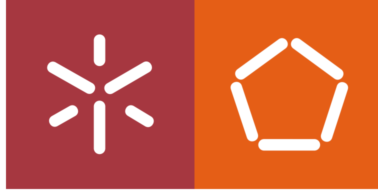
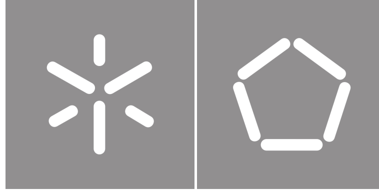


Universidade do Minho
Escola de Engenharia

João Pedro Barros de Sá

Automation of Machine Learning Models Benchmarking

december, 2021



Universidade do Minho
Escola de Engenharia

João Pedro Barros de Sá

Automation of Machine Learning Models Benchmarking

Master Thesis
Master in Informatics Engineering

Work developed under the supervision of:
Professor Doctor João Miguel Lobo Fernandes
Professor Doctor André Leite Ferreira

december, 2021

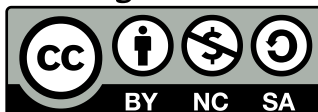
COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

This is academic work that can be used by third parties as long as internationally accepted rules and good practices regarding copyright and related rights are respected.

Accordingly, this work may be used under the license provided below.

If the user needs permission to make use of the work under conditions not provided for in the indicated licensing, they should contact the author through the RepositoriUM of Universidade do Minho.

License granted to the users of this work



Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International
CC BY-NC-SA 4.0

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en

Acknowledgements

This words are dedicated to everyone involved in my academic journey so far, it's success is in part thanks to you.

To my parents, brother and girlfriend for all the support provided during this long year. It was thanks to all of you I got the strength necessary to overcome all the challenges, and finish another chapter in my academic and personal life.

To Professor João Miguel Lobo Fernandes, thank you for providing all the support, guidance and mentorship over the year to achieve this document.

To my work supervisor, André Ferreira, thank your for being my first professional leader and mentor, your guidance and confidence in my work was crucial towards the success of the work achieved throughout the year.

To the MlOps team who together formed great moments of friendship and collaboration, it was a privilege to work with you and thank you for making this work possible in the end.

To Pedro Vieira, for being a great friend and giving the most honest advice, always contributing with positive feedback leading to a higher quality standard in the work performed.

To everyone at Bosch, thank you for providing a great work environment and contributing to this work.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

Resumo

Automation of Machine Learning Models Benchmarking

Na área de ciência de dados, o machine learning está-se a revelar uma ferramenta essencial para resolver problemas complexos. As empresas estão a investir em equipas de ciência de dados e Machine Learning para desenvolver modelos que apresentem valor para os clientes. No entanto, estes modelos são uma pequena percentagem de uma pipeline de projetos de Machine Learning (ML) e, para entregar um produto de ML completo, é necessário um número maior de componentes.

DevOps é uma mentalidade de engenharia e um conjunto de práticas que visa unificar o processo de desenvolvimento e o processo de operações em um software, MLOps é um conceito similar a DevOps mas aplicado ao desenvolvimento e entrega de soluçoes de ML. O nível de automatização das etapas em uma pipeline de ML define a maturidade do processo de ML, que reflete a velocidade de treino de novos modelos com novos dados ou de treino de novos modelos com diferentes implementações. Um sistema de ML é um sistema de software, desenvolvimento e atualizações continuas são necessárias para garantir um sistema que escale conforme as necessidades.

O principal objetivo desta tese é apoiar a criação de um sistema integrado de ML com uma arquitetura que proporcione a capacidade de ser continuamente operada em um ambiente de produção. Um conceito para avaliação de desempenho de algoritmos deve ser elaborado e implementado. O principal objetivo é melhorar e acelerar o ciclo de desenvolvimento de modelos de ML na empresa.

Para atingir este objetivo surge a necessidade de definir uma arquitetura com especificações e a implementação de processos automatizadas num pipeline de ML existente, este processo têm como objetivo alcançar uma ferramenta de benchmark de modelos, com capacidade de analisar o desempenho do modelo, um motor de inferência e um banco de dados para armazenar todas as métricas computadas.

Um sistema baseado em IA em desenvolvimento fornece o caso de estudo para desenvolver e validar a arquitectura. Os avanços atuais na área da condução semiautomática introduz a necessidade de sistemas de monitoramento que podem localizar e detectar eventos específicos no veículo. Os conjuntos de sensores são instalados dentro da cabine para alimentar sistemas inteligentes que visam analisar e sinalizar certos comportamentos que podem impactar a segurança e o conforto dos passageiros.

Palavras-chave: Engenharia Software, Aprendizagem Máquina, Ciência dados, DevOps, MlOps ...

Abstract

Automation of Machine Learning Models Benchmarking

In the field of data science, ML is proving to be a core feature to solve complex real-world problems. Businesses are investing in data science and ML teams to develop AI based models that can deliver business value to their users. However, these models are only a small fraction of an ML project pipeline, and to deliver an end to end ML product, a greater number of components are needed.

DevOps is an engineering mindset and a set of practices that aims to unify the development process and the operation process on software. MlOps is a similar concept to DevOps but applicable to the development and delivery of ML based solutions. The automation of the steps in a ML pipeline defines the maturity of the ML process, reflecting the velocity of training new models given new data or training new models given new implementations. An ML system is a software system that can support development, provide continuous integration and continuous delivery apply to help guarantee that one can reliably build and operate ML systems at scale.

The main objective of this thesis are to support the creation of an integrated ML system with an architecture that provides the ability to be continuously operated in a production-like environment. Furthermore, a concept to evaluate the performance of algorithms shall be devised and implemented. The end goal is to improve and accelerate the ML development lifecycle. To achieve this goal surges the need to define an architecture alongside specifications and the implementation of several automated steps into an existing ML pipeline. To improve and accelerate model development an model engine benchmark tool is devised capable of several features, including the ability to have dashboards for model performance evaluation, an automatic inference engine, performance metrics for the model and a database to store all the computed metrics and metadata.

An AI-based system under development provides the case study to develop and validate this architecture. The current advances of semi-automated driving introduce the need for monitoring systems to scan and detect specific events in the vehicle. Sensor clusters are installed inside the vehicle cabin to feed data to intelligent systems that aim to analyze and red flag certain behaviours that can potentially impact passengers safety and comfort while using the vehicle.

Keywords: Machine Learning, Software, MlOps, DevOps, Data Science, Pipelines, Automation

Contents

List of Figures ix

Acronyms xi

1 Introduction 1

1.1 Context and motivation 1
1.2 Challenge at BOSCH 2
1.3 Thesis goals 2
1.4 Document structure 3

2 State of the art 5

2.1 Software development methodologies 5
2.1.1 The waterfall approach 5
2.1.2 The agile approach 6
2.2 DevOps 6
2.2.1 Technical debt 7
2.2.2 DevOps Culture 7
2.2.3 DevOps techniques 8
2.3 Development in machine learning 9
2.3.1 Crisp-Dm 9
2.3.2 Hidden technical debt in Machine Learning 10
2.4 MLOps 11
2.5 Best practices for model deployment 14
2.5.1 The problem of concept drift 14
2.5.2 MlOps frameworks for the data engine 14
2.5.3 MlOps frameworks for model benchmark engine 15
2.5.4 Model benchmarking methodologies 15
2.6 Model explainability methodologies 16
2.6.1 Lime values explainability 17
2.6.2 Shap values 18

2.7 Model monitoring methodologies 18
2.8 Summary 19

3 Early work **21**
3.1 Data collection - Hardware solution 21
3.1.1 Sensors and computational device 21
3.1.2 Casing solution 22
3.2 Software solution 23
3.2.1 Data visualization tool 24

4 Objectives and results **26**
4.1 Problem definition 26
4.1.1 Benchmark consideration 26
4.1.2 Concept drift in the context 27
4.1.3 Continuous deployment and continuous training 27
4.2 Objectives 27
4.3 Requirements 28
4.3.1 Functional requirements 28
4.3.2 Non functional requirements 29

5 Design and architecture **31**
5.1 The benchmarking engine in integration with the Atlas pipeline 31
5.2 Model benchmark engine design and architecture overview 34
5.2.1 System use cases 35
5.3 Solution architecture 41
5.3.1 Database solution, design, and architecture 43
5.3.2 Benchmarking engine, design and architecture 45
5.3.3 Web engine architecture 51
5.4 Deployment overview 53
5.5 Summary 54

6 Results demonstration and discussion **55**
6.1 Conclusion 62
6.1.1 How does this solution improve the team workflow 62
6.2 Future work 63
6.3 Final remarks 64

Bibliography **65**

List of Figures

1 ML system pipeline . . . 2
2 Model engine benchmark components . . . 3
3 Waterfall development approach, by (Van Casteren, 2017 . . . 6
4 DevOps communication and feedback flow . . . 8
5 Components in a ML system, ("MLOps", 2020 . . . 9
6 Crisp-DM pipeline by Hassanien, 2019 . . . 10
7 First level of maturity, by ("MLOps", 2020 . . . 12
8 Intermediate level of maturity, by ("MLOps", 2020 . . . 13
9 Final level of maturity, by "MLOps", 2020 . . . 13
10 ConvNet layer explainability . . . 17
11 Kullback Leibler . . . 18
12 Hellinger distance . . . 19
13 Average Pairwise distance . . . 19
14 Design's levels design . . . 22
15 Printed multi level design . . . 23
16 Component diagram for data collection tool . . . 23
17 Activity diagram for data collection tool . . . 24
18 Component diagram for data visualization tool . . . 24
19 Activity diagram for visualization tool . . . 25
20 Atlas Pipeline flow . . . 32
21 Atlas pipeline component diagram . . . 33
22 Atlas pipeline activity diagram . . . 34
23 **UC1:** Sequence diagram for manual model insertion . . . 36
24 **UC2:** Sequence diagram for manual CRUD operations . . . 36
25 **UC3, UC4:** Sequence diagram for data filtering . . . 36
26 **UC5, UC6:** Sequence diagram for benchmark and results filtering . . . 37
27 **UC7:** Sequence diagram for ordering a model page. . . . 38
28 **UC8:** Sequence diagram for ordering an explainability page. . . . 38

29 **UC9:** Sequence diagram for ordering a dataset explainability page. 39
30 **UC10:** Sequence diagram for obtaining logging information. 39
31 Model benchmarking engine internal process 40
32 Model benchmarking engine, manual inputted model 41
33 Model benchmarking engine, building block diagram 42
34 Benchmarking engine class diagram 44
35 Benchmarking engine model file explained 46
36 Benchmarking engine subcomponent explained 47
37 Benchmarking engine activity diagram 48
38 Benchmark time in seconds by number of threads 50
39 Benchmarking engine efficiency loss by thread added 50
40 Benchmarking engine scalability comparison, thread BPM comparison 51
41 Web engine subcomponent explained 52
42 Application deployment diagram 53
43 Atlas final architecture 54

44 Initial system webpage 56
45 Model listing webpage 56
46 Benchmark listing webpage 57
47 Dataset listing webpage 57
48 Dataset detail webpage 58
49 Admin dashboard and log information webpage 59
50 Admin dataset management webpage 59
51 Model benchmark page 60
52 Model benchmark page, confusion matrix 61
53 Model benchmark hyperparameters page 61
54 Model benchmark SHAP explainability page 62

Acronyms

AI Artificial Intelligence

CD Continuous Delivery

CI Continuous Integration

CRUD Create / Read / Update / Delete

ML Machine Learning

UC Use Case

Chapter 1

Introduction

This thesis is integrated into the area of Software development and machine learning, explicitly integrating good software development mindsets and practices into an end-to-end machine learning system and fixing core problems and challenges in the current panorama of machine learning development.

1.1 Context and motivation

Software engineering is an ever-evolving field. Software applications development is costly in terms of time and human resources. Over the years, many research and development methodologies have been proposed to optimize the development process.

DevOps is an emerging approach to agile software development, it proposes a new culture in the software engineering field combined with a set of practices and tools to unify the development and the operations process, (Wahaballa, Wahballa, Abdellatief, Xiong, and Qin, 2015). This means that if successfully, a much faster development cycle and deployment can be achieved while facilitating updates and maintenance processes to the application. A successful *DevOps* implementation means that a much faster development cycle and deployment can be achieved while facilitating updates and maintenance processes. A successful DevOps implementation usually relies on automating previously manual and slow steps, thus enabling speed of development, testing, and deployment.

ML and data science are two emerging fields in the IT area and proving to solve complex real-world problems with great success. As a result, businesses invest in data science and ML teams to develop predictive models that deliver real value for customers and companies, (Karamitsos, Albarhami, and Apostolopoulos, 2020). However, the models developed are usually only a tiny part of a much bigger ML product. To deliver an end-to-end ML product, a more significant number of components are needed.

MLOps proposes to be a set of practices that combine software development and IT operations for ?? based systems. The level of automation in an ML pipeline defines the maturity of the ML process.

1.2 Challenge at BOSCH

This thesis was developed as part of Bosch Car Multimedia at Braga in the scope of a product development team.

A specific challenge for the team is the automation of specific tasks to increase the speed at which a new model can be trained and evaluated given new data or given new model implementations. A existing ML development pipeline provides the use case to develop this automation pipeline. This system targets the car-sharing market and aims to scan and red flag certain events outside the vehicle cabin. The sensor cluster installed inside the car collects and feed the data for the intelligent systems. In figure 1 we can see a diagram of the ML pipeline to be created.

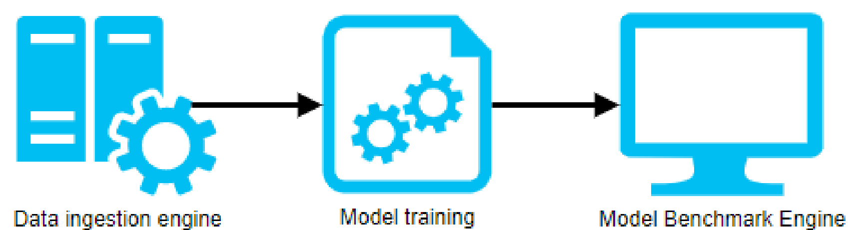


Figure 1: ML system pipeline

This thesis focus on the Model benchmark engine block represented in the diagram. Each model that is trained needs to be evaluated using a test dataset, this evaluation produces key performance indicators that need to be compared between each relevant model that has been produced. All this information is stored in a database which provides ease of accessibility and versioning of models.

This pipeline process allows for a much faster end-to-end development cycle. As a result, models can be trained and tested much faster, increasing the success rate of ML implementation while also providing a good solution for future maintenance required when the models start decaying.

1.3 Thesis goals

In this document, the process followed to develop the pipeline described in figure 1 is explained, from data collection in a real-world environment, data preparation and cleaning using the developed data ingestion engine, model development, deployment, and final but not least, the model benchmark engine.

The goal and focus is to develop the third component, the **Model benchmark engine,** that can interact directly with the previous steps of the pipeline and automate several tasks that speeds up the process to train and evaluate new models. Figure 2 is a representation of the different components that constitute this engine.

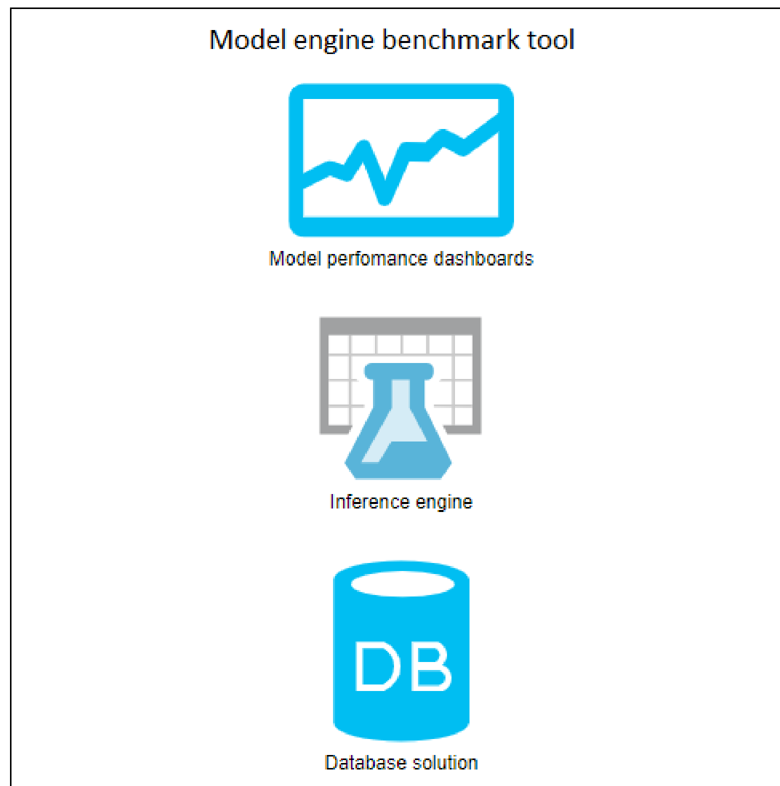


Figure 2: Model engine benchmark components

First, the **Model performance dashboard** is a tool to simplify the process of evaluating an ML model. The dashboard will contain key performance indicators of the model, thus facilitating the comparison process between models.

The **Inference engine** is a component of the ML system pipeline that applies logical rules to a knowledge graph to derive new information. Inference engines have gained much traction due to the current machine learning boom and are extremely useful at providing new insight and knowledge.

Lastly, in the current machine learning panorama, many models algorithms are built, trained, and evaluated, all with different architectures, data pipelines, and parameters, generating an issue. The **Model performance database** is a component designed to automate and facilitate model storage. It stores all the information related to each model, including key performance indicators and data pipelines used. It also allows for a versioning system to know exactly everything about each model and its evaluation.

This solution aims to automate, simplify and speed the whole process of building and deploying models in the ML systems developed.

1.4 Document structure

The remaining document structure is the following:

- **State of the art** - This section targets a complete review of the literature around the topic of

this thesis, including problems in software development methodologies and the solutions that were proposed over the years to solve them. Machine learning systems can be software systems, so the same challenges can also exist in the current ML system development. A complete review of these topics will be presented in this section.

- **Early work** - This chapter shows the early work towards the development of the MlOps automation pipeline.
- **Objectives and results** - This chapter outlines the requirements defined for the solution.
- **Design and architecture** - Full explanation of the implementation process to achieve the architecture defined in the previous steps. All the decisions made, implementation choices, and outcomes.
- **Conclusion** - This chapter provides the experiment setup explanation and the entire outcome of those setups. It concludes the work achieved so far.

Chapter 2

State of the art

This chapter provides an in-depth literature review on machine learning and software systems development approaches. What was used in the past, what is used now, and the main advantages and problems that arose from each. Since machine learning is a sub-field of Artificial Intelligence (AI), some software development problems extrapolated from one to the other, so before we tackle the approaches currently used in Machine Learning (ML), a brief look at the past is needed to further enhance the perception of the current state of the art in development methodologies.

2.1 Software development methodologies

This sub-chapter provides the foundation on how the current state of the art came to be. It reviews some of the technical difficulties and the solutions proposed in the past to understand the present.

2.1.1 The waterfall approach

Before the 21st century, the waterfall approach was a common methodology used to deliver software inside an organization. This approach was divided into multiple phases, as described in figure 3. Each phase had to be completed before proceeding to the following stage, and this meant that changes required after the initial requirements phase were complex and costly to implement, and the process was very rigid, maintenance in the software was also a difficult task, (Van Casteren, 2017).

At the beginning of the century, software was in more demand than ever, and the waterfall approach started to show limitations dealing with the volatility in software requirements, (Rahman, 2019). The development cycle was long, rigid, and could no longer keep up with the increasing demand for software supply and maintenance processes.

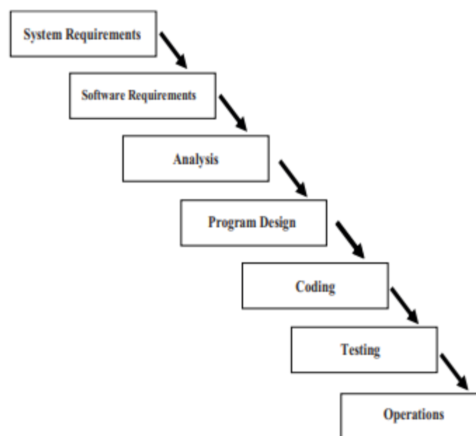


Figure 3: Waterfall development approach, by (Van Casteren, 2017
).

2.1.2 The agile approach

In 2001, due to the limitation of the waterfall approach, a group of developers discussed a new methodology for software development, which led to the creation of the agile manifesto, (Martin Fowler, 2001). This manifesto introduced a set of rules and methods to improve interaction and communication within teams. Agile approaches required face-to-face interaction and short iterative cycles with dynamic feature planning, (Van Casteren, 2017).

The agile approach was meant to make software delivery fast and continuous. It intends to integrate business people with developers, facilitate late requirements change, motivate individuals, improve the overall team performance, and schedule regular meetings to discuss how to become more effective. Simplicity in design choices was welcomed, (Kumar and Bhatia, 2012).

Multiple papers surged over the years on tackling agile development, being scrum one of the most popular. It named the development iterations cycles as sprints, which usually lasts between 1-4 weeks. Team meetings called daily scrums happen every day and serve to discuss what has been done, difficulties, and future work, but the original manifesto's core philosophy remained unchanged.

2.2 DevOps

DevOps is a set of practices that combines software development (Dev) and IT operations (Ops) with one key difference with respect to the agile approach, which while it was great in simplifying the development process between business people and developers, the operations process was left out. To solve this limitation, *DevOps* was born, proposing methods to approximate and combine the development and operations process into one.

2.2.1 Technical debt

Technical debt is a term coined by Ward Cunningham in 1992. The term refers to a metaphor between debt in the financial system and the debt in software development, (Cunningham, 1992).

This debt refers not necessarily to lousy coding. Most of the time, it represents a trade-off between good practices and simplified implementations to save time and speed up the delivery process of new features. As the development process matures over time, testing teams report more and more bugs resulting from these cheap implementations creating a deficit over time that needs to be paid. This phenomenon can create a problem where software maintenance is compromised because as each situation is dealt with, more issues arise in the process.

With the introduction of agile methodologies, dealing with technical debt became more straightforward, but there still was little connection between developers and operators. *DevOps* surged as a way to mitigate the problem of technical debt, streamlining the whole development process, from software requirements to operations. While this helped manage technical debt, it has not eliminated nor was it expected to eliminate the problem.

2.2.2 DevOps Culture

Gene Kim, one of the biggest contributors to the topic states in his book, *The DevOps Handbook* that three principles are sustaining a good *DevOps* practice and implementation, (Kim, Humble, Debois, and Willis, 2016).

The first principle is about enabling **left-to-right workflow and communications** from the development to operations and costumers, removing the barriers between different teams or departments and promoting cooperation. The work is made visible and accessible to maximize flow, the updates are minor but frequent, and quality is preferable over quantity. This leads to a smaller time requirement to fulfil requirements, all while increasing quality. The result of this practice is continuous build, **continuous integration**, **continuous test** and **continuous deployment.**

The second principle is all about enabling **fast feedback flow from right-to-left** from the operations and costumers to developers in all phases of the work. Adopting this principle allows for much quicker problem detection and consequent fix, permitting problems to be solved at the core, creating quality in the product, and reducing fatal failures to occur down the line. This principle is to contrast with the first, see figure 4.

