THOMAS MORE

Decision Support System

2024 - 2025

Table of Contents

Table of Figures

1. Introduction

This document describes the realization of the project proposed in the earlier project plan. In short, the goal was to develop a beta version of an application that serves as a decision support system for researchers and cardiologists to determine the optimal drug and dosage for an individual patient based on the patient’s history, of course, applying the many safeguard rules that govern such sensitive AI-driven decisions. These, however, will not be discussed in this document, as they are outside the scope of this internship and fall under the responsibility of the main research team. The application abstracts the complexity of machine learning inference and makes model predictions accessible through a clean user interface.

In addition to the application itself, a supporting pipeline was developed to streamline model training and evaluation for data scientists. This pipeline is now integrated into the broader application architecture, contributing to a unified and maintainable system.

The document begins with an analysis of the initial requirements and an explanation of the technology choices made during development. It then walks through the end-to-end prototype, highlighting key architectural decisions and how early requirements were addressed. Following this, the beta application is presented in detail, with a focus on architectural improvements, modular design, and security features.

Due to delays in accessing the real clinical dataset, the current version of the application was developed and tested using a dummy dataset. As a result, no production-ready models or real patient data are integrated at this stage. This constraint shaped the project’s scope and allowed focus to remain on designing a scalable and modular system architecture, ready for future integration once the data becomes available.

The pipeline is described in a dedicated section, detailing its components and integration with the main application. The deployment strategy is covered next, outlining the microservice-based architecture and containerization setup. This is followed by a brief overview of the complementary courses taken during the internship, and finally, a conclusion reflecting on the results and suggesting directions for future development.

2. Analysis

This section presents an in-depth evaluation of the technologies selected to implement the internship project. Each technology decision was made following a structured comparison using weighted decision matrices, based on criteria such as scalability, performance, integration, security, and personal expertise. The final choices were validated in consultation with the internship mentor and reflect both the immediate requirements and long-term goals of the application. Where relevant, attachments provide the detailed matrices used in the evaluation.

2.1. Machine Learning Pipeline

To determine the best framework for the machine learning pipeline, I compared Scikit-learn (Scikit-learn, n.d.), H2O AI (H2O.ai, 2025), deployKF (Kubeflow) (deployKF, 2024), Azure ML (Microsoft, 2025), and AWS SageMaker (Amazon, 2025) based on key criteria like, scalability and flexibility. Each has distinct strengths, making it essential to evaluate them against the project's needs. (Attachment A)

Own experience was considered, because of time constraints and the complexity of learning a new framework. Since I have the most experience with Scikit-learn, this framework came out on top in this category and will allow me to work within a familiar environment. Other frameworks like deployKF, Azure ML, and SageMaker require additional time investment due to their platform-specific tooling or infrastructure setup.

Another important criterium that was evaluated is **scalability**. In the future, new models will be trained frequently as new data becomes available. For this, H2O AI and deployKF stood out. H2O is built for large-scale machine learning and handles large datasets efficiently. deployKF, built on Kubernetes via Kubeflow, also provides excellent scalability, making it suitable for more complex workflows. Azure ML and AWS SageMaker also offer strong scalability but require proper resource configuration and come with added complexity.

Distributed computing was another area of focus. H2O AI scored highest, as it includes distributed computing capabilities out of the box. deployKF also performed well in this area due to its Kubernetes-native architecture. Azure ML and AWS SageMaker also support distributed training but require additional setup. Scikit-learn, on the other hand, does not provide distributed computing by default and depends on external tools like Joblib (Joblib, 2021).

Model training speed was evaluated to ensure fast iteration during development. H2O AI and the cloud platforms performed well here, especially when hardware acceleration is enabled. deployKF also offers good performance, though it depends heavily on how the underlying cluster is configured. Scikit-learn performed reasonably, but slower when handling large datasets.

All frameworks supported a good level of **integration** and **flexibility**. Scikit-learn, H2O AI, and deployKF integrate well into Python-based workflows and can be adapted to different pipeline structures. Azure ML and AWS SageMaker also provide flexible solutions, especially when integrated into their broader cloud ecosystems.

Finally, **cost** was included as a critical criterium. Since this project prioritizes open-source tools, cost and licensing was a deciding factor. Scikit-learn, H2O AI, and deployKF are open-source and highly accessible. Azure ML (Microsoft, n.d.) and AWS SageMaker (Amazon, n.d.), while powerful, involve recurring infrastructure costs that may not be sustainable for all use cases.

Taking all criteria into account and applying a weighted ranking matrix, H2O AI and deployKF emerged as the most suitable frameworks for this project. H2O AI scored highest overall due to its performance, scalability, and built-in distributed computing. deployKF followed closely, offering strong technical capabilities with open-source flexibility. While Scikit-learn remains valuable due to familiarity, it lacks the scalability required for future growth. Azure ML and AWS SageMaker provide powerful managed services but were less suitable due to cost and platform complexity.

2.1.1. Evaluation Summary

The table below summarizes the key finding from the analysis of the machine learning pipeline framework above:

Criterion	Highlights
Scalability	- Top performers: **H2O AI**, **deployKF** - Scalable but complex: **Azure ML**, **AWS SageMaker**
Distributed Computing	- Built-in support: **H2O AI** - Strong via Kubernetes: **deployKF** - Supported with setup: **Azure ML**, **AWS SageMaker** - Limited: **Scikit-learn** (requires external tools)
Training Speed	- Fast: **H2O AI**, **Azure ML**, **AWS SageMaker** - Moderate: **deployKF**, **Scikit-learn**
Integration & Flexibility	- Strong Python integration: **Scikit-learn**, **H2O AI**, **deployKF** - Ecosystem-based flexibility: **Azure ML**, **AWS SageMaker**
Cost/Licensing	- Open-source: **Scikit-learn**, **H2O AI**, **deployKF** - Commercial/licensed: **Azure ML**, **AWS SageMaker**
Conclusion	**H2O AI**

2.2. Deployment

To deploy the machine-learning solution, I compared two container orchestration options: Kubernetes and Docker Compose. The decision was based on a set of weighted criteria, including own experience, scalability, complexity, performance, integration, and flexibility. (Attachment B)

Own experience was again a relevant factor. Because the deployment process needs to be efficient, familiarity with the tooling is important. I have more experience with Docker Compose, which allows for a quicker setup and less overhead. Kubernetes, while more powerful, has a steeper learning curve and requires more initial configuration.

Scalability was a key concern. Since the system may need to scale to handle more users or services in the future, Kubernetes scored significantly higher. Its native support for horizontal scaling, load balancing, and rolling updates makes it more suitable for production environments where uptime and scalability are critical. Docker Compose, while simpler, lacks these capabilities out of the box. (Isaiah, 2025)

In terms of **complexity**, Docker Compose is the more straightforward choice. It is easier to set up and manage, especially for smaller deployments or local testing. Kubernetes introduces more abstraction and complexity, which can slow down development if not managed properly.

Performance was another important factor. Kubernetes performed better in this area due to its more robust networking model and better resource scheduling across clusters. Docker Compose is sufficient for smaller deployments but can become a bottleneck under heavier workloads.

Both tools scored equally in terms of **integration**. They both support container-based deployment and integrate well with CI/CD pipelines and monitoring tools.

Flexibility was also evaluated. Kubernetes offers more options for customization, automation, and advanced deployment patterns such as blue-green or canary deployments. Docker Compose is more limited in this regard but still flexible enough for basic container management.

Based on the weighted ranking matrix, Kubernetes scored 8.00, making it the better long-term option due to its scalability, performance, and flexibility. Docker Compose scored 6.90, and while it is easier to use and a good fit for simpler setups, it lacks the advanced orchestration capabilities needed for production-ready systems. Therefore, Kubernetes is recommended as the final deployment platform. However, Docker Compose will be used as an intermediate step towards Kubernetes for which I will only provide an architecture diagram due to time constraints.

2.2.1. Evaluation Summary

The table below summarizes the key finding from the analysis of the deployment framework above:

Criterion	Highlights
Scalability	- Strong scaling support: **Kubernetes**
Complexity	- Simple and lightweight: **Docker Compose** - Higher setup overhead: **Kubernetes**
Performance	- Robust networking and resource scheduling: **Kubernetes** - Sufficient for smaller workloads: **Docker Compose**
Integration	- Good CI/CD and monitoring integration: Both **Docker Compose** and **Kubernetes**
Flexibility	- Advanced deployment strategies: **Kubernetes** - Basic container orchestration: **Docker Compose**
Conclusion	**Kubernetes**

2.3. Backend

To decide on which framework I will use, I made use of a weighted decision matrix. The backend will be written in Python, so only Python frameworks are considered. I compared three major frameworks based on several criteria. The frameworks in question are Flask (Flask, n.d.), Django (Django, n.d.), Streamlit (Streamlit, n.d.), and H2O Wave (H2O.ai, n.d.). These all have their strengths and weaknesses, so it's important to focus on the most relevant features for my project. (Attachment C)

Own experience was one of the factors considered in the framework evaluation, particularly due to the time constraints and the need to maintain development velocity. Familiarity with Scikit-learn allowed for quicker prototyping and reduced the learning curve, which made it a practical choice in the early stages. However, the final comparison considered a broader set of criteria—including scalability, distributed computing, integration flexibility, and cost—to ensure that the selected framework aligned with the long-term goals and technical requirements of the project.

Scalability is another critical factor, as the application needs to be able to support multiple users. Here, Django excels, as it is designed with larger, complex applications in mind, offering built-in features for handling large-scale projects, authentication, and user management. Flask, while scalable to some extent, is typically used for smaller applications, which suits my current needs for this project. Streamlit and H2O Wave, however, are primarily built for quick prototyping and is not designed to scale for production-ready applications. Its limitations in handling high loads or user concurrency make it a less viable option for a production system. (Toxigon, 2025)

Performance is another factor I considered when choosing between Flask, Django, and Streamlit. Flask is lightweight, which can be a benefit in terms of speed for small applications, as it does not add unnecessary overhead. It is also well-suited for API integrations, which is important for the machine learning pipeline in this project. Django's performance is solid but comes with a heavier structure due to its "batteries-included" approach, which might add some overhead for the specific needs of this project. Streamlit, although optimized for displaying data visualizations and simple UIs, is not ideal when considering performance for complex backend processes or large-scale deployments. H2O Wave offers very good performance, especially when it comes to model interactions. (Restack, 2024)

Security is another key criterion to consider for any web application. Django comes with many built-in security features, including protection against common threats such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF). It is a robust choice for securing an application out of the box. Flask does not come with built-in security features like Django but allows for flexibility in choosing the security measures you implement. You will need to manually integrate features like authentication, authorization, and protection against common vulnerabilities. Streamlit and H2O Wave, being focused on rapid prototyping and simplicity, do not offer much in terms of security out of the box and would require extra work to secure the application properly, especially for more sensitive data handling.

