including data access, data preparation, and feature engineering.
- **Experimentation and Model Tracking:** Kubeflow allows one to keep a record of experiments and models, which facilitates the comparison of results and informed decision-making.

- **Model deployment:** Facilitates the deployment of models in production environments through its integration with Kubernetes, enabling scalability and high availability.
- **Elasticity:** Kubeflow leverages the scalability of Kubernetes to adapt resources as needed, allowing it to handle varying machine learning workloads.
- **Extensibility:** Kubeflow's architecture is modular and extensible, which means that additional components and functionality can be added according to the specific needs of the project.
- **Support for Diverse Machine Learning Libraries:** Kubeflow is compatible with a variety of machine learning libraries, such as TensorFlow, PyTorch, scikit-learn, and others, providing flexibility in the choice of technologies.
- **Monitoring and Visualization:** Provides tools to monitor and visualize metrics and results of experiments and models.
- **Web User Interface:** Kubeflow offers an intuitive web user interface that allows users to manage and control workflows, experiments, and models efficiently.

After knowing the main features, we can highlight the following: it is a free and open source code, it is focused on the automation of ML and CI/CD pipelines,, and, in addition, it is based on Kubernetes. The latter, in turn, is a problem, since you have to have a development team with knowledge of containers and Kubernetes, and finally, despite being able to achieve any level of MLOps automation, it does not offer anything by itself.

### 3.2.3 Azure ML

Azure Machine Learning (AzureML) [39] is a cloud service provided by Microsoft Azure that facilitates the development, training, deployment, and management of machine learning models. Here are the top five characteristics of AzureML [39]:

- **Scalable training -** AzureML enables training of machine learning models in a distributed and scalable manner, which speeds up the training process by leveraging cloud resources.

- **Model Lifecycle Automation:** Provides tools to automate the entire model lifecycle, from data preparation and model creation to deployment and continuous monitoring.
- **Integration with Data Science and Development Tools:** Easily integrates with popular data science and development tools, such as Jupyter Notebooks, and provides support for libraries such as TensorFlow and PyTorch, giving data scientists flexibility.
- **Deployment in diverse environments:** Enables models to be deployed in a variety of environments, such as Azure Container Instances (ACI), Azure Kubernetes Service (AKS) and Azure cloud services, making it easy to deploy in different scenarios and environments.
- **Library of pre-trained algorithms and models:** Offers a variety of pre-trained algorithms and models that facilitate rapid development and deployment of machine learning solutions without the need to start from scratch.

Therefore, AzureML is an MLOps level 2 platform with zero code approach, which obviously has global access through Azure and allows one to have on-premise, hybrid, and cloud solutions. However, it is a paid service with scalable functions that make it difficult for companies to calculate the cost of the solution. Furthermore, the platform itself recommends that all components of the system be used without anything external. Finally, as this is a large and complex environment, the learning curve is high.

#### 3.2.4 Vertex AI (Google)

Vertex AI [23] is a comprehensive ML platform from Google that enables both the training and the deployment of ML models and AI applications. In addition, it facilitates the customization of Large language models (LLMs) for integration into your AI-driven applications. Vertex AI fuses data engineering, data science and AI engineering workflows, enabling your teams to collaborate efficiently through a shared toolkit and scale your applications with the advantages offered by Google Cloud.

Key features [23]:

- **End-to-end platform:** Many artificial intelligence solutions seek to provide an end-to-end platform that spans from data preparation to model implementation.

- **Automated Machine Learning (AutoML) Automation:** Automation of the modeling and training process, known as AutoML, is a common feature on many platforms, allowing users without machine learning experience to develop models.
- **Scalability:** AI solutions are often designed to be scalable, which means that they can handle large volumes of data and computational demands, adapting to different project sizes.
- **Integration with Cloud Services:** enables flexible access to and use of computing resources, facilitating the deployment and management of models in distributed environments.
- **Support for Multiple Frameworks:** To provide flexibility, many platforms support multiple deep learning frameworks, such as TensorFlow, PyTorch, or scikit-learn.
- **Model Deployment and Monitoring:** Artificial intelligence solutions often provide tools for deploying models in production environments and monitoring their performance in real-time.

Among the main advantages of its use is level 2, very focused on the life cycle to allow correct deployments, with tools such as Vertex AI Workbench, model training, Vertex AI Feature Store, reusable storage of features, Vertex AI Model Monitoring, automatic alerts of failures in models in production, etc. and finally, total management from the cloud for availability anywhere. On the other hand, as with AzureML, it is a pay-per-use service, with its difficulties in forecasting costs, and the full set of Google tools is necessary.

### 3.2.5 Amazon SageMaker

Amazon SageMaker [6] is an Amazon Web Services (AWS) service designed to simplify the process of building, training, and deploying machine learning models. Therefore, it is a fully managed service that covers the entire machine learning workflow, from data preparation and exploration to the deployment and monitoring of models in production. Here is an overview and some key features [6]:

- **Scalable Model Training:** SageMaker enables you to train machine learning models on a scale, leverage distributed resources and optimize performance during training.

- **Easy deployment:** Makes it easy to deploy models in production environments with just a few clicks or lines of code, streamlining the deployment of models in applications.
- **Automatic Training (AutoML):** Includes AutoML capabilities to simplify the model-building process, which is especially useful for users without machine learning experience.
- **Model and Version Management:** Provides tools for efficient model management, including the ability to create, maintain, and update different versions of models.
- **Integrated Development Environment:** Provides an integrated development environment that includes preconfigured Jupyter notebooks, facilitating data exploration, model development, and team collaboration.
- **Scalability and Flexibility:** Allows scaling vertically and horizontally according to user needs and supports a variety of machine learning algorithms and popular frameworks such as TensorFlow and PyTorch.
- **Deployment Automation and Continuous Monitoring:** Facilitates the automation of the deployment process and provides tools for continuous performance monitoring of models in production, with the ability to adjust them as needed.

In the same way as the two previous ones, since they are hypercaller platforms, we are faced with a complete development ecosystem with a zero code approach designed to accelerate the beginning of development and with global access managed by AWS. It is a level 2 environment of MLOps. However, it is a paid service with a high learning curve due to all the components that have to be used, recommended by the provider to be used as a whole.

#### 3.2.6 Domino

Domino is a MLOps [14] platform that allows data scientists to develop their models more easily and quickly. It works as a Platform as a Service (PaaS)/Software as a Service (SaaS) pay-as-you-go, custom-priced service. Domino transforms the fragmented structure of data science and inefficient workflows through a unified process that spans the entire lifecycle, from exploring new datasets to publishing and managing models in

production. It is also distinguished by its independence from any cloud provider, facilitating on-premises administration. It consists of three essential modules: the System Registry, the Integrated Model Factory, and the Infrastructure Self-Service Portal [14].

- The System Registry maintains version control of code, data, tools and packages, incorporating targets and linking to Git and Jira for effective model traceability.
- The Integrated Model Factory simplifies the lifecycle management of services, enabling their deployment in diverse architectures such as scalable APIs, applications, AWS Sagemaker and Docker images for CI/CD pipeline generation. It also monitors the health and drift of the models over time.
- The Infrastructure Self-Service Portal manages the tools necessary for the creation and deployment of models, allowing the choice of different computer clusters for training and various development environments. Access to the data is done from a secure environment, guaranteeing security throughout the process.

Domino is a service whose offer is practically the same as that of MLFlow, but paying for its services, consisting of an infrastructure with a level 2 automation level and with the possibility of the company's own servers so as not to depend on hyper callers. Although it can be deployed with any available cloud.

### 3.2.7 H2O MLOps

H2O MLOps [25] is a MLOps developed by H2O.ai. It focuses on optimizing and effectively managing the lifecycle of machine learning models, from development to deployment and monitoring in production. Here are five key characteristics of H2O MLOps [25]:

- **Model Lifecycle Automation:** H2O MLOps automates various stages of the model lifecycle, from experimentation and development to deployment and monitoring in production. This streamlines the process and reduces manual intervention, enabling faster and more efficient delivery of models.
- **Centralized Management:** Provides a centralized platform to manage all aspects related to machine learning models. From model registration and tracking to dependency and version management, H2O MLOps provides a unified environment to facilitate team administration and collaboration.



- **Efficient deployment:** Facilitates the deployment of models in production environments by generating model artifacts and integrating them with continuous deployment (CI/CD) systems. This ensures a smooth transition of models from development to production.
- **Monitoring and Governance:** H2O MLOps includes robust monitoring capabilities to assess model performance in real-time. In addition, it offers governance features to track model changes, manage access, and maintain compliance with standards and regulations.
- **Collaboration and Scalability:** Provides tools to facilitate collaboration between data teams, data scientists, and operations. In addition, the platform is scalable and can adapt to model deployment needs at different scales, from small projects to large-scale enterprise deployments.

In general, H2O MLOps aims to optimize the process of developing and deploying machine learning models, improving efficiency, collaboration, and governance throughout the model lifecycle. Despite being paid, the vast majority of the features are open source, as is its ecosystem; it allows hybrid deployment in the cloud and with various providers, and, finally, it allows user role management.

#### 3.2.8 Platform selection

Based on the above, platforms such as AzureML, Amazon SageMaker, or Vertex AI offer a wider range of possibilities and resources, which are scalable. When it comes to deploying a project for a company, choosing one of these options is one of the most coherent decisions, as it allows models to be deployed in a more controlled manner within the same environment. In addition, the management of incidents and resources, as well as their access, is more specific. On the other hand, qualified personnel with knowledge of these offerings are required, as they are slower to learn and require more effort.

Focusing on this project, the only possible tools are Kubeflow and MLFlow, due to the fact that they are open-source platforms and we do not have any budget. Between these two platforms, Kubeflow is a more specific and less flexible environment than MLFlow, as everything is done through containers and Kubernetes; the latter, on the other hand, allows for any programming language and/or libraries and with a low learning curve. Its adaptation to the code is by means of a series of lines in the Python files, and it also

Platform	Open code	MLOps Level	On-premise/cloud/hybrid	Learning Curve
MLFlow	Yes	0	All	Low
Kubeflow	Yes	0	All	Low
Azure ML	No	2	Cloud/On-premise	High
Vertex AI	No	2	Cloud	High
Amazon SageMaker	No	2	Cloud	High
Domino	No	2	All	Medium
H2O MLOps	No	2	Cloud/Hybrid	Medium

Table 3.1: Comparing of end-to-end platforms

has extensive documentation. For this reason, the platform to be used in this project is MLFlow, with its corresponding configuration to store all the generated data.

MLFlow can be used as a Python library; for correct management, virtual environments is used to avoid conflicts between packages. In addition, to have a replicable architecture, Docker is used to be independent of the operating system on which it runs. When initialized, MLFlow is deployed as a web server, which is offered at 0.0.0.0 through port 80 (standard http services).

### 3.3 MLOps Frameworks

A MLOps framework is a set of tools, practices and processes that facilitate the implementation, management, and automation of workflows related to the ML model lifecycle. It aims to effectively integrate the development, deployment and monitoring of ML models in production environments.

#### 3.3.1 Scikit-learn

Scikit-learn [54] is an open source machine learning library for the Python programming language. It provides simple and efficient tools for predictive analytics and data mining

and is designed to be accessible and usable for novice and expert users alike. Here are five key features of Scikit-learn [54]:

- **Consistent interface:** Scikit-learn provides a consistent and easy-to-use application programming interface (API) that makes it easy to build and evaluate machine learning models. The consistency of the API facilitates experimentation with different algorithms.
- **Wide variety of algorithms:** The library includes a wide variety of supervised and unsupervised learning algorithms, as well as tools for data pre-processing, model selection, and performance evaluation. This allows users to explore and apply different techniques according to their needs.
- **Efficiency and performance:** Scikit-learn is designed to be efficient and scalable. It implements optimized algorithms and uses NumPy [45] and SciPy [55] for numerical calculations, which contributes to its performance.
- **Integration flexibility:** Scikit-learn integrates easily with other popular Python libraries, allowing users to combine it with tools such as [50] for data manipulation and Matplotlib [36] for visualization.
- **Comprehensive documentation:** Scikit-learn has extensive and well-organized documentation, including tutorials, examples, and detailed descriptions of methods and parameters. This makes it easy for new users to learn and implement.

### 3.3.2 TensorFlow

TensorFlow [66] is an open-source library developed by Google for machine learning and deep learning. Some of its main features and key concepts are presented below [66]:

- **Computational Graphs:** TensorFlow represents computations as a directed graph. The nodes of the graph represent mathematical operations, and the edges represent the data (tensors) that flow between operations. This allows efficient optimization and parallelization of computations.
- **Deep Learning:** TensorFlow is primarily used in deep learning, a subarea of machine learning that focuses on deep neural networks to solve complex tasks such as computer vision, natural language processing, and more.

- **High-level API:** TensorFlow offers high-level APIs, such as Keras, that facilitate the creation and training of deep learning models. Keras provides a simple and easy-to-use interface for defining and training neural networks.
- **Flexibility:** TensorFlow is highly flexible and can be used on a variety of platforms and devices, including Central Processing Units (CPUs), Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs), which makes it suitable for machine learning tasks in various environments.
- **Extensive ecosystem:** TensorFlow has an extensive ecosystem of tools and extensions for specific machine learning and deep learning tasks. This includes TensorFlow Serving to deploy models in production, TensorFlow Lite for mobile and embedded devices, and TensorFlow.js for web applications.
- **Auto-differentiation:** TensorFlow enables automatic differentiation, which is crucial for optimizing machine learning models by gradient descent. This means that TensorFlow can automatically calculate gradients to adjust the model parameters.
- **Pre-trained Models:** TensorFlow provides access to pretrained models on large datasets, which can accelerate the development of machine learning applications by enabling transfer of learning.
- **Active community:** TensorFlow has an active developer community and extensive documentation, which facilitates problem-solving and the adoption of new features.
- **Interoperability:** TensorFlow is compatible with several programming languages, including Python, C++, and more, making it accessible to a wide audience of developers.
- **Distribution:** TensorFlow is capable of distributing training and prediction tasks across clusters of machines, which is essential for training models on large datasets.

### 3.3.3 Tempo Framework

Tempo [65] is an open-source Python framework that allows the testing and development of machine learning pipelines, which can be created and tested locally or deployed through Seldon in production. Among the main features of the tempo are [65]:

- Optimization of model packages for a shorter execution time on servers.
- Ease of orchestration of process steps.
- Support for any custom Python component.
- Test locally and deploy in production, with the possibility of following GitOps workflows.

The general outline of the workflow of this framework is as follows (Fig. 3.1):

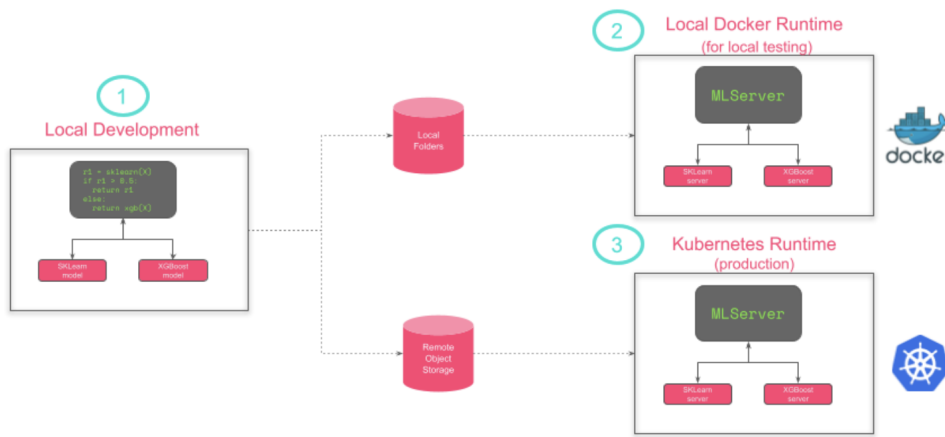


Figure 3.1: General schema of tempo framework [65]

### 3.3.4 PyTorch

PyTorch [33] is an open-source framework for deep learning and machine learning. It was developed by Facebook’s AI Research Lab (FAIR) and has become very popular in the deep learning research and development community. Here is an overview and five main features of PyTorch[33]:

- **Dynamic tensorisation:** One of the most distinctive features of PyTorch is its focus on dynamic tensorisation. Unlike some other frameworks that use static computation graphs, PyTorch allows you to build computation graphs dynamically during program execution. This facilitates experimentation and debugging, as you can change the structure of the computation graph on the fly.

- **Autograd:** PyTorch provides an automatic differentiation system called Autograd. This means that it can perform backpropagation automatically to compute gradients in tensor operations. It makes it easy to implement machine learning algorithms, as you don't have to manually compute gradients.
- **Neural Network Library:** PyTorch includes a rich library for building and training neural networks. The `torch.nn` module provides tools to define layers, loss functions, and complete models in a modular way. This makes it easy to build and experiment with complex neural network architectures.
- **Integration with Numpy:** PyTorch is designed to integrate well with the NumPy library, which is widely used in scientific and numerical computing. The tensors in PyTorch can be easily converted into NumPy matrices and vice versa, facilitating interoperability with other libraries and migration of existing code.
- **Active Community and Full Documentation:** PyTorch has an active community of developers and users. This translates into a wide variety of resources, tutorials, and documentation available online. PyTorch documentation is extensive and well-maintained, making it easy to learn and troubleshoot.

### 3.3.5 Framework selection for machine learning

Once the end-to-end platform, MLFlow, was chosen, a framework had to be chosen that would allow it to be deployed in this environment and perform specific machine learning analyzes. Therefore, scikit-learn is the one that provides the most support and has the best integration with MLFlow. Along with this, frameworks such as pandas, NumPy, and Matplotlib is used for information management and possible visualization of the models from the code, in the development environment. By importing the models and the necessary libraries into the code, and after loading the data and training the model, making the predictions, and evaluating the performance, the whole model is saved in MLFlow as shown in the listing 3.1.