The third principle is all about building the *DevOps* **culture and mindset**, to ensure a high trust mentality that supports a dynamic, disciplined environment. The scientific approach to building products with tolerance to risks, making knowledge gained from failure and success alike, to complement this the shorter feedback loops allow for safer systems which increase the tolerance for taking risks and learning with them. A team adopting these principles makes them evolve faster than the competition.

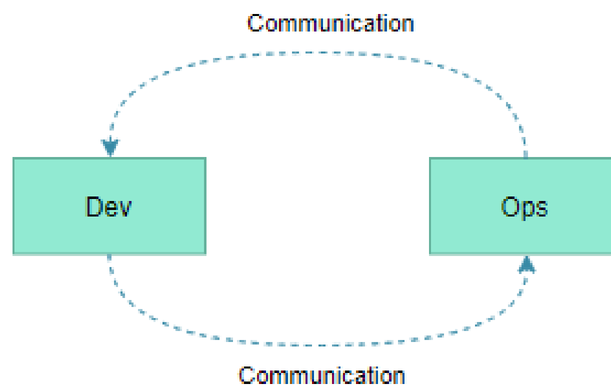


Figure 4: DevOps communication and feedback flow

2.2.3 DevOps techniques

To adopt *DevOps* there are several techniques to apply, Manish Virmani stated numerous techniques, (Virmani, 2015), as follows:

- **Continuous planning** states that business plans need to be agile to meet up with fast market changes. Frequent interim checkpoints to overview the current state and adjust as needed. Without DevOps, it's hard for developers to keep up with change, but with DevOps, the increase in communication and feedback between developers and customers allows for a better overhaul flow.

- **Continuous integration** or Continuous Integration (CI) for short is a process where a team automates most of the tasks between development and operations. It involves teams integrating their work regularly and automating tasks like building, testing, and validating. In the end, this achieves faster bug finding and fixing and a greater standard of quality in the software with lower costs. This is what makes CI a stone pillar of a good **DevOps** implementation.

- **Continuous deployment** is the act of getting all types of updates into the hands of users as fast and safely as possible, without worrying about what type of update it is, whether a bug fix, a new feature or even a large scale update. This reduces friction points inherent in the more traditional deployment or release processes. In other words, the application is always in a production-ready state.

- **Continuous testing** is all about automating every step possible when testing. If an action is repetitive over time, it must be automated. Currently, there is a large number of technologies in the market that simplify this process.

- **Continuous monitoring**, due to testing better and early, there is an opportunity to observe critical parameters and react to unseen events faster.

2.3 Development in machine learning

Some machine learning applications share a set of similar characteristics with more traditional software applications. In many cases, they are almost identical, but with an extra ML model layer, ("MLOps", 2020). Due to this, much of the same conventional problems are present, with the ML system introducing new concerns that need to be addressed. This section provides a complete analysis of ML development methodologies state of the art and the work being done to improve ML system pipelines. In figure 5 a typical ML system structure is presented.

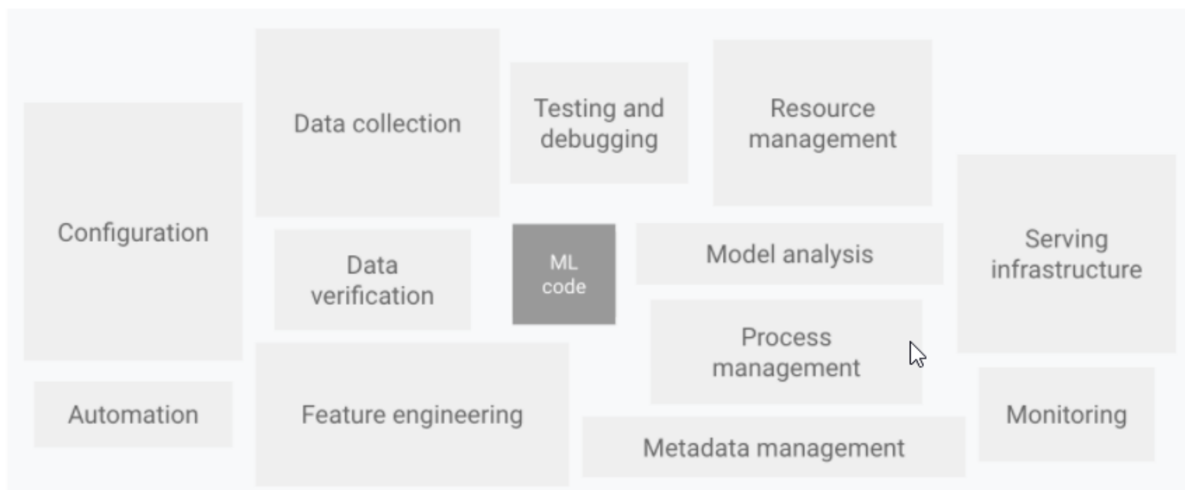


Figure 5: Components in a ML system, ("MLOps", 2020
)

2.3.1 Crisp-Dm

With the rise of machine learning, one of the methodologies that gained the most popularity is *Crisp-Dm*, short for **Cross Industry Standard Process for Data Mining**, This methodology is an industry standard consisting of six phases that describe most machine learning pipelines, see 6.

Each step of the pipeline described is an important phase and are explained as follows:

- **Business understanding** is a crucial step. It consists of understanding the underlying business that we want to apply machine learning, what customers want to accomplish and what data can lead us there. If we start working on a problem without previous knowledge, it won't be easy to know which data to collect and the best way to treat it.

- **Data understanding** is the step where we gain in-depth information about the problem. It usually starts with an extensive exploration of the data with statistic analysis and data visualization.

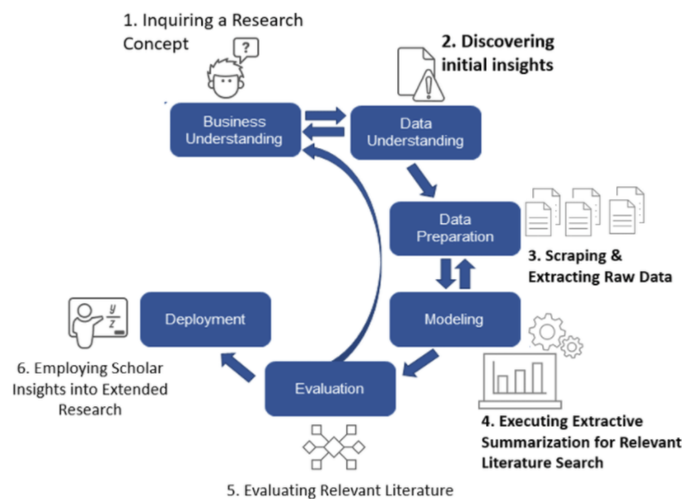


Figure 6: Crisp-DM pipeline by Hassanien, 2019

- **Data preparation** is generally one of the most time-consuming parts of the pipeline. It's infrequent to obtain the data ready to fit the algorithms. Most of the time, we have raw collected data that is unsuited for our goals. Generally, we want each record in a row and each attribute in a column, but this is a general rule of thumb as video or audio data have different needs. In this phase of the pipeline, we already have information on the quality of the data, for instance, how many missing values, if the dataset is balanced, or simply if there is enough data for an ML algorithm to learn.

- **Modelling** is about creating the model. First, there is the need to choose the technique that best suits our problem and data, then a test design is generated that may involve splitting the dataset into training, validation, and test subsets. The next step is to build the model and fine-tune it to get the best results possible. In this phase, we can go back to the data preparation phase and make changes to best suit our current model.

- **Evaluation** is the phase where the model is evaluated, in this step we check if the model performs well and answers to the business needs, then an evaluation of the entire pipeline is done. In this stage a decision is made whether to deploy the model or return to the first phase to start again and change the approach.

- **Deployment** is the last stage of the pipeline since a model isn't useful if it cannot be accessed by its customers. In this stage, a deployment plan is built and a monitoring system is created to help future maintenance in the model.

2.3.2 Hidden technical debt in Machine Learning

In chapter 2.2.1 the term technical debt is introduced, following that definition, (Sculley et al., 2015) introduced in a published article in 2015 the preeminence of ML systems to develop a very specific type of

technical debt, which he called **Hidden technical debt** stating that ML systems have all the traditional problems of traditional coding plus specific problems that arise from the ML components.

Furthermore, in traditional software, technical debt is code related, but in ML applications, Hidden technical debt is code and data-dependent, since an ML model learns from data, any change to the model or it's data changes everything, (Alahdab and Çalıklı, 2019, Sculley et al., 2015 and Sculley et al., 2014) exemplifies this phenomenon in an article studying and finding patterns of hidden technical debt in ML systems.

2.4 MLOps

DevOps revolutionized traditional software development. ML system development is still in an infant state compared to software, but the area has been receiving much more focus in recent years. In the academic community, most of the attention goes to the early stages of the ML pipeline, preparing and exploring data and model development, but in the real world, model deployment and maintenance are as important if we want to give use to our ML systems, so automating our system pipeline becomes a crucial step to have reliable results in real-life applications, (Arnold et al., 2020).

MlOps came to solve many of the issues of ML development by applying *DevOps* practices in ML systems. According to ((“MLOps”, 2020)), ML systems differ from traditional software and fail in different areas. The steps in a traditional data science workflow are as follows:

- **Data extraction** is at the beginning of every data science workflow, since ML isn't possible without data. In this phase, data is collected from one or multiple relevant sources.

- **Data analysis** corresponds to the phase where the raw data collected in the previous phase is analyzed. Once this process is completed, the team gets familiar with the data available, its characteristics, and the work that will need to be performed in the next phase.

- **Data Preparation** is one of the steps that are generally more time-consuming. It involves cleaning the raw data, treat missing values and perform feature engineering. In the end of this step we have the data ready to feed our model.

- **Model training** is where the team implements and tests with different algorithms and different hyper-parameters tune. This outputs the best-trained ML model.

- **Model evaluation**, the model is validated for predictive value, and if it passes, it's ready to be deployed in a real-world scenario.

- **Model serving** is when the model is deployed in the real world in a specific environment, this can be a web API, mobile or desktop application, among other possibilities.

- **Model monitoring** is the final step of a usual Data science workflow, the model is monitored, and if key performance indexes start dropping, maintenance is required, and an ML task in this process is performed.

MlOps is all about automating every single step that is repetitive along multiple ML pipelines, and the level of automation defines the maturity of the MlOps implementation. A bigger maturity means a faster end to end process.

An entry-level ML system has no automation in the pipeline. This is very common in academic projects and companies starting their first ML projects. The typical pipeline in this type of project doesn't include continuous delivery or integration, no autonomous monitoring, see figure 7. This type of system usually doesn't age well due to all the limitations that will contribute to hidden technical debt.

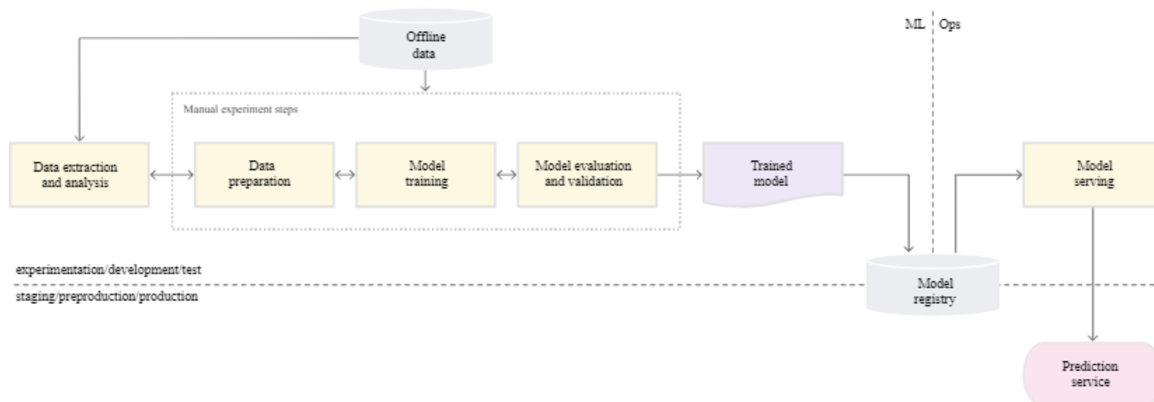


Figure 7: First level of maturity, by ("MLOps", 2020
)

In the next level, we find a type of ML systems that have a more mature *MLOps* implementation. These types of systems have a significant level of automation implemented. This includes the data ingestion pipeline to feed the model that ensures continuous training, finally, the consequent continuous serving of the model to the final customer. At this level, we can train new models given new data very fast. However, the process is still slow and not ideal for making changes in the ML algorithm, see figure 8.

The final level of maturity is the ideal *MLOps* ML system with fully-fledged CI/CD, (continuous integration / continuous delivery), implementations in this stage. The continuous integration usually has features like automatic testing for model performance, convergence and behavior checks between different pipeline components. The continuous delivery implies very fast delivery of changes in the pipeline to a production environment, this level of maturity is expected to have a very fast implementation given new data or given a new algorithm, see figure 9.

One of the key parts of every *MLOps* implementation is to improve the system's quality in the long run, as ML systems don't age in the same way as traditional software systems. To combat this phenomenon,

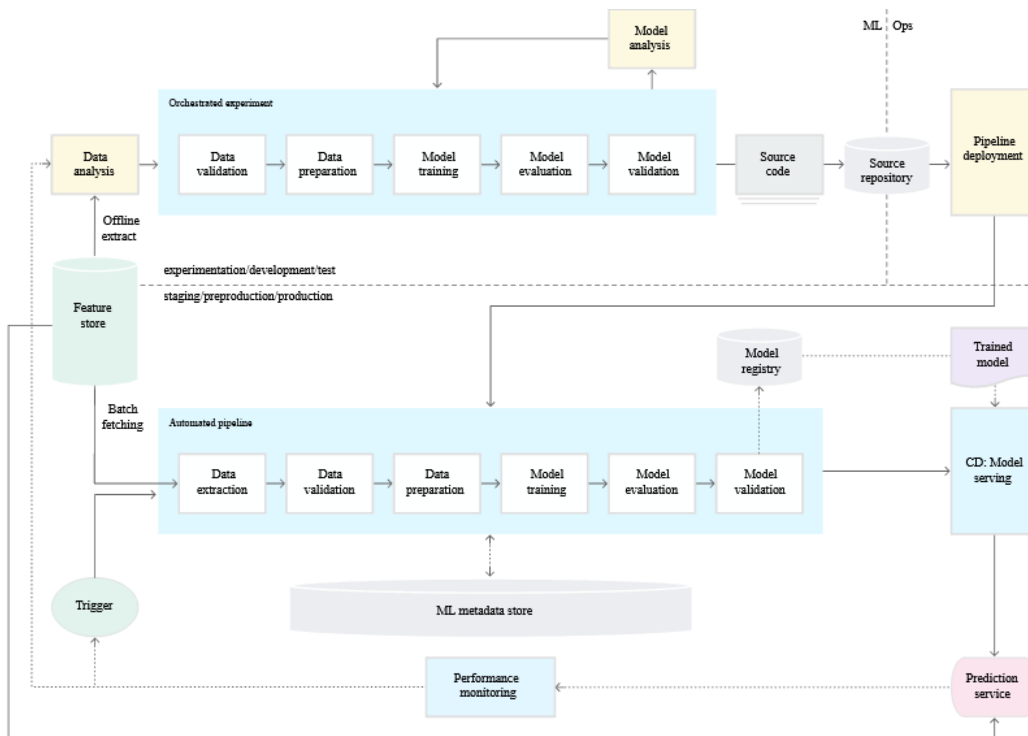


Figure 8: Intermediate level of maturity, by ("MLOps", 2020
)

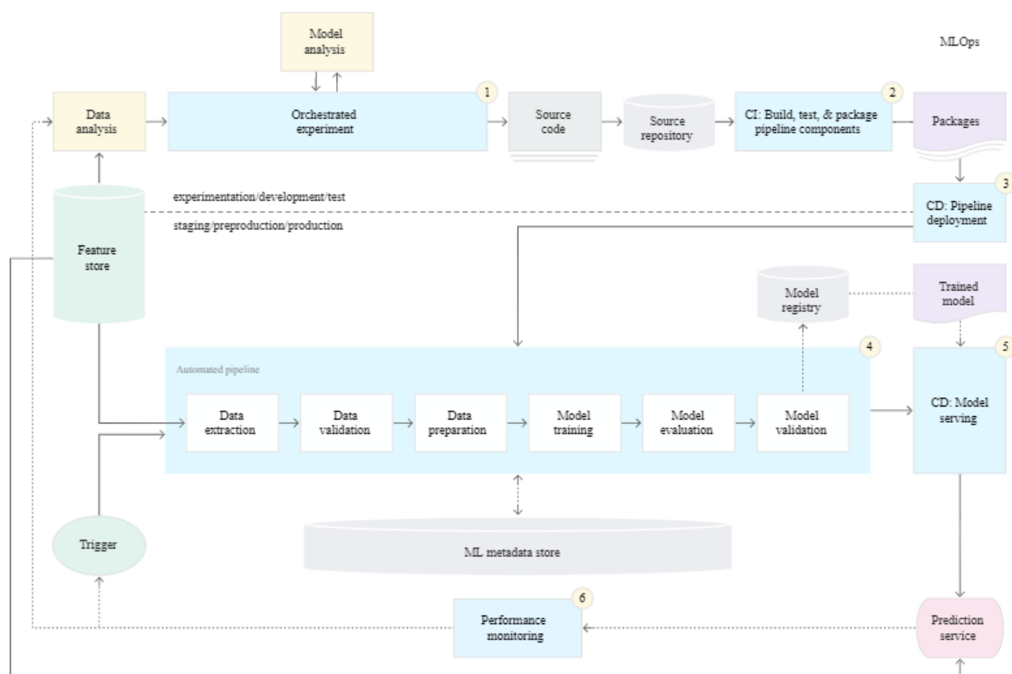


Figure 9: Final level of maturity, by "MLOps", 2020

there is a need for pipelines that can support continuous training and deployment on top of robust monitoring systems capable of detecting deterioration in the model by analyzing its performance metrics and trigger solutions. This phenomenon is called concept drift and is discussed further in the next section.

2.5 Best practices for model deployment

As already seen, there is a significant number of challenges when deploying machine learning systems. This section provides a complete analysis to find the best MlOps frameworks that allow for continuous monitoring of our model key performance index and versioning our models to compare between new iterations and old ones. The goal is to gain knowledge of why some ML systems succeed while other fails.

A division is made between different phases of the pipeline, the first part is called **data engine** and deals with everything that is before the model training. This includes data extraction, analysis, and preparation. The second division is called **model benchmark engine** and includes everything after training the model such as model evaluation, serving, and monitoring, see 1.

2.5.1 The problem of concept drift

In traditional software, a system doesn't lose functionality over time, although it may degrade, see p.25 in Fernandes and Machado, 2015. A well-coded calculator still works twenty years later. However, in machine learning this is not linear. From the moment a model is deployed, it already starts to degrade and perform worse. This is called concept drift, and is deeply explored in (Lu et al., 2020)

Concept drift is a term used in statistical analysis and machine learning that describes the particularity that data tends to change and mutate in unforeseen ways as time progresses. For this reason, concept drift remains one of the biggest challenges in implementing an ML system, it can dictate the catastrophic failure of an ML system in a real-world scenario; however, all of this can be avoided if we can detect it and act appropriately.

Data mutate in various ways, one example can be the different seasons in a year that can change the most bought types of clothes, for instance, or new trends that emerge over time that can be entirely unpredictable, this represents obstacles that any given model needs to overcome over time to be considered successful.

2.5.2 MlOps frameworks for the data engine

Many frameworks are already present in the market that aid in the creation of an ML pipeline. In literature, the process of automating data storage and access is often associated as ***DataOps***. Some MlOps frameworks have a set of features that do not improve the data engine and are particularly useful to later stages, in this phase the discussion is for tools that allow data pipelining, in other words, to apply a set of repetitive processes to incoming data and data versioning to distinguish between different iteration and revisions of the dataset. This allows for a much faster cycle between the data collection and model training, as much of the data manipulation and processing can be automatized.

Some of the best frameworks for this workload are as follows:

- **Pachyderm** is a framework that provides an excellent set of tools for the data. It allows data versioning and pipelining by tracking data revisions along with the duration of the dataset.
- **KubeFlow** is a framework that started as a way to run Tensorflow jobs in Kubernetes. However, it has expanded to run in a multi-cloud, multi-architecture and now can manage an entire pipeline. It is great at experimenting and managing data pipelines and scales really well for any project size, ("Kubeflow", n.d.).
- **DataBricks** is a unified data analytics platform that can process massive-scale data engineering and provides integration with mlflow, apache-spark, and amazon web services.

2.5.3 MlOps frameworks for model benchmark engine

A large number of frameworks are present in the market that can aid with the process of deployment. These tools help to automate the pipeline and deploy it. Such examples are:

- **MlFlow** is one of the most popular frameworks, as it´s use cases, are vast and provide great value, there are four main components in this framework. **MlFlow tracking** allows for model versioning, metrics management and store serialized models. **MlFlow projects** is a convention to package ML code in a standard format that allows it to be reproducible and reusable, it also allows to store chunks of code that addresses specific steps in the pipeline and then chain them together.**MLflow Models** helps deploy ML models in packaging standard. The model registry is the component that stores and manages a model lifecycle and captures all the metadata of the full lifecycle.
- **DataRobot** is a framework that significantly simplifies the training and deployment process by providing tools to train and compare different models against each other in real-time. It also provides model explainability by indicating what features contributed to the results.
- **Arize Ai** is a tool focused on monitoring ML models. It offers great tools for validation, model performance alerts, and prediction explainability. The most popular ML frameworks like Tensorflow, Pytorch, and Sk-learn all work with Arize, and most cloud platforms for ML are also supported.
- **OctoMl** uses artificial intelligence to optimize and benchmark user's custom models. One of the main difficulties in the deployment process is to port all the libraries used for the model and data into completely different hardware; OctoMl has tools to improve this process significantly.

2.5.4 Model benchmarking methodologies

Benchmarking a machine learning model can be a very complex process. At the same time, in an academic workload, results are the most important factor, in industrial and commercial applications performance of the model in given hardware also plays a big role. Hence, it is important to factor in the raw performance the model can achieve but also the cost in hardware it takes to run it in a real world environment.

The main costs associated with an algorithm is the training cost and inference cost. The training cost refers to the cost of making the model appropriate to deliver value to the customer. A complex model like a neural network has a huge training cost that is only feasible due to the recent increases in computation performance, particularly in graphics cards, (Zhu et al., 2018) looks specifically for benchmarking the training cost in neural networks. The inference cost is essential because an amazing algorithm that delivers great accuracy can be thrown out if it can't run in the market audience devices, so there is a need to assess the best performance/cost metric for any given solution provided. One solution for heavy algorithms is to run it through an API on an external server. However, this implies an always-on connection on the target device that may not be feasible and a big load on the host server instead of distributing that load on the application devices. All of this must be considered when developing a product at an Enterprise level.

Different ML algorithms require different benchmarks, a random forest needs to be benchmarked on the accuracy and performance to find the sweet spot, a bigger number of trees can increase the accuracy, but it may not justify the cost associated with running the model, similar thought processes need to be applied to different algorithms.