Integration was another key consideration, as the backend requires integration with APIs that connect to a machine-learning model. Flask is highly flexible and integrates well with various Python libraries, including scikit-learn, TensorFlow, and others. It also offers simplicity in building REST APIs, which is crucial for connecting the backend with the trained model. Django, with its more extensive features and structure, would likely require more effort to integrate. Streamlit, on the other hand, can quickly integrate machine learning models and is well-suited for visualizing model predictions and outputs, but it's not as well-suited for handling complex backend workflows and API integrations in a scalable way. Compared to H2O Wave, which is specifically designed for model interactions.

Flexibility was an important factor, as the project might need to be adapted and changed over time. Flask's flexibility shines here, as it provides a minimalistic framework that lets me design the app exactly as needed without enforcing any strict project structure. This is particularly useful when you have specific requirements and limited time. Django, with its opinionated structure, is more rigid, which could be a benefit in larger applications but could slow down development in smaller, more customized applications. Streamlit and H2O Wave offer great flexibility for building UIs quickly, but it is somewhat limited when it comes to backend features and customization beyond data display.

When adding up the scores based on these criteria, Flask comes out as the best option for my project. It provides the necessary flexibility, scalability (to a reasonable extent), and ease of integration with machine learning models, which makes it ideal for the limited scope and time I have. Django, Streamlit and H2O Wave each have their strengths, but for the specific requirements of this project, Flask strikes the best balance between performance, scalability, and development speed.

2.3.1. Evaluation Summary

The table below summarizes the key finding from the analysis of the backend framework above:

Criterion	Highlights
Scalability	- Strong support: **Django** - Moderate: **Flask** - Limited: **Streamlit, H2O Wave**
Performance	- Lightweight and fast: **Flask** - Robust: **Django** - Optimized for UI, not backend: **Streamlit** - High performance for model interaction: **H2O Wave**
Security	- Built-in protection: **Django** - Customizable: **Flask** - Limited out-of-the-box: **Streamlit**, **H2O Wave**
Integration	- High: **Flask** - Moderate: **Django** - Easy visualization: **Streamlit** - Model-focused: **H2O Wave**
Flexibility	- High: **Flask** - Moderate: **Django** - Lower: **Streamlit, H2O Wave**
Conclusion	**Flask**

2.4. Front-end

To make the user interface simple and straightforward, I used bootstrap (Bootstrap, n.d.) combined with inline styling. This gave me the ability to implement new features very fast and keep the styling consistent and simple. By making use of templates for each section, styling is kept consistent throughout the entire application. The reason for using inline styling is because of PDF generation. For this, I use WeasyPrint (WeasyPrint, n.d.), a library used for generating PDF files from HTML. This library is not compatible with bootstrap directly.

2.5. Authentication

Authentication is an important part of the application. Since the system works with sensitive information, it is important that only authorized users can access it. To choose the right solution, two options were compared: Flask-Security (Flask-security, 2025) and Auth0 (Auth0, 2025). Flask-Security is a module that works inside a Flask application, while Auth0 is a service that handles authentication in the cloud. Even though they are not the same type of tool, they both help with managing users and login functionality. That is why I compared them using a weighted decision matrix. (Attachment D)

The first thing that was considered was **own experience**. I had used Auth0 before in project 4.0, which made it easier to work with. Flask-Security is something I have not worked with, but setting up a user database is something I have done before in other projects. Because of this, Auth0 received a higher score for this criterion.

Scalability and **security** were the most important factors. Auth0 supports large systems, advanced login methods like multi-factor authentication, and protects against common attacks. Flask-Security does not include these features by default, so I would have to build and manage them myself, which would take more time and could be riskier.

Flask-Security scored higher on **complexity**, but this does not mean it is more difficult. In this case, it means that the code and logic are easier to understand and change. Auth0 is more powerful, but also harder to customize.

Both solutions offer great **integration** with the system. Flask-Security works naturally with Flask, and Auth0 provides libraries for Python and many other tools.

In the end, Auth0 was chosen because it offers more built-in features, is more secure, and fits better with the needs of the project. It saves time, requires less maintenance, and reduces the risk of making mistakes in the authentication process.

2.5.1. Evaluation Summary

The table below summarizes the key finding from the analysis of the authentication method above:

Criterion	Highlights
Scalability	- High: **Auth0**
Security	- Strong built-in: **Auth0** - Requires manual setup: **Flask-Security**
Complexity	- Lower code-level complexity: **Flask-Security** - Higher system complexity: **Auth0**
Integration	- Seamless: **Flask-Security** - Broad compatibility: **Auth0**
Conclusion	**Auth0**

2.6. Database

To determine which database will be used in the solution, a weighted decision matrix was used. Important criteria were examined, and the best fitting database was selected. The technologies compared were MySQL (MySQL, 2025), PostgreSQL (PostgreSQL, 2025), and MongoDB (MongoDB, 2024). Each of these systems offers specific strengths, but the final choice was guided by practical project needs and familiarity. (Attachment E)

Own experience was an important factor. Since I have the most experience with MySQL, this choice allows me to develop and maintain the solution more efficiently, especially under time constraints. PostgreSQL and MongoDB, while powerful, would require additional ramp-up time that could impact delivery speed.

Performance was another key consideration. Both MySQL and PostgreSQL demonstrate strong performance for structured data and transactional operations. MongoDB also performed well, particularly in document-based or unstructured use cases, but was slightly less suited for the structured relational model used in this project. (Vercel, 2023)

In terms of **flexibility**, MongoDB stood out due to its schema-less design, offering adaptability for evolving data structures. PostgreSQL also offers advanced features and extensibility. However, MySQL was still sufficiently flexible for the requirements of this application and comes with a more familiar operational model.

Integration with the rest of the stack was also evaluated. MySQL integrates well with Docker, Python, and existing pipeline components, giving it an edge.

Finally, **learning curve** was considered. MySQL scored highest due to existing knowledge, while PostgreSQL and MongoDB would require more time to learn.

Based on these factors, MySQL scored highest for this application. Allowing for rapid development of the database and allowing for flexibility towards the future.

2.6.1. Evaluation Summary

The table below summarizes the key finding from the analysis of the database above:

Criterion	Highlights
Performance	- Strong: **MySQL**, **PostgreSQL** - Good for unstructured: **MongoDB**
Flexibility	- High: **MongoDB** - Advanced features: **PostgreSQL** - Sufficient: **MySQL**
Integration	- Seamless: **MySQL**
Learning Curve	- Lower: **MySQL** - Steeper: **PostgreSQL**, **MongoDB**
Conclusion	**MySQL**

3. Initial Prototype and Feasability

In this chapter, I describe the prototype concept developed during the initial weeks of my internship. This prototype was used to communicate requirements and as a foundation for the beta version of the solution. It consists of two main components: an automated machine learning pipeline and a web application that serves the trained models. To build and test this system, I used a public dataset from Kaggle that closely resembles the real data. (Lapp, 2019)

3.1. Architecture

The current setup, as illustrated in Figure 1, uses two Docker containers to maintain a clean and modular architecture. The first container hosts the web application, which users interact with through their browser. This container is responsible for rendering the user interface and managing both authentication and authorization. These functions are handled through Auth0, which oversees the login flow, user roles, and access permissions using JWT tokens. By using Auth0, the application avoids the need to store passwords or deal directly with sensitive security logic.

Within the same container, a lightweight REST API enables communication with the second container: the model API container. This second container is responsible for handling all model-related functionality. Whenever the web application needs to make a prediction or retrieve information about a model, it sends a request to the model API container, which then loads the appropriate model, performs the necessary computations, and returns the results.

The model API container also stores all files essential to the operation of the machine learning models. These include the serialized model files, which contain the trained machine learning models used for inference, as well as the datasets on which the models were trained and evaluated. These datasets are also necessary for generating SHAP plots. In addition to these, several supporting files are included. A feature mapping file defines how input data collected from the application corresponds to the model's expected input format. Performance plots offer visual representations of how well each model performs, helping users choose the most suitable one. Finally, a scaler file ensures that input data is pre-processed in the same way as during the model's training phase, maintaining consistency in how predictions are made.

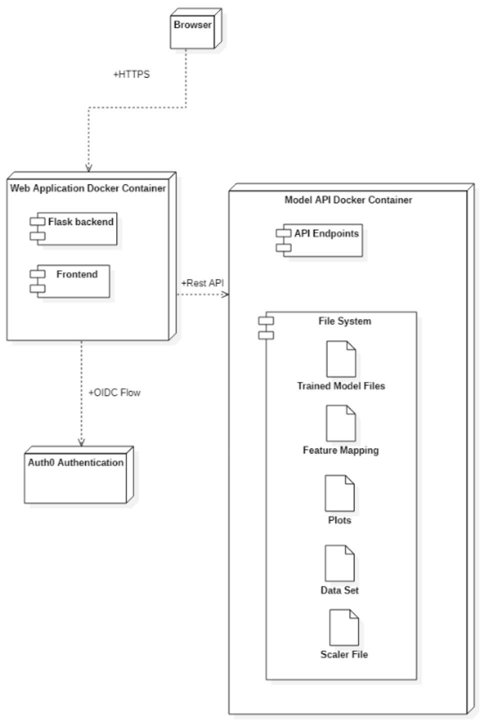


Figure 1. Architecture of the Prototype, showing the backend built with Flask, integration with the ML model, and data flow between components.

3.1.1. API

To enable communication between the two containers, several API endpoints have been implemented. The most important of these is the /predict endpoint, which handles POST request, as illustrated in Figure 2. This endpoint receives input parameters entered by the user through the input page, along with the selected model. These inputs are compiled into a JSON object and sent to the model API container. It is worth noting that this JSON may contain sensitive patient data. As this is a prototype, no security measures have been implemented to protect this data during transmission.