Listing 3.1: Code for saving the model in MLFlow

```
with mlflow.start_run():  
    # Log hyperparameters and metrics
```

```
mlflow.log_param('estimators', 100)
mlflow.log_metric('accuracy', accuracy)
# Log the model
mlflow.sklearn.log_model(model, 'model')
```





## CHAPTER 4

# Requirement Analysis

---

*This chapter carries out an analysis of the main users and actors involved in the entire project.*

## 4.1 Introduction

This sections identifies the use cases of the system. This helps us to obtain a complete specification of the uses of the system, and, therefore, define the complete list of requisites to match. First, we present a list of the actors in the system and a Unified Model Language (UML) diagram that represents all the actors participating in the different use cases. This representation allows us to specify the relationships between them, apart from specifying the actors that interact in the system. These use cases are described in the next sections, including each in a table with their complete specification, and adding a use case diagram to clarify the concepts. Using these tables, we are able to define the requirements to be established.

## 4.2 Actors dictionary

In this section, we resume all the actors involved in the use cases in Tab. 4.1. These use cases are described in detail in the following sections.

Actor identifier	Role	Description
ACT-1	Data Scientist	Is responsible for building, training, and evaluating machine learning models using tools such as MLflow. It registers the trained models in the MLflow Model Registry.
ACT-2	MLOps Engineer	Configures and manages the MLOps system, including integration of MLflow as a model registry, implementation of metrics through Prometheus and workflow orchestration through Jenkins
ACT-3	Systems operator	Oversees the proper functioning of services, including MLflow, Prometheus, and the Flask API for model serving. Intervenes in the event of a problem and ensures system availability and performance.

ACT-4	Prediction User	Inputs data to the API and receives the predictions generated by the model. It can be an end user or a system that consumes the predictions to perform specific actions.
ACT-5	Jenkins Pipeline Manager	Configures and maintains the Jenkins pipeline that is responsible for orchestrating the deployment and update of services through Docker containers.
ACT-6	Model serving API	Service in charge of managing the data entered by the prediction user to provide a prediction.
ACT-7	MLFlow	Central Service in charge of registering the different models and providing the metrics through the API to Prometheus.
ACT-8	Data taker API	Service in charge of porting the data from the MLFlow server to the Prometheus server with the correct format.
ACT-9	Prometheus	Server that receive the data to be able to keep track of the statistics.
ACT-10	Jenkins	Orchestrator for the deployment of all elements in the project.
ACT-11	Docker	Service used for constructing and packaging all the pipeline components.

Table 4.1: Actors List

### 4.3 Use case description

The use cases of the actors identified previously are shown in Fig. 4.1., which are detailed below.

Each of these requirements is of equal importance, being fundamental elements with-

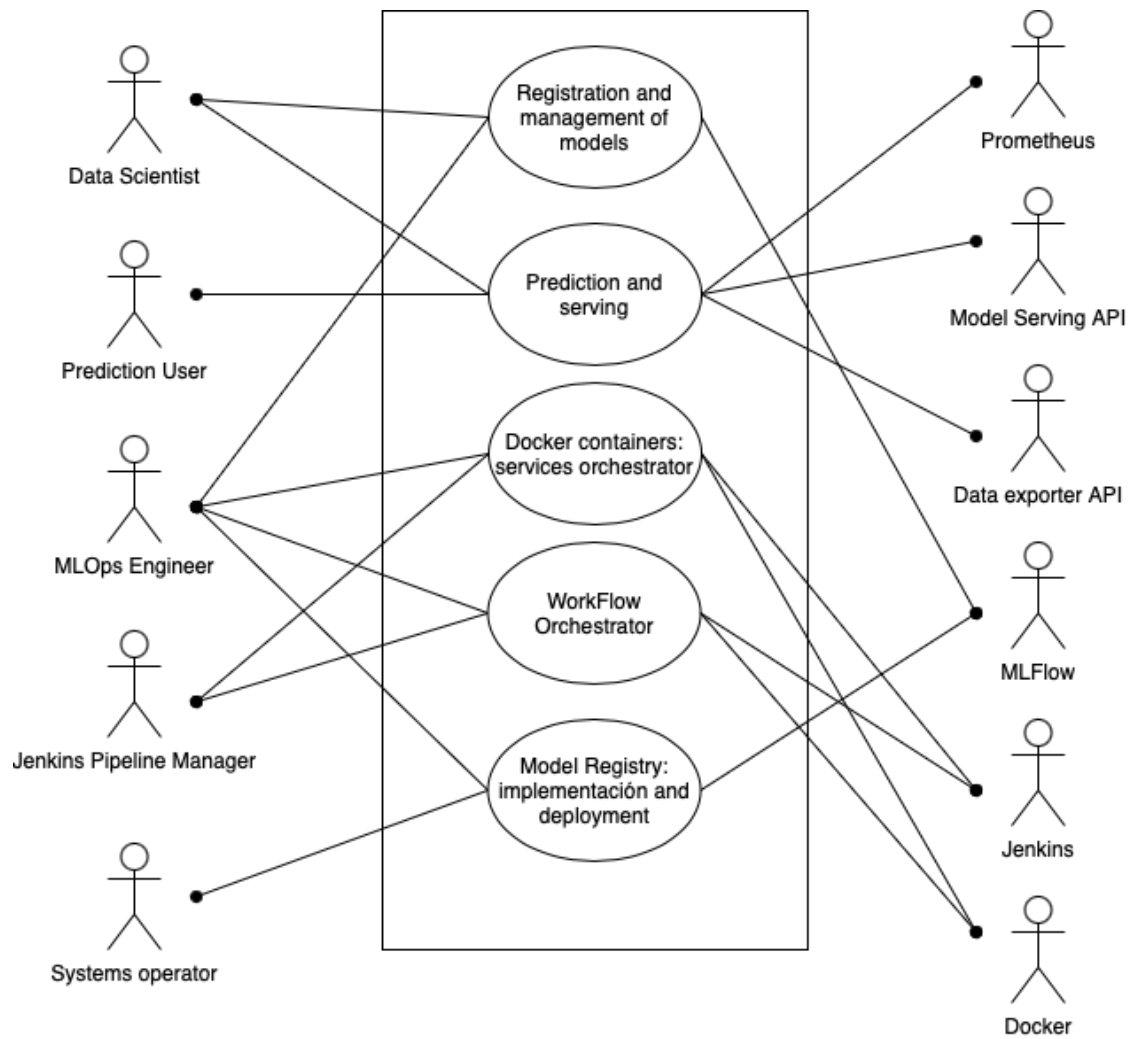


Figure 4.1: Use cases diagram

<b>Use Case Name</b>	Registration and Management of Models
<b>Use Case ID</b>	UC1
<b>Actors</b>	Data scientist, MLOps engineer, MLFlow
<b>Pre-Conditions</b>	The MLFlow server must be linked to GitHub to receive the models.
<b>Post-conditions</b>	-
<b>Flow of events</b>	The data scientist uses MLflow to register and manage models. The MLOps engineer configures and administers the MLFlow platform.

Table 4.2: UC1: MLflow: Registration and Management of Models

out which the comprehensive completion of the various functionalities of the project would be hindered. It is imperative to stress that the absence of any one of these requirements could compromise the viability and overall effectiveness of the initiative.

Each use case is described in the tables described in the following (Table 4.2, Table 4.3, Table 4.4, Table 4.5, and Table 4.6).

<b>Use Case Name</b>	API for prediction and serving
<b>Use Case ID</b>	UC2
<b>Actors</b>	Data Scientist, Prediction User, Model Serving API, Data Exporter API, Prometheus
<b>Pre-Conditions</b>	The model has to be already trained
<b>Post-conditions</b>	-
<b>Flow of events</b>	The model serving API offers the metrics to third parts, such as Prometheus. Data scientists and prediction users use the prediction API interface to obtain predictions.

Table 4.3: UC2: APIs for prediction and service

<b>Use Case Name</b>	MLFlow Model Registry: Implementation and maintenance
<b>Use Case ID</b>	UC3
<b>Actors</b>	MLOps engineer, Systems operator, MLFlow
<b>Pre-Conditions</b>	The models have to be already trained. The flow of data between Grafana and Prometheus has already been established.
<b>Post-conditions</b>	-
<b>Flow of events</b>	The MLOps engineer uses the MLFlow Model Registry to manage model versions and deploy specific models to production environments. The system operator monitors the implementation.

Table 4.4: UC3: MLFlow Model Registry: Implementation and maintenance

Use Case Name	Workflow Orchestration
Use Case ID	UC4
Actors	Jenkins pipeline manager, MLOps engineer, Jenkins, Docker
Pre-Condition	-
Post-condition	-
Flow of events	The Jenkins Pipeline Manager configures and maintains a Jenkins pipeline that automates the deployment and update of services using Docker Compose. The MLOps engineer assists with configuration and monitoring.

Table 4.5: UC4: Workflow Orchestration

Use Case Name	Docker containers: services orchestrator
Use Case ID	UC5
Actors	Jenkins pipeline manager, MLOps engineer, Jenkins, Docker
Pre-Condition	-
Post-condition	-
Flow of events	The Jenkins Pipeline Manager uses Docker Compose to orchestrate the execution of containers containing services such as MLFlow, Prometheus, and the Flask APIs. The MLOps engineer verifies the consistency and replication of the environment.

Table 4.6: UC5: Docker containers: services orchestrator





## CHAPTER 5

# Architecture

---

*This chapter presents the architecture used in this work along with all its components, their connections, its possible technologies, and finally a selection of the technologies used throughout the process.*

## 5.1 Architecture

In this section, the general architecture of the project is described (Fig. 5.1). To do so, all parts involved are described one by one, and then the necessary technologies are selected.

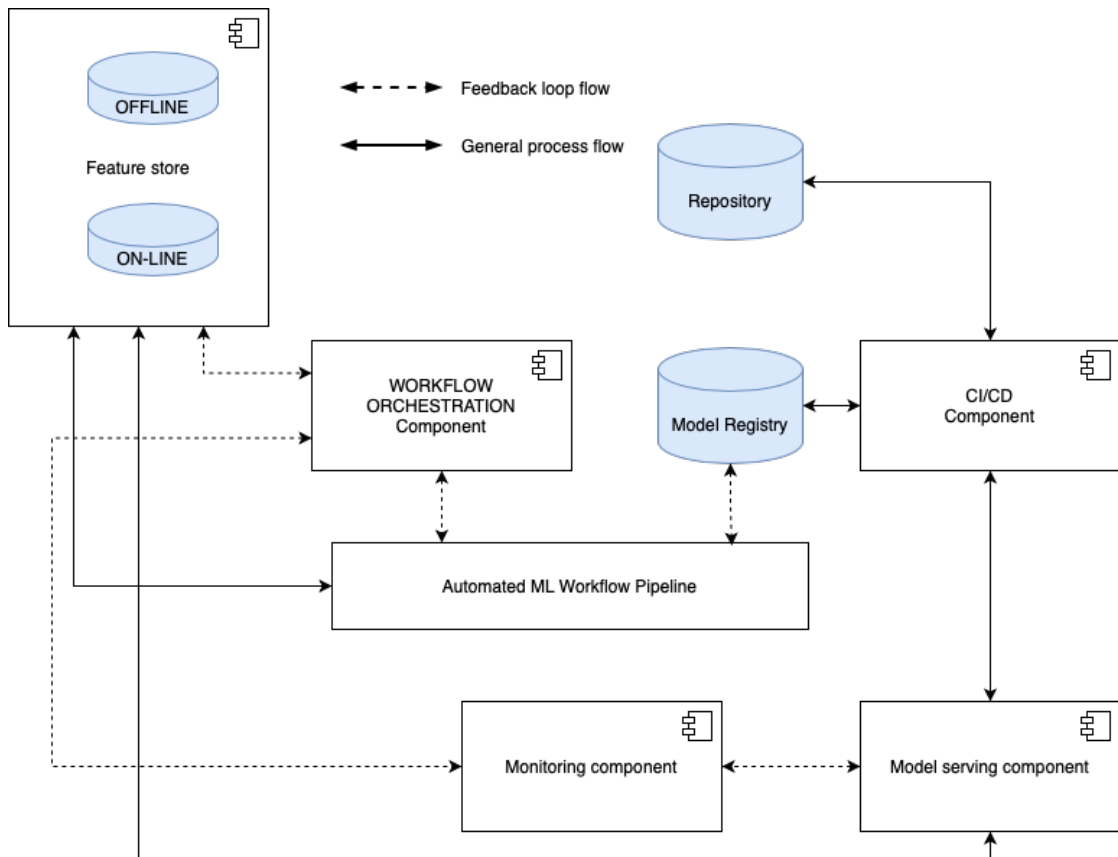


Figure 5.1: Architecture

The main premise when implementing an architecture has been, first of all, the possibility of carrying it out without incurring paid applications, that the whole process could be carried out free of charge, without any subscription. This allows the use of any type of technology as long as it allows integration with the rest of the elements of the system. This freedom facilitates the deployment of the application, although it makes it difficult to use, as you have to know and understand how to use each of the technologies involved. Despite these obstacles, a solution to the proposed problem is implemented,

although with a series of complications or extra aspects, such as the connections between the different elements of the architecture or the flow of data between its structures, which can lead us to use other types of tools, whose use is more integrated throughout the development. The alternatives would have been hypercallers, such as AWS with Amazon SageMaker and Google with Google ML, where the entire process is carried out under the umbrella of a single company, with its support and orchestration already pre-established for greater ease of use and deployment. On the other hand, they tend to be more limited in the technologies with which they can be used and communicate, and, above all, the learning curve and knowledge are higher to be able to use them correctly, without forgetting, of course, that they are paid tools whose method of payment is pay-per-use.

The following explains how this architecture works. First, each team deposits its code and model in the repository. The repository is organized by folders and branches for better information management. With the possibility to set up a trigger with the push to git as a trigger for the start of the project, Jenkins is triggered to build the entire containerized infrastructure via Docker. This infrastructure includes the model training, the Model Registry, the prediction service, and the monitoring applications. First, the model is trained on a container, and all its information is registered in the Model Registry. The model registry allows all the information to be registered and served to the other components that need it. The model serving component provides an interface for real-time predictions, thanks to the CI/CD component. The necessary data are served by an API after they have been collected from the Model Registry. With the same operation, the data is served to the different monitoring components. Finally, all metadata related to model training is deposited in the feature store to enable certain features in future stages of training.

Explaining how the architecture was made, we have to explain that it is based on the principles of the open code CI / CD model, the roles involved in the activity, and the general architecture models given for DevOps methodologies. Therefore, by combining all these premises, we obtain a first outline with the implementation of MLOps principles(Fig. 5.2):

The following describes each of the elements first, explaining their characteristics and their function within the process. We continue with an analysis of the different alternatives currently on the market for each of the elements previously described: char-

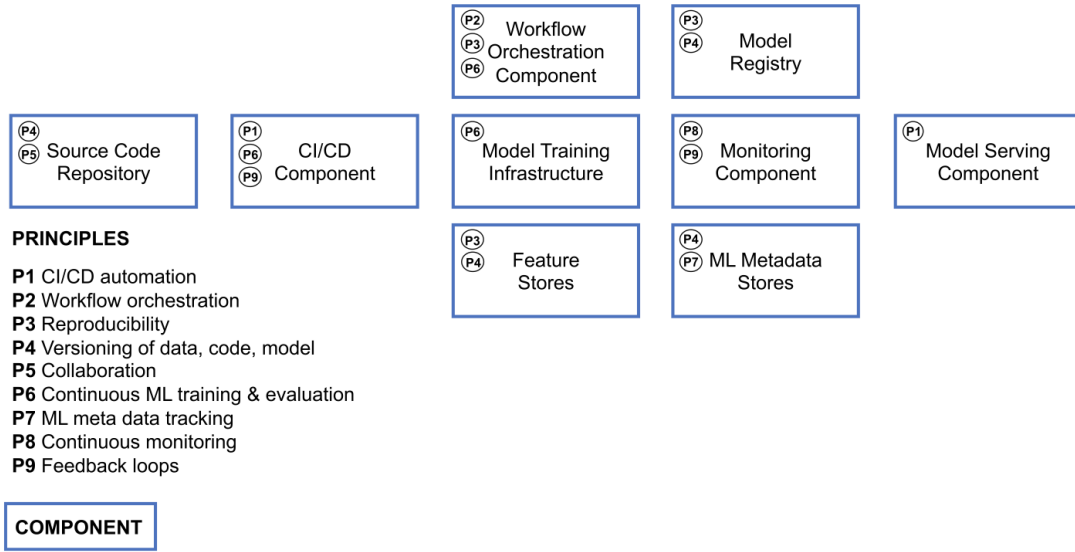


Figure 5.2: MLOps principles for an Architecture [31]

acteristics, advantages, and disadvantages. Finally, the most suitable technologies for each part are selected based mainly on their integration with the other tools, their free availability and ease of use.

### 5.1.1 Source Code Repository

The training and application code is versioned in a repository [34, 9]. This allows developers to merge their code and commit it to it. From the repository, this code will be served to the various CI/CD components to build the infrastructure. Some of the options available on the market are Bitbucket [2], GitLab [20], GitHub [17] and Gitea [10].

### 5.1.2 CI/CD Component

The CI/CD component [59, 44] ensures continuous integration, continuous delivery and continuous deployment. Such tools implement the construction, testing, deployment, and servicing of the infrastructure. Examples are Jenkins [3] or GitHub actions [19], which provide quick feedback from the customer on the successes and failures of deployments. This process has to be done dynamically and independently.

### 5.1.3 Workflow Orchestration Component for ML

Workflow Orchestration Component for ML allows the organization of the different machine learning tasks by means of Directed acyclic graphs (DAG). The graphs graphically show the execution steps and the different artifacts used in the process. The steps that can be performed by this orchestration are data extraction, training, and embedding the model in an application. These same steps could be performed by CI/CD tools, but due to the complexity of the models and the meticulous treatment required by the data, specific tools have been developed. Examples of such tools are: Apache AirFlow [61], KubeFlow Pipelines [5], AWS SageMaker Pipelines [57] and Azure Pipelines [40]. Within our project, there is no need for such tools, as the model is extremely simple and learning does not require large steps or packages.