Neural networks analysis has a tool integrated into the *tensorflow* library called *tensorboard*, ("Tensorflow", 2017) allows for an excellent metric analysis for a specific model or to compare it with others; however, it falls short when comparing the actual cost of running such model comparing with others. In this article by (Coleman et al., 2017), a solution to benchmark ML models is provided called *DawnBench*, it analyzes the performance of the algorithm and the cost of training and inference, one particular studied case observes the changes of changing a parameter in convolutional neural networks called mini batches have in the time to train the network and the accuracy obtained. In one example, increasing the batch size from 32(best value for accuracy) to 256 reduced accuracies by 0.43 percent but was almost 2x as fast. This is the type of analysis that is really important when considering a mass production product.

("Baidu", 2017) has an exciting approach for inference benchmark that includes latency testing as a possible key metric. This is very important as some applications require a fast inference time, like vehicle collision prediction. Moreover, a few milliseconds can make a difference between the algorithm being effective or ineffective even if the prediction is correct, resulting in another key performance index to be considered.

2.6 Model explainability methodologies

Machine learning models can easily become black boxes that take an input and output a prediction, with the internal decision making process potentially being quite complicated. Linear regression, decision trees, SVMs (support-vector machines) are categories of models that can be easily explained with charts, for instance, with linear regression, that function line will immediately inform the decision for any given input. Neural networks are a completely different problem to tackle. The internal processes are too complicated to be easily explained in a chart or infographic, so they tend to be black boxes, this creates mistrust in the

model, and it makes it very hard to explain to someone without a machine learning background. There are some exceptions. ConvNets can be explained by highlighting the features in the image that the model considers essential. This process can be explained in figure 10, and the green areas are the features that activate on each square, meaning that in the end, depending on the pattern, a different output is defined. This is one of the simplest ways to provide explainability, but some models require or gain value with more advanced techniques.

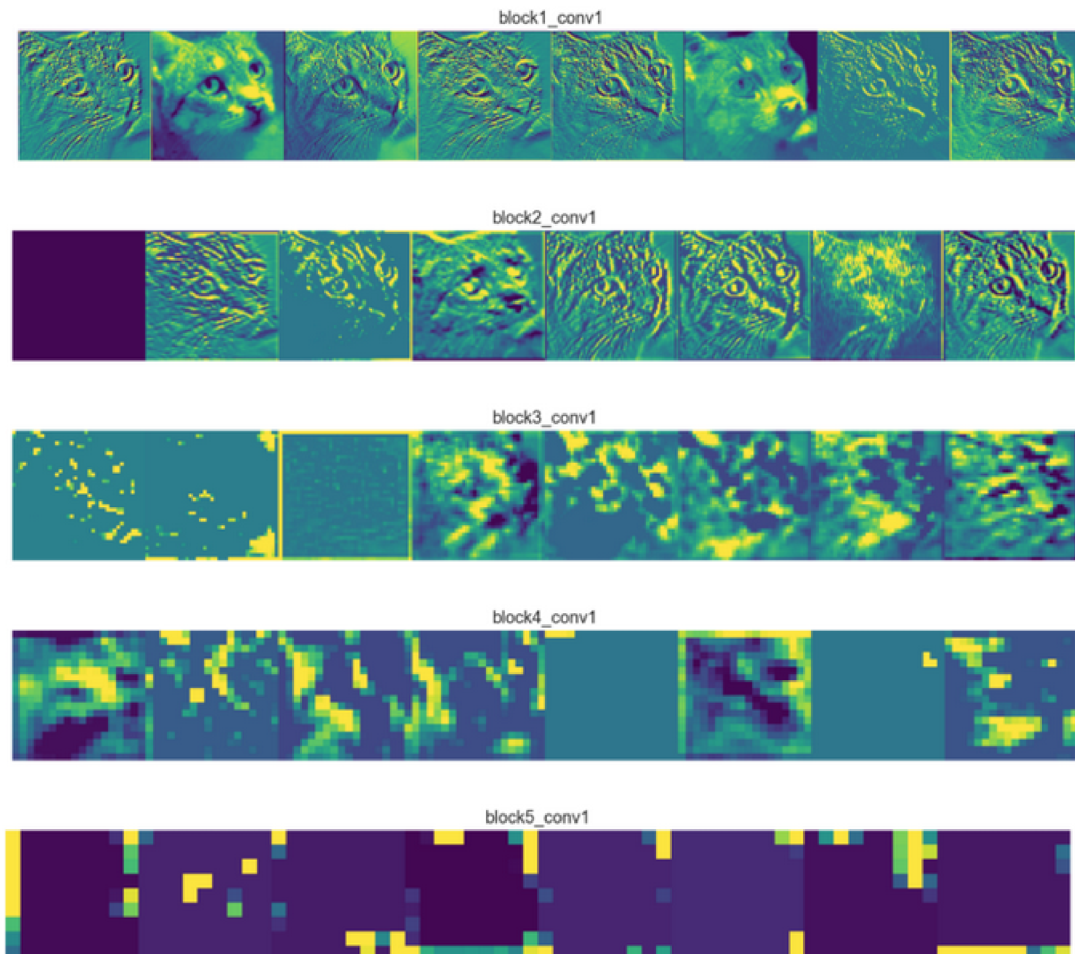


Figure 10: ConvNet layer explainability

2.6.1 Lime values explainability

Local Interpretable Model-Agnostic Explanations, or lime for short is a model-agnostic technique first introduced in (Ribeiro, Singh, and Guestrin, 2016), it can be applied to any algorithm and works by manipulating the data batch entering the model to see differences in the prediction outcome. Using this technique. It can estimate the relative importance each feature has for each prediction. It is a very powerful tool, as it matches how the human intuition perceives outcomes.

Humans predict outcomes by judging the environment and its features. If a patient has cough the disease can be a common cold, however if cough and fever are present it may be a flu virus. The feature fever changes the prediction. Using lime values, the data scientist can present a similar explanation for each outcome, and each feature importance for the specific prediction.

Lime has the advantage of being very fast to execute, however model interoperability does not mean causality, so it may be dangerous to change model training methods and features just by interpreting its explainability report.

2.6.2 Shap values

Shapley additive explanations is a methodology built on the Shapley value, introduced in (Lundberg and Lee, 2017). A Shapley value is the average of the marginal contributions across all permutations. This approach is united and provides both local and global interpretability for the results.

This methadology provides feature importance for each prediction, but it can also compute features statistics for the dataset as a whole, making it possible to generalize and correlate feature values and outcomes, for instance.

Shap provides the same advantages and risks as Lime values, it is better because it can also provide a global outlook of the problem, however it is significantly slower to compute which may be a problem depending on the use case. It is also important to understand the SHAP interoperability does not mean causality.

2.7 Model monitoring methodologies

The model is trained, tested, and ready to be deployed. However, due to the phenomenon of concept drift explored above, it is expected that it will start to degrade over time. Due to this complication, a good monitoring system for the model is just as important. Different algorithms may age better than others. Nonetheless, the problem affects them all because it comes from the data. As ML becomes more used at a large scale, many companies invest in monitoring tools and frameworks to solve the problem.

Unitary tests and integration tests only work for the standard software components. To observe the model behaviour in the real-world, the testing is directed to the data being feed to the model and the predictions the model is outputting. ((Sarnelle, Sanchez, Capo, Haas, & Polikar, 2015)) explore three metrics to quantify the deviation between drift rates. Kullback Leibler(KL) measures how far two distributions are from each other, formula in figure 11, This allows to compare generic data from training and the real world and try to quantify the drift rate happening in the real world.

$$\mathbf{KL}(X \parallel Y) = \sum_{x \in (X \cup Y)} \ln\left(\frac{P(X(x))}{P(Y(x))}\right) P(X(x))$$

Figure 11: Kullback Leibler

The Hellinger distance measures the similarity between two distributions. The formula to calculate the Hellinger distance is in figure 12, the formula returns values closer to zero if both distributions are very similar and closer to one when they are very different from each other.

$$\mathrm{H}(X,Y) = \frac{1}{\sqrt{2}}\sqrt{\sum_{x \in X \cup Y}\left(\sqrt{P(X(x))} - \sqrt{P(Y(x))}\right)^2}$$

Figure 12: Hellinger distance

The average pairwise distance iterates over all observation in both distribution and returns the average computed distance, the formula is in figure 13, euclidean distance is the distance formula in this case, but any formula that can predict the distance between observation work with the average pairwise formula.

$$D_{avg}(X,Y) = \frac{1}{|X||Y|}\sum_{x \in X}\sum_{y \in Y}(x-y)^2$$

Figure 13: Average Pairwise distance

In the literature, (Breck, Cai, Nielsen, Salib, and Sculley, 2017) explores interesting approaches like data statistical analysis and data schemes that include the normal range expected for the feature, this way if data starts deviating from the normal standard at the time the model was trained, a warning can be provided to the developer who can act and avoid malfunction in the system due to data drifting, this process allows scoring and quantify changes in data. However, metrics analysis needs to be observed as well, which present challenges. Furthermore, there is no way to compare the predictions with the ground truth as the data is not labelled in real-time and requires human intervention, so this presents challenges that need to be solved.

Data drift is a normal phenomenon that will always present itself over time, like for example consuming habits changing year after year. However, certain events can render an ML algorithm useless overnight, for example, a global pandemic that forced the entire population to use a face mask, making a huge shift towards online sales and led the global markets to an astonishing volatility rate, this led to huge implications in facial recognition applications, recommendation algorithms underperforming and stock market algorithms making huge mistakes.

2.8 Summary

In conclusion, developing an end to end ML application has many factors to consider. Adopting the best **MlOps** practices means a faster development cycle and post-deployment updates, the gains obtained with **MlOps** depend on the maturity of the implementation. To assist with the automation of the pipeline, many frameworks are present in the market that help the process. Each has its main advantages and disadvantages for each use case.

Choosing the best algorithm is complex, and good benchmark methodologies in the market allow for comparisons between different models. Some methodologies consider inference cost, training cost, or both, and depending on the use case all have their advantages and disadvantages.

Even the best algorithm won't stand the test of time as data drift can and will impact the model's performance. Many monitoring methodologies are being studied to combat this phenomenon, which applies techniques to control and signal potential data drift.

Chapter 3

Early work

A car has unique particularities for collecting data due to its fast pace movement, resulting in potential significant changes in the air quality and properties for the duration of the trip. This early work subsection intends on prototyping a solution for collecting vast amounts of data from real-world circulating vehicles, and study their properties to find value in them. This value can be to detect dangerous levels of smoke or distinct noise that can indicate danger to the passenger or physical harm to the vehicle.

Before the data is ready to be fed into Ml algorithms, many steps are needed, like collecting it, treating it, and ensuring reliability in the data. This process is aimed to be automated, so a pipeline among the algorithms and the data source is constructed.

After the model is trained, a pipeline between the training and deployment will also be constructed to streamline and facilitate deployment, benchmarking, and monitoring new models. This automation will lead to a much faster end to end cycle and provide the ability to update and maintain the Ml application easily.

3.1 Data collection - Hardware solution

3.1.1 Sensors and computational device

To collect data from the vehicle, a computational device and sensors are needed. Sensors collect raw data from the environment, decode the data, and record it to an appropriate storage solution. The following hardware was selected

- **Pm2.5 Sensor** is a particle sensor capable of detecting multiple sizes of particles in the air.
- **Uma8 microphone array** is a high definition 8 channel microphone array that that collects audio from the environment.

- **Bme680** is a gas sensor capable of measuring a wide range of gases, pressure, humidity, and temperature.

- **Raspberry pi4 model B** is the computation device that manages the sensors and collect the data from them in a database

These sensors were mainly selected because they can capture the data needed and were available in the company stock of sensors.

The computational device is chosen because it offers the computational power required and can run a Linux distribution that runs the software necessary to manage the sensors.

3.1.2 Casing solution

The results obtained in a given test vehicle needs to be reproducible and generalized for all vehicles. The sensors must be installed in identical locations with identical configurations to avoid different noise between different setups and to minimize noise from the environment. This requires a generalized setup.

To solve the installation problem, a casing design was developed using a 3d Printer. This design needs to allow for the correct accommodation of all the sensors. For instance, the microphone cannot have sound blocked, and proper airflow needs to be supplied to the gas and particle sensor; otherwise, they will not function correctly. Furthermore, the installation location is in the car windshield, so it needs to be attached without scratching or damaging the glass when installed.

With all the requirements addressed, the design process can start, a free CAD software was used, and a two-level design was conceptualized. This allows for airflow and sound to pass to the sensors while not overcomplicating the manufacturing(printing) process. The results can be seen in figure 14

Figure 14: Design's levels design

This design is printed and the final result is displayed in figure 15. The design provides a simple, effective, and pleasant casing solution that won't distract the driver and can blend in with the natural aesthetics of the car.

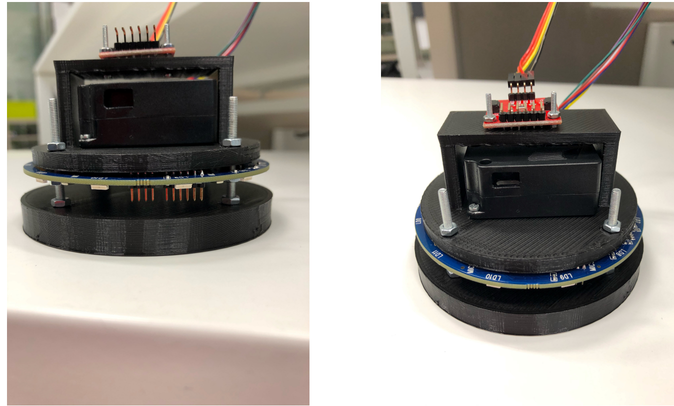


Figure 15: Printed multi level design

3.2 Software solution

To manage and coordinate all sensors, Robot operating system (Ros) is used. ROS allows us to treat each sensor as a node that can communicate with a central node, usually called the listener that subscribes to channels and receives messages from the sensor nodes publishing in those channels. This approach allows to achieve true sync between data from different sources, a time synchronizer filter is used to eliminate incoherent data that arrives in different time sources, the data is then stored in a file structure called *bagfile* that allows storing raw data with different formats which are highly beneficial since each sensor has a unique data schema, this architecture can be visualized in figure 17.

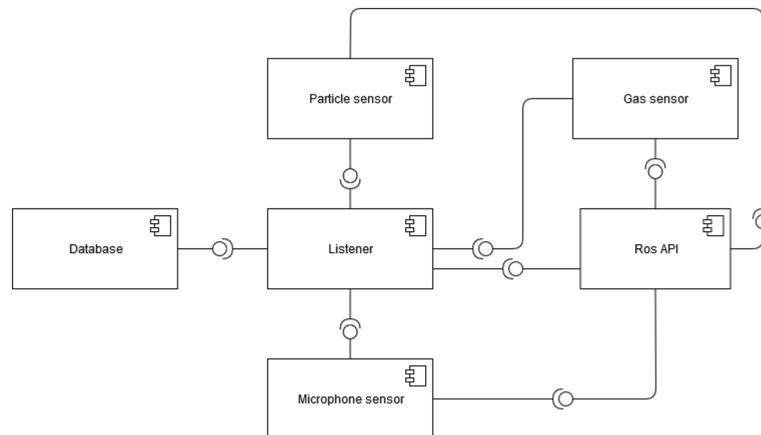


Figure 16: Component diagram for data collection tool

This application uses a launch script to initiate all nodes at the same time. The node sensor registers their channels in the ROS API, and the listener node subscribes to them and starts waiting for messages. The sensors start collecting data and are timed in predefined intervals to publish the data in the specific data schema. The listener hears the messages arriving in different channels and verifies the time synch between them to check if consistency is maintained; if everything is good, they are stored in a database or otherwise deleted. This flow can be observed in 17

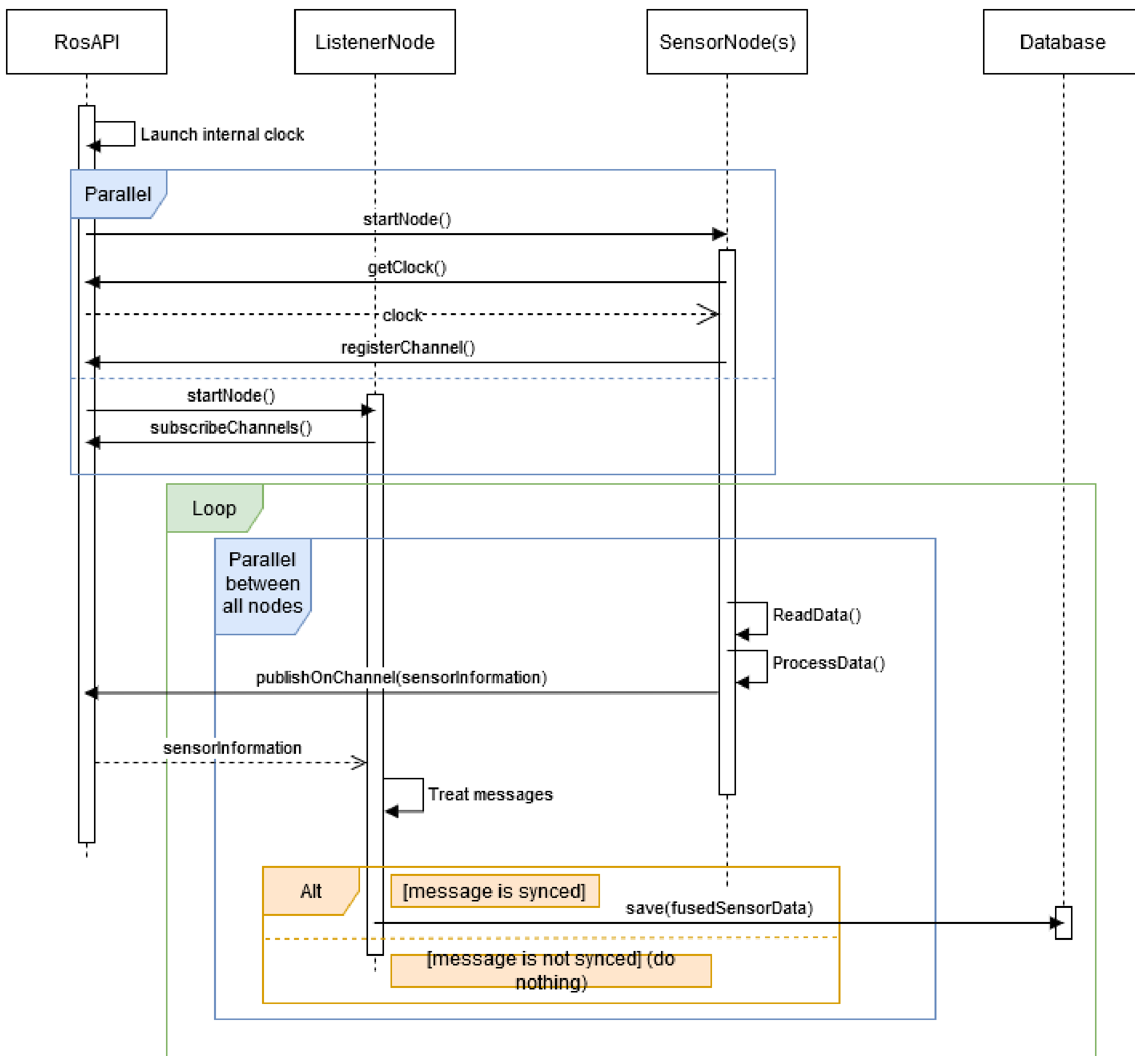


Figure 17: Activity diagram for data collection tool

3.2.1 Data visualization tool

The data collection tool is created and has a standardized output meaning the data treatment can be automatized, for this reason, the data visualization tool has requirements specified to allow the automation of data between the database and the visualization tool. To achieve this the tool must interpret and decode the *bagfiles* that are the output of the collection tool, then extract and treat the data by interpreting how many sensors are registered and give the user the choice to select the ones that have interest for the current analysis, then show the graphs and statistical information from the data, see figure 18.

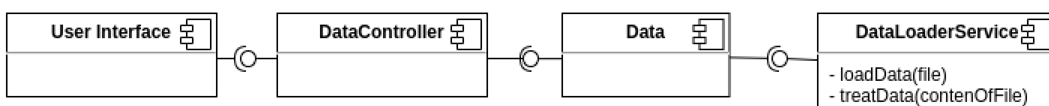


Figure 18: Component diagram for data visualization tool

This four components in figure 18 work as follows:

- **DataLoaderService** is a collection of functions that loads raw data and executes the necessary processes to make it usable
- **Data** is the component that represents the instance of data, providing functions and information on it's domain
- **DataController** is a comunication layer between the model class **Data** and the **UserInterface**, this apprach intents on centralizing all the requests.
- **UserInterface** is the visual interface of the software and makes interaction with the tool simplified.

This application is designed to be used through the user interface, all the functions are explained in the activity diagram in figure 19

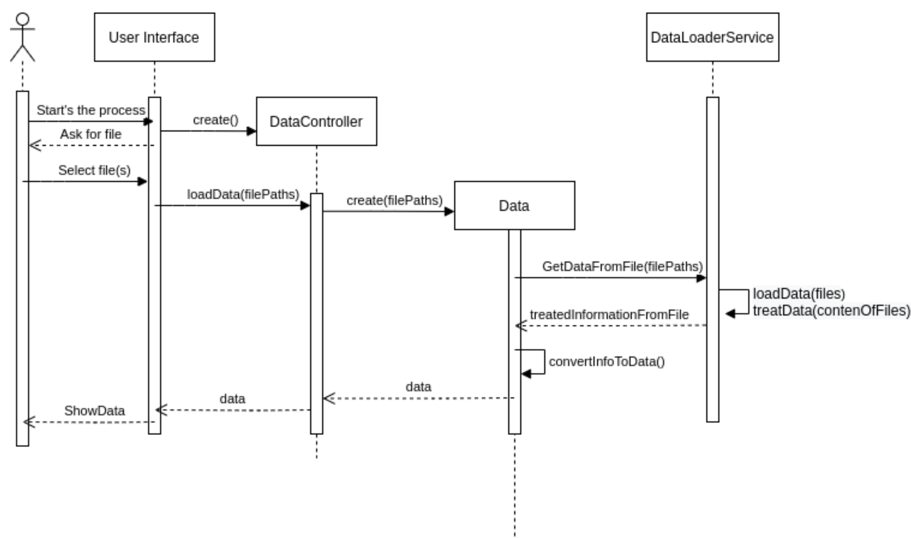


Figure 19: Activity diagram for visualization tool

Chapter 4

Objectives and results

4.1 Problem definition

The development of an end to end Machine Learning (ML) pipeline is a very complex process. A concept to evaluate the performance of algorithms shall be devised and implemented to support the choice of the best model and an inference engine that provide feedback for the predictions and assure model reliability.

The end goal of the thesis is to accelerate and improve the company already existing ML development pipeline by automating and improving the existing steps. The research found that the best model is a highly subjective matter, with the use case has significant implications on benchmark metrics. In this section, a few considerations are made to achieve the final goal.

4.1.1 Benchmark consideration

The ultimate problem for this project is to make impact detection and classification, this means the model receives data from the sensors and predicts if an impact happened and what type of impact to understand damages done to the car, based on this goal a better decision for benchmark metrics can be made.

In many ML projects inference times are an essential subject, however, the goal of the current ML implementations is to detect and classify damage impacts on a car exterior, since we have no intention on predicting upcoming events, inference times are not important. This means that the correct metrics are the single most important factor in evaluating models.