Upon receiving the request, the model API loads the specified machine learning model and performs a prediction based on the provided parameters. To enhance interpretability, a LIME (Ribeiro, 2016) explainer is used to generate feature importance scores for the input data. The resulting prediction and explanation are returned in a JSON format, which the Flask-based web application can then render on the frontend.

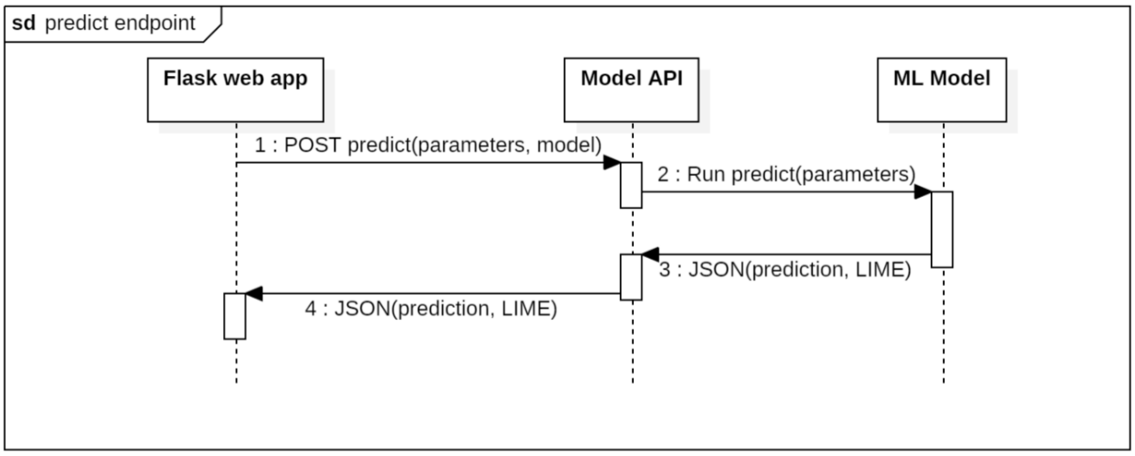

sd predict endpoint
Flask web app
Model API
ML Model
1 : POST predict(parameters, model)
2 : Run predict(parameters)
3 : JSON(prediction, LIME)
4 : JSON(prediction, LIME)

Figure 2. /predict API endpoint for handling machine learning inference by using POST requests.

In addition to the /predict endpoint, four other endpoints have been created, all of which use GET requests and follow a similar pattern, as shown in Figure 3. The /metrics endpoint requires the name of a model as a parameter and returns a JSON object containing model performance metrics such as accuracy, precision, recall, and the shape of the training data. The /models endpoint requires no parameters and returns a list of all available models in JSON format. Similarly, the /feature-mapping endpoint accepts a model name and returns the corresponding feature mapping, which is used to align the input data with the model's expected format. Lastly, the /plots endpoint also takes a model name as input and returns a JSON object containing base64-encoded plots, including a receiver operating characteristic (ROC) curve, a precision-recall curve, and a SHAP summary plot.

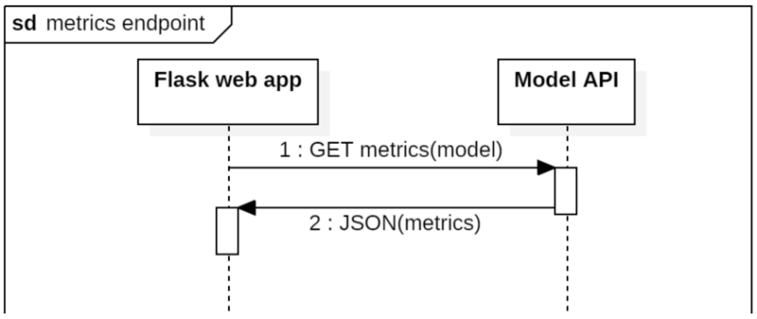


Figure 3. /metrics API endpoint for retrieving metrics using GET requests.

3.2. Pipeline

The first requirement for the prototype was to design and implement a pipeline that could train multiple machine learning models, gather key performance metrics, and prepare the models for deployment. For this purpose, a publicly available dummy dataset named “Heart Disease” was selected, as it closely aligns with the goals of the assignment. After selecting the dataset, a conscious decision was made to use Scikit-learn rather than H2O. This choice was based on the instruction to prioritize the development of the application rather than the automation of machine learning pipelines. Scikit-learn allowed for faster iteration and delivery, avoiding the steep learning curve that H2O might have introduced. The resulting pipeline consists of several interconnected steps, as illustrated in Figure 4.

The first phase is data ingestion, followed by data preprocessing. This begins with loading the CSV file into a Pandas DataFrame, enabling efficient manipulation for the upcoming steps. After the data is loaded, it is split into features and the target variable. The target variable is extracted from the DataFrame and stored separately to allow the models to train by comparing predictions against the actual labels.
Subsequently, the dataset is divided into a training set (80%) and a validation set (20%). The training set is saved in a pickle file within the model directory, as it is required later for generating Local Interpretable Model-agnostic Explanations (LIME). After splitting, the data is standardized using StandardScaler, ensuring that all numeric features are transformed to a uniform scale. The scaler object is also saved in the model directory so that the same transformation can be applied during inference.
An important part of preprocessing is the creation of a feature mapping. This involves generating a dictionary that maps feature indices to their corresponding column names. This mapping is essential for producing SHAP and LIME plots, which rely on feature names to provide interpretability. Although all models currently use the same mapping, it is saved individually in JSON format for each model to support future scenarios where models may be trained on different datasets. Additionally, the shape of the training data is saved for reference.

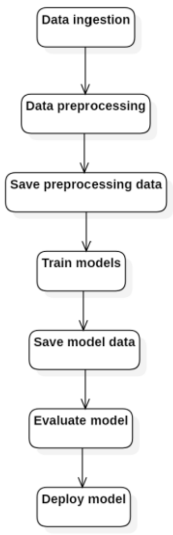
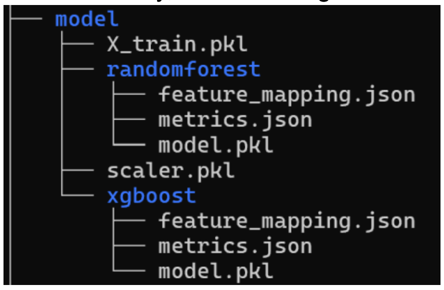


Figure 4. Pipeline flow.

The next phase of the pipeline is model training. For the prototype, two models were trained: an XGBoost classifier and a Random Forest classifier. These were selected to demonstrate the system's ability to manage and serve multiple models, rather than to optimize performance. After training, each model's performance is evaluated using accuracy, precision, and recall. These metrics are saved as JSON files, referred to as metrics.json, within each model's designated subdirectory, shown on Figure 5.
Each trained model is then serialized and stored as model.pkl. This file is placed in the same subdirectory alongside its corresponding metrics.json, the feature_mapping.json, and other supporting files, ensuring an organized and modular directory structure. Once the models are saved, further evaluation is conducted by generating three diagnostic plots: a Receiver Operating Characteristic (ROC) curve (Google, 2025), a Precision-Recall (PR) curve (Scikit-learn, n.d.), and a SHAP summary plot (Lundberg, 2018).



```
── model
    ├── X_train.pkl
    ├── randomforest
    │   ├── feature_mapping.json
    │   ├── metrics.json
    │   └── model.pkl
    ├── scaler.pkl
    └── xgboost
        ├── feature_mapping.json
        ├── metrics.json
        └── model.pkl
```

Figure 5. File structure prototype.

The ROC curve, shown in Figure 6, evaluates a model's classification performance across various thresholds by plotting the true positive rate against the false positive rate. The diagonal line represents a random classifier, while curves that bulge toward the top-left indicate better performance. The Area Under the Curve (AUC) score provides a single metric for evaluation—higher AUC values (closer to 1) indicate better discrimination between classes.

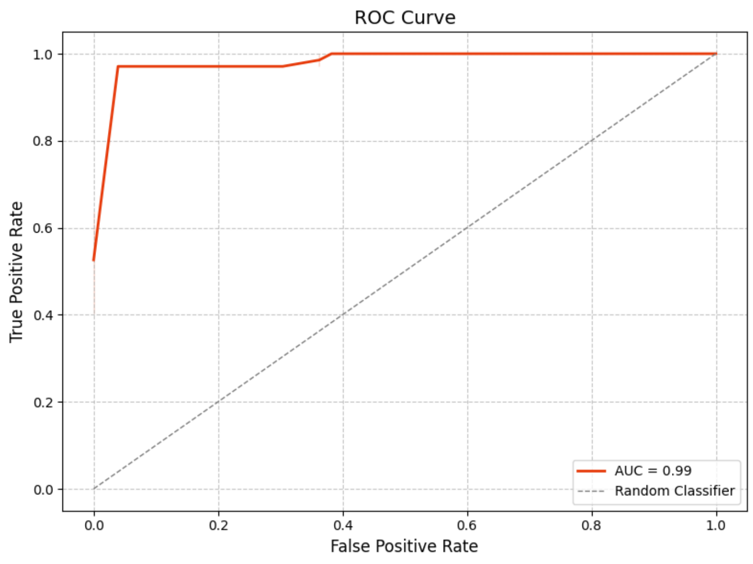


Figure 6. Receiver operating characteristics curve of dummy XgBoost model.

In contrast, the Precision-Recall curve, shown in Figure 7, focuses on the trade-off between precision (the accuracy of positive predictions) and recall (the ability to find all positive instances). This curve is particularly useful in cases of class imbalance. A strong model will have a PR curve that approaches the top-right corner. As with the ROC curve, the area under the PR curve summarizes overall performance.