### 5.1.4 Feature Store System

A Feature storage system [30] allows for centralized storage of features. Typically, they usually have two types of storage within them: the first is an offline database with normal response for experimentation, and a second database for running projects where latency is low, allowing a faster flow of work. These types of warehouse can be scalable or not, depending on the size of the project in question. For scalable projects, it is normal to have them in the cloud. Some alternatives for the feature store system are: Amazon AWS Feature Store [56], Google Feast [22] and Tecton.ai [64]. In our project, a scalable system is not necessary, so a simple database can do the job.

### 5.1.5 Model Training Infrastructure

The model training infrastructure provides the resources that allow the model to be trained and delivered. This type of infrastructure in production environments tends to be distributed and scalable, allowing greater adaptability to resource needs, as well as greater isolation from potential problems. The most widely used distributed and scalable platforms include Kubernetes [16] and Red Hat Openshift [26]. In our project, due to the complexity it requires, it has the possibility of being scalable, but it is not distributed since the computation node is the computer where it is executed.

### 5.1.6 Model Registry

As explained in Chapter 3, a model registry is a centralized repository of trained ML models together with their metadata. Its main tasks are very simple but necessary: to store the ML artifact and to store the metadata. Some of the most commonly used examples of a model registry are MLFlow [43], Kubeflow [32], AWS SageMaker Model Registry [58], Microsoft Azure ML Model Registry [42] or Neptune.ai [47].

### 5.1.7 ML Metadata Stores

A ML metadata store [51] is a system or database designed to store metadata related to the development and execution of machine learning models. These metadata may include information on the data sets used, trained models, hyperparameters, performance metrics, model versions, and other relevant details. Furthermore, it allows the tracking of different types of metadata for each task within the ML.

### 5.1.8 Model Serving Component

Model Serving refers to the ability of a system to serve predictions based on machine learning models. It involves the deployment of models in production environments where they can receive real-time queries and generate predictions efficiently. The most recommended model is scalable and distributed, and there are two main examples for this component:

- Use of Kubernetes for scalability and Docker [13] technology to containerise the ML model, so you can leverage a Python web application like Flask [49] with an API to serve.
- Use of Kubernetes for scalability and KServing from KubeFlow with TensorFlow Serving [66].

### 5.1.9 Monitoring Component

The monitoring component is responsible for monitoring the performance of the different components of the architecture at all times, as well as the different metrics of the model. Common examples are Prometheus [4], Grafana [62], and the ELK stack [8]. Within our project, the objective is to monitor the metrics and use of the query infrastructure. In

this way, consumption can be monitored for scalability and, above all, the objective of this project, machine learning, can be controlled.

## 5.2 Analysis of tools

After learning how each service works within the process, the possible technologies that can be used in each case are explained, detailing the main characteristics and comparing them with each other, to finally choose the ideal, or at least the one that best suits our project, at the next point. The pre-selection of these tools is mainly due to their ease of use, as they are already known to me, and the amount of information and documentation found on the subject.

### 5.2.1 Code version control

Using version control tools, development teams can track changes in source code, allowing them to manage and organize the deployment of applications.

#### 5.2.1.1 Git

Git [17] is currently the most widely used version control system in the world, both professionally and educationally. It is open source, free, cross-platform and allows nonlinear code tracking. It has a desktop application that graphically displays all the changes that have occurred in the code. It is a distributed tool Peer-To-Peer (P2P), whose queries and management are quite simple, both through the command line and the graphical interface.

#### 5.2.1.2 Apache Subversion

Apache Subversion (SVN) [63] is the most widely used code management system after git. It was created to solve all the errors generated by Secure Verification Code (SVC). SVN supports atomic operations to prevent or at least prevent file corruption. In addition, it supports empty dictionaries, and branch operations are more affordable than CVS. Compared to its rival Git, it has better support on Windows, but does not have as many commands for repository management, and is slower. It has plugins for almost all Agile tools and Agile tools.

### 5.2.2 Version control of datasets

Dataset version control allows development teams to track the changes made and to manage and act more effectively in the implementation of ML, base

#### 5.2.2.1 DVC

Data Version Code (DVC) [29] is a tool that allows both dataset versions and the ML model to be managed. Among its main features is its ability to interpret any language, both in the datasets and in the ML models. DVC is command-driven, like Git, so it is quite easy to use for those familiar with git. The files that DVC generates are stored locally, along with the datasets and models.

#### 5.2.2.2 Git LFS

Git Large File Storage (LFS) [18] allows developers to store large binary files in conjunction with a Git repository. Unlike DVC, LFS only allows storing and managing versions of data and code, while DVC also allows tracking, so when it comes to tracking changes or experiments in ML it is rather inefficient.

#### 5.2.2.3 MLFlow

Another possibility we have is to use one of the functions of the model registry, such as version control of datasets. This allows us to combine more functions in the same tool and reduce the failure of elements, as well as using innate functions within tools such as MLFlow [43]. MLFlow provides mechanisms to log and track the version of the dataset along with code and parameters, seamlessly integrating with existing version control systems.

### 5.2.3 Model deployment tools as CI: Build and package pipeline components

For correct management of dependencies in a homogeneous environment, deployments must be orchestrated by containerization software. This control allows dependencies to be controlled when using certain libraries in the correct functioning of the models. Pipeline orchestration tools allow for the execution of the necessary processes in the



process in a sequential and orderly manner, synchronizing the different tools. With this in mind, only Jenkins has been evaluated in this project.

#### **5.2.3.1 Jenkins**

Jenkins [3] is a Java-written code tool that allows developers and architects to perform CI tasks with DevOps methodology in an automated way. In our project, it allows the sequence of tasks of packaging, model training, container deployment, and application deployment to be performed. Deployments are usually done through pipelines written in Groovy language. These pipelines mark the different steps, or stages, to be performed, allowing errors to be located at any stage. They have a very large community that facilitates the resolution of problems. They can be connected to Git to be able to read code and obtain the different files necessary for the process, and to Docker to allow the deployment of containers. Later, it is specified how both configurations have been made.

### **5.2.4 Model deployment tools as CD**

Continuous Delivery (CD) tools are critical in the software development lifecycle to automate the delivery of applications efficiently and reliably. These tools enable development and operations teams to deploy code changes quickly and repeatedly in production environments, reducing the time between writing code and making it available to end users.

#### **5.2.4.1 Docker and kubernetes**

Docker [13] is a container management framework. Containers are a form of virtualization of the operating system and its dependencies that are very useful for creating, testing, and running applications in different environments regardless of the machine on which they are being mounted, as well as being extremely portable. With this solution, resources are saved, and the scalability of the resources is simplified once they are put into production. In our project, Docker allows the packaging of the model, a correct containment of the dependencies used, as they are created and managed by the selected resources, and, above all, the possibility of replicating the environment exactly in the different work teams. To better manage resources and allow for project scalability, Kubernetes [16] could be used. Kubernetes is, therefore, a container orchestration

framework that allows the management of container resources and the connections between them, with scalability. In our project, it is not needed since the number of calls and requests is not high enough.

### 5.2.5 Workflow orchestration component for ML

The platforms mentioned in section 5.1.3 are for complex projects where the data pipeline has to be managed from a single tool in order to be managed more efficiently and with a high degree of isolation from potential problems. In our project, this is not necessary, as it is a simpler machine learning model. Therefore, the learning deployment is a Docker container from a .py file. Within this file, the pipeline established in our project will be executed: obtaining the dataset, training with the different models, and exporting metrics to the model registry. The priority has never been the model itself, or what is done on it, but how any model can be managed on a full production platform.

### 5.2.6 Recording of artifacts, experiments, metrics and hyperparameters

In this project, MLFlow is deployed as a web server at the localhost location on port 80. During the whole process, it generates files with information about the models; all this information is stored in two different ways, depending on whether the information is stored in a database or not.

First, a Database (db) is used for experiments, metrics, and hyperparameters. The db allows a more controlled concurrent environment than other file systems. MLFlow supports dbs through the SQLAlchemy library, which allows connection to MySQL [48], MariaDB [15], MSSQL [41], SQLite [11] and PostgreSQL [24]. The premises for choosing the database are as follows. It needs few resources to be able to run, and it must be free. As a personal preference, it should have enough documentation and not be difficult to manage. Therefore, the possibilities were MariaDB and PostgreSQL. The chosen one is PostgreSQL. To ensure that the database runs independently in different environments and with the same information and persistence, it is also necessary to include a persistent data volume, where db is stored.

Artifact logging is mostly for distributed systems, so it will be explained but not implemented in our project. It is used for all data generated during the training and construction of the model and cannot be stored in a db. The stores can or cannot be distributed as infrastructure.

### 5.2.7 APIs as Model Serving Component

Once the models are put into production, a communication interface is needed for the models to receive new data and for the model to launch its predictions. For this purpose, an API is used, which works as an intermediary between two independent applications that need to communicate with each other. This type of interface is also used to communicate the central model register with the monitoring applications, so it will also be used for this purpose in our project.

#### 5.2.7.1 Flask

Flask [49] is a library whose components are packaged and minimalist in nature. Its code is simple, but does not allow for synchronization. Among its virtues is that it does not automatically generate code or files with its operation. To be used, it is implemented within a Python code. Unlike FastAPI, it is easier to render html files for the prediction page, rather than just an empty page with cubes to fill.

#### 5.2.7.2 FastApi

FastApi [68] works similarly to Flask, also within Python code in our project. Like Flask, it is asynchronous in nature, but it does generate documentation while running in Swagger. It is faster than Flask and separates the server code from the business code.

### 5.2.8 Monitoring

Once all the metrics calculated from the trained models are generated and served, it is necessary to control our infrastructure in order to generate alarms when the uses are erroneous or the workload is excessive. Therefore, first of all, a tool is used to manage all the metrics generated in the ML. In addition, this tool is intended to be monitored by another tool that allows one to manage its use and access to calculate the workflow and average times of the requests; therefore, the following tools are proposed.

#### 5.2.8.1 Prometheus

Prometheus [4] is an open source monitoring and alerting system designed to record system and application metrics. It uses a time-series-based data model and allows for

efficient querying and visualization of data. In addition, it offers flexible alerting capabilities to notify about problems in real time. Its modular architecture makes it highly scalable and adaptable to different infrastructure environments. It is widely used in container and cloud environments to ensure the health and performance of systems.

### 5.2.8.2 Grafana

Grafana [62] is an open source data visualization platform that integrates seamlessly with Prometheus and other monitoring systems. It allows you to create interactive dashboards and custom graphs to visualize metrics, logs, and trace data in real time. With a wide range of plug-ins and an intuitive interface, Grafana is widely used to build monitoring dashboards and scorecards in a variety of environments, from IT infrastructures to enterprise and Internet of Things (IoT) applications. Its flexibility and ability to connect to multiple data sources make it a powerful tool for real-time data visualization and analysis.

### 5.2.8.3 ELK stack

Elk Stack [8] is an open source toolkit that includes Elasticsearch, Logstash, and Kibana. Elasticsearch is a distributed search and analysis engine to store, search, and analyze large volumes of data in real time. Logstash is a data processor that facilitates the ingestion, transformation, and enrichment of data from multiple sources. Kibana is a visualization platform that enables the creation of interactive dashboards and graphs from the data indexed in ElasticSearch. Together, these components form a comprehensive solution for log management and analysis, as well as for monitoring systems and applications in distributed environments. Elk Stack is widely used for observability of infrastructure, anomaly detection, and real-time reporting.

## 5.3 Selection of tools

After knowing which possible technologies can be used for each part of the process, a tool is selected for each process. With this in mind, the main search criteria are ease of use and learning, that they have a zero cost, and that they can be integrated with each other to achieve the necessary solution.

### 5.3.1 GitHub as Source Repository

The tool ultimately used as the source repository is GitHub [17]. First, it offers a robust version control system, making it easy to track and manage code changes. Additionally, its branching functionality allows working on different features simultaneously, making it possible to simulate development, preproduction, and production environments. GitHub would also allow one to simulate collaboration between developers by providing tools to efficiently review and merge code. Finally, its integration with continuous deployment services and the ability to manage entire projects make GitHub an integral choice for software development, as in our case is its work alongside Jenkins.

### 5.3.2 MLFlow as Model Registry and version control of the datasets

The tool used as a model registry is also the tool used as an end-to-end platform: MLFlow [43]. its advantages include centralization of models in an organized repository. It allows for the tracking and comparison of multiple model versions, facilitating reproducibility and auditing. Furthermore, MLflow is compatible with various machine learning libraries, ensuring flexibility in development. Its intuitive interface and ability to manage metadata, metrics, and model artifacts make it a solid choice for organizing and collaborating on machine learning projects. It also offers integrations with other popular tools, making it easy to deploy and monitor models continuously.

### 5.3.3 PostgreSQL as ML metadata store

PostgreSQL is a robust choice as a metadata store for machine learning for several reasons. First, its relational structure allows for efficient organization of information about models, datasets, and experiments. Additionally, PostgreSQL offers ACID capabilities, ensuring data integrity and consistency. Its support for complex queries facilitates the retrieval of specific metadata, such as hyperparameters and metrics. Scalability and the ability to handle large data sets make PostgreSQL suitable for large-scale ML environments. Finally, the active community and broad support for tools and programming languages make PostgreSQL a versatile and reliable choice for managing metadata in machine learning projects.

### 5.3.4 Jenkins for pipeline deployment

Jenkins [3] is a popular choice for pipeline implementation for several reasons. First, its open-source nature and large community offer flexibility and ongoing support. Additionally, Jenkins provides robust integration with a variety of tools and services, enabling the construction of custom pipelines. Its user-friendly interface makes it easy to configure and visualize workflows. The ability to run automated tasks and manage deployment in diverse environments contributes to process efficiency. Additionally, Jenkins offers scalability options and the ability to integrate with monitoring tools, making it a versatile choice for continuous deployment.

### 5.3.5 Docker for build and package pipeline components

Docker [13] is a popular choice for constructing and packaging pipeline components for several key reasons. First, it provides a consistent execution environment, eliminating dependency issues, and ensuring reproducibility of the build process. Additionally, its ability to encapsulate applications and dependencies in containers facilitates portability, allowing components to run consistently in different environments. Docker also improves efficiency by separating dependencies from the host system, avoiding conflicts. Easy integration with orchestration tools such as Kubernetes facilitates deployment and scalability. Finally, Docker fosters collaboration by standardizing configuration and sharing preconfigured environments easily.

### 5.3.6 Flask for model serving

Flask [49] is a great choice for deployment services for several reasons. First, its simplicity and lightness enable fast and efficient API development for model exposure. The flexibility of Flask facilitates integration with various machine learning frameworks and libraries. In addition, its modular approach and ability to scale vertically make it suitable for small and large implementations. Flask also offers an active community and extensive documentation, simplifying the development and maintenance process. Its ability to handle HTTP requests and its minimalist approach make it an efficient choice for implementing predictive services. Within our project, Flask has been chosen over Fast to serve the model and metrics to the different monitoring elements. The main reason is its ease of use and simplicity in allowing its use with Prometheus.

### 5.3.7 FastAPI for prediction service

Fast [68] has been used as a prediction API, mainly due to the ease of configuration of its main page, as it allows for the management of different html. In addition, the management of the information entered is simpler, as will be seen later in the configuration of the use case.

### 5.3.8 Prometheus y Grafana for monitoring

Recapping, the monitoring tools would be used first to manage all the metrics generated in the ML, and then to manage its use and access to calculate the workflow and average times of requests. Therefore, Prometheus will be used as a query tool due to its ability to efficiently collect data, its flexibility to adapt to various infrastructures, and its powerful query system that allows detailed analysis and proactive alerts on system performance. In addition, its modular architecture and integration with visualization tools such as Grafana make it a powerful option for monitoring and improving application and system performance.

Because of this tight integration, Grafana is ideal for monitoring Prometheus usage and all the metrics from the Machine Learning, due to its ability to create custom visualizations and intuitive dashboards. It allows a clear and detailed representation of the metrics collected, thus facilitating data-driven analysis and decision making.

Therefore, after choosing all the elements of our architecture, the architecture of our particular platform would be:

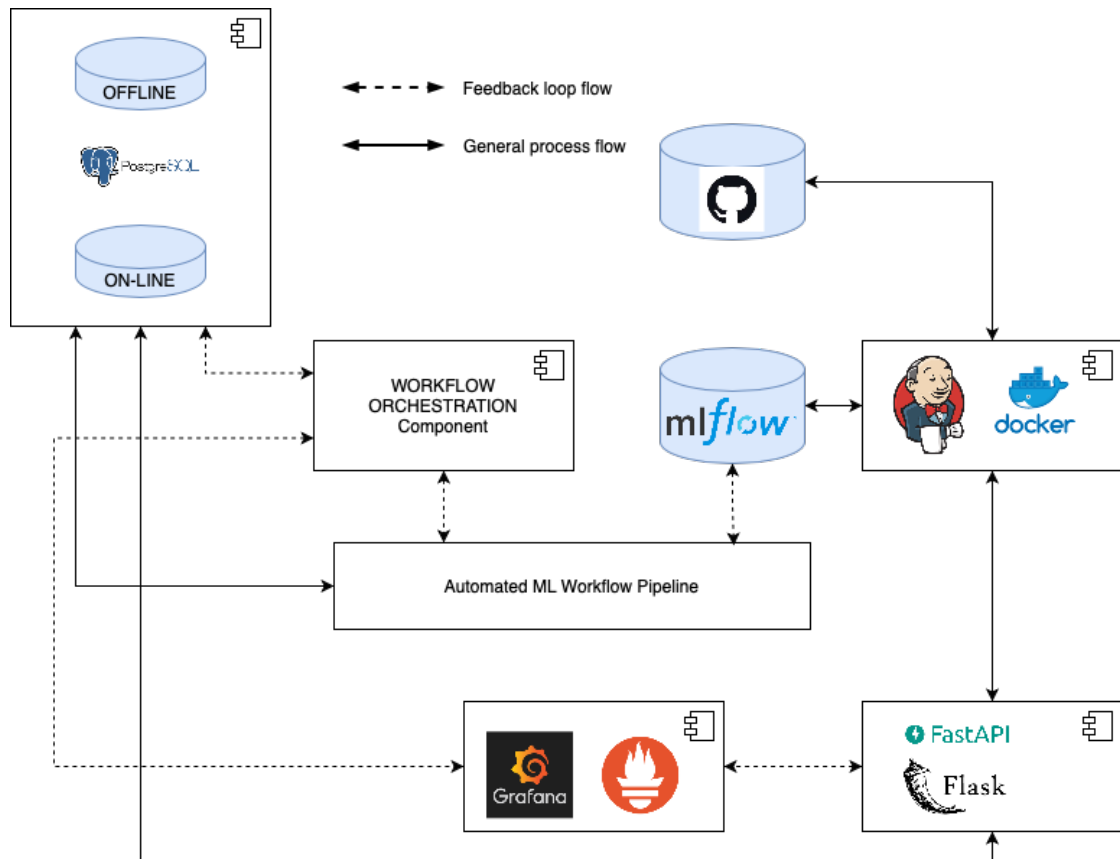


Figure 5.3: Architecture with the tools



# CHAPTER 6

## Case study

---

*This chapter presents a use case for a system with the architecture explained above. It develops and breaks down how it has been configured and created from the beginning for each of the elements involved.*

## 6.1 Introduction of the case study

Throughout this chapter, we break down a practical use case for the architecture mentioned above.