Different models with different role all flow from the pipeline and arrive at the benchmarking engine, each must take into consideration different evaluation metrics to correctly assess its usefulness and performance. The data scientist must be the responsible actor who chooses the correct metrics to evaluate the model.

The benchmark engine platform's role is to have a standard that allows for the benchmark of every metric and to display the relevant information on the dashboard. The web dashboard has the role to intuitively display model key performance indexes and showcase the best models for each model type present in the system.

4.1.2 Concept drift in the context

Data drift is certain to occur in every data-related problem, but it's essential to consider how it can affect the performance in this specific use case. In this context, data drift can occur due to several reasons, some can be easily identifiable and some not. For instance, if a sensor starts to misbehave, it can be easily identified with a monitoring tool. However, suppose the car manufacturing process or the materials it's made changes over the years to make it safer. In that case, the data to a similar incident can change drastically, like the sound of the collision or less g-force registering, making the model obsolete.

A correct monitoring approach must be deployed to detect the occurrence of data drift in the data received by the model, this can be done for instance by applying the Hellinger distance, figure 12. If the same type of collision data starts differentiating a lot from training to real-world, then this formula will start to output values closer to 1. A case study needs to be performed to discover the best metric to measure such deviations or if such metrics can be helpful in the dataset used.

This system is also responsible for keeping benchmarking each model has new data that corresponds to the original query is available. This process helps detect model performance over time and ensure the model is still performing as intended.

4.1.3 Continuous deployment and continuous training

Ensuring a pipeline capable of continuous deployment and training is essential. Due to the enterprise nature of this application, downtime must be reduced to the minimum, so it's important to design a pipeline that can perform updates in real-time to the model. So continuous training ensures concept drift doesn't make the model outdated, and continuous deployment ensures the model is updated. The benchmark engine integrated into the pipeline helps choose the best model in the current setting.

4.2 Objectives

The end goal is to design and build a pipeline capable of improving and accelerating the current ML development cycle in the organization, this implies to facilitate, improve and automate most of the steps and tasks already existing. In the end, time and human costs must be significantly reduced.

This thesis end goal is to provide a solution that can tackle the final steps of the pipeline. After the model is trained, the solution must automate every action necessary and display the final key performance indexes and comparisons between different models in the web dashboard. The **model benchmarking**

engine needs to work and communicate with the previous components from the pipeline and respect the standard defined by the team. To tackle the problem, three-component must be devised. These components are as follows:

- Model benchmark engine, capable of indexing, versioning and benchmarking each model that arrives while also performing continuous evaluations on existing models.
- Web engine, capable of displaying all information about each model and benchmark while also providing a comparison tool
- Database solution, capable of storing and managing all the information required by the system.

4.3 Requirements

To solve the problem defined above, a combination of three components will be developed, each to solve a set of challenges related to the end stage of the pipeline. Combined, they form the model benchmark engine referenced in this document. These tools address the general complication in ML development, from model training to deployment and monitoring.

The following requirements are defined for the system.

4.3.1 Functional requirements

The system must allow to:

- **Receive new models from the pipeline.** Models will flow from the model training engine automatically, a standard is necessary to send and receive models for communication and usability purposes.
- **Probe if new Data is available.** A model comes associated to a dataset using its embedded metadata and a dataset **Tag**, new data relevant to the model may be available with time making it necessary to index and process it for the appropriate models.
- **Trigger benchmark action automatically for new models or new data.** The first benchmark needs to be triggered when possible by the system, however this doesn't mean the end of the model evaluation process, each model is set to be continuously benchmarked as new data is made available.
- **Store all the relevant metadata about each model / benchmark / dataset.** The end user must have access to all relevant model information, this information is processed and stored in the database for further utilization.

- **Compute all metrics defined by each model.** Each model has its metrics computed after the benchmark process and the data stored in the database.
- **Filter relevant models for comparison.** A filtering tool capable of selecting models by name or type is required.
- **Filter results by dataset used.** Finding the best performing model for a given dataset is essential.
- **Provide statistical information for each dataset.** Each dataset must have a dashboard that displays statistical information about it, like damage events, non damage and cars present in the dataset.
- **Manual CRUD operations to the database.** Since the system is fully automated, manual access to the database is required to remove unwanted information.
- **Provide model explainability using SHAP values methodology.** Model explainability is an important resource for a data scientist to debug and exploit additional information about a model after the benchmarking process terminates.
- **Continuous benchmarks capability for each model as data is made available.** The system must have the capability to benchmark each models as new data is available and show the model progress in function of time.
- **Comparison charts and tables between different metrics for each benchmark.** Charts to simplify the visualization of a benchmark.

4.3.2 Non functional requirements

- **Interactive and simple to use web dashboard.** The system must be designed to be as simple as possible and not requiring a learning curve.
- **Manual model insertion into the system.** A manual model insertion function must be present for models that do not come from the pipeline and need to be evaluated by the system.
- **Automatic system restart after malfunctions.** Automatic system recover after exceptions or hardware crashes is required.
- **System documentation** System documentation is required due to the continual update of the project and possible introduction of new developers in the future.
- **Exceptions treatment** Exceptions should be treated and managed in order to avoid application malfunction and downtime.

- **Correct hardware split with other components.** The Atlas pipeline is a complex system with multiple components running in the same server, correct resource management between components is a requirement.
- **Async benchmark functions.** Functions between different components need to run asynchronous, as to not slow down or stop the system.
- **Parallel benchmark execution.** Benchmarks can take a long time to process, to optimize the hardware capabilities multiple benchmarks can execute at the same time.
- **New and different model algorithms in the future** New ML algorithms may be added in the future, and it is important to leave the system ready for this potential changes.
- **New metrics in the future.**. Metrics are very subjective and relative to each model, it's important to include flexibility and design a system that allows the data scientist to use the metrics he sees necessary for any given model.
- **Scaling for new features.** Additional system functionalities may become necessary in the future, modular and upgradable architecture is a requirement.
- **Logging system for functions, errors, and crashes.** System crashes need to be debugged, full logs for each crash, exception, and even relevant system functions need to be present.
- **Web dashboard**. A web dashboard makes the system portable and easy to use in any device, the system is compiled and hosted in a single server and able to be used in the entire company.
- **SQL storage**. Ideal for relations between the different tables in the system and licensing terms.
- **Containerization framework**. Easy to be deployed everywhere without effort with all the prerequisites automatically installed.
- **LDAP authentication**. Already used by the company, easy to be integrated with current systems.
- **Interaction with data ingestion engine interface and comply with the standard.** The data ingestion engine serves as the data API and data management system for the entire pipeline, complying with its standard is essential.
- **Interaction with model training engine and comply with the standard.** The model training engine trains and delivers the ML models, complying with its standard is essential.
- **Bosch's hardware and systems.** Each company has its deployment hardware to be utilized.

The requirements allow understanding the client needs for the project, in this case the ML engineers. With the requirements correctly assessed the design and development can be planned to correctly meet expectations, but also predict and allow the application to scale for future demands.

Chapter 5

Design and architecture

This chapter documents the goals, use cases, requirements, software planning and development for the final solution.

5.1 The benchmarking engine in integration with the Atlas pipeline

This system is composed of three individual components that together compose the Atlas system pipeline. The Atlas pipeline is an end-to-end solution designed to accelerate the Artificial Intelligence (AI) development cycle, and it consists of :

- **The data ingestion engine**, this component serves as the ingestion tool for all the data collected by the team. After a collection exercise, the data is deposited in the engine, which indexes and manages the data into an internal SQL database. This database supplies a custom query engine through an API that can provide all the data requested using the custom query language. It also solves the problem of reproducibility. The moment a dataset is built using this system, it can be reproduced forever.

- **The model training engine** The model training engine receives a set of parameters from a configuration file and automatically trains new models, it fetches the required data on the data ingestion engine and uploads the models to the model benchmarking engine for evaluation, this system serializes the models and their internal pipelines using a contract standard that makes standardization with all different models possible.

- **The model benchmark engine** is the final component in the pipeline. Its goals are to automatically interpret the serialized models and benchmark them when outputted from the model training engine, while also providing the option to upload models manually. Each benchmark uses the provided dataset, which dynamically updates with time as new data is available respecting the original query. This means the models are continuously evaluated through time. All the benchmark information, key performance indexes and model information is then stored on a database and served on a web dashboard which also answers the best performing models for each model type and each dataset that is used, thus making it easier to compare performance between different models.

These three components are represented in figure 20. This doesn't represent the flow of information as all three components communicate with each other but the flow of processes in a regular Machine Learning (ML) pipeline.

Figure 20: Atlas Pipeline flow

The component diagram in figure 21 depicts the individual components that partake in the pipeline. As can be seen, the ingestion engine is consumed by both the training engine and the benchmark engine. A query is made by the training engine, which generates a training set and evaluation set. When the model is trained, it is sent to the benchmark engine, which consumes the tag and asks for the evaluation set to the ingestion engine corresponding to the query ID associated with the tag. The end of the pipeline is a dashboard provided by the benchmark engine that serves all the data about each model, provides a comparison tool that can filter and display the best model for each use case and each dataset. The integration between the three main components needs to feel seamless, so a rigid communication standard needs to be defined.

The activity diagram on figure 22 showcases how the system will be used, the user, which will mainly be a machine learning engineer or data scientist. The activity sequence starts with introducing collected data in the company repository. The ingestion engine automatizes the following processes of data persistence and indexing, saving time for the user. The next primary interaction with the system is in the model training engine, where the user defines the training specs for the ML model and, using the data ingestion engine's query language, creates and tags a custom dataset for training and evaluation. This information is submitted, and a model training is scheduled without further user input. The trained model is then submitted to the model benchmarking engine, where the model is indexed and scheduled for benchmarking. The benchmark metadata and metrics are stored in the database to be shown later on the dashboard. The user's only interaction with the model benchmarking engine is with the end dashboard, where all

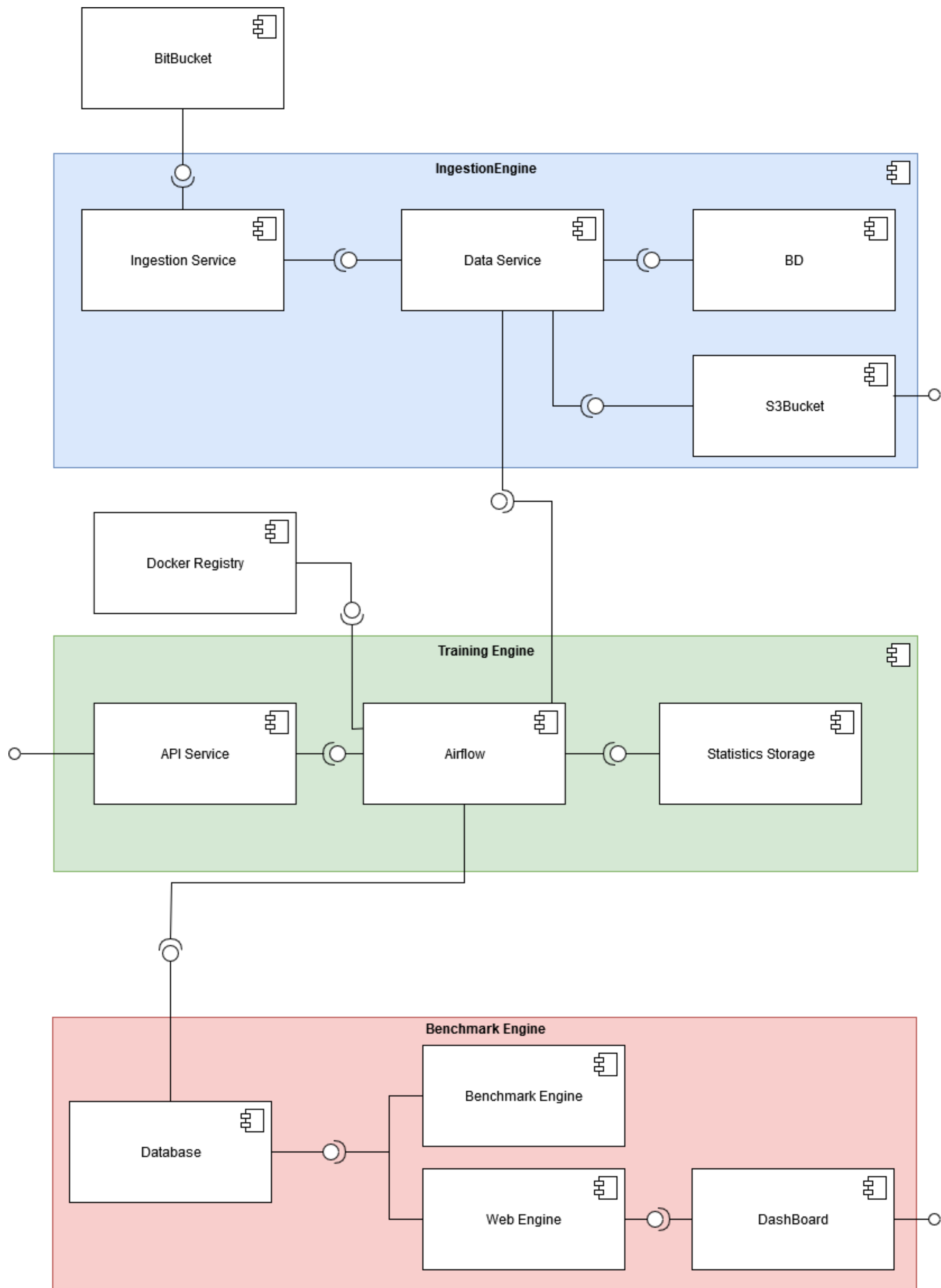


Figure 21: Atlas pipeline component diagram

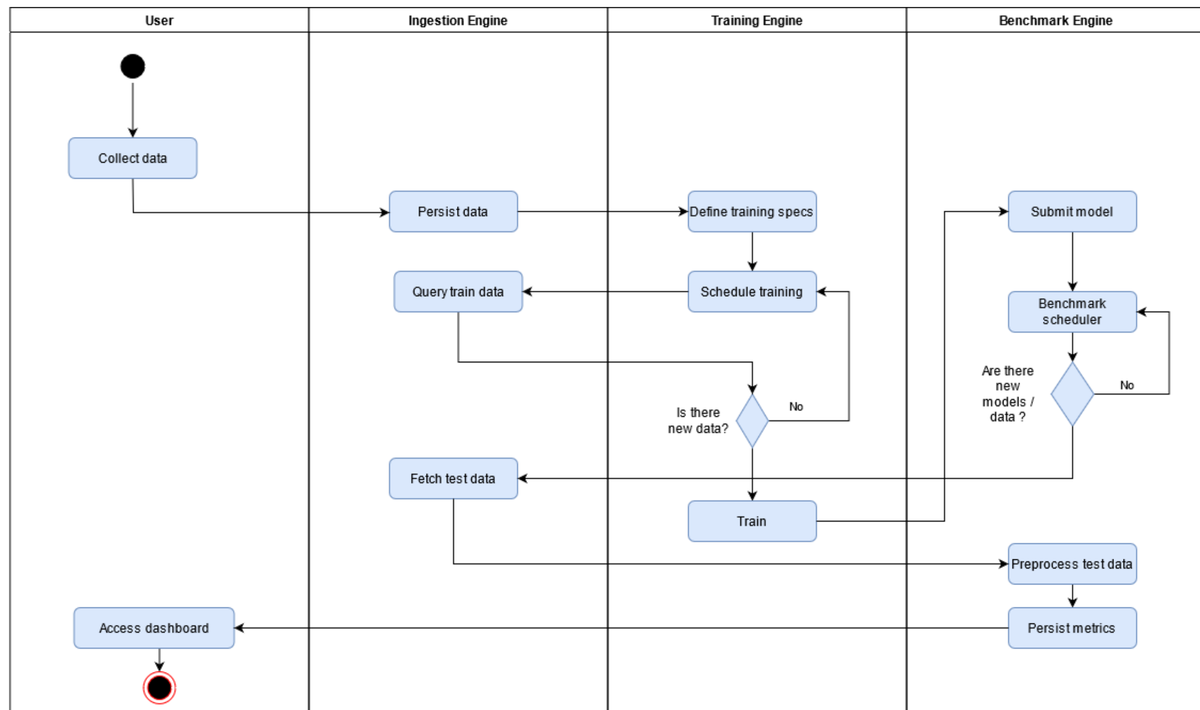


Figure 22: Atlas pipeline activity diagram

the information is present, and all remaining activities and functions are fully automatized. Following this design, the ML engineer/ data scientist has three main inputs:

- Inserting data from the data collection exercise.
- Defining training specification.
- Defining datasets.
- Consult the model benchmark dashboard.

Each component is implemented as a piece of the pipeline but can be utilized individually as well. This thesis will focus on the development of the final component, **model benchmarking engine**.

5.2 Model benchmark engine design and architecture overview

The model benchmark engine is the component where this chapter and thesis will focus, this component solves and automates the evaluation / benchmarking action on ML models and also continue to benchmark models as time progresses and new data is available. This is called **continuous benchmark** in the literature and is helpful to detect data drift and model decay that usually happens with time. Therefore, all machine learning models and datasets referenced in the following sections imply their origin from the pipeline's previous components.

5.2.1 System use cases

Considering the problem defined, the user of the model benchmarking engine, a data scientist, does not have any interaction with the underlying system and only consults the information outputted and provided by the web dashboard. The user has the following use cases:

- **UC1**: The user must be able to manually insert or delete models from the database.
- **UC2**: The user must be able to access the database and perform CRUD operations.
- **UC3**: The user must be able to filter each information table in the dashboard.
- **UC4**: The user must be able to order each information table by column.
- **UC5**: The user must be able to compare different model benchmarks.
- **UC6**: The user must be able to have results filtered by dataset.
- **UC7**: The user must be able to see all the metadata collected about the model.
- **UC8**: The user must be able to access a model explainability dashboard.
- **UC9**: The user must be able to access dataset explainability dashboard.
- **UC10**: The user must be able to access all logging information in the system.
- **UC11**: The user must be able to download individual models.

Most of these use cases require that the automated underlying function work as expected. The system is responsible for receiving new models, storing the metadata and dataset information in the database, performing benchmark operations, and storing the metrics. Finally, it also runs operations to compute *SHAP* values and charts.

The main actor of the system is the data scientist, which is denominated as the main user of the system. Most features and use cases implemented have the purpose to improve the data scientist ability to gather knowledge about each model and the decision-making process.

Each Use Case (UC) is explained in the following use case diagrams. For example, in figure 23 the **UC1** is explained, the model is inserted using the web dashboard which uploads the file to an object store solution, an internal scheduled process fetches for new models in a specified interval and if it founds one makes a request for the file, it then parses the embedded metadata and stores it in the database respecting the data schema defined.

The sequence diagram in figure 24 showcases manual CRUD operations to the database, **UC2** that may be necessary to perform. The web dashboard provides the interface for these operations, and when confirmed, the changes are executed in the database engine, making the process as user-friendly as possible for the user.

All data tables in the web dashboard must be dynamic. This means each column must have the possibility to order by ascending or descending and a custom search bar to filter the content in each table. **UC3, UC4** is described in the sequence diagram in figure figure 25.

Another essential use case is **UC5 and UC6**. A tool must be devised that allows comparison between benchmarks of the same model or different models, and it is also important to filter all results by dataset to

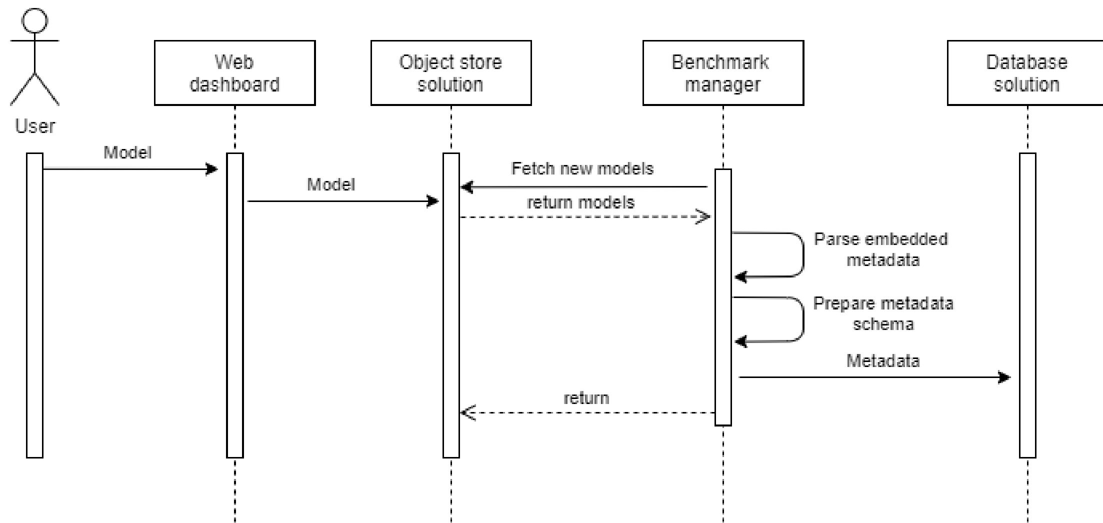


Figure 23: **UC1:** Sequence diagram for manual model insertion

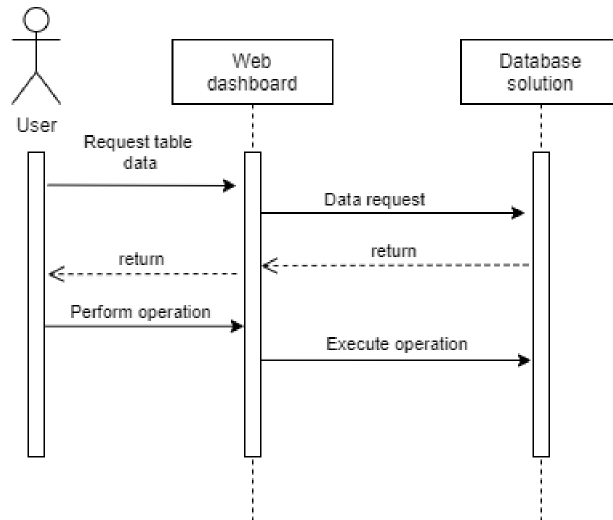


Figure 24: **UC2:** Sequence diagram for manual CRUD operations

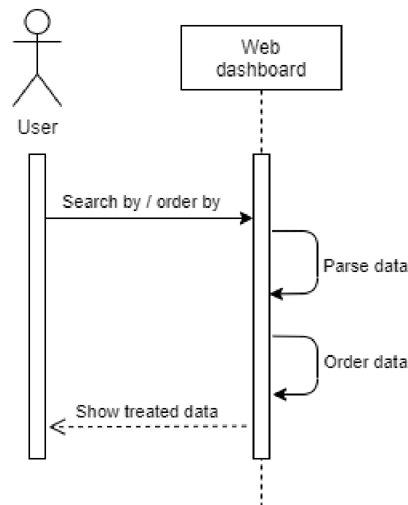


Figure 25: **UC3, UC4:** Sequence diagram for data filtering

find the best performing model for a given dataset or the best performing model for a project use case like a damage detector or damage locator, the sequence diagram in figure 26 explains the intended sequence for this process.