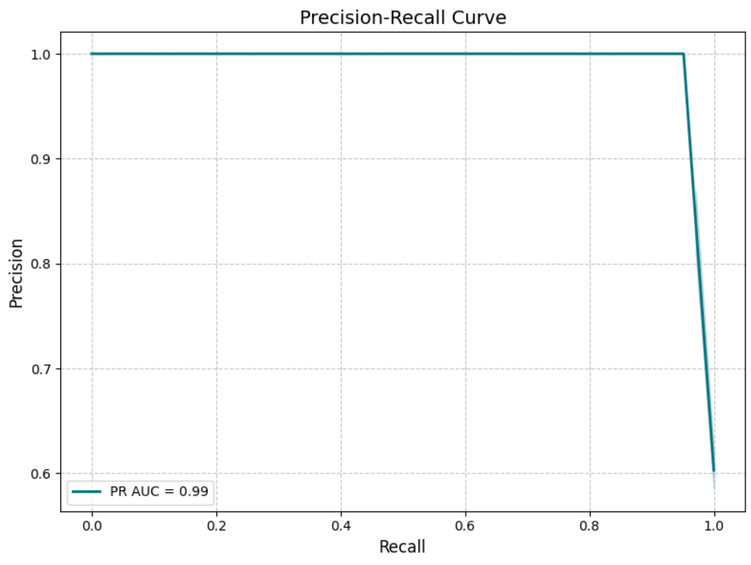


Figure 7. Precision recall curve of dummy XgBoost model.

The final evaluation tool is the SHAP summary plot, seen in Figure 8. SHAP (SHapley Additive exPlanations) values offer a way to understand how each feature influences the model's predictions. Features are ranked by importance, and each dot on the plot represents a single prediction. The horizontal position indicates whether the feature pushes the prediction higher or lower, while the color shows whether the feature's value is high (red) or low (blue). For example, in the case of the binary feature sex (0 = female, 1 = male), high values (male) tend to have a negative impact on the prediction, pulling it toward a "no disease" classification. Conversely, lower values (female) tend to push predictions toward a "disease" outcome.

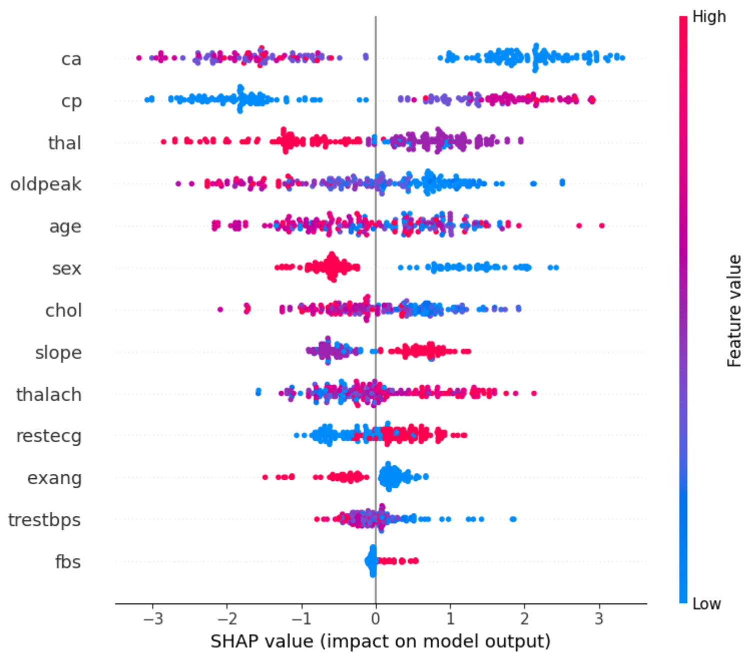


Figure 8. SHAP summary plot, of the XGBoost model, showing the impact of each feature on the model's output. Each dot represents a single prediction: red indicates higher feature values, while blue indicates lower ones. Features appearing further to the right contribute more strongly to a higher predicted risk of disease; those on the left have a lesser effect.

Abbreviation	Full Name	Description
Ca	Major Coronary Arteries (Fluoroscopy)	Number of major coronary arteries (0–3) coloured by fluoroscopy.
Cp	Chest Pain Type	0: Typical angina, 1: Atypical angina, 2: Non-anginal pain, 3: Asymptomatic
Thal	Thalassemia Test Result	3: Normal, 6: Fixed defect, 7: Reversible defect (from thallium stress test)
Oldpeak	ST Depression (Exercise)	ST depression induced by exercise compared to rest
Age	Age	Age of the patient in years
Sex	Biological sex	0: Female, 1: Male
Chol	Serum Cholesterol	Serum cholesterol level in mg/dL
Slope	ST Segment Slope	0: Upsloping, 1: Flat, 2: Downsloping (during exercise)
Thalach	Max Heart Rate Achieved	Maximum heart rate achieved during exercise
Restecg	Resting ECG Results	0: Normal, 1: ST-T abnormality, 2: Left ventricular hypertrophy
Exang	Exercise-Induced Angina	1: Yes, 0: No
Trestbps	Resting Blood Pressure	Resting blood pressure in mm Hg
Fbs	Fasting Blood Sugar > 120 mg/dL	1: True, 0: False

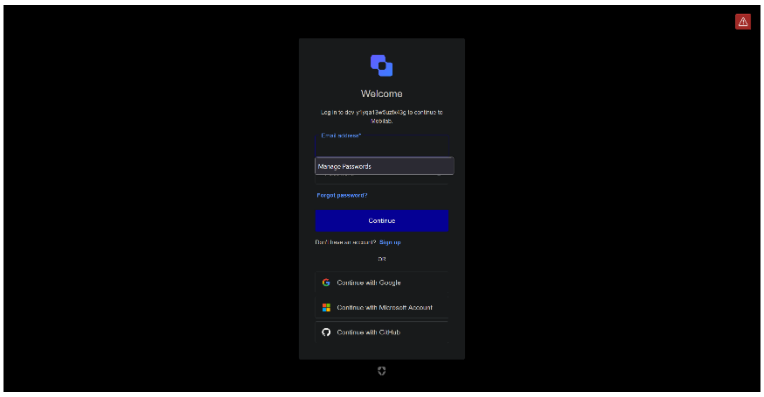
With the models trained, evaluated, and visualized, they are now ready for deployment. The next step in the system is to serve these models through an API that enables interaction via several endpoints. This API will form the backbone of the application's inference and interpretability features.

3.3. Application

To enable users to make predictions using the trained models, a user-friendly application was developed. This application functions as a decision support system for clinicians, helping them input patient parameters, generate predictions, and understand the reasoning behind those predictions. In developing the application, both user experience and security were prioritized, clinicians need fast, intuitive interactions, and at the same time, the application handles sensitive health data.

Upon opening the application in a browser, users are greeted with the standard login screen provided by Auth0 as seen on Figure 9. All authentication and authorization responsibilities are delegated to Auth0, ensuring modern, secure practices.

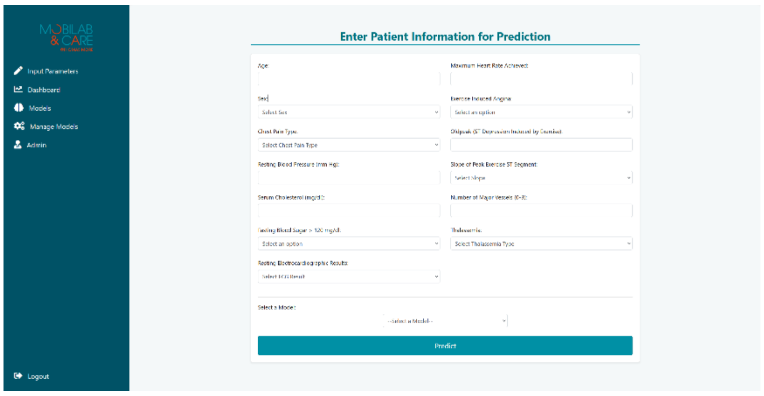
When a new user registers, they are not immediately granted access. Instead, they are redirected to a pending approval screen. During registration, metadata is attached to the user profile with an "approved: false" flag. This flag is embedded in the JWT token issued by Auth0, which also includes user roles. Until approval, the token reflects that the user is unauthorized, and they cannot access any functionalities of the application. Administrators manage user approvals through a dedicated admin panel (described later).



Log in to dev-y1yqa13w6uztk43g to continue to Mobilab.

Figure 9. Auth0 login screen.

Once an approved user logs in, they are taken to the Input Parameters page, seen on Figure 10. This is the main inference interface where users can enter relevant patient data, choose a model, and initiate a prediction by sending a POST request to the model API. The form input is packaged into JSON format and sent to the /predict endpoint, which returns another JSON response. This response includes the model's prediction result and LIME values (used for interpretability).

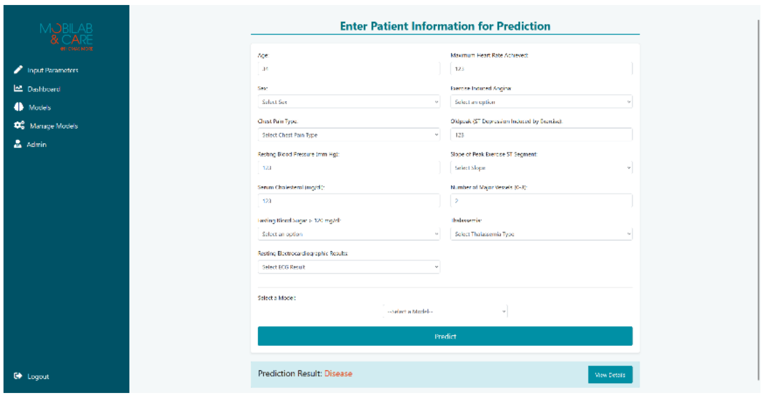


Enter Patient Information for Prediction

Age:
Maximum Heart Rate Achieved:
Sex:
Select Sex
Exercise Induced Angina:
Select an option
Chest Pain Type:
Select Chest Pain Type
Oldpeak (ST Depression Induced by Exercise):
Resting Blood Pressure (mm Hg):
Slope of Peak Exercise ST Segment:
Select Slope
Serum Cholesterol (mg/dl):
Number of Major Vessels (0-3):
Fasting Blood Sugar > 120 mg/dl:
Select an option
Thalassemia:
Select Thalassemia Type
Resting Electrocardiographic Results:
Select ECG Result
Select a Model:
--Select a Model--

Figure 10. Input parameters screen before making a prediction. Here, clinicians can enter patient biomarkers and select a trained machine learning model for inferencing.

Upon receiving a response from the model API, the prediction result is displayed under the form shown on Figure 11, along with a button linking to the dashboard. The application currently predicts whether a hypothetical patient is likely to have heart disease, based on the dummy dataset used.