For this purpose, a new scenario is proposed. A Spanish wine shop is responsible for classifying wines from three different wineries. They request the deployment of a model that allows the identification of which wine belongs to which winery. Currently, this process is performed manually, delimiting the different characteristics of each individual wine and classifying them according to average values. To end this tedious and inaccurate process, they have proposed to hire the services of a company to create an infrastructure that allows them to make predictions more quickly and efficiently. The project proposed by the company is an MLOps infrastructure. It is based on all previous records, and correctly characterized, it automatically predicts what kind of wine they are dealing with. They also ask for the possibility of managing which characteristics are most important and which criteria are used for their selection at any given moment. They want to be able to visualize the use of the application graphically; the volume of queries is so high that they need response times to be fast and below certain limits. Their budget is limited, and they consider that the services of a hyperscaler are not necessary to achieve this service, so they only pay for the service itself and its maintenance, not for the tools used.

To do this, we need to know what our objective is and what requirements we have in order to achieve our objectives. The final objective is to achieve a MLOps system that adapts to the needs of a production system in practice: adaptability to different environments, handy in its characteristics, traceability of the different changes made and monitoring of the use of the infrastructure. First, the general requirements are that it must be a free architecture, that it can carry out the complete CI/CD cycle, i.e. that all changes made to the code or configuration of the system are implemented automatically and continuously, that it deploys a machine learning model and that predictions can be made on this model, that changes can be monitored and tracked, and that the use of the different services of the system can be monitored.

For this purpose, a complete solution is implemented which the client uses on his localhost. Depending on its correct functioning, new data is introduced to the dataset, as well as new functionalities, as the wine library requests them.

## 6.2 Configuration of the project

The architecture to be implemented is as follows:

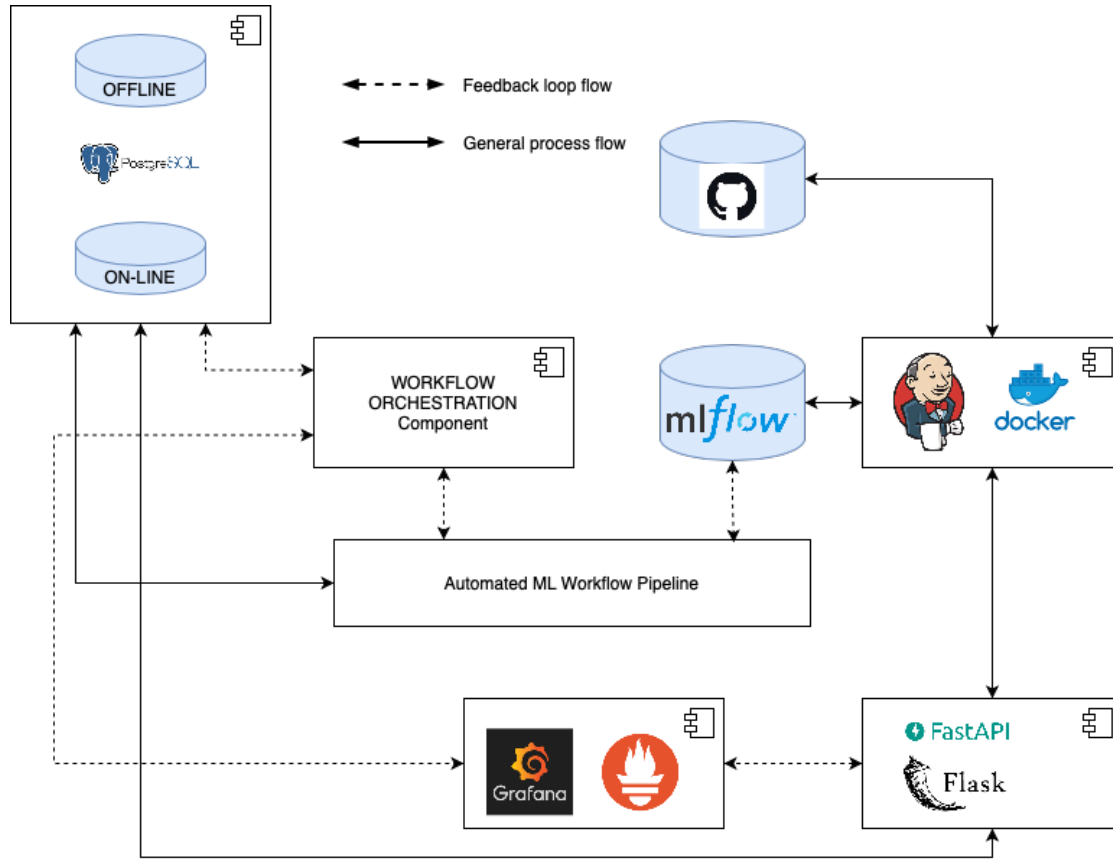


Figure 6.1: Architecture with the tools

So, in this section, we break down how each element of the required system has been configured and created to obtain the expected solution. To do so, we go into detail with the configurations between services and also follow the order in which the solution has been implemented, as this is quite important when it comes to achieving the expected results. In addition to all the sections presented above, two additional servers have been implemented to visualize the different metrics. A Prometheus server for the control and visualization of the metrics obtained from the models and a Grafana server, which allows for the graphic visualization of different diagrams of use and access to the Prometheus server. As presented later, in this use case, it has been decided to use a classification

machine learning model, which has been provided by scikit learn as a toy model.

### 6.2.1 GitHub

The GitHub repository is the first required element. A repository where all the code used throughout the process can be included and is capable of visualizing and tracking the different changes made to it. The source repository is included in `https://github.com/amrg1902/codigoTFM.git`, and from there its code is used by Jenkins, creating the different components within the process. The git structure corresponds practically to the different containers that are going to be implemented, with the docker compose [27] that deploys them *docker-compose.yaml* and with the code of the jenkins pipeline, made in groovy [1], *deployment-container.groovy*.

### 6.2.2 Jenkins

After having the git repository, the Jenkins server was set to localhost following the steps in the guide [46]. After this configuration, the Jenkins server would be at `http://localhost:8080`, starting with the command `brew services start jenkins - lts`. Since this command had to be run every time the computer was started, it was decided to configure it to start the server automatically every time the device was started. As our goal is to create a system with CI/CD methodology, it was necessary to adjust a series of parameters with respect to credentials and download some plugins that made possible the connection and access of Jenkins to GitHub and the connection of Jenkins to Docker and its local daemon. First of all, all the necessary plugins that would allow the configurations of each service to be carried out for GitHub and Docker were downloaded: Docker plugins:

- Docker API
- docker-build-step

Git plugins:

- Git
- Git client
- GitHub API
- GitHub Authentication

- GitHub Branch Source
- GitHub Pipeline for Blue Ocean
- Pipeline: GitHub

From Git, external connections were enabled, and a token was created so that it could be exported to the application in question. Then, in Jenkins, an environment variable called *PATH* was configured to point to the address of the Docker daemon on our device, and 3 access credentials were created: 2 Git access credentials, via ssh and token, and the last one to Docker via token. (Fig. 6.2).

#### Credentials







T	P	Store ↓	Domain	ID	Name
		System	(global)	GitHub_access_SSH	amrg1902 (Jenkins a repositorio TFM)
		System	(global)	68041f7d-8155-43e6-9375-0d8599f19a25	amrg1902/***** (GitHub con token)
		System	(global)	ce19e7ac-fe9e-47b6-aa1d-65aaf111e2d8	Token de acceso docker

Figure 6.2: Jenkins Credentials

With this configuration, adding the Groovy code manually in the pipeline configuration, we were already building Docker images and containers from a Dockerfile [28] on GitHub.

The intention was that the pipeline would run manually or be launched after a GitHub push, but our priority was that the pipeline build code would not have to be entered each time it was run. To solve this problem, *container deployment.groovy* was created. A code in which each stage within the pipeline was declared that would allow us to configure everything in a faster way, and together with the Git functions, changes and versions could be tracked. This file was inside the git repository, and thanks to the previously installed plugins, along with the necessary configuration, the information within the source of these data, the GitHub repository, was configured in the pipeline.

This code would execute three stages plus the one defined by Jenkins when executing code from Jenkins. The three stages implemented are: ***Access to Dockerfile in the workspace***, after copying all the data from Git, this stage checks that the different Dockerfiles are in the locations where they should be. ***Delete previous images and containers in Docker***, which removes all images and containers in the workspace,

making it easier to avoid cross-referencing data from previous versions, since all the data that need to be saved are contained in persistent docker volumes. **Docker Compose** builds the new images and deploys the containers as defined in the *docker-compose.yaml* (see Appendix A.1). This file contains the information necessary to deploy the seven services needed in this project, both APIs, the MLFlow server, Grafana and Prometheus for visualization, the database and the ML training, and the network that interconnects them.

### 6.2.3 Machine Learning

The main container is in charge of data training. To this end, a data set had to be chosen that contained a sufficient amount of data and features because if it were too simple, all the metrics of the training with the different models would always obtain their maximum values, regardless of which model had been trained with. ScikitLearn offers a series of toys datasets [53] to perform all types of machine learning processes, I choose a dataset for the classification of different types of wines, whose features are not scaled and the predictions are made correctly.

This dataset classified three different types of wine: type 1, type 2 and type 3 according to the different characteristics of their features (Tab. 6.1). The dataset itself had already undergone the cleaning process, which would allow working directly with it. In the process, 80% of the total dataset has been taken for training and the remaining 20% for execution.

In addition, the correlation matrix (Fig. 6.3) between the different features was calculated in case there were any that could be eliminated from the process because they were too closely related to each other. Finally, and despite the data obtained, the dataset was kept intact, as the objective of this project was not a machine learning work but how it can be used and deployed in a CI/CD environment, so the machine learning results are independent of this deployment.

When it came to training our models, we created a python file *train\_model.py* (see Appendix A.10) where, once trained, all their data were saved in the MLFlow server, as we see later. The models chosen to be trained, all of them because of their simplicity and because they had already been worked on previously in the master's degree courses, are the following:

- Random Forest Classifier

Feature	Type	Mean	Std	min	MAX
Alcohol	Float 64	13	0.81	11.03	14.83
Malid acid	Float 64	2.83	1.11	0.74	5.80
Ash	Float 64	2.36	0.27	1.36	3.23
Alcalinity of ash	Float 64	19.5	3.33	10.6	30
Magnesium	Float 64	99.74	14.28	70	162
Total Phenols	Float 64	2.29	0.62	0.98	3.88
Flavanoids	Float 64	2.02	0.99	0.34	5.08
Nonflavanoid phenols	Float 64	0.36	0.12	0.13	0.66
Proanthocyanins	Float 64	1.59	0.57	0.41	3.58
Color intensity	Float 64	5.05	2.31	1.28	13
Hue	Float 64	0.95	0.22	0.48	1.71
od280/od315	Float 64	2.61	0.70	1.27	4
Proline	Float 64	746.89	314.90	278	1680

Table 6.1: Description of the features

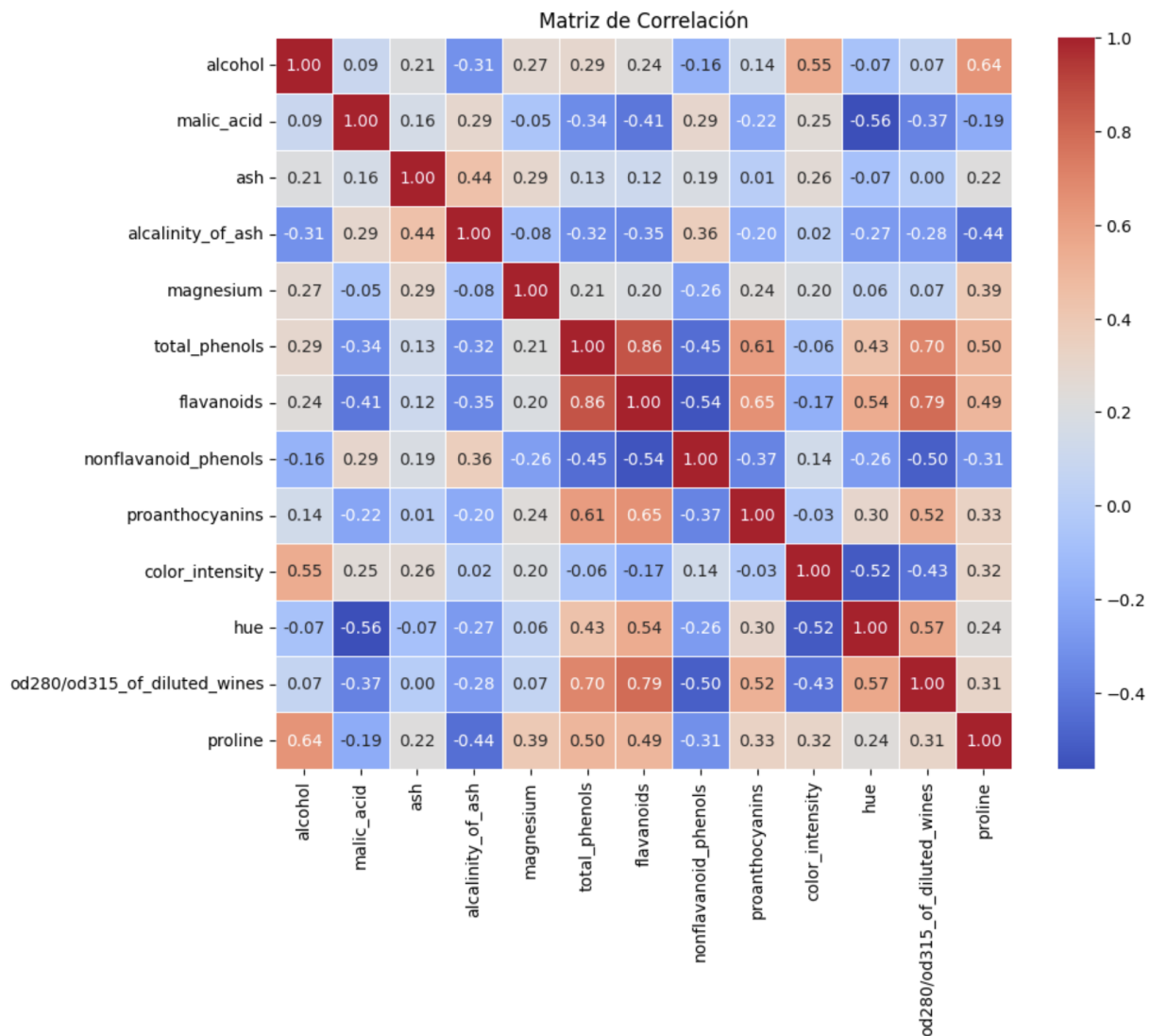


Figure 6.3: Correlation Matrix



- KNeighbours Classifier
- SVC
- Gradient Boosting Classifier

For each of the trainings, a series of metrics have been calculated which are the validators of each of these models, and also be the criteria used to choose between one model or another, in our case, accuracy. The metrics used are the following:

- Accuracy

$$\text{Accuracy} = \frac{\text{Number of correct predictions in } y_{\text{test}}}{\text{Total number of elements in } y_{\text{test}}}$$

- Precision

$$\text{Precision} = \frac{\text{Number of True Positives in } y_{\text{test}}}{\text{Number of True Positives} + \text{Number of False Positives in } y_{\text{test}}}$$

- Recall

$$\text{Recall} = \frac{\text{Number of True Positives in } y_{\text{test}}}{\text{Number of True Positives} + \text{Number of False Negatives in } y_{\text{test}}}$$

- F1

$$F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

#### 6.2.4 MLFlow server

Once the models were trained and the different metrics calculated, all the information had to be exported to a hub, in this case, the MLFlow server. By doing this, we achieve a record and a point from which to export all the information to the different steps of our process. With this intention, a container was deployed through docker-compose from a Dockerfile (see Appendix A.2). This Dockerfile specifies the connection between the database and the server, in the server launch command the database is enabled as a backend save, as well as declaring the necessary variables to expose data to Prometheus. The port selected to be exposed is port 80, so the server would be available at *http://localhost:80*.

With the server up, we could access it. In the upper left-hand side of the server there is a space where all the saved experiments appear, at the beginning of the process



Figure 6.4: MFlow experiments

there is a first experiment "Default" ((Fig. 6.4)). In addition, the main information appear in the centre of the screen, where information pertaining to all models in the experiment, as well as their metrics, can be displayed either in table form or in a much more representative form such as a chart ((Fig. 6.5)).