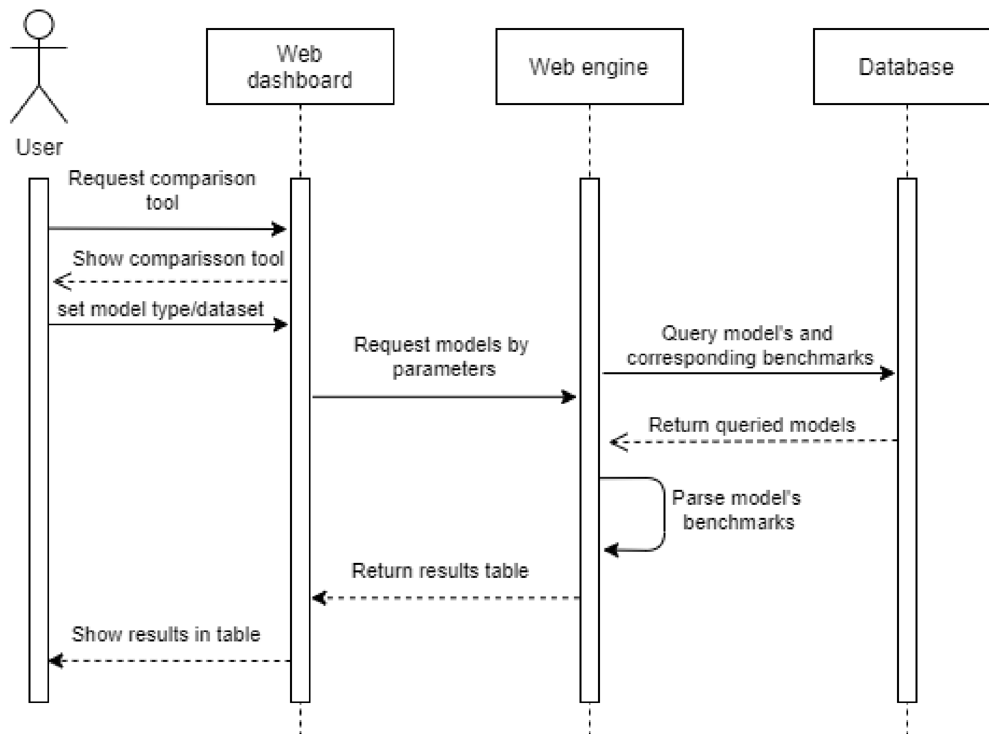


Figure 26: **UC5, UC6:** Sequence diagram for benchmark and results filtering

UC7 describes the process of a user getting all the information available about a model. This includes all its metadata like the name, timestamp introduced, and all the benchmarks information performed to the model like the metrics results, datasets used, distribution matrices and more. This information can be consulted by requesting the model page, and the process is described in the diagram in figure 27

UC8 is about the user requesting and accessing a model explainability page. Explainability refers to the model hyperparameters and tuning information and the SHAP explainability charts explored in state of the art. Each model should have SHAP explainability. The process of displaying this information is in figure 28. This information should already be computed after the benchmark is performed and the data stored in the database.

UC9 UC9 is a feature requested so the end-user can understand what is inside a specific dataset used for a benchmark. Although the process is explained in the sequence diagram of figure 29, this is different from other information pages, all data statistics are computed outside the *benchmarking engine* component and instead computed on the **data ingestion engine**, due to interoperability requirements between the two, a standard must exist to guarantee the correct dataset is requested.

UC10 is a use case requirement because the end-user will only access the web dashboard. If an error happens in the background components, it is important to have access to the logs. Logs include system failures, exceptions and also critical performed operations like benchmarking being executed on a model.

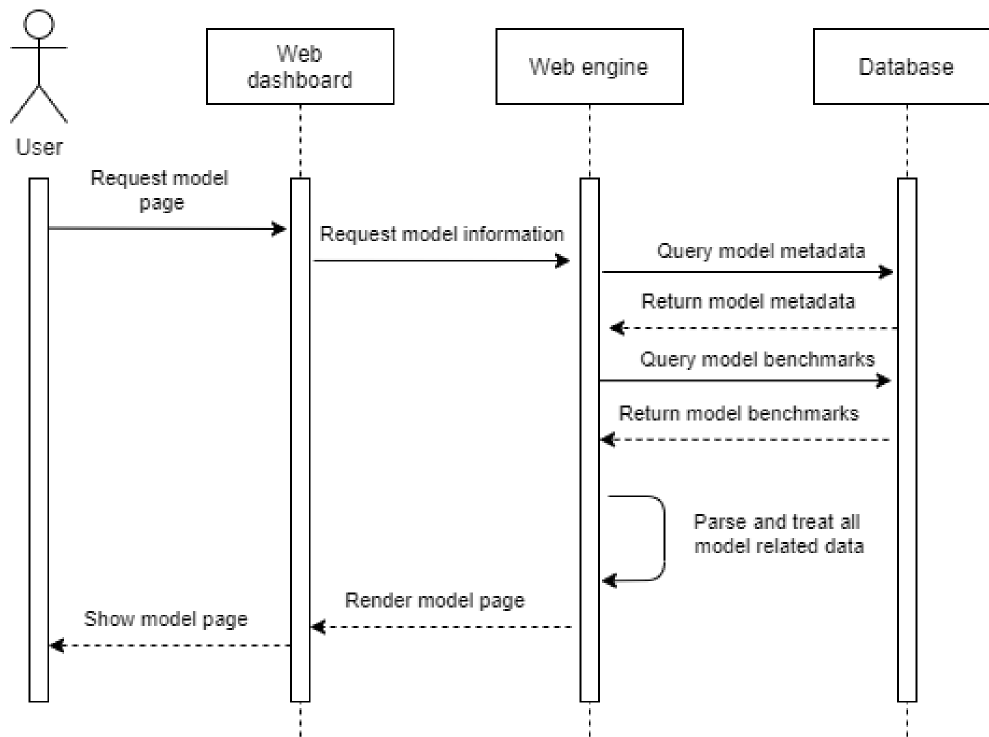


Figure 27: **UC7:** Sequence diagram for ordering a model page.

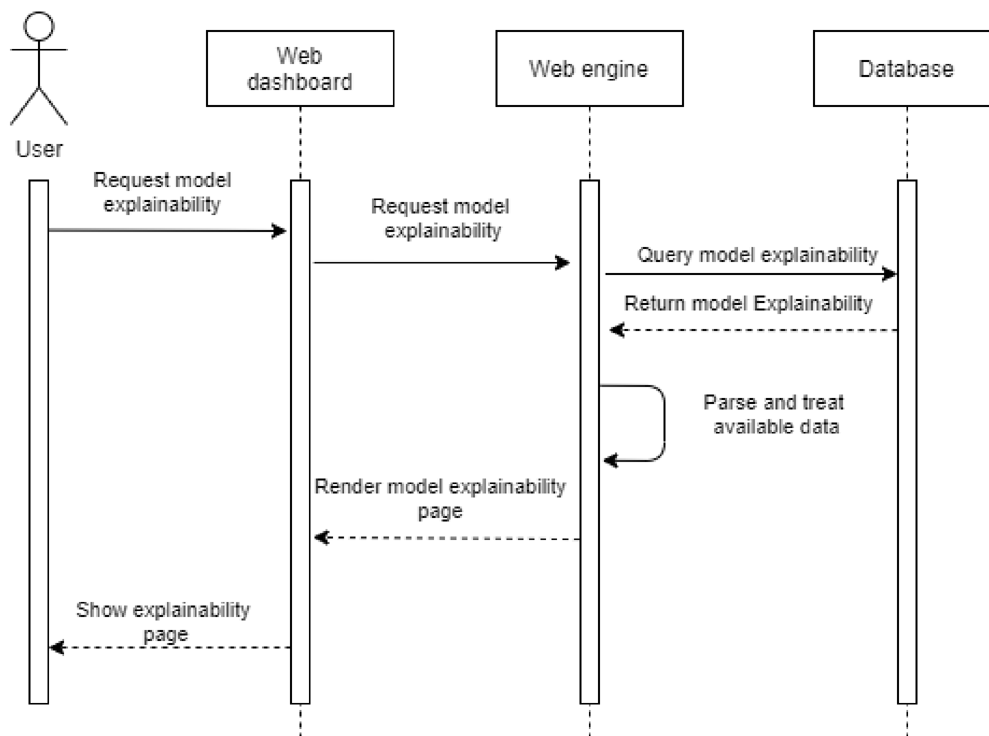


Figure 28: **UC8:** Sequence diagram for ordering an explainability page.

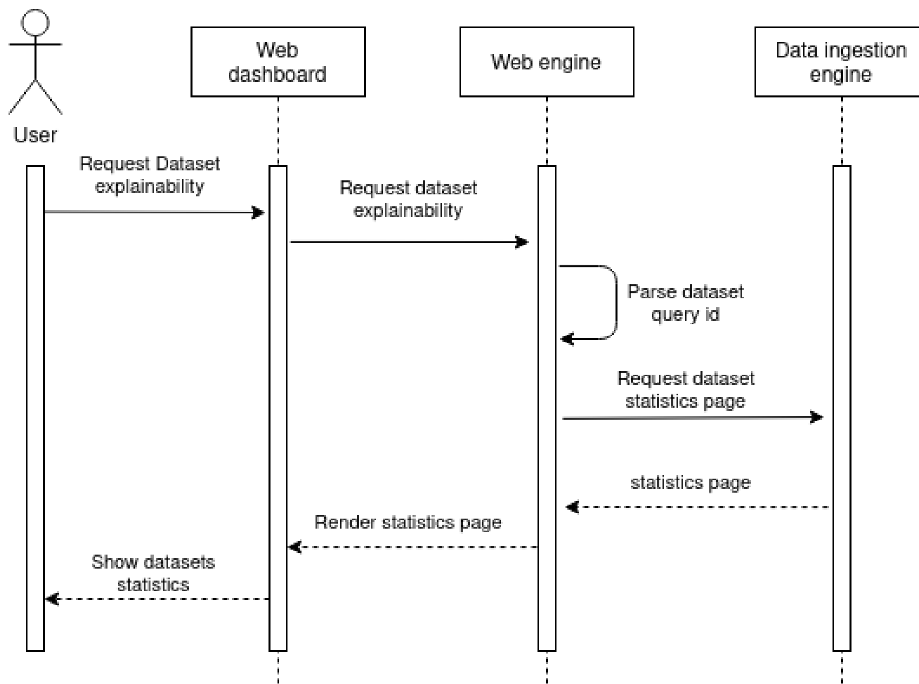


Figure 29: **UC9:** Sequence diagram for ordering a dataset explainability page.

The sequence diagram in figure 30 explains the log collection and storing process, and the dashboard provides the information.

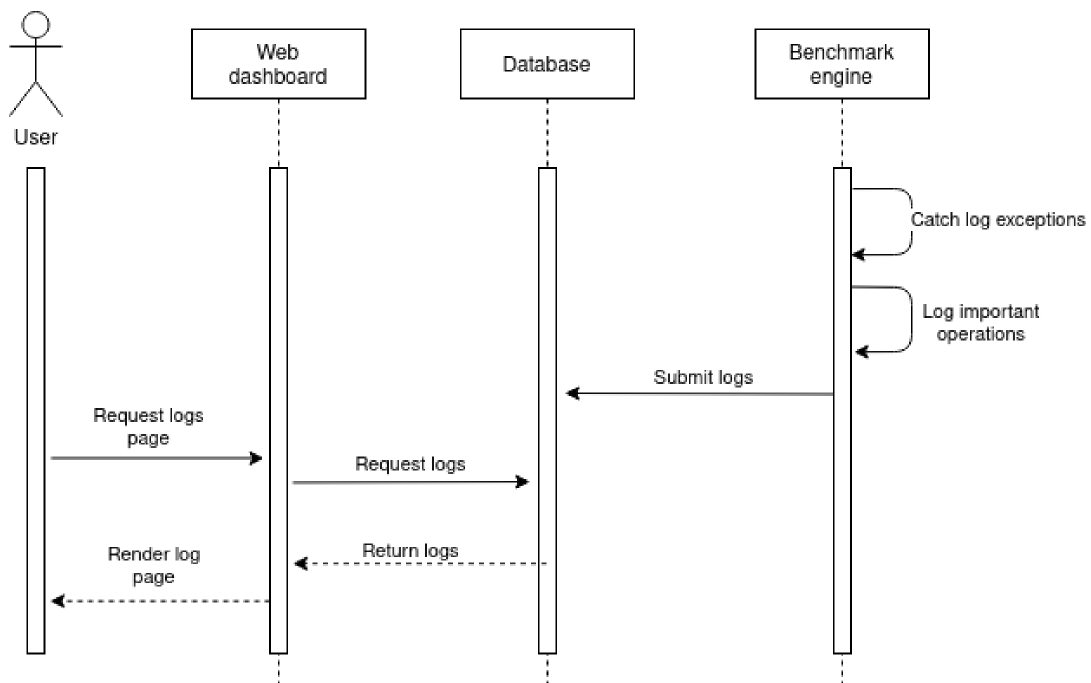


Figure 30: **UC10:** Sequence diagram for obtaining logging information.

UC11 is a part of requesting the model page, so figure 27 explains the process and then a download button is available in the model page.

To achieve this set of functionalities described in the use cases, the sequence diagram in figure 31 showcases the steps needed and the order to be executed by the system. All three main pipeline components are present and represented inside a rectangle which is an entity. The sequence starts with the model training engine submitting a trained ML model to the Model benchmarking engine. This system indexes the incoming model in a database to be picked by the benchmark engine to be evaluated. The correct dataset tag is in the model dataset, so a request for the evaluation dataset corresponding to the tag is sent to the data ingestion engine, which returns the data, the benchmark is performed, and the resulting metrics are stored in the database. Finally, all the data in the database is available to consult in the end-user's dashboard.

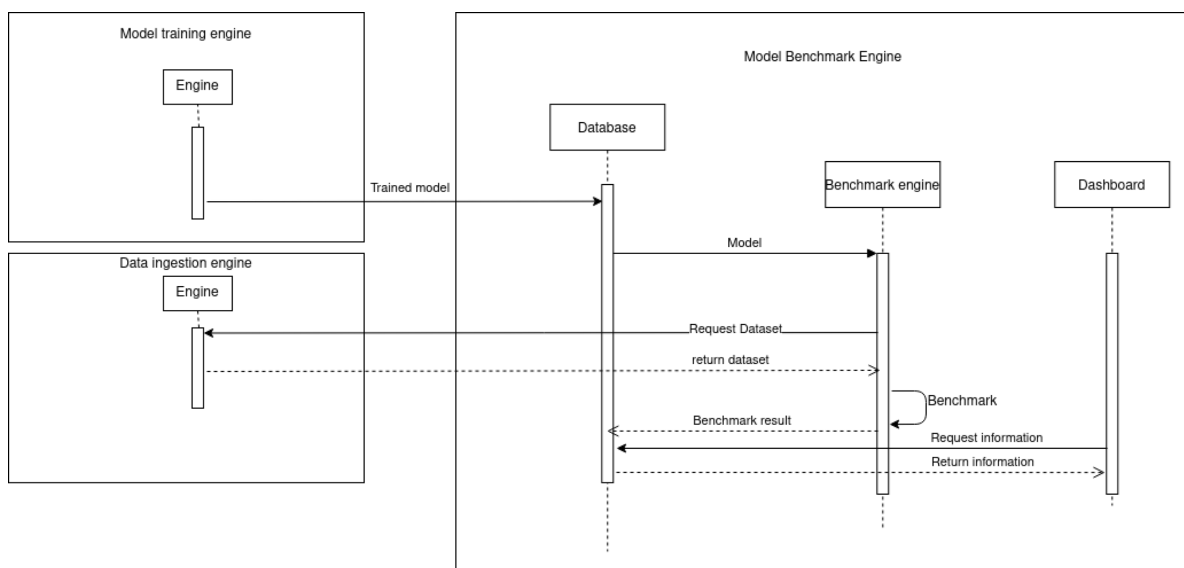


Figure 31: Model benchmarking engine internal process

With all the automated steps described in figure 31 a pipeline introduced model is benchmarked. However, one of the use cases implies that the user can also occasionally insert models that weren't produced by the pipeline. The sequence is described in figure 32. It is important to understand that a model from the pipeline comes with predefined metadata, some necessary and some optional, that should also be present in manual inputted models, or else the system may be incapable of performing an automatic benchmark.

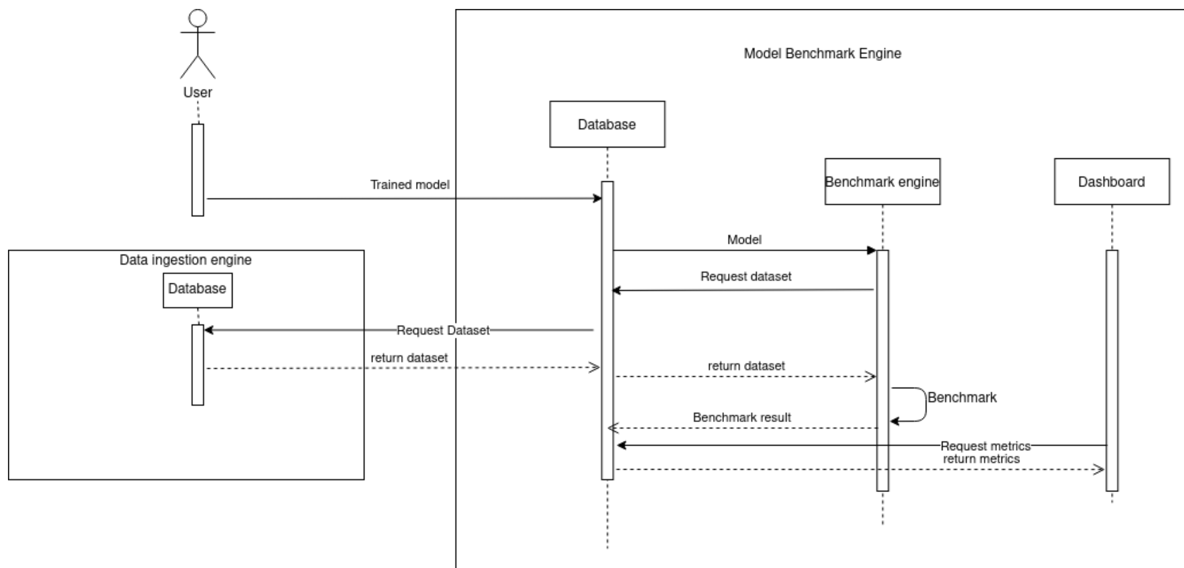


Figure 32: Model benchmarking engine, manual inputted model

An important use case is to quickly produce value from the data stored on the database to the end-user. The dashboard must provide the tools to parse the data and display it so the data scientist can see and perceive the information as efficiently as possible, for instance, finding the best performing algorithm for a specific use case or the best algorithm in a specific dataset.

5.3 Solution architecture

Inside the model benchmarking engine, three main components are designed to answer the project requirements and use cases. This section specifies and explains the design choices associated with each.

The first component requirement is to have a storage solution for the data associated with the system, the **database** component is designed specifically for this problem.

The second component requirement is to have a **benchmarking engine** capable of managing all the operations related to a machine learning problem, fetching, scheduling and updating benchmarks for each model automatically without user input.

The third component requirement is to have a full **web dashboard** capable of answering all the questions about the collected data simply and effectively. The dashboard container accomplishes this.

To understand the general scope of each subcomponent in integration with the pipeline, the building block diagram in figure 33 explains the scope in three layers, each diving in more detail.

In layer two, we have a more detailed look into the three components that together compose the model benchmarking engine. They are the database solution, benchmark engine and the dashboard. Finally, in layer three, we have a more detailed look into each component.

The following subsections explain in detail the functioning of each subcomponent.

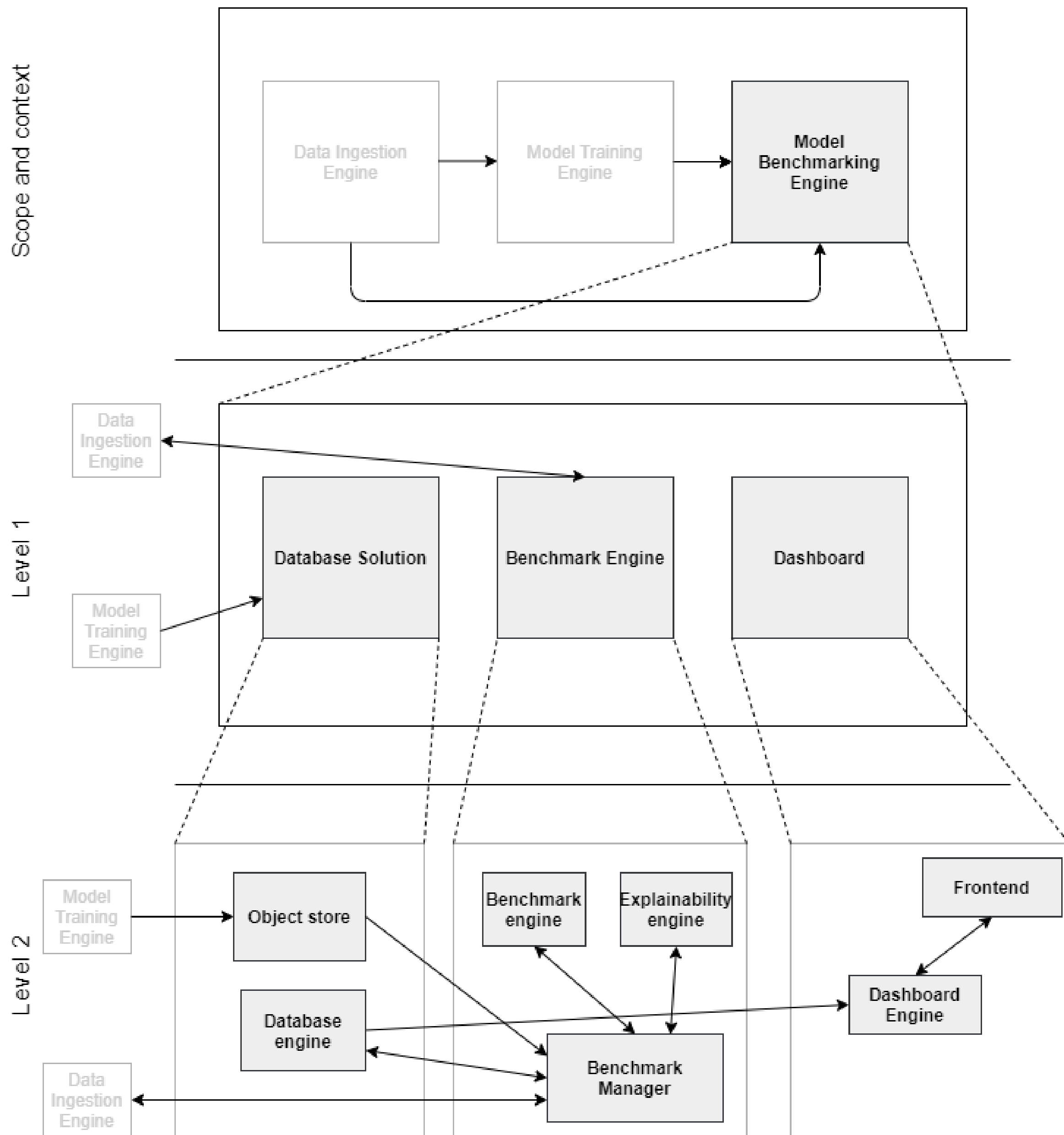


Figure 33: Model benchmarking engine, building block diagram

5.3.1 Database solution, design, and architecture

The first component is responsible for the system data storage solution. When considering a database solution, many factors are involved in the type and structure of data stored and the optimization of each database engine. The following question help to decide on which solution to choose:

- Is the data scheme fixed or will it change?
- Is there relations between different tables in the database?
- How much data will the system store?
- Does the database engine scale as required?
- Is the database engine open source or does a licensing agreements with the company exists?