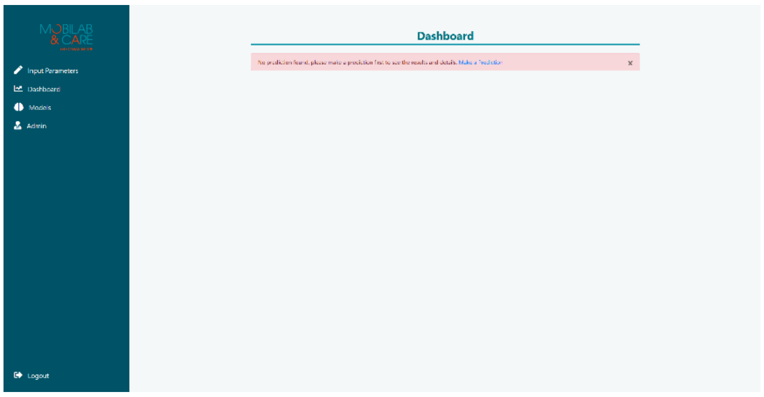
Enter Patient Information for Prediction

Age: 34
Maximum Heart Rate Achieved: 123
Sex: Select Sex
Exercise Induced Angina: Select an option
Chest Pain Type: Select Chest Pain Type
Oldpeak (ST Depression Induced by Exercise): 123
Resting Blood Pressure (mm Hg): 123
Slope of Peak Exercise ST Segment: Select Slope
Serum Cholesterol (mg/dl): 123
Number of Major Vessels (0-3): 2
Fasting Blood Sugar > 120 mg/dl: Select an option
Thalassemia: Select Thalassemia Type
Resting Electrocardiographic Results: Select ECG Result
Select a Model: --Select a Model--

Prediction Result: Disease

Figure 11. Input parameters screen after making a prediction with outcome disease.

The Dashboard page gives a detailed view of the prediction and interpretability results. If no prediction has been made yet, the application prompts the user to return to the input page and submit data, illustrated on Figure 12.

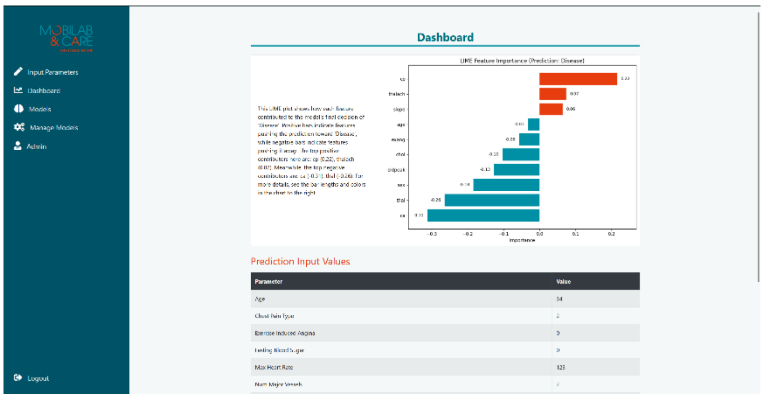


Dashboard

No prediction found, please make a prediction first to see the results and details. Make a Prediction

Figure 12. Dashboard without prediction.

Once a prediction is available, the LIME plot is displayed. LIME (Local Interpretable Model-agnostic Explanations) offers feature-level interpretation of an individual prediction, showing how specific input features influenced the result. Below the plot, the user's input parameters are listed in a table for reference, as shown on Figure 13.



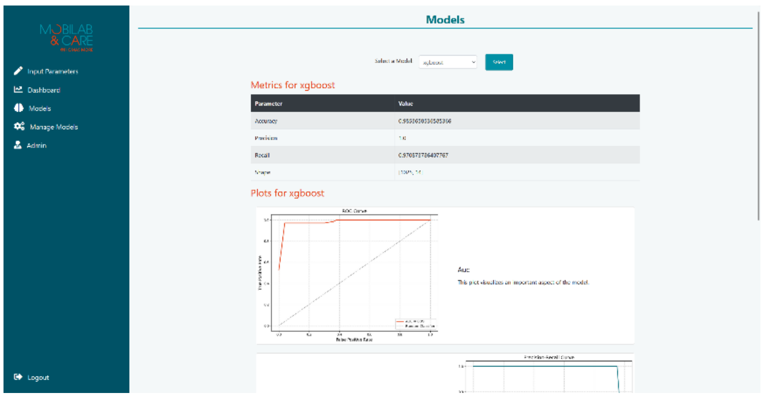
This LIME plot shows how each feature contributed to the model's final decision of 'Disease'. Positive bars indicate features pushing the prediction toward 'Disease', while negative bars indicate features pushing it away. The top positive contributors here are: cp (0.22), thalach (0.07). Meanwhile, the top negative contributors are: ca (-0.31), thal (-0.26). For more details, see the bar lengths and colors in the chart to the right.

Prediction Input Values

Parameter	Value
Age	54
Chest Pain Type	2
Exercise Induced Angina	0
Fasting Blood Sugar	0
Max Heart Rate	125
Num Major Vessels	2

Figure 13. Dashboard with prediction, providing an overview of the impact of each biomarker on the disease risk for an individual patient.

The Model page, seen on Figure 14, allows users to inspect performance metrics of a selected model, helping clinicians in their choice for the correct model. These metrics are the same as those calculated in the pipeline: accuracy, precision, and recall. The application retrieves this data via the /metrics and /plots endpoints. Metrics are returned as JSON, while plots are encoded in base64 and decoded on the frontend. Despite the minor performance cost, this method has minimal impact thanks to caching and the small number of plots involved.



Metrics for xgboost

Parameter	Value
Accuracy	0.9853658536585366
Precision	1.0
Recall	0.970873786407767
Shape	(1025, 14)

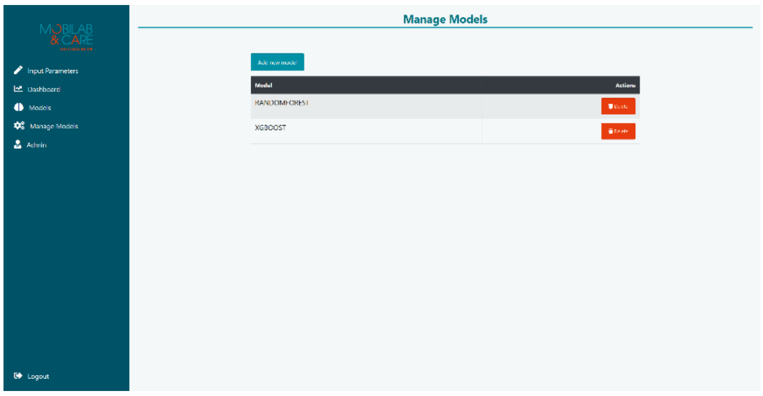
Plots for xgboost

Auc

This plot visualizes an important aspect of the model.

Figure 14. Models page for XgBoost model displaying metrics such as precision, recall, accuracy, and training shape. Plots highlighting the model performance are also included.

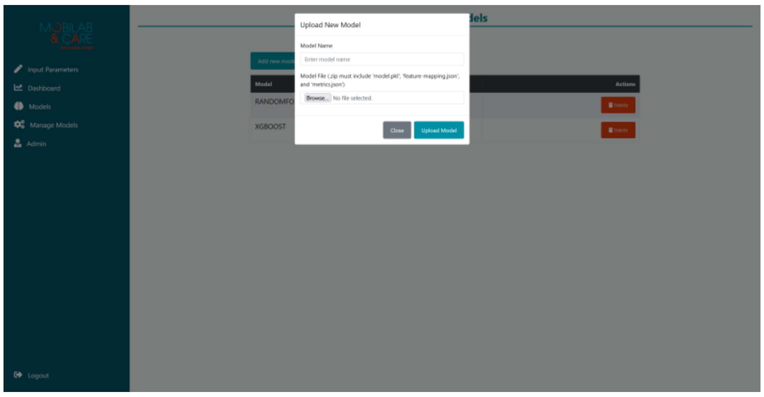
The Manage Models page is accessible only by administrators and can be seen on Figure 15. This page allows the admin to add or delete models used for inference. When uploading a new model, the administrator provides a name and uploads a .zip file containing: model.pkl, feature_mapping.json, and metrics.json. The backend checks if all files are present, repackages the zip with the correct name, and sends it to the model container, where it is extracted and placed in the /models directory. The container then scans for available models and updates the list accordingly.



Model	Actions
RANDOMFOREST	Delete
XGBOOST	Delete

Figure 15. Manage models page used for adding new machine learning models to the application or removing old ones.

To add a model, the process is initiated from the form shown below on Figure 16. This design ensures file validation is handled within the application container to reduce API interactions and protect the model container from overload.

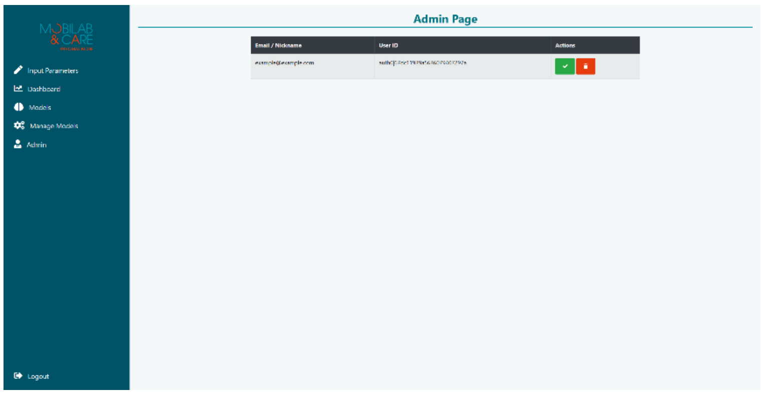


Model Name: Enter model name
Model File (.zip must include 'model.pkl', 'feature-mapping.json', and 'metrics.json')
No file selected.

Figure 16. Manage models page adding a new machine learning model by uploading a zip file with the appropriate files.

The final section is the Admin Panel, accessible only by users with the admin role encoded in their JWT token, displayed on Figure 17. This role is currently only assignable via the Auth0 dashboard. On this page, administrators can approve or reject new users.