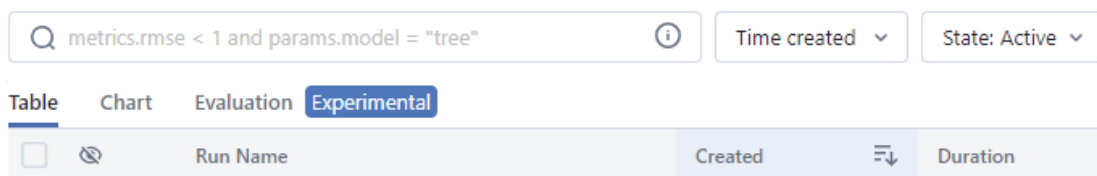


Figure 6.5: MFlow models table and chart

Having already explained where the information would appear, it was necessary to register all the models and save the metrics so that they could be consulted or used in future operations, as in our case. To achieve this, in the code where the different models were trained, *train\_model.py* (see Appendix A.10), the URI of the server address and the name of the experiment carried out had to be defined. This configuration is parameterized so that this information is entered into Jenkins at the time of deployment.

```
mlflow.set_tracking_uri("http://mlflow_container:80")
mlflow.set_experiment("Entrenamiento dataset vino")
```

Subsequently, the data is entered into MLFlow, starting with the metrics and ending with the model in question:

```
with mlflow.start_run(run_name=run_name):
    # Log de métricas
```

```
mlflow.log_metric("Accuracy", accuracy)
mlflow.log_metric("Precision", precision)
mlflow.log_metric("Recall", recall)
mlflow.log_metric("F1", f1)
mlflow.sklearn.log_model(model, model_name)
```

After verifying the server working correctly and with all the information entered in the way is meant to be, the next steps to carry out would be the two APIs, the first one, to export the data to the Prometheus server with the correct format, the second one to raise a FastApi that would allow having a page to be able to carry out the predictions.

The database also had to be configured so that all information obtained during the process could be saved, exported, or retrieved in the event of an error.

### 6.2.5 Postgres database

The database service allows the storage of all data that come from or are hosted in MLFlow. To do this, a database is started on port 5432 using a Dockerfile (see Appendix A.5). The user, the password, and the database in question have been declared in the initialization of the MLFlow server so that it has access to it. Furthermore, by means of the configuration file `textitinit-user-db.sql` (see Appendix A.6) all necessary database permissions are granted to the MLFlow server.

### 6.2.6 Exporter to Prometheus FlaskAPI

The first of the APIs we had to set up is the one that takes data from the MLFlow server, converts and adapts it, and makes it readable on the Prometheus server, which is where it is served.

To do this, a Dockerfile (see Appendix A.3) was configured, which would simply execute a python `mlflowexporter.py` (see Appendix A.7), where all the logic of the API is found, and the working port of the API, 8000, is exposed.

The API works as follows:

- First, you configure the URI where the tracking is done, in this case, the address of localhost and port 80, the MLFlow server.
- Then, we continue to define the location within our localhost port 8000 where the Prometheus server can scrape the metrics correctly, in our project `/metrics`.

- Next, the main function, which is called in *http://localhost:8000/metrics*, performs a search for the experiment in question, and all executions of the experiment are obtained for all the models that have been trained. Iterating on each of the models performed in each of the executions, we obtain all the metrics that have been collected in the MLFlow server. To facilitate the work and to be able to visualize their behavior, all metrics are also printed on the API location. To do that, a simple rendering is used, where all the information provided is configured in such a way that it is readable by the Prometheus server, which is our goal.

```
response = make_response(metricas_prometheus)
response.headers["Content-Type"] = "text/plain"
```

After this configuration, the metrics had to be collected by Prometheus, the configuration of which is explained later.

### 6.2.7 Prediction FastAPI

The second of the APIs was responsible for generating a data collection template, which would be used to make a prediction with our selected model. The reason for using a Fast API instead of the Flask was the ease of rendering and obtaining the information as expected. For its implementation, an identical structure to the Prometheus scraping API was followed, a Dockerfile (see Appendix A.4) was configured, which would simply execute a python *api.py* (see Appendix A.8), where all the logic of the API is found, and the working port of the API, 7654, is exposed.

The structure of the API to perform the given tasks was:

- First, you configure the URI where the tracking is done, in this case, the address of localhost and port 80, the MLFlow server. Also, the name of the current experiment.
- Next, an auxiliary function was generated that returned the exact URI of the experiment in question from its name, as this was necessary for us to be able to make predictions. This was done by selecting the model whose accuracy was the best, that is, closest to one. The selection of this metric as the method of choice is arbitrary and could be any other metric generated.

- Finally, an html (see Appendix A.9) is rendered, in a static way, where all the data to be entered are displayed. Once entered, it makes a prediction based on the data introduced, and, in the experiment selected in the previous auxiliary function, the result would be displayed on the screen once the predictions are done.

### 6.2.8 Prometheus

Prometheus is a tool for visualization of the different metrics obtained during machine learning training. Once the metrics have been collected, they have to be scraped by Prometheus so that all the metrics can be visualized and represented. To do this, from the docker-compose (see Appendix A.1) a Prometheus image is created and enabled on its default port, 9090. This image depends on the MLFlow server container and is connected to the common network that has been enabled. When collecting metrics, the application configuration file, *prometheus.yaml* (see Appendix A.11), is created and configured. In this file, the scraping period is defined, as well as the possibility of defining tags to be able to better search and collect all searches. The different scrape locations are defined in `scrape_configs`:

```
scrape_configs:
  - job_name: 'mlflow'
    static_configs:
      - targets: ['mlflow_exporter:8000']
  - job_name: "prometheus"
    static_configs:
      - targets: ["prometheus:9090"]
```

In our case, it is configured to scrape the API container. With this, we can check all the metrics from Prometheus; in case any of them were wrong or not with the expected values, an update process of the model would be initiated, for example, and the Prometheus container itself. This is due to the subsequent use of Grafana. In our system, Grafana is used as a monitoring tool for the different Prometheus resources, although it could also be used for data related to machine learning, so the Prometheus metrics are scraped.

### 6.2.9 Grafana

In this project, Grafana is used to visualize metrics for the different accesses and uses of the Prometheus server, alongside with the metrics from the training. The concept of the Grafana server configuration is very similar to that of Prometheus. First, it is deployed from the docker-compose (see Appendix A.1) with the Grafana image, allowing port 3000 for its operation. For the correct configuration of Grafana, 4 files must be configured:

- *dashboard.json* (see Appendix A.12): Configuration file for the graphs, where it appears what has to be represented and how. After starting the Grafana server for the first time, it was empty in terms of graphics. For this reason, all the graphs had to be configured that first time, and the configuration file had to be saved so that it could be exported at any time. Grafana creates this file by itself.
- *dashboard.yml* (see Appendix A.13): Configuration file that tells where the information has to be collected in order to create the graphs, i.e. it points to the location of *dashboard.json*.
- *datasource.yml* (see Appendix A.14): Configuration file where the data source is defined, in our case the Prometheus server on port 9000, as well as the type of access, proxy type.
- *grafana.ini* (see Appendix A.15): Configuration file indicating where the Grafana server is to be initialized.

Once the Grafana configuration was complete, the whole solution was made. The solution required by the wine cellar to be able to offer the classification of wines in a quicker and more efficient way. The following section shows the results of the project for our use case.

## 6.3 Results

Once the infrastructure configuration has been described, we proceed to show the results of the whole system working, following the order in which the system works.

First, from Jenkins, we run the pipeline, introducing the parameters to take into consideration in our project (Figs. 6.6 and 6.7), described in the *.groovy* file inside the repos-

itory mentioned above: <https://github.com/amrg1902/codigoTFM.git> (Fig. 6.8). This builds our infrastructure using containers, one for each service.

Esta ejecución requiere parámetros adicionales:

**URI**  
URI a utilizar

**nombre\_experimento**  
Nombre del experimento

 Ejecución  Cancel

Figure 6.6: Jenkins parameters

All containers remain up throughout the process, except for the *model\_training\_container*. This is the container that, as its name suggests, trains the model and loads all the data into the MLFlow server, which, once it has done its job, is shut down. The addresses to visit the locations of the different services are:

- Jenkins localhost:8080
- MLFlow server localhost:80
- MLFlow exporter localhost:8000\metrics
- Prediction Api localhost:7654
- Prometheus localhost:9090
- Grafana localhost:3000

After the process carried out during the training of the different models, always with the server started, all the data we required has been logged in the MLFlow server. Visiting its location in port 80, on the upper left side we find a breakdown of all the experiments carried out, in our case, only the one corresponding to the experiment

Pipeline pipelineTFM

Full project name: TFM/pipelineTFM  
Full process Pipeline.

Stage View

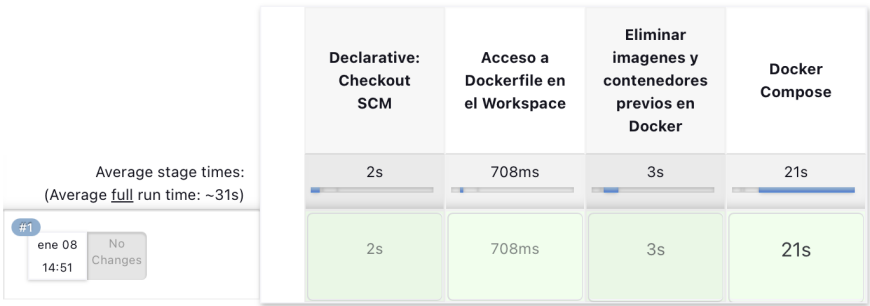


Figure 6.7: Jenkins Pipeline

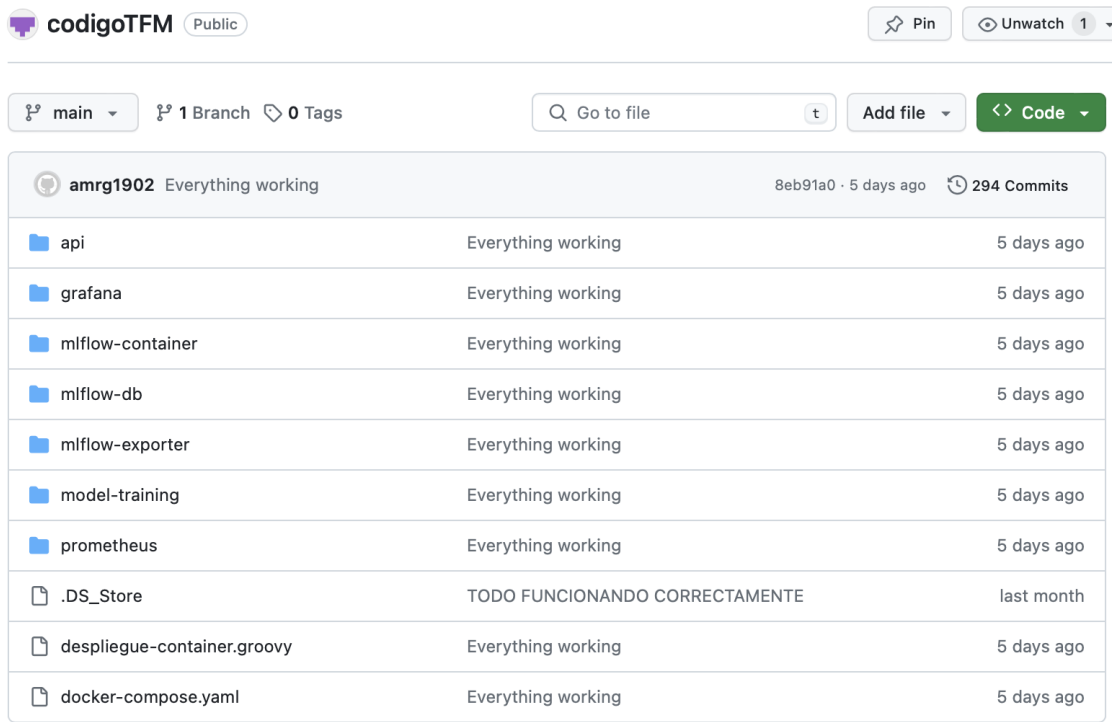


Figure 6.8: GitHub repository





Figure 6.9: MFlow experiments

*Entrenamiento dataset vino* (Fig. 6.9) . But the principal information contained in the main page are the four trained models, with the execution time, as well as the metrics corresponding to each of them (Fig. 6.10). In addition, the same information can be seen in a much more representative way if we select the chart as a representation method.

Table   Chart   Evaluation <b>Experimental</b>								
					Metrics			
<input type="checkbox"/>	<input type="checkbox"/>	Run Name	Created	Duration	Accuracy	F1	Precision	Recall
<input type="checkbox"/>	<input type="checkbox"/>	RandomForestClassifier	58 minutes ago	3.3s	1	1	1	1
<input type="checkbox"/>	<input type="checkbox"/>	GradientBoostingClassifier	57 minutes ago	1.7s	0.9444444...	0.9439974...	0.9462962...	0.9444444...
<input type="checkbox"/>	<input type="checkbox"/>	SVC	57 minutes ago	2.0s	0.8055555...	0.8024265...	0.8010582...	0.8055555...
<input type="checkbox"/>	<input type="checkbox"/>	KNeighborsClassifier	57 minutes ago	2.1s	0.7222222...	0.7222222...	0.7222222...	0.7222222...

Figure 6.10: MFlow models table and chart

With these types of representations, plus those that we wish to configure, we are able to choose the model that best suits the requirements of the process in which we would be involved. In our case, to facilitate the selection process, we choose the model whose accuracy is better, i.e., closer to one.

With all the information already in our model registry, it was time to export it to the different services available. First of all, to the prediction API. As you can see, all the boxes appear to be able to enter the different input data to make the prediction, once made, this appears on the right with the type of wine that the selected model has predicted (Fig. 6.11).

The other configured API is in charge of providing the different data collected from MFlow to Prometheus in the appropriate format. Once everything was implemented, it was not necessary to represent the metrics visually (Fig. 6.12), as it was possible to see from Prometheus that it was scraping correctly. Even so, it was decided to leave such a visualization as it allows to control and isolate errors in a better way in case of future

The screenshot shows a web browser window with the address bar set to 'localhost'. The page title is 'Wine Prediction Form'. The form itself is titled 'Wine Prediction Form' and contains the following fields and values:

Field	Value
Alcohol:	13
Malic Acid:	2.83
Ash:	2.36
Alcalinity of Ash:	19.5
Magnesium:	99.74
Total Phenols:	2.29
Flavanoids:	2.02
Nonflavanoid Phenols:	0.36
Proanthocyanins:	1.59
Color Intensity:	5.05
Hue:	0.95
OD280/OD315:	2.41
Proline:	745.89

At the bottom of the form is a green 'Predict' button. To the right of the form, the text 'Prediction: 0' is displayed.

Figure 6.11: Prediction

improvements or updates. It was also decided that the metrics should show the metric itself and the model to which it belonged. This was not necessary when performing the prediction calculations since all this information was contained in our Model Registry and could be accessed, but it was necessary when using Prometheus, since otherwise there would be no way to distinguish between the metrics of one model or another. For future use, you can also add the different model runs to be able to track variations based on your input data.

After exporting all the metrics and checking that they were in the correct format to be used in Prometheus, it was necessary to check in the Prometheus targets that the correct locations were being scraped and that these links were healthy. These locations corresponded to the endpoints of the exported API container and the Prometheus container itself. As can be seen in the representation (Fig. 6.13), both endpoints correspond to the container/metrics. This is the location where Prometheus scrapes its locations, which is why the declarative API code uses this location.

After checking that both endpoints were up and running, it was time to start searching the different metrics. (Fig. 6.14) At this point, the data scientists would check the

```

Accuracy{run_name="GradientBoostingClassifier"} 0.9444444444444444
Precision{run_name="GradientBoostingClassifier"} 0.9462962962962962
Recall{run_name="GradientBoostingClassifier"} 0.9444444444444444
F1{run_name="GradientBoostingClassifier"} 0.9439974457215836
Accuracy{run_name="SVC"} 0.8055555555555556
Precision{run_name="SVC"} 0.801058201058201
Recall{run_name="SVC"} 0.8055555555555556
F1{run_name="SVC"} 0.80242656449553
Accuracy{run_name="KNeighborsClassifier"} 0.7222222222222222
Precision{run_name="KNeighborsClassifier"} 0.7222222222222222
Recall{run_name="KNeighborsClassifier"} 0.7222222222222222
F1{run_name="KNeighborsClassifier"} 0.7222222222222222
Accuracy{run_name="RandomForestClassifier"} 1.0
Precision{run_name="RandomForestClassifier"} 1.0
Recall{run_name="RandomForestClassifier"} 1.0
F1{run_name="RandomForestClassifier"} 1.0

```

Figure 6.12: Metrics in the API

## Targets

All scrape pools ▾
All
Unhealthy
Collapse All

**mlflow (1/1 up)** [show less](#)

Endpoint	State	Labels
<a href="http://mlflow_exporter:8000/metrics">http://mlflow_exporter:8000/metrics</a>	UP	instance="mlflow_exporter:8000" job="mlflow" ▾

**prometheus (1/1 up)** [show less](#)

Endpoint	State	Labels
<a href="http://prometheus:9090/metrics">http://prometheus:9090/metrics</a>	UP	instance="prometheus:9090" job="prometheus" ▾

Figure 6.13: Prometheus targets

uniformity, consistency, and results of the metrics themselves. Within our project, these searches are necessary to check the subsequent performance of the generated Grafana graphs.