Following these questions, the solution can start to be formulated. This problem is defined in an object-oriented language. The class diagram in figure 34 helps to understand the data structure and relations that exist. The scheme is expected to be mostly fixed so that no significant migrations will happen to the database with time. Most tables have relationships between each other, meaning a relational database engine will be ideal when only considering this fact. Full data size isn't expected to be an issue, as actual model files will be stored independently.

After consideration, the PostgreSQL database engine is chosen as it meets all the requirements. However, models files still need an appropriate storage solution. This solution is an independent object storing service called minio. Minio is an open-source, high performance and cloud-based storing solution that provides all the requirements for all pipeline components, so all the data lake brute files will also be stored in a minio bucket. Furthermore, since some datasets can reach multiple gigabytes of size, requests need to be processed as fast as possible, minio can write/read at up to 183 GB/s and 171 GB/s respectively.

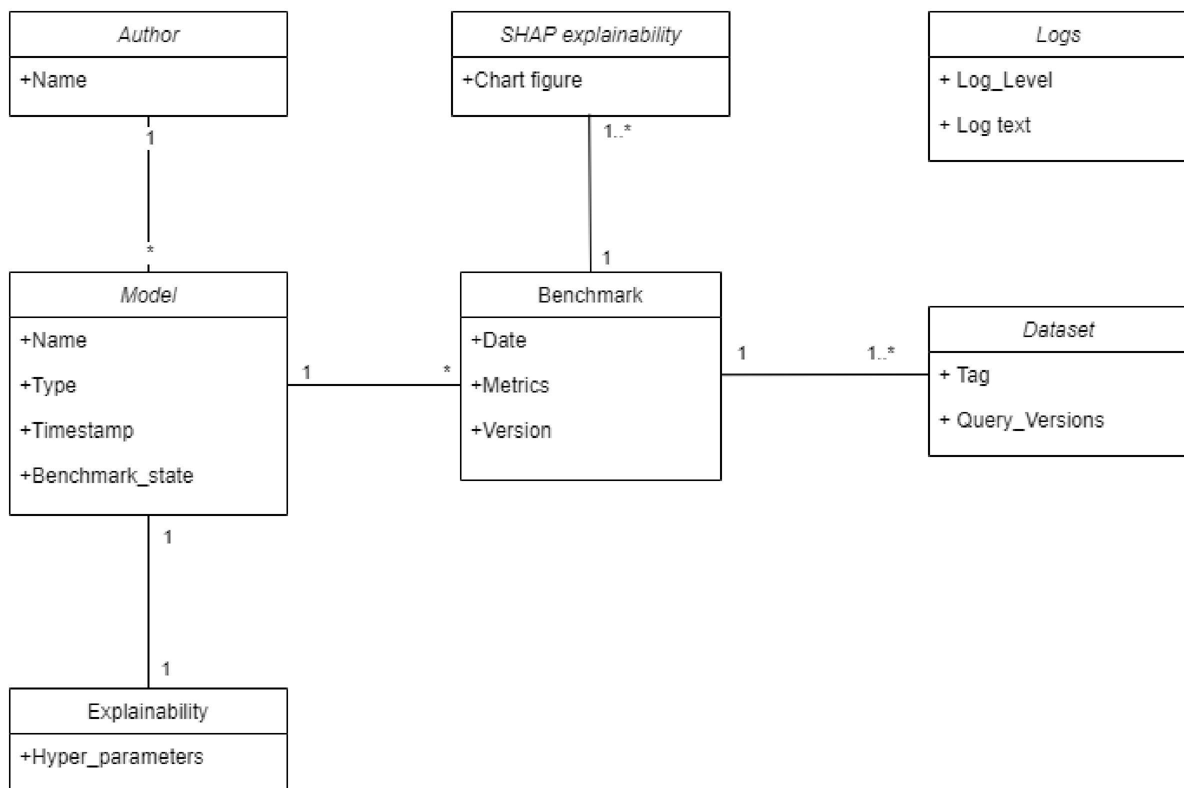


Figure 34: Benchmarking engine class diagram

Following the diagram in figure 34. Each model has associated a unique author and an '*explainability*', which is where the different model hyperparameters are stored. Last but not least are the benchmarks performed on the model, with each benchmark having associated a dataset and its SHAP explainability charts. Log tables are also present to store system operations, malfunctions and exceptions. The data structure is not very complex but achieves all requirements, and with a unique model file, the component should be able to fill all this information automatically.

The final consideration is for the interoperability standard between the database and the object storage solution, the model name works as the unique ID for each model and is stored in both storage systems. When a system wants to pick a given model, it asks the file storage solution for the model name stored in the relational database, and it returns the model file. Since model names must be unique the system guarantees the correct functionality.

Finally, after a model is trained in the previous steps of the pipeline, it is inserted into the object storage solution. Then, an automatic process is scheduled to run in a given time interval which checks the models inside the object solution bucket and the ones indexed in the database. If new models are found, they are added to the system.

The database solution is made of two distinct components as discussed above and shown in figure 33 on level 2 of the diagram:

- **Database engine** - PostgreSQL

- **Object solution** - MinIO

These two components connect to all the other internal components, as shown in the next section, and an external pipeline component, the **model training engine** that submits trained models automatically to the object storage solution.

5.3.2 Benchmarking engine, design and architecture

The benchmark engine component has one main goal to solve, the continuous evaluation of machine learning models. An evaluation is called a benchmark and is performed using the model to perform a set of tasks. A benchmark consists of :

- **Model metrics:** These metrics are defined by the data scientist that created the model. After the model predicts all the data in the test set, these metrics are computed.
- **Informational metrics:** These metrics refers to the additional information collected when benchmarking a model, like time clock for the evaluation, number of predicts.

All models are benchmarked at least one time, and that is when they arrive at the system, an initial benchmark runs using the given test-set tag, with time if new data becomes available that respects the initial test-set tag, a new benchmark will run and ensure the continuous evaluation of the model.

5.3.2.1 Defining standards and interoperability

An important tool for this mechanism is the **tag**, so it's important to understand how it works. The **data ingestion engine** designs the tag. A tag is a human-readable ID defined by the user who creates a dataset. The dataset is created by providing a query using the designed query language. This system allows to always reproduce a dataset as long as the original tag is known, it also allows to update the test-set as new data is available that respects the original query, when the tag is inspected, it may return multiple versions of the dataset because the system automatically versions the tag.

Standardization between all arriving models is a fixed requirement. Without a proper standard, there is no guarantee that a model can be benchmarked, each time a model is referenced, it is not just a conventional machine learning model but a serialized file that contains much more information and functions than just an ML model.

Figure 35 explains what is inside a model, there are three main things, the first is the **Atlas metadata** composed of the dataset **tag** and the **author** information. The second is the actual machine learning model, and the third is the data processing pipeline capable of transforming the raw data into the processed data the particular model requires. These processes are all standardized and work with every ML model that arrives in the system.

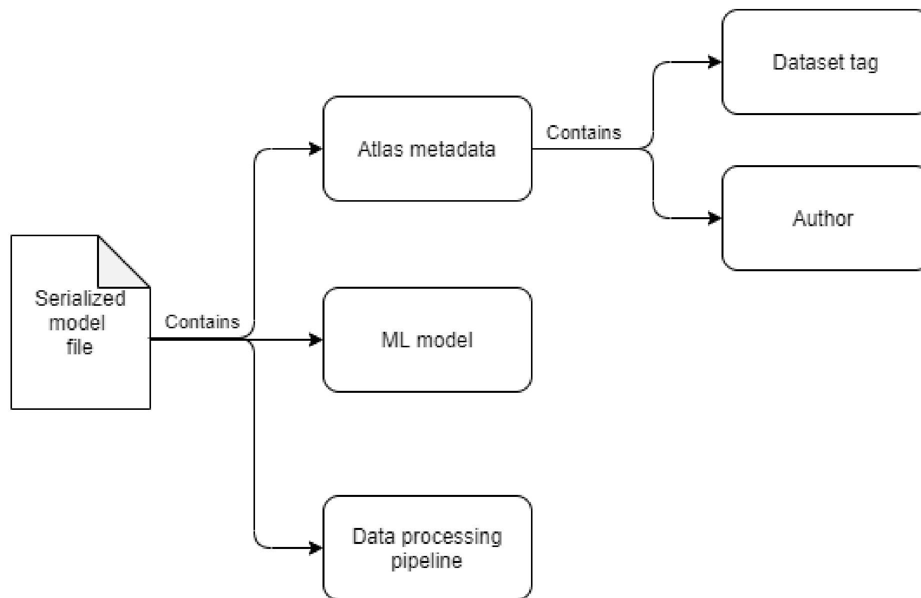


Figure 35: Benchmarking engine model file explained

With a communication standard between all pipeline components, all requirements to develop a benchmark engine are finalized. The benchmarking engine subcomponent communicates with the storage solution to request models, benchmark info, datasets info and actual model files. It also stores all collected metadata about each arriving model and each benchmark performed.

In figure 36 the subcomponent **benchmarking engine** is explained in a component diagram, and it's integration with the **storage solution** component, four main subcomponents exist inside the **benchmarking engine** which are as follows:

- **Connector**: This subcomponent is the bridge between the storing solution and the benchmarking engine component, and its primary goal is to implement a function to query the database. Using this method, no repeated code needs to be written, and all operations needed can be standardized and called using this component.

- **Model manager**: This subcomponent has one main goal: to manage all system models. It queries the object storage and verifies if new models have arrived, then stores those models metadata into the SQL storage. It also manages model benchmarks and orders them as necessary.

- **Benchmark engine**: This subcomponent executes a benchmark operation in a model when called by the model manager, all resulting information from the benchmark is stored in the SQL storage solution.

- **Model explainability engine**: This subcomponent performs the explainability process in a model. This is achieved in two main tasks, retrieve and store the algorithm hyperparameters and execute a SHAP computation to draw explanation charts stored in the SQL storage solution.

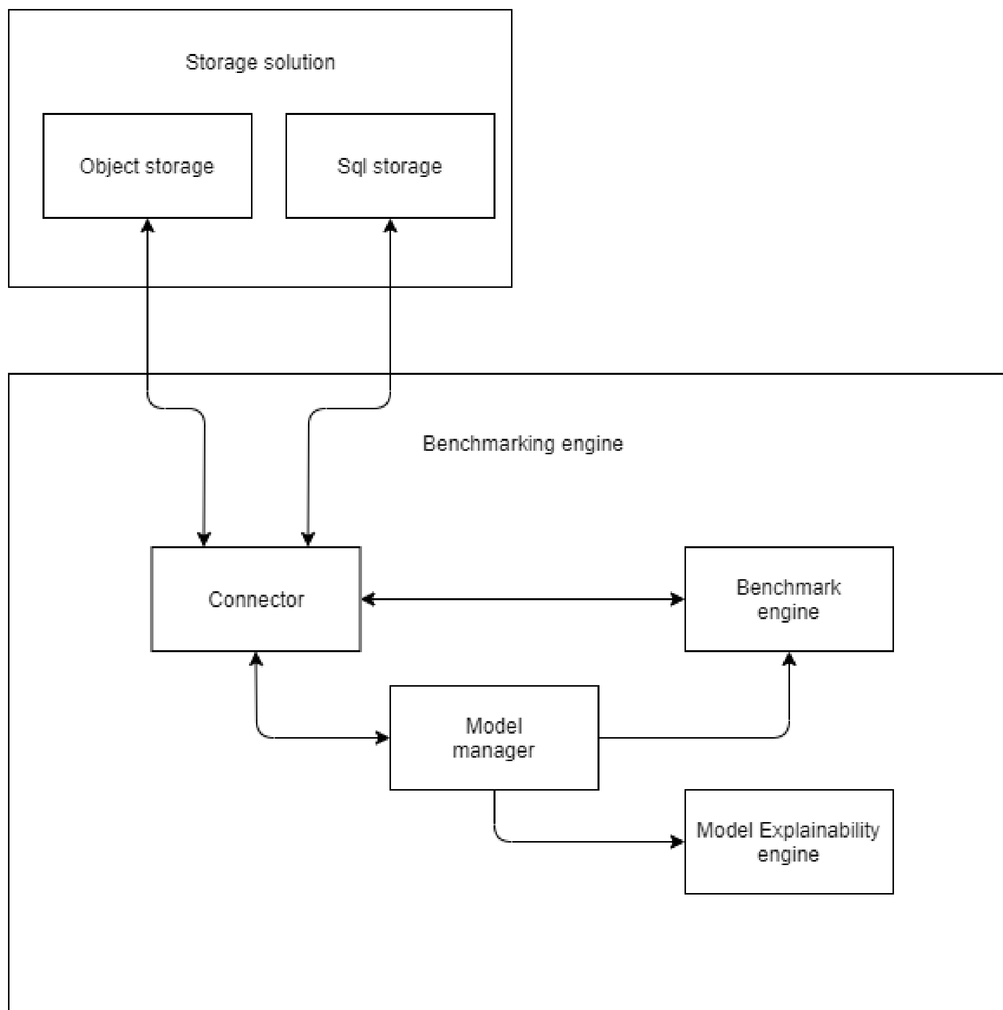


Figure 36: Benchmarking engine subcomponent explained

5.3.2.2 Architecture design

The activity diagram in figure 37 explains the logic behind the component. Two schedules run continuously, the first is the model scheduler which verifies if new models have arrived at the object storage solution, and the second scheduler verifies if a dataset used by a model has been updated with new data. When one of the scheduler's verifier gets triggered, a new benchmark must be performed. The next step is to get information about the dataset that will be used and store the version. Then all models get organized by version, and all that share the same version run in sequence. This allows optimizing dataset fetching times and downloading the dataset only one time for several different possible benchmarks. The first model in the pool starts the benchmark process. The first step is to download the dataset, then a preprocessing set of functions prepare the raw data for the model finishing the **model** manager set of processes.

The benchmark engine receives a dataset and a model from the **model manager** component, then predicts the dataset using the model and computes all the resulting metrics. Finally, all the benchmark information gets stored in the storage solution. A verifier is also executed in the model to check the type of algorithm, if the algorithm is in the compatibility list for the **explainability engine** this component is

executed.

The Explainability engine receives the model prediction list for each event, the dataset and the model. The first operation is to extract the model hyperparameters, then SHAP values are computed, which returns the information to produce all the SHAP charts that provide explainability in the model, this component set of processes end with the storage of all gathered information.

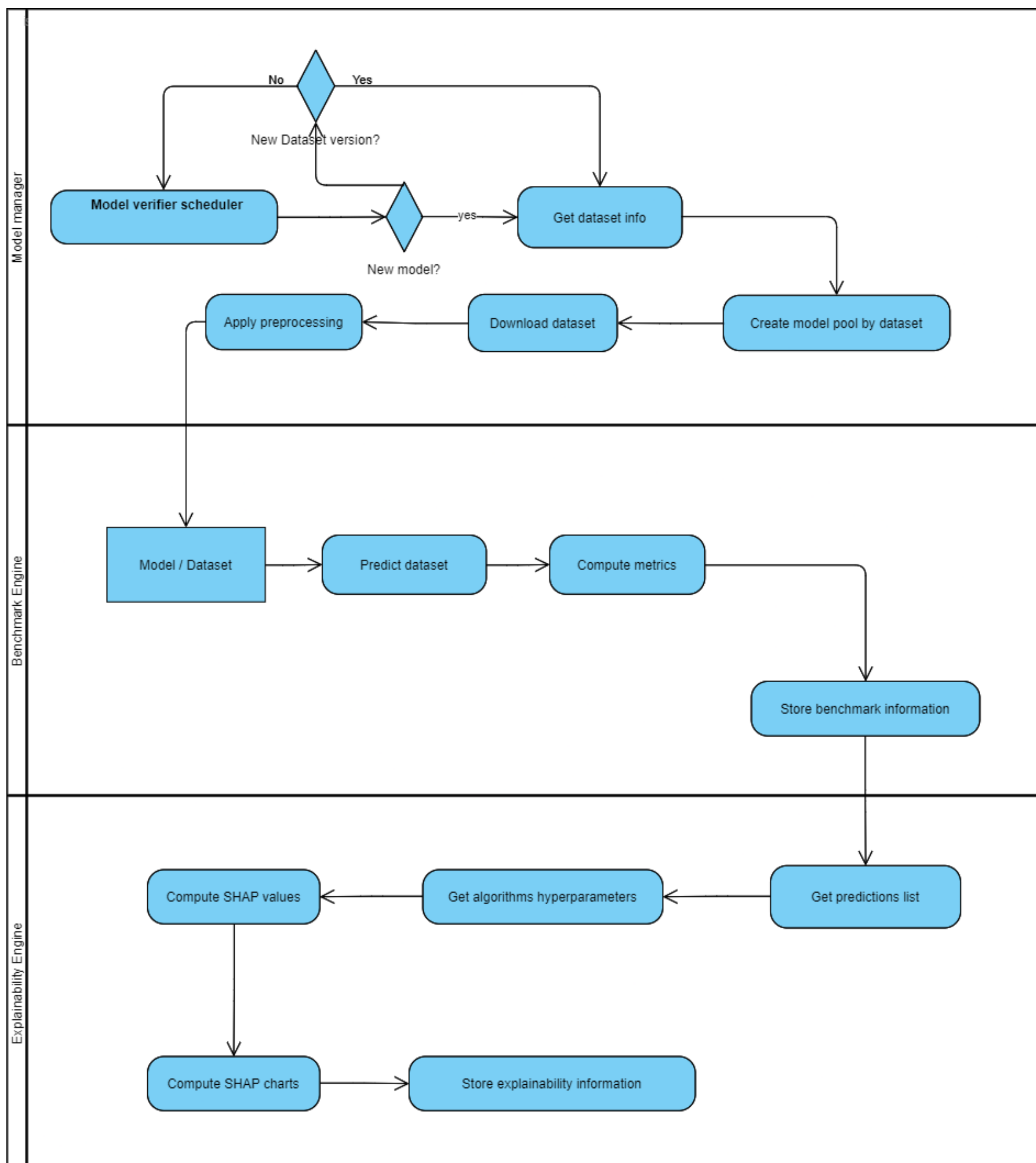


Figure 37: Benchmarking engine activity diagram

The architecture follows a modular design with each component being independent and the only requirement for it to function is to provide adequate entry parameters, this design choice makes new updates and features to the architecture possible and straightforward, leaving room for system improvements in the future.

The benchmark pipeline is very complex and demanding, each benchmark can take from several minutes to hours to be performed, and as the system grows and more and more models need to be processed and continuously evaluated, it might be unfeasible without further improvements in the architecture. To solve these limitations, a parallel architecture is going to be explored in the next section, with vertical and horizontal scalability approaches considered.

5.3.2.3 Increasing benchmark throughput, parallelization

A benchmark requires many operations that are demanding in processing power and network bandwidth, implementing the architecture in figure 37, we can see this will evaluate a single model each time it runs. There are two main bottlenecks. The first is dataset download times, a dataset can take several minutes to download, and the system is unable to continue until this download process stops. To improve on this problem, the architecture will organize all pooled models by dataset and download the dataset a single time to run all the models that utilize it. This technique by itself will severely improve system performance and benchmark throughput. Another bottleneck is running a single benchmark each time, this can be solved in two ways:

- **Horizontal scalability** refers to adding more computing nodes to the system, for instance distributing the workload in multiple workstations to achieve a higher computing power.
- **Vertical scalability** refers to leveraging the current computing capability in the workstation node to achieve a higher computing power, this can be achieved by parallelizing the workload or adding stronger hardware to the physical system.

With the limitations of this project, vertical scalability is the option chosen to provide scalability for the system. After a system inspection, there are up to 6 cores available to parallelize the system, however, it is important to measure the system throughput as the number of cores increases. For this, we measure benchmarks per minute as the metric.

For the following testing methodology, a baseline model that uses the latest algorithm architecture and a small dataset is used to keep results accurate and with real-world value.

Following figure 38, a time comparison for a benchmark execution is registered. In an ideal world, a six thread benchmark parallelized would take the same 118 seconds as it took in a one thread workload, however as can be observed, efficiency loss is a real problem as more threads are being added to the parallel execution.

Due to efficiency loss in the system, another important test is to check how much performance is lost by each added thread compared to a single thread execution. Figure 39 shows the efficiency loss each thread takes compared to a baseline value of 1 thread, adding a second thread, the benchmark is 14 percent slower, three threads are 29 percent and then 47, 63 and 84 percent lower compared with baseline values. The analysis of this chart introduces two main observation, the first is that the benchmark execution gets slower as parallelization increases and the second is, how many threads justify the performance gain

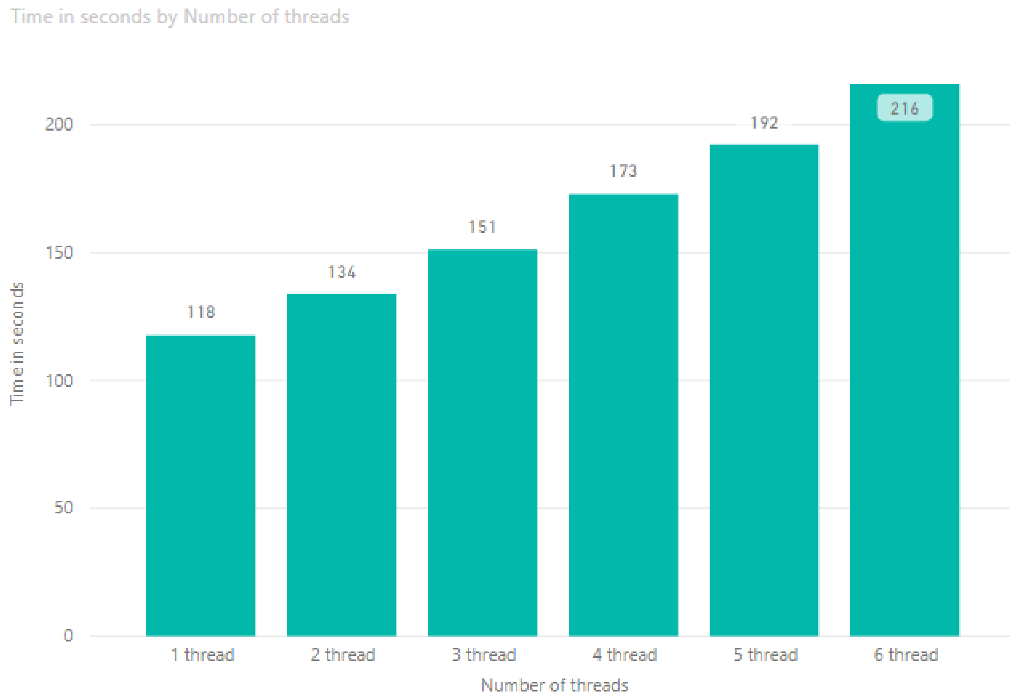


Figure 38: Benchmark time in seconds by number of threads

and when does the system stop scaling, to solve this a new metrics must be considered which is the benchmarks per minute the system can process.