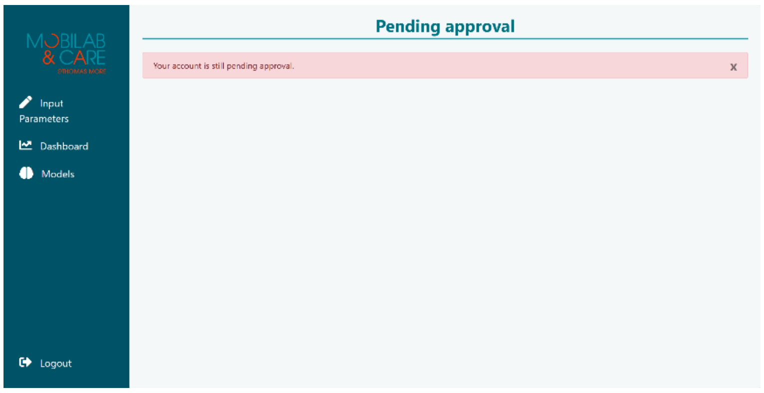
When a user registers, they must be approved before gaining access. Therefor, they are met with a warning message stating their unapproved state as show on Figure 18. If approved, a request is sent to the Auth0 Management API to change their metadata to approved: true. If rejected, the user is removed from the system. No notification is sent—the user must attempt to log in again to discover whether they've been approved.



Admin Page

Email / Nickname	User ID	Actions
example@example.com	auth0|[illegible]	✓

Figure 17. Admin page with a new user pending approval.



Pending approval

Your account is still pending approval.

Figure 18. New user awaiting approval to gain access to the application.

3.4. Possible Expansions

For this prototype, I did not include every possible feature. The main goal was to communicate requirements and test them out, so certain things were left out during development. One of the biggest missing features is a database, which could allow users to store past predictions and parameters. This would make inference faster since users wouldn’t have to fill in all the parameters manually each time. Additionally, storing models, plots, and other required files in a database could improve organization and retrieval. However, a shared volume could also be a viable alternative, as it would require fewer resources while still enabling efficient access to stored models and files.

Another feature that could be useful is the ability to upload a CSV file to automatically populate the input fields. This would be a great improvement for users who need to input large amounts of data. At the moment, the application also does not support generating or retrieving plots for uploaded models. This could be addressed in different ways. One option is for the administrator to include the plots in the zip file when uploading a model, ensuring they are placed in the correct folder and served to the frontend when needed. Another approach would be to allow users to upload or select a dataset on which the model was trained and then let the model container generate the necessary plots. However, this would introduce additional security concerns, as handling potentially sensitive data would require stricter access control.

Right now, the application uses Auth0 for authentication and authorization. The free plan of Auth0 allows up to 25,000 monthly active users and one active domain, which should be more than enough for the current scope of the project. It is also an easy-to-implement and low-maintenance solution, making it a good fit for this prototype.

Another improvement that should be considered for future development is refactoring the application's routing logic. Currently, the routes directly interact with the API for data retrieval, which works but could be better structured. A service layer should be introduced to keep the routes lean and ensure that all business logic is handled separately. This would improve maintainability and make it easier to expand the application later.

4. Application

To create the final application, the prototype was refactored into a more modular design. While the core functionality remains unchanged, the architectural structure of the code was reworked to better support future expansion and maintainability. To achieve this, a proper service layer was developed to handle all business logic within the application, using complementary design patterns that ensure a clean separation of concerns. The new version also introduces a database to store model metadata, a feature not present in the prototype. This addition further separates the prediction logic from the retrieval and management of metadata, streamlining the overall architecture. Furthermore, security measures were strengthened to protect against common web vulnerabilities, ensuring that the application can be deployed securely.

4.1. Architecture

The overall microservice architecture of the application is containerized using Docker Compose, as shown in Figure 19 below. This setup includes three containers that together form a modular and loosely coupled system. These containers run the web application, the model-serving API, and the MySQL database. Each container has a single, clearly defined responsibility, making the system easier to manage, test, and expand.

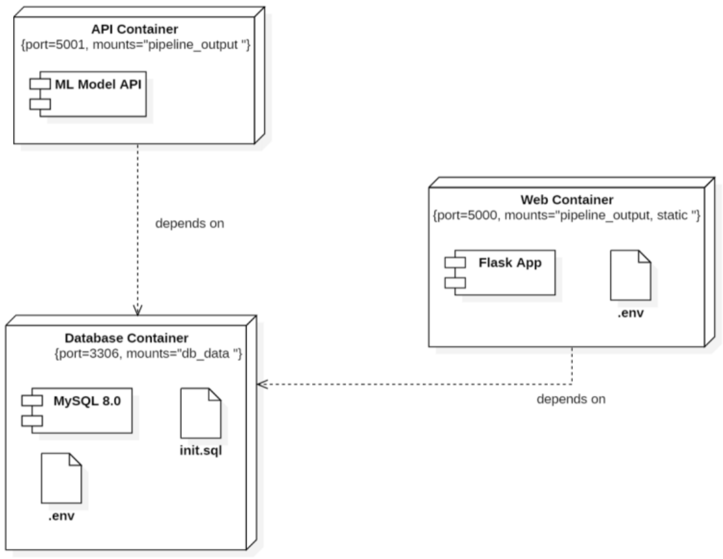

API Container
{port=5001, mounts="pipeline_output "}
ML Model API
depends on
Web Container
{port=5000, mounts="pipeline_output, static "}
Flask App
.env
Database Container
{port=3306, mounts="db_data "}
MySQL 8.0
init.sql
.env
depends on

Figure 19. Docker deployment diagram using three containers to deploy the micro service architecture.

At the centre of the setup is the container for the web application. This is the main access point for users and handles everything from rendering pages to sending out requests for predictions. This container includes both frontend and backend logic, bundled together for simplicity and easier local use. The application runs inside the container using Gunicorn (Gunicorn, n.d.), which is better suited for production environments compared to Flask’s default server.

The application depends on the database for its functionality. Without the database, no prediction can be made, no models can be trained, and no models can be viewed. To ensure that the application starts after the database, a health check has been added, more on that in the deployment chapter. On the application, a reconnection system was made so the application tries to reconnect to the database after losing connection. The application will try to reconnect to the database on its own five times, to mitigate small outages of the database. If the outage is longer, the application will try to reconnect with every SQL query executed so a connection can be made when needed again.

The database container runs a MySQL 8.0 instance and stores all model-related metadata. The schema and initial data are automatically set up using an SQL script that is mounted into the container. The database is linked to a persistent volume, which means the data remains available even if the container is restarted or rebuilt. This setup keeps the metadata consistent between runs and helps prevent data loss during development or testing.

The third container is used to serve the machine learning models. It exposes a simple API that accepts structured HTTPS requests containing prediction variables and a model name. It loads the appropriate model from a shared volume and returns a prediction. Like the web application, this API also runs on Gunicorn for better performance. Since this container is stateless, it can be rebuilt or restarted at any time without affecting its functionality, as long as the model files remain in place. In a cluster environment like Kubernetes these files will be placed inside a persistent volume claim, to ensure stability.

Using Docker Compose I created this intermediate step for the final deployment with Kubernetes. This was done to make the Kubernetes deployment easier and as a fallback solution for unforeseen circumstances that make the Kubernetes deployment unfeasible in the given timeframe.

4.2. Database

The database, seen in Figure 20, is designed to store only model metadata. This is the same type of data used in the earlier prototype, with the addition of two new plots—propensity (Geeks, 2025) and Love (CausalWizard, n.d.) plots, which can be seen on Figure X and X respectively—used specifically for causal inference models (Pearl, 2010). This was a deliberate decision. Storing sensitive information such as patient data or prediction results would introduce privacy risks and stricter security requirements that go beyond the scope of this project. It would also require more resources to manage and maintain. By focusing only on metadata, the database remains lightweight, secure, and easier to maintain.

The database structure has been kept intentionally simple and consists of three main tables: Model, Plot, and Metric. Together, these cover the most important information needed to track and manage trained machine learning models. This setup also supports versioning and future extension if needed.

The Model table stores general information about each trained model. This includes its name, type, version number, and feature mapping. The feature mapping shows which input features were used during training and in what order. This is important to ensure the model is used correctly during inference. The mapping is saved as a JSON object within a single column, which makes it easy to inspect and reuse when needed.

The Plot table tracks the visualizations generated during training, such as SHAP value plots, performance curves, or causal model diagnostics like propensity and Love plots. Rather than saving the images directly in the database, only their file paths are stored. This avoids making the database unnecessarily large. For the local version of the app, these plots are stored in the /static/plots folder inside the container. In a production setup, these would be stored on a shared persistent volume claim (PVC) to make them accessible even when containers are restarted or scaled.

The Metric table contains the performance values for each model. This includes metrics such as accuracy, a report, and the shape of the training data. For causal models, traditional accuracy does not apply. Instead, global summary metrics like average treatment effect (ATE) (CauzalWizard, n.d.) can be stored here. The report field can be used for any other metrics that need to be included. This flexible design allows each model to report metrics that are most relevant to its type. One global metric is always stored in a dedicated accuracy field, even for causal models—though in that case, the label may be adjusted accordingly to represent a meaningful summary value. This is done to give a quick overview of model quality in the UI without requiring users to dig into the full report. All performance data is linked to a specific model via foreign keys, making it easy to compare different models or model versions.

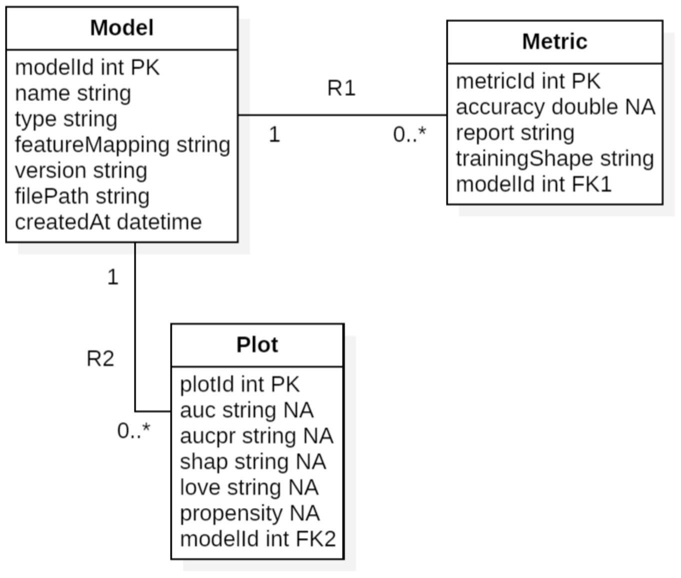


Figure 20. Database schema of the beta solution, describing three tables used for storing model meta data.