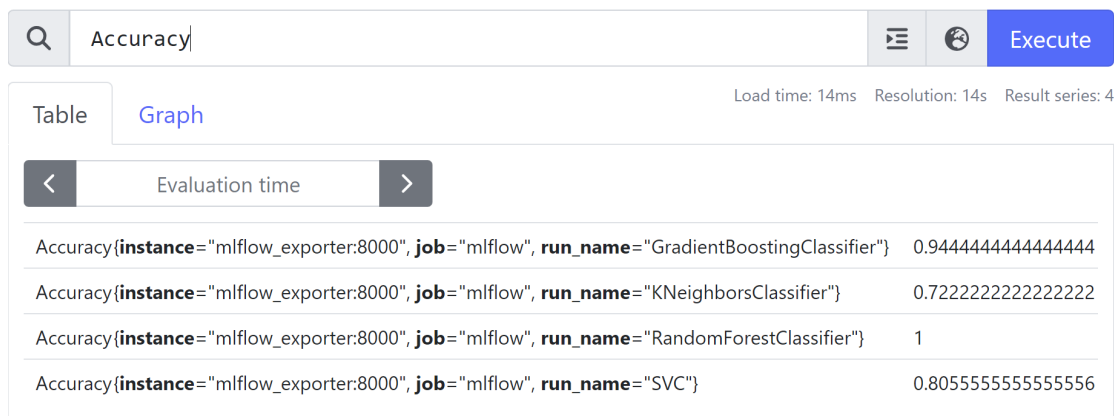


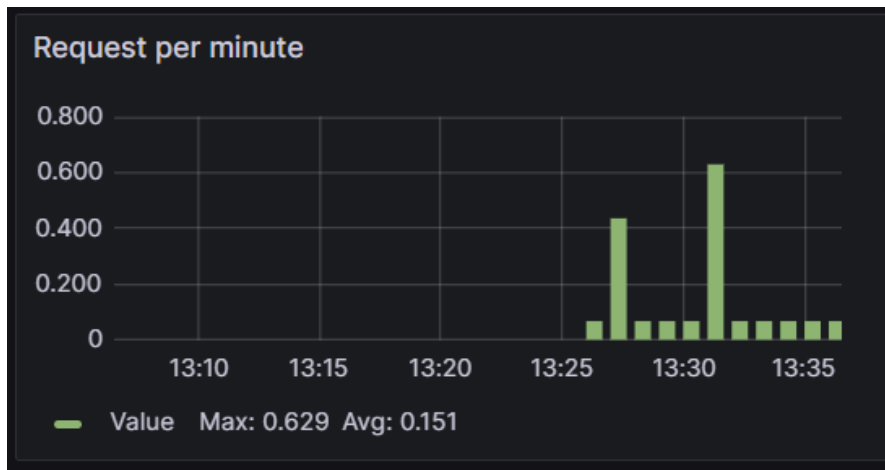
Figure 6.14: Prometheus metrics

As explained in the previous points, Grafana was going to be used for the graphical visualization of the different metrics of the Prometheus server and ML. First, in a colloquial way, it was going to control access and times destined to the return of the requested search values. (Fig. 6.15)

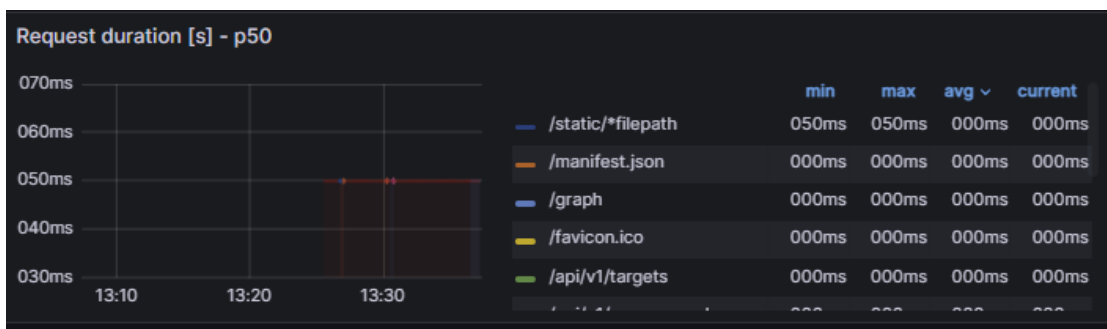
In this case, the four graphs displayed as an example, and which thanks to the configuration carried out, do not have to be configured each time a project is started with these files, are:

- Request per minute: number of access requests per minute. (Fig. 6.15a)
- Ruquest duration [s] -p90: acumulative amount of request duration in periods of 90 seconds. (Fig. 6.15b)
- Ruquest duration [s] -p50: acumulative amount of request duration in periods of 50 seconds. (Fig. 6.15c)
- Request under 100ms (%): request whose time is under 100ms in percentage. (Fig. 6.15d)

Then, all the graphs belonging to the ML, where the variation of Accuracy and Precision (Fig. 6.16) over time is exposed once you train the models.



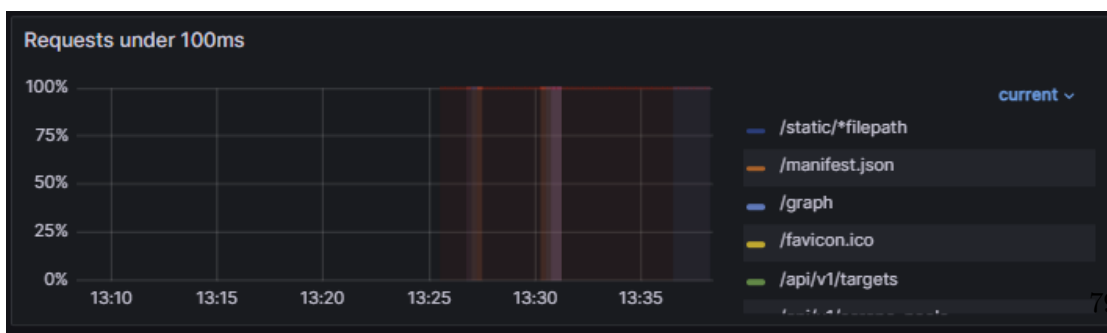
(a) Request per minute



(b) Request duration -p50



(c) Request duration -p90



(d) Request under 100ms (%)

Figure 6.15: Grafana graphics from Prometheus



(a) Accuracy (%)



(b) Precision (%)

Figure 6.16: Grafana graphics from Machine Learning

With the presentation of the Grafana application, all the elements of this system have been explained. With it in its entirety, we have a scalable, reproducible machine learning service with the possibility of adjusting to different environments and with a variety of different trainings.





## Conclusions

---

*This chapter describe the goals achieved by the master thesis following some of the key points developed in the project.*

### 7.1 Introduction

The research and development process carried out within the framework of this thesis has culminated in a number of significant results that deserve to be highlighted. Throughout this work, we have sought to address the challenges commonly encountered in the deployment of machine learning models in production environments, with the main objective of improving the efficiency and reliability of such deployment.

This chapter presents a detailed analysis of the objectives set at the beginning of the project and critically examines the extent to which these objectives have been achieved. Through a comprehensive review of the achievements, it aims to provide a comprehensive assessment of the impact and relevance of this research in the field of operational machine learning.

Furthermore, the importance of the tools and techniques implemented to achieve these objectives is discussed, highlighting their contribution to the overall success of the project. It also reflects on the lessons learned during the development of this research and outlines possible future directions for the continuation of this work.

Ultimately, this concluding chapter serves as a reflective and analytical closure to the master thesis, highlighting both achievements and potential areas for future research and development in the exciting field of operational machine learning.

### 7.2 Achieved Goals

Once the project is finished, it is time to see which of the goals we set at the beginning of the project have been achieved.

- **Seamless Model Deployment:** The MLOps framework successfully facilitated smooth model deployment, reducing friction from development to production through automated processes and version control.
- **Model Versioning and Management:** MLFlow Model Registry effectively managed model versioning, providing a systematic approach to tracking changes, dependencies, and performance metrics throughout the model lifecycle.
- **Monitoring and Alerting:** Integration of Grafana and Prometheus enabled real-time monitoring, ensuring prompt responses to deviations in model behavior and providing insights into resource utilization and system health.

- **Performance optimization:** Continuous monitoring allowed for the identification of optimization opportunities, leading to the implementation of strategies for automated or manual retraining based on model performance.
- **Database Integration:** The connection with a reliable database ensured the consistency of the data, allowing seamless storage and retrieval of model-related metadata, configurations, and performance metrics.
- **Scalable Infrastructure:** The MLOps architecture demonstrated scalability, efficiently handling varying workloads, and accommodating growing data volumes through the utilization of containerization and orchestration technologies.
- **Predictive API:** The development of a user-friendly API for model predictions met operational requirements, offering high availability, low latency, and scalability.
- **Export API:** Implementation of the Export API facilitated secure sharing and collaboration by allowing the export of trained models, ensuring intellectual property protection.
- **Continuous Integration and Continuous Deployment (CI/CD):** The established CI/CD pipeline automates testing, validation, and deployment, ensuring rapid and reliable updates to deployed models in response to changing data or business requirements.

## 7.3 Conclusion

This MLOps initiative, driven by the imperative to address the prevalent challenges in the deployment of machine learning models into production, has yielded substantial and transformative outcomes. The project was conceived in response to industry-wide issues highlighted by Deborah Leff's observations, indicating that a significant percentage of data science projects fail to reach production. The overarching goals were designed to tackle these problems and improve the success rate of machine learning deployments.

Incorporation of the MLFlow Model Registry, Grafana, Prometheus, and the development of Predictive and Export APIs formed the cornerstone of a comprehensive MLOps project. This strategic amalgamation successfully navigated the complexities of seamless model deployment, versioning, monitoring, and optimization.

Throughout the project, the team conducted a meticulous requirement analysis, resulting in a custom architecture that not only met organizational needs but also demonstrated scalability, security, and compliance. The automated process of the continuous integration and deployment pipeline, fostering agility and reliability in model updates.

The case study provided a practical illustration of the efficacy of the MLOps framework, showcasing successful model deployment, lessons learned, and performance metrics. The project's ability to address past deployment challenges, integrate robust security measures, and measure business value through key performance indicators exemplifies its comprehensive approach.

In conclusion, this MLOps initiative stands as a testament to the organization's commitment to overcoming industry challenges and making the most of AI investments. By fostering a culture of collaboration, automation, and continuous improvement, the project not only achieved its defined goals, but also laid a foundation for future advancements in the organization's machine learning endeavors. The outcomes underscore the importance of a systematic and holistic MLOps approach in realizing the full potential of machine learning technologies in production environments.

## APPENDIX **A**

### Code

---

All the code developed in this project

## A.1 docker-compose.yaml

```
version: '3'
services:
  mlflow_postgres:
    build:
      context: .
      dockerfile: mlflow-db/Dockerfile
    container_name: mlflow_postgres
    networks:
      - mlflow_network
    ports:
      - "5432:5432"
  api:
    build:
      context: .
      dockerfile: api/Dockerfile
    container_name: api
    networks:
      - mlflow_network
    ports:
      - "7654:7654"
    depends_on:
      - mlflow_container
  mlflow_container:
    build:
      context: .
      dockerfile: mlflow-container/Dockerfile
    container_name: mlflow_container
    networks:
      - mlflow_network
    ports:
      - "80:80"
```

```
mlflow_exporter:
  build:
    context: .
    dockerfile: mlflow-exporter/Dockerfile
  container_name: mlflow_exporter
  networks:
    - mlflow_network
  ports:
    - "8000:8000"
model_training_container:
  build:
    context: .
    dockerfile: model-training/Dockerfile
  container_name: model_training_container
  networks:
    - mlflow_network
  depends_on:
    - mlflow_postgres
    - mlflow_container
prometheus:
  image: prom/prometheus:latest
  container_name: prometheus
  ports:
    - "9090:9090"
  volumes:
    - ./prometheus:/etc/prometheus
  networks:
    - mlflow_network
  depends_on:
    - mlflow_container
grafana:
  image: grafana/grafana:latest
  container_name: grafana
```

```
ports:
  - "3000:3000"
volumes:
  - ./grafana/grafana.ini:/etc/grafana/grafana.ini
  - ./grafana/datasource.yml:/etc/grafana/provisioning/datasources/
    datasource.yml
  - ./grafana/dashboard.json:/etc/grafana/provisioning/dashboards/
    dashboard.json
  - ./grafana/dashboard.yml:/etc/grafana/provisioning/dashboards/
    default.yml
environment:
  - GF_SECURITY_ADMIN_PASSWORD=admin
networks:
  - mlflow_network
depends_on:
  - prometheus
networks:
  mlflow_network:
    driver: bridge
```

### A.2 Dockerfile MLFlow

```
# Utiliza una imagen base de Python con MLflow preinstalado
FROM python:3.9
#Actualizar pip
RUN pip install --upgrade pip
# Instala MLflow y el exportador de Prometheus desde el requirements.txt
COPY ./mlflow-container/requirements.txt /app/requirements.txt
RUN pip install -r /app/requirements.txt
# Instala wait-for-it
ADD https://raw.githubusercontent.com/vishnubob/wait-for-it/
master/wait-for-it.sh /usr/local/bin/wait-for-it
RUN chmod +x /usr/local/bin/wait-for-it
# Actualiza
```



```
RUN apt-get update && \
    apt-get install -y wait-for-it
# Creo la conexión entre la base de datos y MLflow
ENV MLFLOW_DATABASE_URI postgresql://mlflow:mlflow@mlflow_postgres:5432/mlflow_db
# Establece el valor de MLFLOW_EXPOSE_PROMETHEUS
ENV MLFLOW_EXPOSE_PROMETHEUS true
# Establece el puerto en el que se ejecutará el servidor de MLflow
EXPOSE 80
# Establecer el directorio de trabajo
WORKDIR /app
# Establece el comando por defecto al iniciar el contenedor
CMD ["sh", "-c", "wait-for-it mlflow_postgres:5432 -- mlflow ui --host 0.0.0.0 --port 80 --backend-store-uri postgresql://mlflow:mlflow@mlflow_postgres:5432/mlflow_db --expose-prometheus $MLFLOW_EXPOSE_PROMETHEUS"]
```

## A.3 Dockerfile Flask

```
# Dockerfile_mlflow_exporter
FROM python:3.8
WORKDIR /app
COPY ./mlflow-exporter/ .
# Instala MLflow y el exportador de Prometheus desde el requirements.txt
COPY ./mlflow-exporter/requirements.txt /app/requirements.txt
RUN pip install -r /app/requirements.txt
CMD ["python", "mlflow_exporter.py"]
```

## A.4 Dockerfile Fast

```
# Dockerfile_mlflow_exporter
FROM python:3.8
WORKDIR /app
COPY ./api/ /app/.
# Instala MLflow y el exportador de Prometheus desde el requirements.txt
```

```
COPY ./api/requirements.txt /app/requirements.txt
RUN pip install -r /app/requirements.txt
ENV MLFLOW_TRACKING_URI=http://mlflow:80
EXPOSE 7654
#CMD ["sh", "-c", "uvicorn api:app --port 7654 --host 0.0.0.0"]
CMD ["python", "/app/api.py"]
```

### A.5 Dockerfile Postgres

```
FROM postgres:14.1
ENV POSTGRES_DB mlflow_db
ENV POSTGRES_USER mlflow
ENV POSTGRES_PASSWORD mlflow
EXPOSE 5432
COPY ./mlflow-db/init-user-db.sql /docker-entrypoint-initdb.d/
```

### A.6 init-user-db.sql

```
GRANT ALL PRIVILEGES ON DATABASE mlflow_db TO mlflow;
```

### A.7 mlflow\_exporter.py

```
from flask import Flask, render_template, make_response
from prometheus_flask_exporter import PrometheusMetrics
import mlflow

from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

app = Flask(__name__)
# Configura la URI de seguimiento de MLflow
mlflow.set_tracking_uri("http://mlflow_container:80")
@app.route('/metrics') #Para que prometheus los raspe correctamente
def mostrar_experimentos():
```

```
# Nombre del experimento
nombre_experimento = "Entrenamiento dataset vino"
# Obtén el ID del experimento por su nombre
experimento_id = mlflow.get_experiment_by_name(nombre_experimento)
.experiment_id
# Obtén todas las ejecuciones del experimento
runs = mlflow.search_runs(experiment_ids=experimento_id)
# Inicializa metricas_prometheus
metricas_prometheus = ""
# Itera sobre las ejecuciones y muestra las métricas
for index, run in runs.iterrows():
    run_id = run.run_id
    run_info = mlflow.get_run(run_id).info
    run_name = run_info.run_name
    metrics = mlflow.get_run(run_id).data.metrics
    for metric_name, metric_value in metrics.items():
        # Incluye el run_name en las métricas Prometheus
        metricas_prometheus += f'{metric_name}{{run_name="{run_name}"}}'
        {metric_value}\n'
        #metricas_prometheus += f'{{run_id="{run_id}"}}'
        {{run_name="{run_name}"}} {{run_info="{run_info}"}}\n'
    print(f"Metrics for run {run_id} ({run_name}): {metrics}")
# Renderiza la plantilla HTML con la lista de experimentos
response = make_response(metricas_prometheus)
response.headers["Content-Type"] = "text/plain"
return response
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=8000)
```