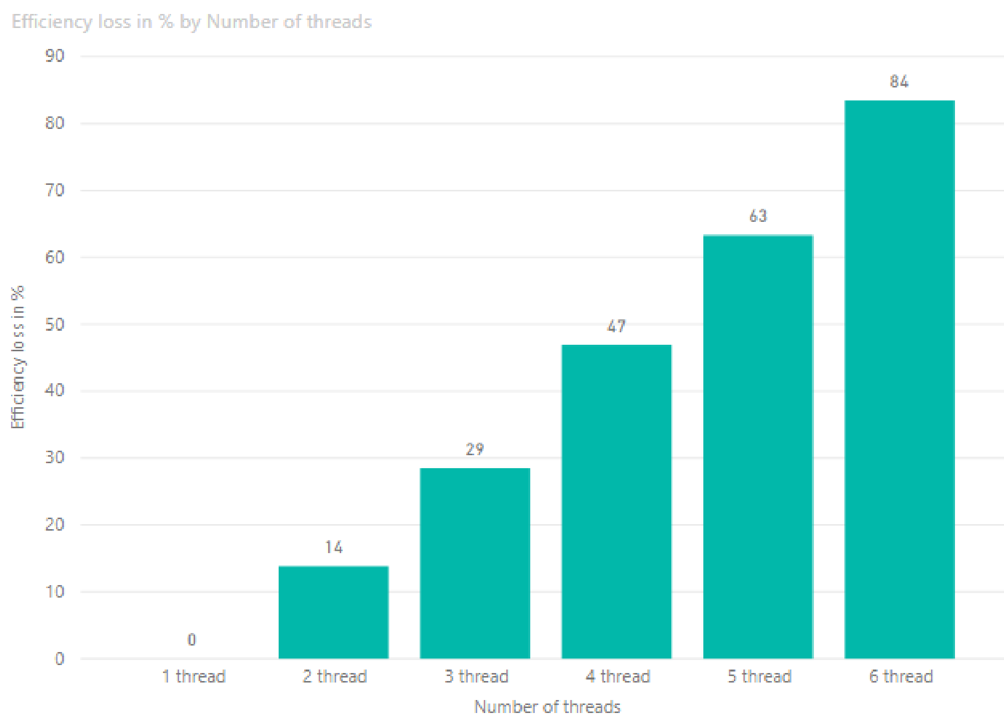


Figure 39: Benchmarking engine efficiency loss by thread added

The chart in figure 40 showcases the key metric to understand the raw output of the system in terms of benchmarks, the metrics benchmarks per minute is referenced in the chart as **BPM**, as can be observed, benchmarks per minute are at 0.51 in the single-threaded test, using two threads the value rises to 0.90 and then 1.19, 1.39, 1.56 and finally 1.67 using all six threads allocated to the problem. Further increase in threads is not sustainable on the hardware side, and returns are already diminishing, indicating that the parallelization process is optimized for the hardware available.

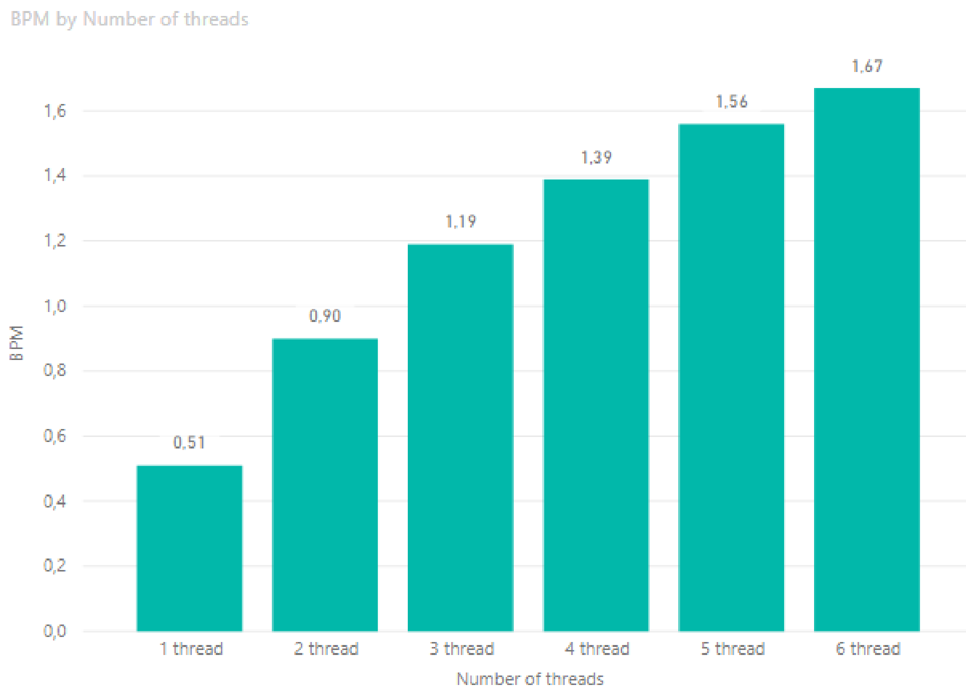


Figure 40: Benchmarking engine scalability comparison, thread BPM comparison

Using this scaling technique, the system now has a speed up of **327** percent comparing to the baseline single-threaded architecture, which significantly improves the response capabilities that it provides to continue to evaluate machine learning models, further in the future, if the system needs more scaling, a horizontal scaling approach might be required, which implies adding computational devices.

5.3.3 Web engine architecture

The final subcomponent is the web engine, which is responsible for providing all the information collected by the system to the end-user in the simplest and most informative way possible. This goal is achieved by having a web framework handling the backend data and a powerful frontend dashboard to showcase all the information.

All the data is fetched in the database solution. This means that no communications occur between the benchmark and web engines, which simplifies the architecture. The technologies to develop the solution are as follows:

- **Django**: Django is a powerful python web frameworks that allows the creation of large scale web applications, it also has a powerful **ORM** database connector that works natively with the chosen SQL database solution and simplifies the querying process.
- **Bootstrap**: Bootstrap is a powerful HTML, CSS and JS library that allows the creation of interactive and dynamic web pages, it is specially powerful to create dashboard like applications which are the goal of this application.

The architectural pattern is the Model-View-Controller, which separates the application into three logical components, allowing for greater modularity, easier maintenance and a bigger upgrade path for the future. Figure 41 showcases the component structure and the connection to the storage solution.

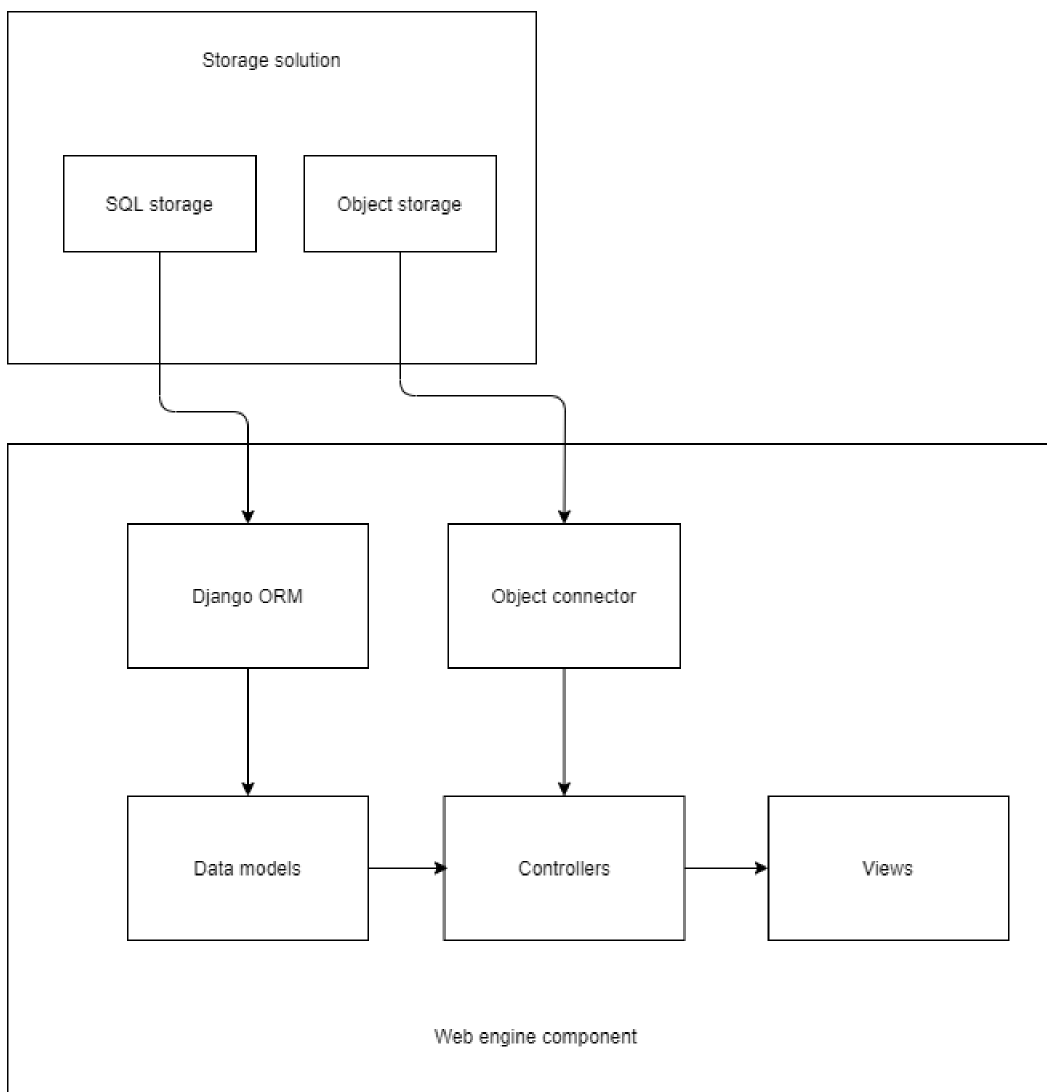


Figure 41: Web engine subcomponent explained

5.4 Deployment overview

The deployment of the application showcases the requirements and techniques used to deploy the application and the hardware utilized and required to guarantee the correct functionality. The figure 42 is a deployment diagram and displays all three main components in the pipeline. TCP-IP connects the **model benchmarking engine** with the **data ingestion engine** and the **model training engine**. Inside the model benchmarking engine, there are three main docker containers, each running a subcomponent. Communication between subcomponents only happens with the storage solution. The final users or the system administrator interacts with the system using a web browser and establishing an HTTPS communication that displays a frontend dashboard.

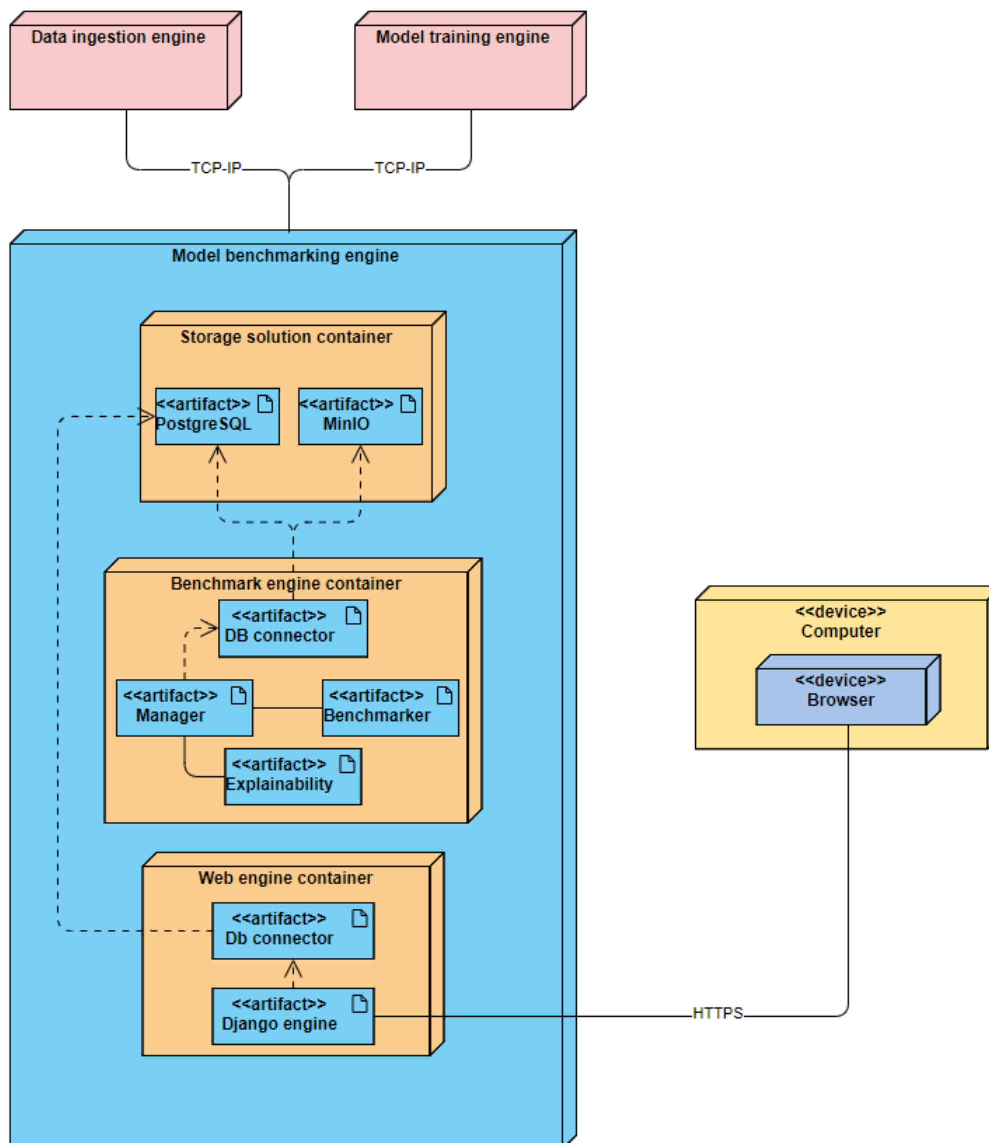


Figure 42: Application deployment diagram

5.5 Summary

Now that the architectural design for the application is built, there is a solid ground to start the development process, the architecture for the whole **Atlas** system is completed, the general overview can be seen in figure 43.

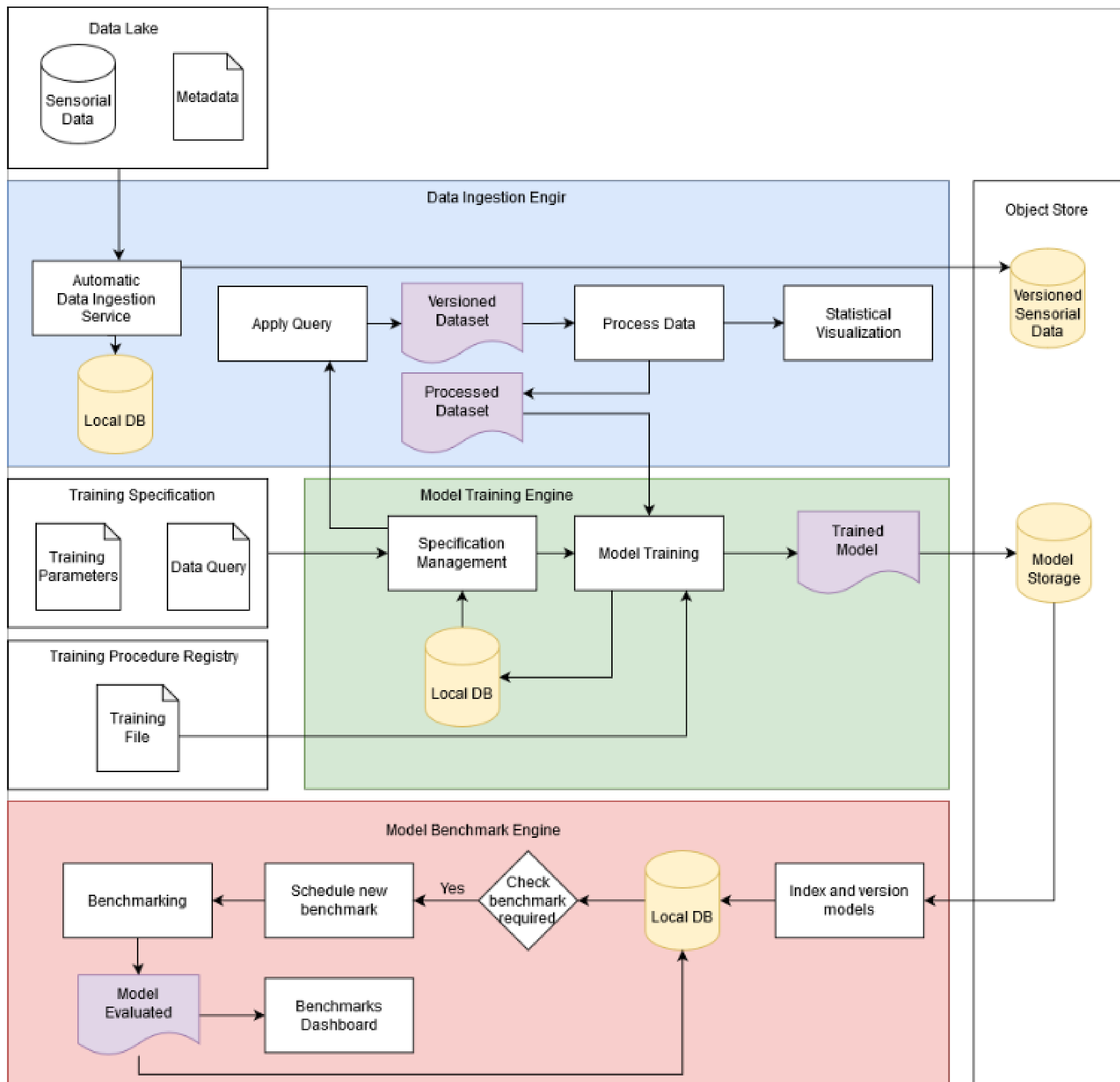


Figure 43: Atlas final architecture

Chapter 6

Results demonstration and discussion

The focus of this chapter will be to demonstrate the web dashboard, all the information displayed in the dashboard is computed in the other components as shown in the previous section without additional user input. This is key for the system automation and time-saving benefits, as it allows the user to have its models continuously benchmarked without performing any action in the system.

The initial page and the first interaction with the system is the default front page. It is designed to show key information about the **model benchmarking engine** and the **Atlas** pipeline and must be intuitive and straightforward, such as allowing the user to use the system without external help.

Figure 44 demonstrates the initial page with multiple numbered rectangles highlighting different information available, Area **1** represents the side menu that gives the user quick access to all major functions available. Area **2** contains information about the system state like models loaded, benchmarks executed and pending an execution, number of different datasets present, and the number of the different authors who submitted models to the system. Area **3** are quick access tables with the latest five performed models and the latest five performed benchmarks. It contains the name of the model, the author that created it, date of submission and a benchmark state with the latter being red if the benchmark is not yet performed or a green button if it is available, this allows the user to find its latest submitted model with as few clicks as possible. Last, area **4** represents the **Atlas pipeline status**, since the pipeline depends on all components working properly, it is important to have all components' status readily available and easy to obtain.

In the initial page and following the side menu, the user can choose the *models page, benchmarks, datasets, and the alchemy page* which will be explained shortly. The *model page* is provided as shown in figure 45, most columns are self-explanatory and contain relevant model information, the last two columns provide information about the benchmark state and explainability state, if the icon is grey, the system hasn't computed the information yet. A red icon means that some content may not be possible

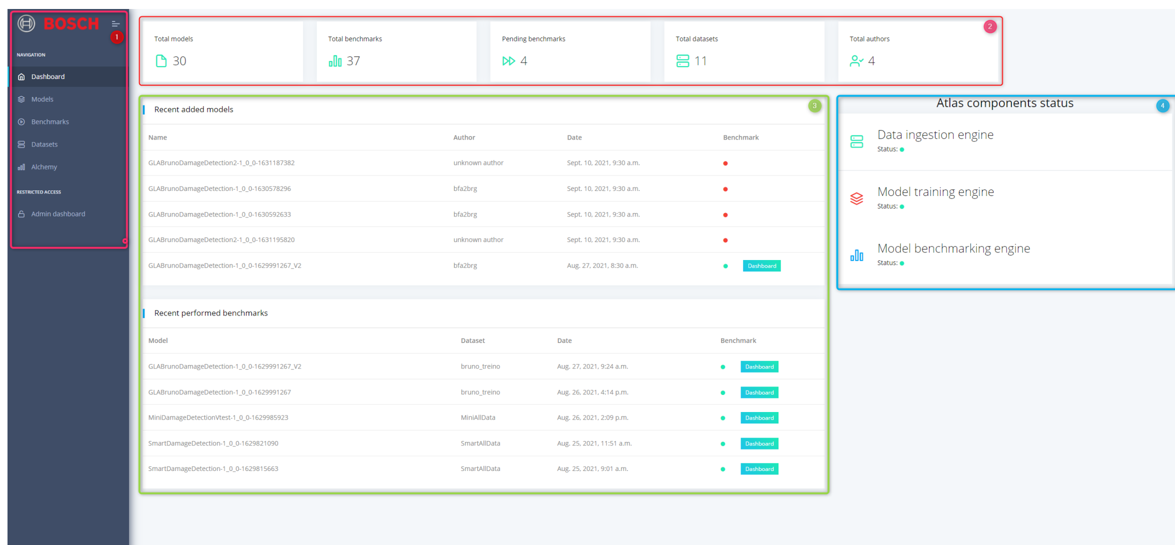


Figure 44: Initial system webpage

to obtain, and a blue icon indicates that all information is available. Clicking the button will take the user to the respective page. All information in the table can be ordered by column, searchable by text and pagination customizable by the user.

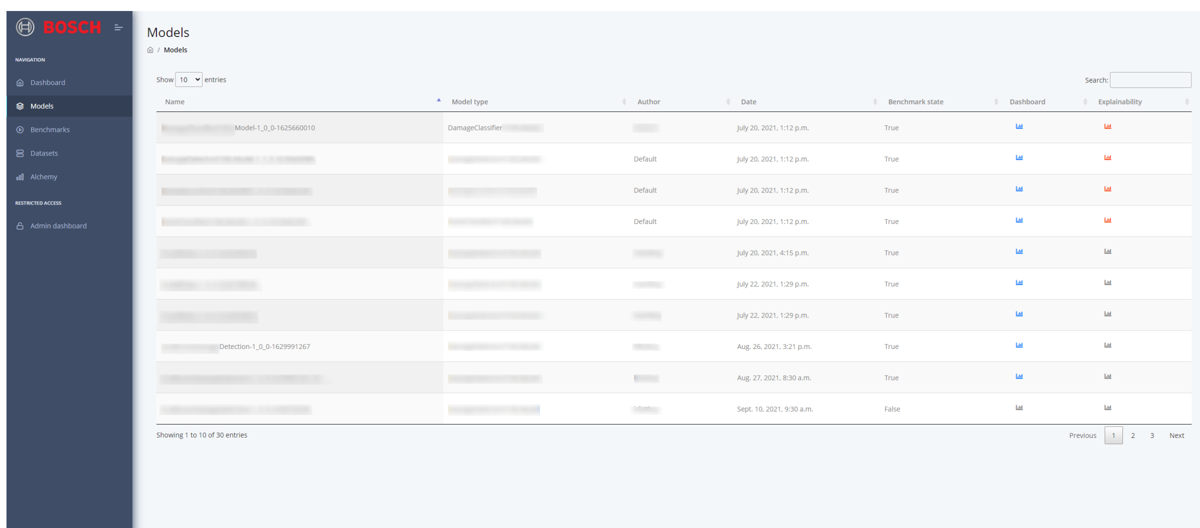


Figure 45: Model listing webpage

Similar to the model page, the *benchmark page* shown in figure 46 shows the benchmarks performed by the system, the main difference is that a model can have one or more benchmarks associated so this page will show each benchmark independently, the user can also access its dashboard and see all the information available.