4.3. API

The application’s model-serving logic is encapsulated in a dedicated Flask-based API, deployed as its own container within the Docker Compose environment. This API exposes a single /predict endpoint, which has the same workflow as depicted on Figure 2, for serving predictions using a range of pre-trained machine learning models. Upon container startup, the API automatically loads all available models into memory from three predefined directories.

Two of these—/h2o_model and /pkl_model—contain test models used primarily during development. These allow for flexible experimentation with H2O and Scikit-learn formats, respectively. In addition, a shared volume (/pipeline_output) is mounted into the container, containing production-ready models trained using the application’s internal pipelines. This includes not only traditional classifiers, but also causal inference models and any future models generated by automated workflows. By standardizing model discovery at container boot time, the API ensures inference can be executed rapidly without per-request loading overhead.

The /predict endpoint expects a JSON payload with two keys: "model" and "features". The "model" field identifies the model to use by name (typically corresponding to its filename), and "features" contains a dictionary of input features, where each key is a feature name and each value is numeric. The API validates this input, checking that the specified model exists and ensuring the features are both complete and numerically valid. This robust validation layer prevents common runtime errors and protects against malformed input.

Inference is performed using either the H2O or Scikit-learn engine, depending on the model type. H2O models are passed data in H2OFrame format, while Scikit-learn models receive pandas. DataFrame objects. Both return a class prediction, the API also computes feature contribution scores (via SHAP for H2O or model introspection for Scikit-learn). These contributions are returned in the response alongside the prediction, enabling model interpretability and supporting downstream explanation visualizations in the frontend.

Extensive logging is implemented throughout the API. Key events—including model loading, requests received, input validation results, prediction execution, and error traces—are logged both to the console and to a persistent file. This aids in observability and debugging, especially in multi-container deployments.

To improve developer experience, the API integrates Swagger documentation, seen in the snippet below, via Flasgger. This provides an automatically generated web interface that describes the /predict endpoint, its parameters, expected payload structure, example input, and possible response formats. The underlying OpenAPI specification also serves as a machine-readable contract, facilitating easy integration and automated client generation.

```
 Predict endpoint for returning predictions and built-in contributions.
    ---
    tags:
      - Prediction
    consumes:
      - application/json
    parameters:
      - in: body
        name: body
        description: JSON payload containing the features and optional model
name.
        required: true
        schema:
          type: object
          properties:
            features:
              type: object
              additionalProperties:
                type: number
              example:
                {
                    "age": 12,
                    "sex": 1,
                    "chest_pain_type": 2,
                    "resting_blood_pressure": 123,
                    "serum_cholesterol": 12,
                    "fasting_blood_sugar": 0,
                    "resting_electrocardiographic": 1,
                    "max_heart_rate": 123,
                    "exercise_induced_angina": 0,
                    "oldpeak": 123.0,
                    "slope_of_peak_st_segment": 1,
                    "num_major_vessels": 2,
                    "thal": 1
                }
            model:
              type: string
              example: "DRF_1_AutoML_5_20250327_133650"
    responses:
      200:
        description: Successful prediction and explanation.
        schema:
          type: object
          properties:
            prediction:
              type: string
              example: "1"
            contributions:
```

Finally, the API includes comprehensive error handling. Bad requests (e.g. missing or invalid input) return 400-level errors with descriptive messages, while unexpected server-side issues are captured with 500-level responses. This ensures clarity for both developers and client applications, while abstracting away the complexity of model selection, format conversion, and inference logic.

4.4. Backend

The backend was developed with a clear architectural separation between routing, service logic, and utility functions, designed from the start to encourage modularity and long-term maintainability. While the initial prototype relied on inline logic inside the route definitions, the final iteration of the system is structured around the idea that the routing layer should be as thin and declarative as possible. Routes simply act as entry points into the system. They determine the context, validate user input where appropriate, and then defer responsibility to a well-scoped service class. This results in minimal logic per route, allowing anyone reading the code to quickly understand what each endpoint is doing without needing to process any domain-specific complexity.

A class diagram of the backend architecture is provided in Attachment F. It shows the relationships between core services, utility components, and the design patterns used (Factory and Builder). Providing a clear visual overview of how responsibilities are divided and how different parts of the system interact. The architecture is built around classes that handle core functionality, supported by lightweight helper functions for tasks such as formatting or session handling.

One note on the diagram: it includes a Routes class for illustration purposes only. In Flask, routes are typically defined as standalone functions and grouped using Blueprints, rather than being implemented within a class. The Routes class in the diagram serves to represent the routing layer conceptually and clarify how it fits into the overall structure.

All routes live under a single blueprint, grouped in a main module that reflects the primary flow through the application — from login to model interaction, to dashboard rendering to downloading reports. Despite serving a wide range of functionality, this routing layer remains lightweight by offloading responsibilities to services that are injected at runtime and made accessible via the Flask app context. A key benefit of this setup is that the app can be tested or extended without altering core routing logic.

At the centre of this structure is the Factory pattern, which is responsible for managing dependency injection across the entire backend. Rather than instantiating services manually or scattering object creation throughout the codebase, the factory serves as the single point of construction for all services and their dependencies. When the application starts, each service is instantiated once and registered with the Flask app context. These include, among others, the ModelDAO, which handles all database interactions; the FeatureService, which processes and validates form input for model compatibility; the PredictionService, which orchestrates the complete inference workflow; the APIClient, a focused utility for handling requests to the external model container; and the PipelineService which abstracts common pipeline functionalities to a central location. By centralizing object construction, the factory makes it trivial to mock, extend, or swap services as needed, without altering any of the surrounding logic.

Each service encapsulates a clearly defined set of responsibilities. For instance, the PredictionService owns everything involved in sending a prediction request to the model container, interpreting the results, and preparing a response for the user interface. Similarly, the ModelDAO abstracts away all interactions with the database, exposing simple methods that execute raw SQL statements to fetch available models, performance metrics, SHAP plots, and more. These services make no assumptions about request context or session state — they simply return structured data based on their input. This strict separation ensures business logic remains decoupled from the framework itself, making it easier to port or test in isolation.

Where data transformation or formatting is required, lightweight helper functions are used instead of embedding this logic within service classes. These functions handle tasks such as parsing form input, managing session data, or preparing prediction outputs for display. They are grouped by related functionality and follow the single-responsibility principle, ensuring clarity and reusability while avoiding unintended dependencies between components.

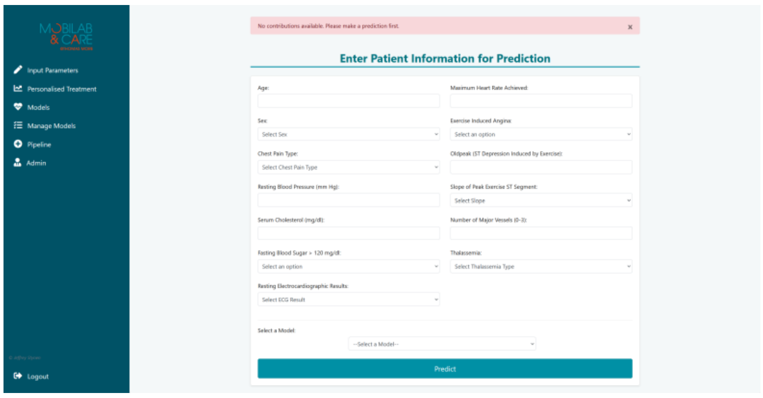
When more structured output is needed — particularly for reports — the backend applies the Builder pattern through the ReportBuilder and ReportDirector classes. These components work together to dynamically assemble structured content for both the dashboard and PDF generation. The builder defines how each section of the report is constructed in the form of templates, while the director coordinates the overall assembly. This pattern proved especially useful when maintaining parity between the web and PDF versions of the report, since it centralizes the content logic while allowing each output to customize its presentation. Without this pattern, rendering two report formats would have led to code duplication or tight coupling between view and data layers.

4.5. Frontend

Visually, the frontend has remained close to the original prototype, with the styling largely unchanged. However, beneath the surface, the way pages are constructed has been significantly improved to increase maintainability and flexibility. Instead of hardcoding each page individually, dynamic page generation was introduced, allowing templates such as the model overview, dashboard, and PDF report to be assembled from reusable building blocks.

This change was made possible by leveraging the builder pattern, already described in the backend section above, which allows frontend pages to be dynamically constructed based on the data provided. Thanks to this modular approach, updates to the layout or content structure can now be made centrally, rather than having to edit multiple templates manually. For example, if new insights or visual elements need to be added to the dashboard or report, they can simply be incorporated by extending the corresponding building blocks, without having to rewrite entire pages.

However, some pages underwent minor styling changes. The first page that was changed was the dashboard page, now renamed to personal treatment. As seen on Figure 21, when the user tries to navigate to the page, they are redirected to the input page and get a warning message that a prediction must be made first.



No contributions available. Please make a prediction first.

Enter Patient Information for Prediction

Age:

Maximum Heart Rate Achieved:

Sex: Select Sex

Exercise Induced Angina: Select an option

Chest Pain Type: Select Chest Pain Type

Oldpeak (ST Depression Induced by Exercise):

Resting Blood Pressure (mm Hg):

Slope of Peak Exercise ST Segment: Select Slope

Serum Cholesterol (mg/dl):

Number of Major Vessels (0-3):

Fasting Blood Sugar > 120 mg/dl: Select an option

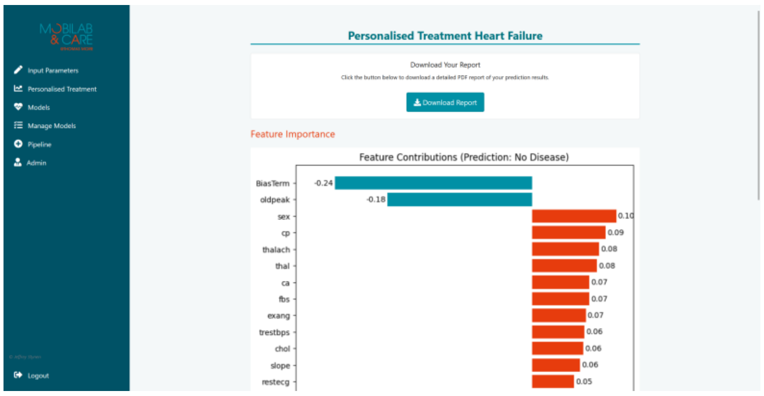
Thalassemia: Select Thalassemia Type

Resting Electrocardiographic Results: Select ECG Result

Select a Model: --Select a Model--

Figure 21. Input parameters page displaying a no prediction error. The user has to make a prediction first before they are able to navigate to the personal treatment page.