## A.8 api.py

```
from fastapi import FastAPI, HTTPException, Query
from fastapi.responses import PlainTextResponse, HTMLResponse
from fastapi.staticfiles import StaticFiles
```

```
import pandas as pd
import mlflow
import uvicorn
import numpy as np

app = FastAPI()

# Configura la URI de seguimiento de MLflow
mlflow.set_tracking_uri("http://mlflow_container:80")
nombre_experimento = "Entrenamiento dataset vino"

def fetch_best_model_uri():
    lowest_mse = float('inf')
    experimento_id = mlflow.get_experiment_by_name(nombre_experimento)
    .experiment_id
    runs = mlflow.search_runs(experiment_ids=experimento_id)
    highest_accuracy = 0
    for index, run in runs.iterrows():
        run_id = run.run_id
        run_info = mlflow.get_run(run_id).info
        run_name = run_info.run_name
        metrics = mlflow.get_run(run_id).data.metrics

        for metric_name, metric_value in metrics.items():
            if metric_name == "Accuracy":
                current_accuracy = metric_value
                if (current_accuracy > highest_accuracy)
                    & (current_accuracy <= 1):
                    highest_accuracy = current_accuracy
                    best_model = run_name

    if run_name == best_model:
```

```
best_model_run_id = run_id

model_uri = f"runs:{best_model_run_id}/{best_model}"
return model_uri

# Montar la carpeta 'static' para servir archivos estáticos
# (como el HTML)
app.mount("/static", StaticFiles(directory="static"), name="static")

@app.get("/", response_class=HTMLResponse)
def read_form():
    return open("static/index.html", "r").read()

@app.get("/predict/")
def model_output(
    feature_1: float = Query(..., description="Feature 1"),
    feature_2: float = Query(..., description="Feature 2"),
    feature_3: float = Query(..., description="Feature 3"),
    feature_4: float = Query(..., description="Feature 4"),
    feature_5: float = Query(..., description="Feature 5"),
    feature_6: float = Query(..., description="Feature 6"),
    feature_7: float = Query(..., description="Feature 7"),
    feature_8: float = Query(..., description="Feature 8"),
    feature_9: float = Query(..., description="Feature 9"),
    feature_10: float = Query(..., description="Feature 10"),
    feature_11: float = Query(..., description="Feature 11"),
    feature_12: float = Query(..., description="Feature 12"),
    feature_13: float = Query(..., description="Feature 13"),
):
    logged_model = fetch_best_model_uri()
    if logged_model:
        loaded_model = mlflow.pyfunc.load_model(logged_model)
        input_data = np.array([
```

```
[feature_1, feature_2, feature_3, feature_4, feature_5,
feature_6, feature_7, feature_8, feature_9, feature_10,
feature_11, feature_12, feature_13]
])

# Crea el DataFrame
predictions = loaded_model.predict(input_data)

return PlainTextResponse(str(predictions[0]),
media_type="text/plain")

else:
    raise HTTPException(status_code=500, detail="No model available.")

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=7654)
```

### A.9 html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Wine Prediction Form</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f4f4f4;
            margin: 0;
            padding: 0;
            display: flex;
            justify-content: center;
            align-items: center;
```

```
        height: 100vh;
    }

    form {
        background-color: #fff;
        padding: 20px;
        border-radius: 8px;
        box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
        max-width: 300px;
        width: 100%;
        display: flex;
        flex-wrap: wrap;
        gap: 8px;
    }

    label {
        flex-basis: 100%;
        color: #555;
    }

    input {
        flex-basis: 100%;
        padding: 8px;
        box-sizing: border-box;
        border: 1px solid #ccc;
        border-radius: 4px;
    }

    button {
        background-color: #4caf50;
        color: #fff;
        padding: 10px;
        border: none;
```

```
        border-radius: 4px;
        cursor: pointer;
        width: 100%;
    }

    button:hover {
        background-color: #45a049;
    }

    h2 {
        margin-top: 20px;
        color: #333;
    }

    #predictionResult {
        color: #555;
        margin-top: 10px;
    }
</style>
</head>
<body>
    <form id="predictionForm">
        <h2>Wine Prediction Form</h2>

        <label for="feature_1">Alcohol:</label>
        <input type="number" id="feature_1" name="feature_1" required>

        <label for="feature_2">Malic Acid:</label>
        <input type="number" id="feature_2" name="feature_2" required>

        <label for="feature_3">Ash:</label>
        <input type="number" id="feature_3" name="feature_3" required>
```



```
<label for="feature_4">Alcalinity of Ash:</label>
<input type="number" id="feature_4" name="feature_1" required>

<label for="feature_5">Magnesium:</label>
<input type="number" id="feature_5" name="feature_5" required>

<label for="feature_6">Total Phenols:</label>
<input type="number" id="feature_6" name="feature_6" required>

<label for="feature_7">Flavanoids:</label>
<input type="number" id="feature_7" name="feature_7" required>

<label for="feature_8">Nonflavanoid Phenols:</label>
<input type="number" id="feature_8" name="feature_8" required>

<label for="feature_9">Proanthocyanins:</label>
<input type="number" id="feature_9" name="feature_9" required>

<label for="feature_10">Color Intensity:</label>
<input type="number" id="feature_10" name="feature_10" required>

<label for="feature_11">Hue:</label>
<input type="number" id="feature_11" name="feature_11" required>

<label for="feature_12">OD280/OD315:</label>
<input type="number" id="feature_12" name="feature_12" required>

<label for="feature_13">Proline:</label>
<input type="number" id="feature_13" name="feature_13" required>

<button type="button" onclick="submitForm()">Predict</button>
</form>
```

```
<div id="predictionResult"></div>

<script>
  function submitForm() {
    const features = [
      'feature_1', 'feature_2', 'feature_3', 'feature_4'
      , 'feature_5', 'feature_6', 'feature_7', 'feature_8'
      , 'feature_9', 'feature_10', 'feature_11', 'feature_12'
      , 'feature_13'
    ];

    const data = features.reduce((acc, feature) => {
      acc[feature] = document.getElementById(feature).value;
      return acc;
    }, {});

    fetch(`/predict/?${new URLSearchParams(data)}`)
      .then(response => response.text())
      .then(prediction => {
        document.getElementById('predictionResult').innerText=
          `Prediction: ${prediction}`;
      })
      .catch(error => console.error('Error:', error));
  }
</script>
</body>
</html>
```

### A.10 train\_model.py

```
import pandas as pd
import numpy as np
import os
```

```
import mlflow
import mlflow.sklearn
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error,
accuracy_score, precision_score, recall_score, f1_score
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier

#Configura la URI de la base de datos y la dirección del servidor de MLflow
mlflow.set_tracking_uri("http://mlflow_container:80")
mlflow.set_experiment("Entrenamiento dataset vino")

# Cargar el dataset de vino
wine = load_wine()
X, y = wine.data, wine.target

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Función para evaluar y registrar métricas
def evaluate_model(model, model_name):
    # Entrenar el modelo
    model.fit(X_train, y_train)

    # Realizar predicciones
    y_pred = model.predict(X_test)

    # Métricas
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted')
```

```
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')

# Definir el nombre del run
run_name = f"{model_name}"

# Log en MLflow
with mlflow.start_run(run_name=run_name):
    # Log de métricas
    mlflow.log_metric("Accuracy", accuracy)
    mlflow.log_metric("Precision", precision)
    mlflow.log_metric("Recall", recall)
    mlflow.log_metric("F1", f1)
    mlflow.sklearn.log_model(model, model_name)

# Modelos
models = {
    "RandomForestClassifier": RandomForestClassifier(),
    "KNeighborsClassifier": KNeighborsClassifier(),
    "SVC": SVC(),
    "GradientBoostingClassifier": GradientBoostingClassifier()
}

# Evaluación de modelos y registro en MLflow
for model_name, model in models.items():
    evaluate_model(model, model_name)
```

### A.11 prometheus.yaml

```
# global:
#   scrape_interval: 15s
#   evaluation_interval: 15s

# scrape_configs:
```

```
#   - job_name: 'mlflow'
#     static_configs:
#       - targets: ['mlflow_container:80']

global:
  scrape_interval: 15s
  evaluation_interval: 15s
  external_labels:
    monitor: "api"

scrape_configs:
  - job_name: 'mlflow'
    static_configs:
      - targets: ['mlflow_exporter:8000']

  - job_name: "prometheus" # Para que monitoree las métricas de prometheus
    static_configs:
      - targets: ["prometheus:9090"]
```

## A.12 dashboard.json

```
{
  "annotations": {
    "list": [
      {
        "builtIn": 1,
        "datasource": {
          "type": "datasource",
          "uid": "grafana"
        },
        "enable": true,
        "hide": true,
        "iconColor": "rgba(0, 211, 255, 1)",
        "name": "Annotations & Alerts",
```

```
        "target": {
          "limit": 100,
          "matchAny": false,
          "tags": [],
          "type": "dashboard"
        },
        "type": "dashboard"
      }
    ]
  },
  "editable": true,
  "fiscalYearStartMonth": 0,
  "graphTooltip": 0,
  "links": [],
  "liveNow": false,
  "panels": [
    {
      "aliasColors": {
        "4xx": "red"
      },
      "bars": true,
      "dashLength": 10,
      "dashes": false,
      "datasource": {
        "type": "prometheus",
        "uid": "PBFA97CFB590B2093"
      },
      "fill": 1,
      "fillGradient": 0,
      "gridPos": {
        "h": 6,
        "w": 7,
        "x": 0,
```

```
    "y": 0
  },
  "hiddenSeries": false,
  "id": 13,
  "interval": "60s",
  "legend": {
    "avg": true,
    "current": false,
    "max": true,
    "min": false,
    "show": true,
    "total": false,
    "values": true
  },
  "lines": false,
  "linewidth": 1,
  "links": [],
  "nullPointMode": "null",
  "options": {
    "alertThreshold": true
  },
  "percentage": false,
  "pluginVersion": "9.1.5",
  "pointradius": 5,
  "points": false,
  "renderer": "flot",
  "seriesOverrides": [
    {
      "$$hashKey": "object:255",
      "alias": "HTTP 500",
      "color": "#bfb000"
    }
  ],
```

```
"spaceLength": 10,
"stack": true,
"steppedLine": false,
"targets": [
  {
    "$$hashKey": "object:140",
    "datasource": {
      "type": "prometheus",
      "uid": "PBFA97CFB590B2093"
    },
    "editorMode": "code",
    "expr": "sum by (status)
(rate(prometheus_http_requests_total[1m]))",
    "format": "time_series",
    "interval": "",
    "intervalFactor": 1,
    "legendFormat": "{{ status }}",
    "range": true,
    "refId": "A"
  }
],
"thresholds": [],
"timeRegions": [],
"title": "Request per minute",
"tooltip": {
  "shared": true,
  "sort": 0,
  "value_type": "individual"
},
"type": "graph",
"xaxis": {
  "mode": "time",
  "show": true,
```



```
    "values": []
  },
  "yaxes": [
    {
      "$$hashKey": "object:211",
      "format": "short",
      "logBase": 1,
      "min": "0",
      "show": true
    },
    {
      "$$hashKey": "object:212",
      "format": "short",
      "logBase": 1,
      "show": true
    }
  ],
  "yaxis": {
    "align": false
  }
},
{
  "aliasColors": {},
  "bars": false,
  "dashLength": 10,
  "dashes": false,
  "datasource": {},
  "description": "",
  "fill": 1,
  "fillGradient": 0,
  "gridPos": {
    "h": 6,
    "w": 9,
```

```
        "x": 9,
        "y": 0
    },
    "hiddenSeries": false,
    "id": 15,
    "interval": "15s",
    "legend": {
        "alignAsTable": true,
        "avg": true,
        "current": true,
        "max": true,
        "min": true,
        "rightSide": true,
        "show": true,
        "sort": "avg",
        "sortDesc": true,
        "total": false,
        "values": true
    },
    "lines": true,
    "linewidth": 1,
    "links": [],
    "nullPointMode": "null",
    "options": {
        "alertThreshold": true
    },
    "percentage": false,
    "pluginVersion": "9.1.5",
    "pointradius": 5,
    "points": false,
    "renderer": "flot",
    "seriesOverrides": [],
    "spaceLength": 10,
```

```
"stack": false,
"steppedLine": false,
"targets": [
  {
    "$$hashKey": "object:426",
    "datasource": {
      "type": "prometheus",
      "uid": "vdXA9nn4k"
    },
    "expr": "histogram_quantile(0.5,
rate(prometheus_http_request_duration_seconds_bucket
{handler!=\"none\"}[30s]))",
    "format": "time_series",
    "interval": "",
    "intervalFactor": 1,
    "legendFormat": "{{ handler }}",
    "refId": "A"
  }
],
"thresholds": [],
"timeRegions": [],
"title": "Request duration [s] - p50",
"tooltip": {
  "shared": true,
  "sort": 0,
  "value_type": "individual"
},
"type": "graph",
"xaxis": {
  "mode": "time",
  "show": true,
  "values": []
},
```

```
"yaxes": [  
  {  
    "$$hashKey": "object:1280",  
    "format": "clocks",  
    "logBase": 1,  
    "show": true  
  },  
  {  
    "$$hashKey": "object:1281",  
    "format": "short",  
    "logBase": 1,  
    "show": true  
  }  
],  
"yaxis": {  
  "align": false  
}  
},  
{  
  "aliasColors": {  
    "none": "red"  
  },  
  "bars": false,  
  "dashLength": 10,  
  "dashes": false,  
  "datasource": {},  
  "fill": 1,  
  "fillGradient": 0,  
  "gridPos": {  
    "h": 6,  
    "w": 9,  
    "x": 0,  
    "y": 6  
  }  
}
```

```
,
"hiddenSeries": false,
"id": 11,
"interval": "15s",
"legend": {
  "alignAsTable": true,
  "avg": false,
  "current": true,
  "max": false,
  "min": false,
  "rightSide": true,
  "show": true,
  "sort": "current",
  "sortDesc": true,
  "total": false,
  "values": true
},
"lines": true,
"linewidth": 1,
"links": [],
"nullPointMode": "null",
"options": {
  "alertThreshold": true
},
"percentage": false,
"pluginVersion": "9.1.5",
"pointradius": 5,
"points": false,
"renderer": "flot",
"seriesOverrides": [],
"spaceLength": 10,
"stack": false,
"steppedLine": false,
```

```
"targets": [
  {
    "$$hashKey": "object:1079",
    "datasource": {
      "type": "prometheus",
      "uid": "vdXA9nn4k"
    },
    "expr": "increase
(prometheus_http_request_duration_seconds_bucket{le=\"0.1\"}
[1m]) \n/ ignoring (le) increase
(prometheus_http_request_duration_seconds_count[1m])",
    "format": "time_series",
    "instant": false,
    "interval": "",
    "intervalFactor": 1,
    "legendFormat": "{{ handler }}",
    "refId": "A"
  }
],
"thresholds": [],
"timeRegions": [],
"title": "Requests under 100ms",
"tooltip": {
  "shared": true,
  "sort": 0,
  "value_type": "individual"
},
"type": "graph",
"xaxis": {
  "mode": "time",
  "show": true,
  "values": []
},
```

```
"yaxes": [  
  {  
    "$$hashKey": "object:1137",  
    "format": "percentunit",  
    "logBase": 1,  
    "max": "1",  
    "min": "0",  
    "show": true  
  },  
  {  
    "$$hashKey": "object:1138",  
    "format": "short",  
    "logBase": 1,  
    "show": true  
  }  
],  
"yaxis": {  
  "align": false  
}  
,  
{  
  "aliasColors": {},  
  "bars": false,  
  "dashLength": 10,  
  "dashes": false,  
  "datasource": {},  
  "fill": 1,  
  "fillGradient": 0,  
  "gridPos": {  
    "h": 6,  
    "w": 9,  
    "x": 9,  
    "y": 6
```

```
    },
    "hiddenSeries": false,
    "id": 16,
    "interval": "15s",
    "legend": {
      "alignAsTable": true,
      "avg": true,
      "current": true,
      "max": true,
      "min": true,
      "rightSide": true,
      "show": true,
      "total": false,
      "values": true
    },
    "lines": true,
    "linewidth": 1,
    "links": [],
    "nullPointMode": "null",
    "options": {
      "alertThreshold": true
    },
    "percentage": false,
    "pluginVersion": "9.1.5",
    "pointradius": 5,
    "points": false,
    "renderer": "flot",
    "seriesOverrides": [],
    "spaceLength": 10,
    "stack": false,
    "steppedLine": false,
    "targets": [
      {
```



```
    "$$hashKey": "object:426",
    "datasource": {
      "type": "prometheus",
      "uid": "vdXA9nn4k"
    },
    "expr": "histogram_quantile(0.9,
rate(prometheus_http_request_duration_seconds_bucket
{handler!=\"none\"}[30s]))",
    "format": "time_series",
    "interval": "",
    "intervalFactor": 1,
    "legendFormat": "{{ handler }}",
    "refId": "A"
  }
],
"thresholds": [
  {
    "$$hashKey": "object:98",
    "colorMode": "critical",
    "fill": true,
    "line": true,
    "op": "gt",
    "value": 0,
    "yaxis": "left"
  }
],
"timeRegions": [],
"title": "Request duration [s] - p90",
"tooltip": {
  "shared": true,
  "sort": 0,
  "value_type": "individual"
},
```

```
    "type": "graph",
    "xaxis": {
      "mode": "time",
      "show": true,
      "values": []
    },
    "yaxes": [
      {
        "$$hashKey": "object:1078",
        "format": "clocks",
        "logBase": 1,
        "show": true
      },
      {
        "$$hashKey": "object:1079",
        "format": "short",
        "logBase": 1,
        "show": true
      }
    ],
    "yaxis": {
      "align": false
    }
  }
],
"refresh": "3s",
"schemaVersion": 37,
"style": "dark",
"tags": [],
"templating": {
  "list": []
},
"time": {
```

```
    "from": "now-30m",
    "to": "now"
  },
  "timepicker": {
    "refresh_intervals": [
      "3s"
    ],
    "time_options": [
      "5m",
      "15m",
      "1h",
      "6h",
      "12h",
      "24h",
      "2d",
      "7d",
      "30d"
    ]
  },
  "timezone": "",
  "title": "Prediction application",
  "uid": "_eX4mpl3",
  "version": 1,
  "weekStart": ""
}
```

## A.13 dashboard.yml

```
apiVersion: 1
```

```
providers:
```

- name: Default # A uniquely identifiable name for the provider
- folder: Services # The folder where to place the dashboards
- type: file

```
options:
  path: /etc/grafana/provisioning/dashboards/
```

## A.14 datasource.yml