The *dataset's page* is demonstrated in figure 47, and it contains information about all the datasets indexed by the system, the query tag, which is the human-readable identifier, and all the versions of the datasets present and the latest query ID correspondent to the tag. Finally, the last column takes the user

Figure 46: Benchmark listing webpage

Figure 47: Dataset listing webpage

to the dataset page, shown in figure 48 and displays all available versions and a full statistical report about the given dataset.

The dataset detail page is provided by the **data ingestion engine,** possible due to the tight integration between all the components in the pipeline and the achieved interoperability. it is of the highest relevance to provide dataset information, as users might want to consult the dataset used for training and evaluation for each model to replicate the results. The dataset information is very detailed with statistic information about a number of events, environment conditions and metadata information, giving the user a fully-fledged report on the data.

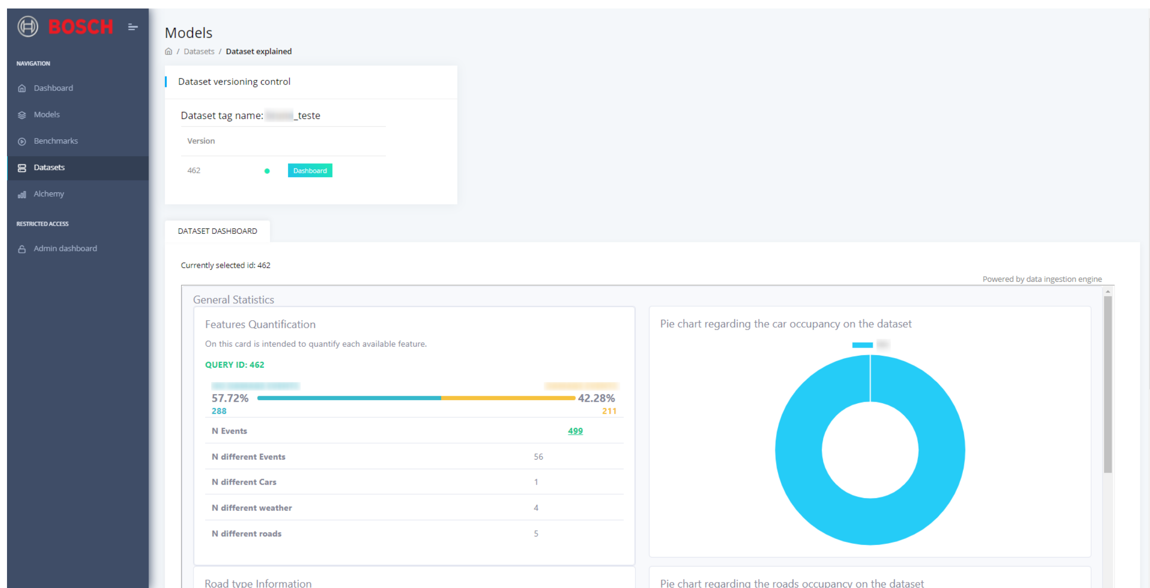


Figure 48: Dataset detail webpage

One of the main issues of having a fully automated system that computes all information in the backend without user input is the lack of knowledge about the status of those operations. To tackle this issue, a logging system was devised that catches checkpoints in common operations and stores their state; it can be seen in figure 49, the logging system also stores exceptions that happen in the system. To give the user the logging information, the admin dashboard was created. It provides all the latest logs from the system classified as info or warning, it is also important that new checkpoints for logs can be added with user feedback. This page also provides information about the system component status, a manual option to upload machine learning models that do not come from the pipeline, but the user still wants to introduce into the system, and a button to access the database management system which provides CRUD operations.

Accessing the *Manage database* button will take the user to the page represented in figure 50. This page provides full control for the tables in the database, allowing the system manager to solve potential exceptions or faulty models that arrive in the system. It is expected that most hiccups can be treated by consulting the exception in the logs page and solving the underlying problem in the database management page.

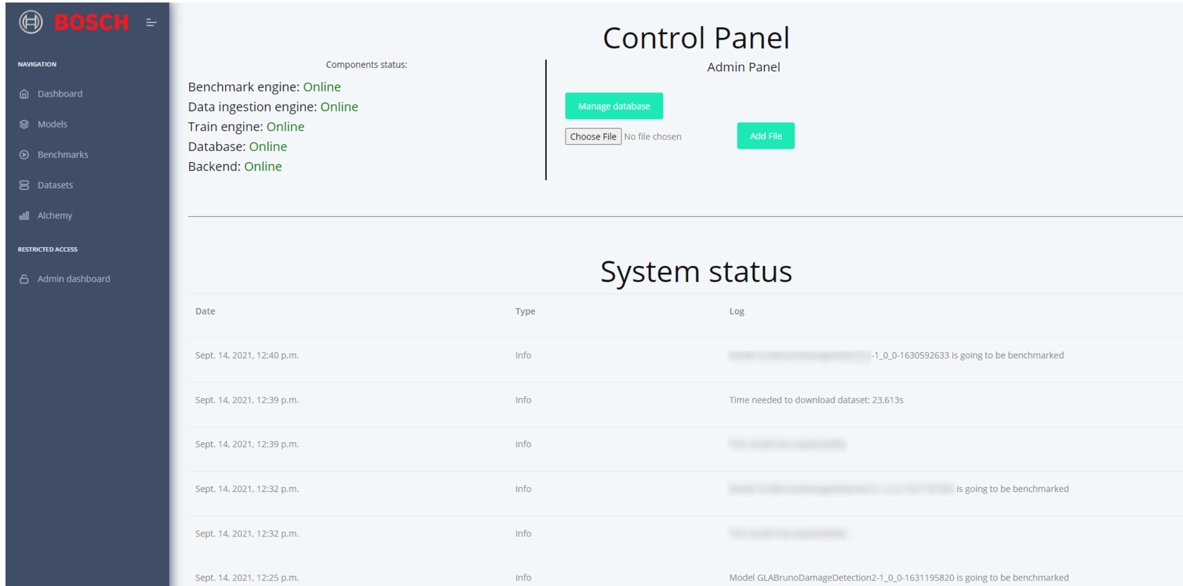


Figure 49: Admin dashboard and log information webpage

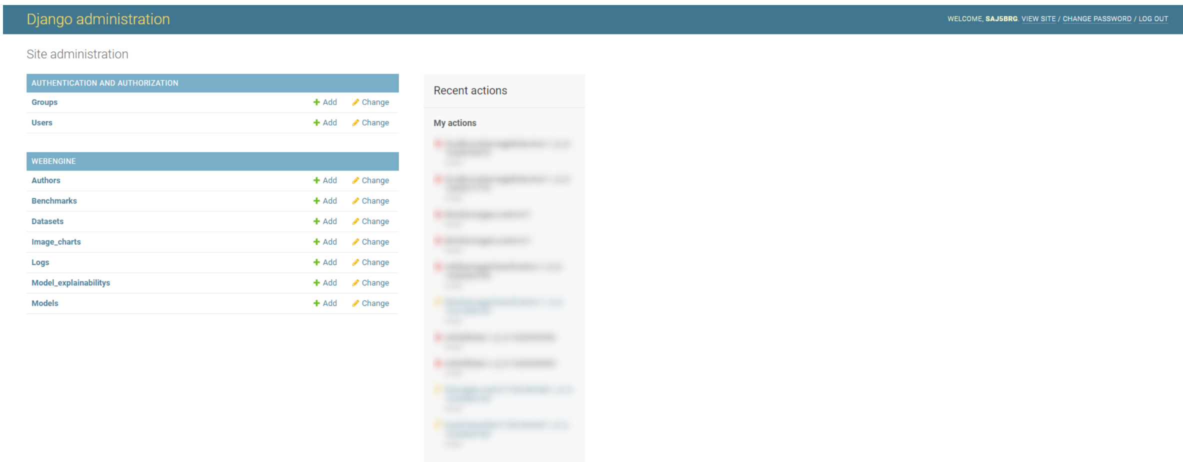


Figure 50: Admin dataset management webpage

The model benchmark results need an appropriate dashboard that is easy to interpret and delivers as much information as possible without overwhelming the user, it is also important to show the multiple benchmarks a model might have and appropriately identify them.

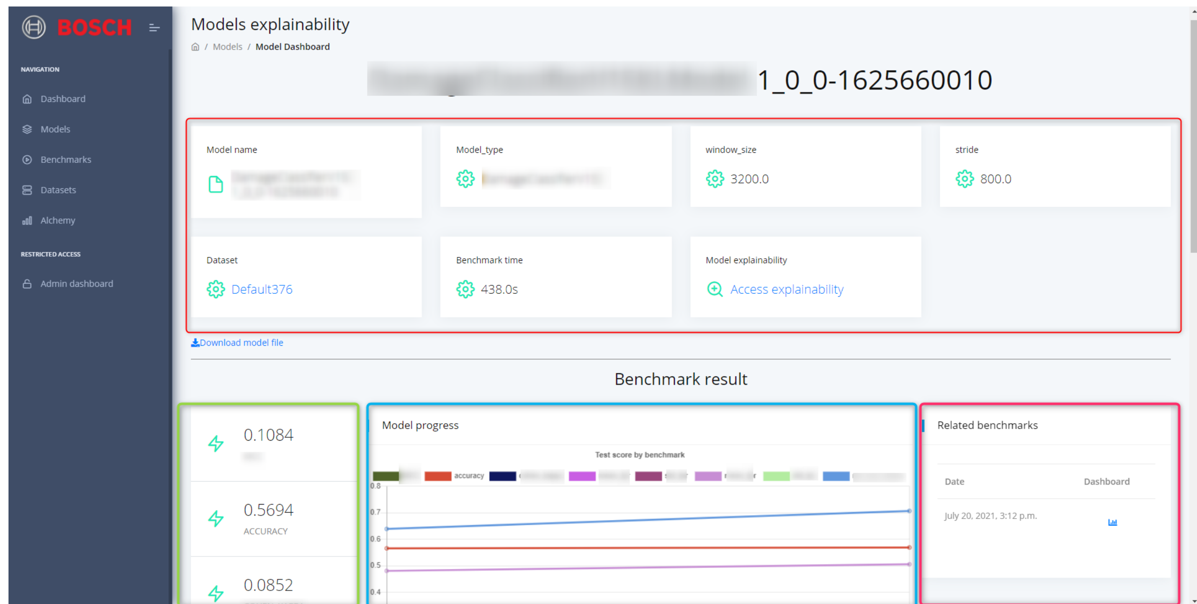


Figure 51: Model benchmark page

The model benchmark page is shown in figure 51. Each drawn rectangle in the picture references a different informational section. Area **1** showcases model information like the dataset tag it uses, data window for prediction and even benchmark time, it also has a shortcut for the model explainability which will be explained later. Area **2** is the metrics section, it displays all the metrics considered relevant by the data scientist for the specific machine learning model with its name and value. Area **3** is the chart section that displays benchmarks results in time for each benchmark, this chart is crucial to understand the model evolution for each benchmark performed. Area **4** is the section that displays all the other benchmarks performed on the model to give the user a good reference, the date of benchmark is the first column which allows to identify the timeframe for each benchmark.

Figure 52 showcases the benchmark distribution section of page 51, it displays the labels predicted by the model and the actual ground truth value and is very useful to understand the model bottlenecks and if the training process needs to be improved.

The model hyperparameters page is an important feature to have since it allows for other users to replicate and improve upon already good models, it makes each machine learning model reproducible, this page is displayed in figure 53, each box represents one hyperparameter, this information is dynamic and can work for any given compatible machine learning algorithm.

Finally, the last feature is the model explainability, to achieve this functionality SHAP framework is used, which is leading state of the art in algorithm explainability. ML explainability is currently a hot topic and helps demystify the *black box* feeling about ml algorithms, it is specially important for two main purposes:

Figure 52: Model benchmark page, confusion matrix

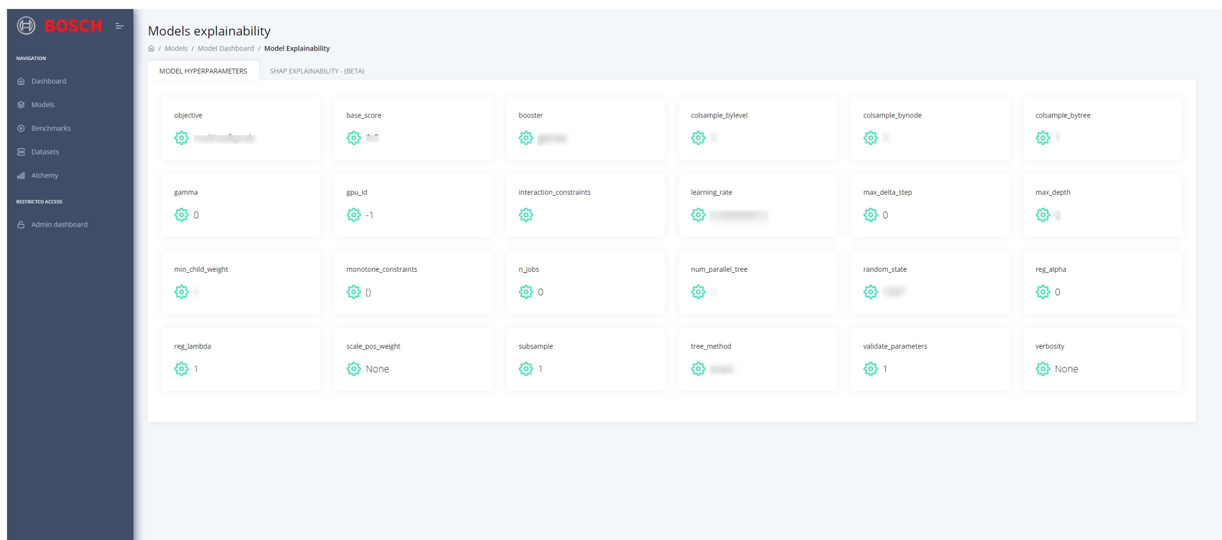


Figure 53: Model benchmark hyperparameters page

- **Explain to management leaders and clients**: Management and clients might not have technical capabilities to understand ML at a deeper level, **SHAP** charts provide an intuitive and effective way to explain the decision-making process behind every prediction.

- **Understand bad predictions**: Sometimes an algorithm has trouble detecting a certain event and can be quite costly to solve these anomalies, **SHAP** explainability can be used to understand where the decision-making process goes wrong to try to fix it.

In figure 54, the dashboard to observe the explainability report is shown. Right now, three charts are displayed, the feature importance calculated by the *SHAP* framework, then the feature value correlation with the prediction outcome and finally, the decision tree by feature. This feature is supported for specific

Figure 54: Model benchmark SHAP explainability page

algorithms only, and not all models can run this successfully yet. However, it is expected to perfect the automation process and fix these few exceptions.

6.1 Conclusion

6.1.1 How does this solution improve the team workflow

In general, the **model benchmarking engine** is a powerful tool that empowers the ML team at the company. All the underlying components have their results displayed in this final dashboard, which is simple and intuitive to use, adding no learning curve to the team.

In its final state, the **model benchmarking engine** tool reveals itself to be a very powerful decision-making assistant. It automatically provides all relevant metrics about each model in each dataset, while also verifying its performance with new incoming data to see performance decay with time.

The interoperability between all three components of the **Atlas** pipeline was also a success achieved in the design stage, allowing for features present that can only be possible by the close integration with all three components.

In general, the team benefits from these solutions in the following ways:

- **Continuous automatic model benchmark**: Previously, the data scientist had to manually benchmark each model, in addition, each time new data arrived a new manual benchmark had to be performed to check the performance. This solution automatically benchmarks all the models from the team providing all the relevant metrics, in addition to also scheduling regular updates

for new data which allows having a continuous monitoring solution of the model performance with time.

- **Model versioning**: Previously, each data scientist had to manually version the models he created. This solution automatically versions models by date, datasets in which it was trained, type of model, and it's performance.

- **Results overview**: Previously, the data scientist had to manually keep track of the performance each model had in order to know the best performing algorithm. This solution automatically tracks performance by dataset and by algorithm type, allowing the data scientist to effortlessly know which is the best performing model for each use-case at any time.

6.2 Future work

Like many development cycles, the initial requirements and set of features sound right in the beginning, but as the application gets widely used and adopted, a whole new set of features get requested for functionalities that would further improve the utility of the application, with users feedback a future roadmap has been created.

The first **model benchmarking engine** feature to add is **multiple dataset tag support**, right now the model is sent with a single dataset tag to continuous benchmark, but it has been pointed out that the data scientist should have the option to have that model be benchmarked with a different dataset tag in the future.

The second feature is to deploy and monitor selected machine learning models using live data that is regularly collected, the **model benchmarking engine** is, as the name suggests, a very capable benchmark engine capable of performing dozens to hundreds of benchmarks a day depending on the dataset size. Therefore, adapting this system to monitor models on live data is only natural and implies that the live data can be tagged using the **data ingestion engine** and a creation of a custom subpage dashboard to monitor these results.

The third feature is to improve **manual model uploads**. Currently, there is already a function that allows the upload of manual machine learning models, however the proper metadata like the dataset tag needs to be embedded in the file. To improve this functionality when a model is uploaded, a form is displayed that allows the user to fill the proper metadata without manipulating and changing the actual model file.

Adding these features to the roadmap will further improve the value of the model benchmark engine and improve the development cycle of machine learning models in the company, which is the project's ultimate goal.

6.3 Final remarks

The project roadmap planed in the beginning of the January has been fulfilled, all the requirements drawn by the team have been built, deployed and can now be fully utilized as demonstrated in the results section.

The project's goal pushes the state of the art in the MlOps field, as machine learning automation is still in its infancy, this made the initial requirements tricky to assemble due to the lack of knowledge of what is possible and feasible in the area, but also the pain points that are to be expected in this type of solutions. After the application received its initial users, while well received, a few improvements were given that are explained in the future work section above.

Another challenge that had to be surpassed was the interoperability between all the components that partook in the Atlas pipeline, in the end all the work towards a successful communication standard panned out and a seamless transition between each component is present in the final product.

The product achieved its goal, the **Atlas** pipeline is already helping data scientist at the company to have a fully fledged continuous training and continuous benchmarking system capable of providing real-world value. Automatic deployment is already in the roadmap, and it is a feature that will further increase the power of the **Atlas** pipeline.

The **model benchmarking engine** is a powerful tool, and its intuitive dashboard helps to automatically deliver all the metrics the data scientist need to judge each model. This set of features makes it a powerful decision-making tool that can always provide the best performing models for each use case or each dataset, and continuously evaluate models to observe for decay with time.

Bibliography

Alahdab, M., & Çalıklı, G. (2019). Empirical Analysis of Hidden Technical Debt Patterns in Machine Learning Software, 195–202. doi:10.1007/978-3-030-35333-9_14

Arnold, M., Boston, J., Desmond, M., Duesterwald, E., Elder, B., Murthi, A., Navratil, J., & Reimer, D. (2020). Towards automating the ai operations lifecycle.

Baidu Benchmarks. (2017). Retrieved from https://github.com/baidu-research/DeepBench

Breck, E., Cai, S., Nielsen, E., Salib, M., & Sculley, D. (2017). The ML test score: A rubric for ML production readiness and technical debt reduction, 1123–1132. doi:10.1109/BigData.2017.8258038

Coleman, C. A., Narayanan, D., Kang, D., Zhao, T., Zhang, J., Nardi, L., Bailis, P., Olukotun, K., Ré, C., & Zaharia, M. A. (2017). Dawnbench : An end-to-end deep learning benchmark and competition.

Cunningham, W. (1992). Experience report- the wycash portfolio management system. *OOPSLA'92*. doi:10.1145/157710.157715

Fernandes, J., & Machado, R.-J. (2015). *Requirements in engineering projects*. doi:10.1007/978-3-319-18597-2

Hassanien, H. E.-D. (2019). Web scraping scientific repositories for augmented relevant literature search using crisp-dm. *Applied System Innovation*, *2*, 37. doi:10.3390/asi2040037

Karamitsos, I., Albarhami, S., & Apostolopoulos, C. (2020). Applying devops practices of continuous automation for machine learning. *Information*, *11*, 363. doi:10.3390/info11070363

Kim, G., Humble, J., Debois, P., & Willis, j. (2016). *The DevOps Handbook*.

Kubeflow, An introduction to kubeflow. (n.d.). Retrieved from https://www.kubeflow.org/docs/about/kubeflow/

Kumar, G., & Bhatia, P. (2012). Impact of agile methodology on software development process. *International Journal of Computer Technology and Electronics Engineering (IJCTEE)*, *2*, 2249–6343.

Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., & Zhang, G. (2020). Learning under Concept Drift: A Review. doi:10.1109/TKDE.2018.2876857

Lundberg, S., & Lee, S.-I. (2017). A unified approach to interpreting model predictions.

Martin Fowler, J. H. (2001). The agile manifesto. doi:10.13140/RG.2.2.10021.50403

MLOps: Continuous delivery and automation pipelines in machine learning. (2020). Retrieved November 27, 2020, from https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning

Rahman, M. M. (2019). Waterfall model: The scientific method of software engineering.

Ribeiro, M., Singh, S., & Guestrin, C. (2016). “why should i trust you?”: Explaining the predictions of any classifier. doi:10.18653/v1/N16-3020

Sarnelle, J., Sanchez, A., Capo, R., Haas, J., & Polikar, R. (2015). Quantifying the limited and gradual concept drift assumption, 1–8. doi:10.1109/IJCNN.2015.7280850

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., & Young, M. (2014). Machine Learning: The High Interest Credit Card of Technical Debt. *Google Research*. Retrieved December 3, 2020, from https://research.google/pubs/pub43146/

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., & Dennison, D. (2015). Hidden technical debt in machine learning systems. *28*. Retrieved from https://proceedings.neurips.cc/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf

TensorFlow Benchmarks. (2017). Retrieved from https://www.tensorflow.org/performance/benchmarks

Van Casteren, W. (2017). The waterfall model and the agile methodologies : A comparison by project characteristics - short. doi:10.13140/RG.2.2.10021.50403

Virmani, M. (2015). Understanding DevOps bridging the gap from continuous integration to continuous delivery, 78–82. doi:10.1109/INTECH.2015.7173368

Wahaballa, A., Wahballa, O., Abdellatief, M., Xiong, H., & Qin, Z. (2015). *Toward unified devops model*. doi:10.1109/ICSESS.2015.7339039

Zhu, H., Akrout, M., Zheng, B., Pelegris, A., Jayarajan, A., Phanishayee, A., Schroeder, B., & Pekhimenko, G. (2018). Benchmarking and Analyzing Deep Neural Network Training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)* (pp. 88–100). doi:10.1109/IISWC.2018.8573476