```
# grafana/provisioning/datasources/prometheus.yml

apiVersion: 1
deleteDatasources:
  - name: Prometheus
    orgId: 1
datasources:
  - name: Prometheus
    type: prometheus
    access: proxy
    orgId: 1
    url: http://prometheus:9090
    basicAuth: false
    isDefault: true
    editable: true
    # <map> fields that will be converted to json and stored in json_data
    jsonData:
      graphiteVersion: "1.1"
      tlsAuth: false
      tlsAuthWithCACert: false
    # <string> json object of data that will be encrypted.
    secureJsonData:
      tlsCACert: "..."
      tlsClientCert: "..."
      tlsClientKey: "..."
    version: 1
    # <bool> allow users to edit datasources from the UI.
```

## A.15 grafana.ini

```
# grafana.ini
```

```
[server]
```

```
root_url = http://localhost:3000
```



## Aspectos éticos, económicos, sociales y ambientales

---

En este capítulo se explica cuáles son los aspectos en los que este proyecto tiene implicación.

## B.1 Introducción

En este anexo se abordan aspectos éticos, económicos, sociales y ambientales relacionados con el proyecto de evaluación, selección y despliegue de entornos MLOps. Se examinan los impactos relevantes asociados con el desarrollo e implementación de sistemas para la gestión de proyectos de Machine Learning.

## B.2 Descripción de impactos relevantes

Durante el desarrollo del proyecto, se identificaron diversos impactos que merecen atención particular. Entre ellos se encuentran:

Consideraciones éticas en la utilización de algoritmos de aprendizaje automático, incluyendo sesgos algorítmicos y riesgos de discriminación. Implicaciones económicas de la implementación de sistemas MLOps, tales como costos de infraestructura, licencias de software y mantenimiento a largo plazo. Impactos sociales derivados de la automatización de procesos y la posible sustitución de tareas realizadas por humanos por sistemas de inteligencia artificial. Aspectos ambientales relacionados con el consumo de recursos computacionales y la huella de carbono asociada con el entrenamiento y despliegue de modelos de Machine Learning.

## B.3 Análisis detallado de los principales impactos

### B.3.1 Aspectos Éticos

Privacidad y Seguridad de los Datos: Es crucial garantizar que los datos utilizados para entrenar y alimentar los modelos no violen la privacidad de los individuos y que se implementen medidas adecuadas de seguridad para protegerlos de accesos no autorizados. Equidad y Sesgo: Los modelos de aprendizaje automático pueden perpetuar sesgos existentes en los datos, lo que puede llevar a decisiones discriminatorias. Es importante mitigar estos sesgos y garantizar la equidad en las predicciones y decisiones basadas en los modelos. Transparencia y Explicabilidad: Los modelos de aprendizaje automático deben ser transparentes y explicables para que los usuarios puedan comprender cómo se tomaron las decisiones y puedan confiar en los resultados.



### B.3.2 Aspectos Legales

Cumplimiento Normativo: El proyecto debe cumplir con todas las regulaciones y leyes aplicables, como el Reglamento General de Protección de Datos (GDPR) en Europa o leyes de privacidad en otras jurisdicciones. Propiedad Intelectual: Se deben considerar los derechos de propiedad intelectual sobre los modelos desarrollados y los datos utilizados en ellos, así como los acuerdos de licencia relevantes. Responsabilidad Legal: Quién es responsable en caso de que el modelo produzca resultados incorrectos o dañinos debe ser claramente definido y documentado.

### B.3.3 Aspectos Económicos

Costos de Infraestructura: La implementación y mantenimiento de una infraestructura adecuada para el desarrollo, despliegue y monitoreo de modelos de aprendizaje automático puede ser costosa. Rendimiento del Modelo: Los costos asociados con el rendimiento del modelo, como la adquisición de datos de alta calidad y el procesamiento de grandes volúmenes de datos, también deben ser considerados. Valor Agregado: Es importante evaluar el retorno de la inversión (ROI) del proyecto de MLOps y asegurarse de que esté alineado con los objetivos comerciales de la organización.

### B.3.4 Aspectos Sociales

Impacto en el Empleo: La automatización impulsada por el aprendizaje automático puede afectar los empleos existentes, lo que podría tener repercusiones sociales y económicas en la fuerza laboral. Accesibilidad: Es importante garantizar que los modelos desarrollados sean accesibles para todos los usuarios, independientemente de su origen socioeconómico o habilidades técnicas. Educación y Formación: Se pueden requerir programas de educación y formación para capacitar a las personas en el uso y comprensión de los modelos de aprendizaje automático, así como en la ética relacionada con su desarrollo y aplicación.

### B.3.5 Aspectos Sociales

Consumo de Recursos: El entrenamiento y la ejecución de modelos de aprendizaje automático pueden requerir grandes cantidades de recursos computacionales y energéticos, lo que puede tener un impacto ambiental significativo. Sostenibilidad: Se deben explorar

enfoques para mejorar la eficiencia energética y reducir la huella de carbono asociada con el desarrollo y operación de sistemas de aprendizaje automático. Impacto en el Medio Ambiente: Se deben evaluar los posibles efectos ambientales adversos de los proyectos de MLOps y tomar medidas para mitigarlos o compensarlos.

## **B.4 Conclusiones**

El análisis de aspectos éticos, económicos, sociales y ambientales revela la complejidad y la interconexión de los desafíos involucrados en el despliegue de sistemas de aprendizaje automático. Se concluye que una evaluación holística y continua de estos impactos es esencial para garantizar un desarrollo tecnológico responsable y sostenible.

## Presupuesto económico

---

## C.1 PRESUPUESTO ECONÓMICO

	Horas	Salario bruto/hora	Coste Seguridad Social empresario/hora	Total
<b>COSTE DE MANO DE OBRA (coste directo)</b>				
Investigación	250	19	5,7	6175
Implementación	200	19	5,7	4940

	Precio de compra	Uso en meses	Amortización (en años)	Total
<b>COSTE DE RECURSOS MATERIALES (coste directo)</b>				
Ordenador Personal	2750	3	7	98,21

<b>GASTOS GENERALES (costes indirectos)</b>	15,00%	Sobre CD	1681,98
<b>DISEÑO INDUSTRIAL</b>	6,00%	Sobre CD+CI	773,71
		Subtotal	13668,91
		IVA (21%)	2870,470725
		<b>TOTAL</b>	<b>16539,38</b>

Figure C.1: PRESUPUESTO TOTAL

# Bibliography

---

- [1] The apache software foundation. The apache groovy programming language. <https://groovy-lang.org>, 2023. Accessed on 3rd January 2023.
- [2] Atlassian. Git solution for teams using Jira. <https://bitbucket.org>, 2023. Accessed on 27rd January 2023.
- [3] Creative Commons Attribution-ShareAlike. Jenkins: build great things at any scale. <https://www.jenkins.io/>, 2023. Accessed on 27th November 2023.
- [4] Prometheus Authors. Monitoring system and time series database. <https://prometheus.io>, 2023. Accessed on 27rd January 2023.
- [5] The KubeFlow Authors. An introduction to the goals and main concepts of Kube-flow Pipelines. <https://www.kubeflow.org/docs/components/pipelines/v1/introduction/>, 2023. Accessed on 27rd January 2023.
- [6] AWS. Machine Learning: Amazon SageMaker. <https://aws.amazon.com/es/sagemaker/>, 2023. Accessed on 10th November 2023.
- [7] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development. *AgileManifesto.org*, 2001.
- [8] Elasticsearch B.V. Conoce la plataforma de búsqueda que te ayuda a buscar, resolver y tener éxito. <https://www.elastic.co/es/elastic-stack>, 2023. Accessed on 27rd January 2023.
- [9] Prithwi Raj Chakraborty, Alok Kumar Chowdhury, Sujana Chowdhury, and Shahed Al Hasan. A new source code repository for dynamic storing, browsing, and retrieval of source codes. In *2013 International Conference on Informatics, Electronics and Vision (ICIEV)*, pages 1–6, 2013.
- [10] CommitGo. Private, fast, reliable DevOps Platform. <https://about.gitea.com>, 2023. Accessed on 27rd January 2023.
- [11] SQLite Consortium. SQLite. <https://www.sqlite.org/index.html>, 2023. Accessed on 27rd January 2023.

- [12] Edgar Dijkstra. Go To Statement Considered Harmful. *Communications of the ACM*, 11(3), 1968.
- [13] Docker. Docker: make better, secure software from the start. <https://www.docker.com/>, 2023. Accessed on 27th November 2023.
- [14] Domino. Domino Data Lab: Build and operate AI at scale. <https://domino.ai/>, 2023. Accessed on 10th November 2023.
- [15] MariaDB foundation. MariaDB Server: The open source relational database. <https://mariadb.org>, 2023. Accessed on 27rd January 2023.
- [16] The Linux Foundation. Kubernetes: Orquestación de contenedores para producción. <https://kubernetes.io/es/>, 2023. Accessed on 27th November 2023.
- [17] Git. Git-everything-is-local. <https://git-scm.com/>, 2023. Accessed on 27th November 2023.
- [18] Git. Git Large File Storage: An open source Git extension for versioning large files. <https://git-lfs.com/>, 2023. Accessed on 27th November 2023.
- [19] Inc GitHub. Documentación de GitHub Actions. <https://docs.github.com/es/actions>, 2023. Accessed on 27rd January 2023.
- [20] GitLab. The DevSecOps Platform. <https://about.gitlab.com>, 2023. Accessed on 27rd January 2023.
- [21] Google. MLOps: canalizaciones de automatización y entrega continua en el aprendizaje automático. <https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning?hl=es-419>, 2022. Accessed on 7th November 2023.
- [22] Google. Introducing Feast: an open source feature store for machine learning. <https://cloud.google.com/blog/products/ai-machine-learning/introducing-feast-an-open-source-feature-store-for-machine-learning>, 2023. Accessed on 27rd January 2023.
- [23] Google. Vertex AI: Google Cloud. <https://cloud.google.com/vertex-ai?hl=es-419>, 2023. Accessed on 10th November 2023.
- [24] PostgreSQL Global Development Group. PostgreSQL: The World's Most Advanced Open Source Relational Database. <https://www.postgresql.org>, 2023. Accessed on 27rd January 2023.
- [25] H2O. H2O MLOps: H2O.ai. <https://h2o.ai/resources/product-brief/h2o-mlops/>, 2023. Accessed on 10th November 2023.

- [26] Red Hat. Red Hat OpenShift enterprise Kubernetes container platform. <https://www.redhat.com/en/technologies/cloud-computing/openshift>, 2023. Accessed on 27rd January 2023.
- [27] Docker Inc. Docker dogs — Docker Compose Overview. <https://docs.docker.com/compose/>, 2023. Accessed on 3rd January 2023.
- [28] Docker Inc. Docker dogs — Dockerfile reference. <https://docs.docker.com/engine/reference/builder/>, 2023. Accessed on 3rd January 2023.
- [29] Iterative.ai. Data Version Control - (Not Just) Data version control. <https://dvc.org/>, 2023. Accessed on 27th November 2023.
- [30] Jayanth Kumar M J. Feature store for machine learning: Curate, discover, share and serve ml features at scale. In *Feature Store for Machine Learning: Curate, discover, share and serve ML features at scale*, 2022.
- [31] Dominik Kreuzberger, Niklas Kühn, and Sebastian Hirschl. Machine Learning Operations (MLOps): Overview, Definition, and Architecture. *IEEE Access*, 11:31866–31879, 2023.
- [32] KubeFlow. Kubeflow: The machine learning toolkit for Kubernetes. <https://www.kubeflow.org>, 2022. Accessed on 8th November 2023.
- [33] Facebook’s AI Research Lab. PyTorch. <https://pytorch.org/>, 2023. Accessed on 17th November 2023.
- [34] HaeJun Lee, Bon-Keun Seo, and Euseong Seo. A git source repository analysis tool based on a novel branch-oriented approach. In *2013 International Conference on Information Science and Applications (ICISA)*, pages 1–4, 2013.
- [35] Qianying Liao. Modelling ci/cd pipeline through agent-based simulation. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 155–156, 2020.
- [36] Matplotlib. Matplotlib: Visualization with Python. <https://matplotlib.org/>, 2023. Accessed on 17th November 2023.
- [37] Steve Mezak. The origins of devops: What’s in a name? *DevOps Journal*, 2018.
- [38] Microsoft. Modelos de madurez de operaciones de Machine Learning. <https://learn.microsoft.com/es-es/azure/architecture/ai-ml/guide/mlops-maturity-model>, 2022. Accessed on 8th November 2023.
- [39] Microsoft. Azure Machine Learning: ML. <https://azure.microsoft.com/es-es/products/machine-learning>, 2023. Accessed on 10th November 2023.

- [40] Microsoft. Compile, pruebe e implemente continuamente en cualquier plataforma y nube. <https://azure.microsoft.com/es-es/products/devops/pipelines>, 2023. Accessed on 27rd January 2023.
- [41] Microsoft. SQL Server. <https://www.microsoft.com/es-es/sql-server/sql-server-downloads>, 2023. Accessed on 27rd January 2023.
- [42] Microsoft. Work with models in Azure Machine Learning. <https://learn.microsoft.com/en-us/azure/machine-learning/how-to-manage-models?view=azureml-api-2&tabs=cli%2Cuse-local>, 2023. Accessed on 27rd January 2023.
- [43] MLFlow. MLFlow: A platform for machine learning lifecycle. <https://mlflow.org>, 2023. Accessed on 8th November 2023.
- [44] A Narendiran, Abhishek D, Adithya P, Debaditya Ray, Prafullata K. Auradkar, and H. L. Phalachandra. Integrated log-aware ci/cd pipeline with custom bot for monitoring. In *2023 8th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, pages 257–262, 2023.
- [45] NumPy. NumPy. <https://numpy.org/>, 2023. Accessed on 17th November 2023.
- [46] Andrés Felipe Ocampos. Cómo instalar y configurar Jenkins en MacOS. <https://andresfelipeocampo.medium.com/cmo-instalar-y-configurar-jenkins-en-mac-os-517f5c5a>, 2023. Accessed on 4rd January 2023.
- [47] Stephen Oladele. MLOps Landscape in 2023: Top Tools and Platforms. *Neptune.ai blogs*, 2023.
- [48] Oracle. The world’s most popular open source database. <https://www.mysql.com>, 2023. Accessed on 27rd January 2023.
- [49] Pallet. FLask: welcome to Flask documentation. <https://flask.palletsprojects.com/en/3.0.x/>, 2023. Accessed on 27th November 2023.
- [50] Pandas. Pandas: Python data analysis library. <https://pandas.pydata.org/>, 2023. Accessed on 17th November 2023.
- [51] Google Cloud Platform. GCP: configura el almacen de metadatos de tu proyecto. <https://cloud.google.com/vertex-ai/docs/ml-metadata/configure?hl=es-419>, 2023. Accessed on 28th November 2023.
- [52] Sam Ransbotham, Shervin Khodabandeh, Ronny Fehling, Burt Lafountain, and David Kiron. Winning with ai. *MITSloan*, 2019.



- [53] scikit learn. 7.1 Toy datasets - scikit-learn. [https://scikit-learn.org/stable/datasets/toy\\_dataset.html](https://scikit-learn.org/stable/datasets/toy_dataset.html), 2023. Accessed on 4rd January 2023.
- [54] Scikit-learn. Scikit-learn: machine learning in Python. <https://scikit-learn.org/stable/>, 2023. Accessed on 13th November 2023.
- [55] SciPy. SciPy: Fundamental algorithms for scientific computing in Python. <https://scipy.org/>, 2023. Accessed on 17th November 2023.
- [56] Amazon Web Services. Almacén de características de Amazon SageMaker. <https://aws.amazon.com/es/sagemaker/feature-store/>, 2023. Accessed on 27rd January 2023.
- [57] Amazon Web Services. Canalizaciones de Amazon SageMaker. <https://aws.amazon.com/es/sagemaker/pipelines/>, 2023. Accessed on 27rd January 2023.
- [58] Amazon Web Services. Register and Deploy models with Model Registry - Amazon SageMaker. <https://docs.aws.amazon.com/sagemaker/latest/dg/model-registry.html>, 2023. Accessed on 27rd January 2023.
- [59] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.
- [60] Sngular. La metodología para poner en orden los proyectos. <https://www.sngular.com/es/data-science-crisp-dm-metodologia/>, 2020. Accessed on 7th November 2023.
- [61] The Apache software Foundation. Apache Airflow. <https://airflow.apache.org>, 2023. Accessed on 27rd January 2023.
- [62] Grafana Cloud Status. Grafana: The open observability platform — Grafana Labs. <https://grafana.com>, 2023. Accessed on 27rd January 2023.
- [63] Subversion. Apache subversion: Enterprise-class centralized version control for the masses. <https://subversion.apache.org>, 2023. Accessed on 27th November 2023.
- [64] Inc Tecton. TRANSFORM. SERVE. SCALE. Your Journey to Production Machine Learning Starts Now. <https://www.tecton.ai>, 2023. Accessed on 27rd January 2023.
- [65] Tempo. Tempo Framework: Tempo: The MLOps Software Development Kit. <https://mlflow.org>, 2023. Accessed on 8th November 2023.
- [66] TensorFlow. Crea modelos de aprendizaje automático de nivel de producción con TensorFlow. <https://www.tensorflow.org/?hl=es-419>, 2023. Accessed on 8th November 2023.

## BIBLIOGRAPHY

---

- [67] Matteo Testi, Matteo Ballabio, Emanuele Frontoni, Giulio Iannello, Sara Moccia, Paolo Soda, and Gennaro Vessio. MLOps: A Taxonomy and a Methodology. *IEEE Access*, 10:63606–63618, 2022.
- [68] Tiangolo. FastApi framework, high performance, easy to learn, fast to code, ready for production. <https://fastapi.tiangolo.com/>, 2023. Accessed on 27th November 2023.
- [69] Mark Treveil, Nicolas Omont, Clément Stenca, Kenji Lefevre, and Du Plan. Introducing MLOps. *Dataiku*, 2020.
- [70] Unknown. What the waterfall project management methodology can (and can’t) do for you. *LucidChart*, 2023.
- [71] VBStaff. Why do 87 *VentureBeat*, 2019.
- [72] Ingeniería y tecnología. ¿Qué es la programación estructurada? *UNIR*, 2022.