The content of the renamed personal treatment page was also restyled to accommodate the builder pattern. Now the contributions plot stands central on the page with the explanation below, making for better readability of the plot as seen on Figure 22. A new feature was implemented as well on this page and that is the ability to download a PDF report of the prediction.



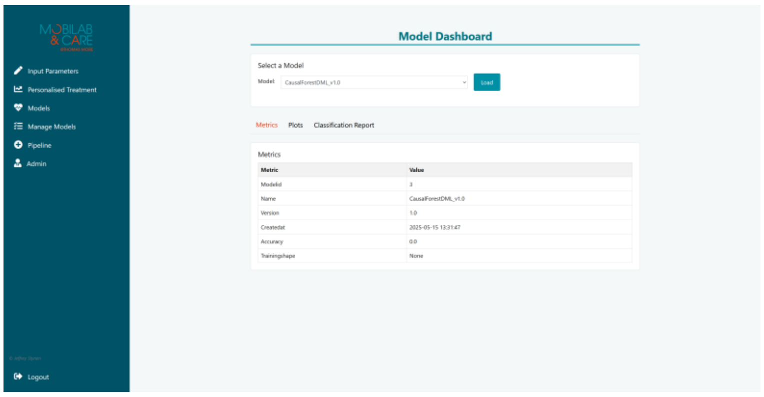
Personalised Treatment Heart Failure

Download Your Report

Click the button below to download a detailed PDF report of your prediction results.

Figure 22. Personal treatment page where the user can download a report of the prediction that was made or look at it directly. A contribution plot is provided displaying how much each feature contributed to the final prediction for an individual patient.

Another page that was changed by making use of the builder pattern was the model page. It now offers a tabular design seen on Figures 23, 24, and 25, making it easier for the user to find specific information about the model. The page now also offers a classification report which makes it very easy for the user to get a quick overview of the model performance illustrated on Figure 25.



Model Dashboard

Select a Model

Model: CausalForestDML_v1.0

Metrics

Metric	Value
Modelid	3
Name	CausalForestDML_v1.0
Version	1.0
Createdat	2025-05-15 13:31:47
Accuracy	0.0
Trainingshape	None

Figure 23. Model metrics, the first tab of the model dashboard. This tab gives a quick overview of the selected model.

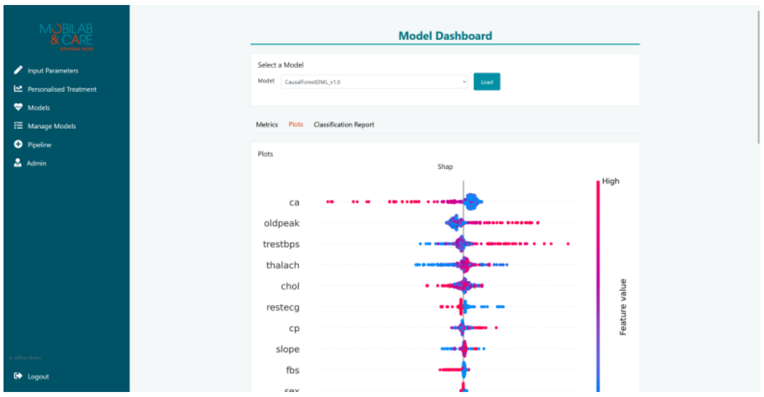
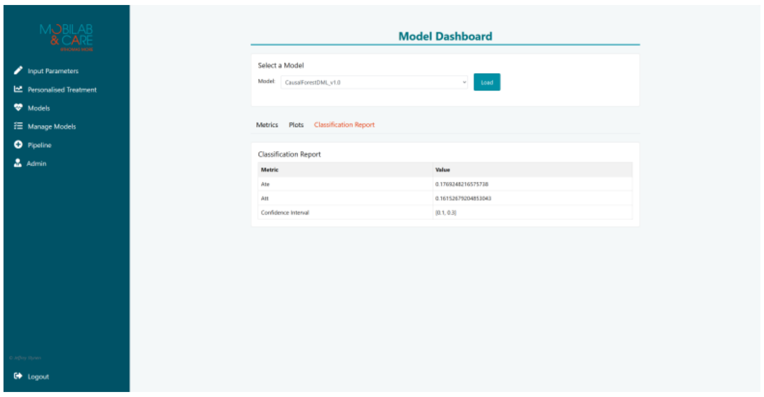


Figure 24. Model plots, the second tab of the model dashboard. This tab displays all available plots for the selected model.



Model Dashboard

Select a Model

Model: CausalForestDML_v1.0

Load

Metrics Plots Classification Report

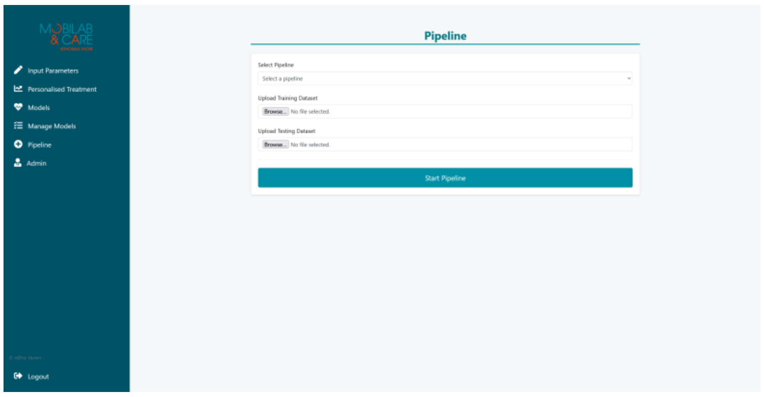
Classification Report

Metric	Value
Ate	0.1769248216575738
Att	0.16152679204853043
Confidence Interval	[0.1, 0.3]

Figure 25. Model classification report, the third tab of the model dashboard. This tab displays all metrics stored in the report field of the database dynamically.

An important consideration in generating the PDF reports is the styling. The library used for this task, WeasyPrint, does not support Bootstrap classes, which required the use of inline CSS styling. While this introduces some limitations — notably that Bootstrap styles do not carry over into the PDF — the inline styling provides the flexibility needed to ensure the reports are visually clear and well-structured. For the web-based pages, Bootstrap is still used where applicable to maintain a responsive and professional appearance.

Additionally, the manage models and pipeline page were introduced. The pipeline page, on Figure 26, is used for training new machine learning models. The user can upload a train and test dataset and select a desired pipeline. The results, the new model and its details, can be seen on the models page after the pipeline has finished. This page is only available for administrators.



Pipeline

Select Pipeline: Select a pipeline

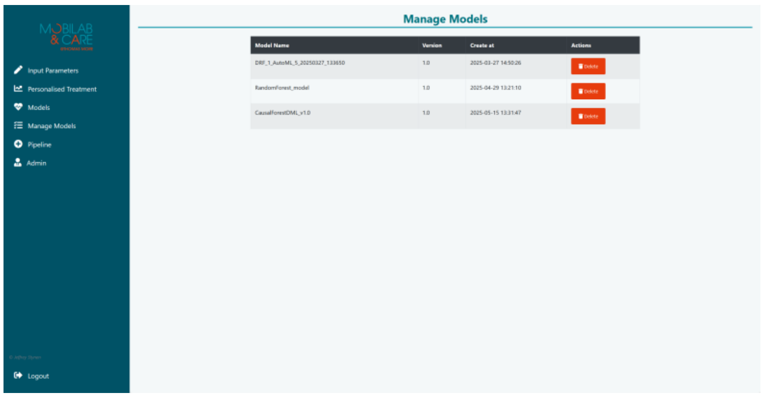
Upload Training Dataset: Browse... No file selected.

Upload Testing Dataset: Browse... No file selected.

Start Pipeline

Figure 26. Pipeline page, allowing an administrator to train a new machine learning model according to a predetermined pipeline that can be selected. The administrator has to provide a train and test dataset.

On the manage models page, seen on Figure 27, administrators can get an overview of all machine learning models present in the database. Here the user can delete models from the database, disabling them for inference. It is important to note that models are not completely deleted and will remain on the server to preserve historical models for future reference.



Manage Models

Model Name	Version	Create at	Actions
DRF_1_AutoML_5_20250327_133650	1.0	2025-03-27 14:50:26	Delete
RandomForest_model	1.0	2025-04-29 13:21:10	Delete
CausalForestDML_v1.0	1.0	2025-05-15 13:31:47	Delete

Figure 27. Manage models page, allowing administrators to delete a machine learning model from the database.

4.6. Security considerations

Throughout the development of the application, several security measures have been integrated to ensure safe operation, both for the local prototype and for future scalable deployments. These include authentication and authorisation, rate limiting, CSRF protection, as well as HTTP security headers.

Authentication and authorization within the application are managed using Auth0, a secure external identity provider. The application uses the OAuth 2.0 Authorization Code Flow with Proof Key for Code Exchange (PKCE). This flow is designed specifically for public clients, such as single-page or desktop applications, and offers enhanced security by avoiding the need to store client secrets.
In short, this flow works by redirecting the user to Auth0's login page, where they authenticate. Once authenticated, the application receives a temporary authorization code, which is exchanged for an access token using a secure, one-time-use code verifier. This ensures that only the legitimate application that initiated the login can complete the process. (Auth0, n.d.)

To enforce authorization, role-based access control (RBAC) is implemented throughout the application. Roles are defined in Auth0 and validated both in the frontend and backend. The backend uses custom decorators that inspect the user's session token for associated claims (such as roles), ensuring that only authorized users can access specific routes or actions within the app. Additionally, a JWT token is used to assign an approval status to users. Meaning, that a user must be approved by an administrator before they can get access to the application.

In addition to user authentication, rate limiting has been applied to critical routes, such as login and prediction endpoints. This limits the number of requests a user can make in a given time frame, protecting the application from brute-force attacks or accidental misuse that could lead to denial of service.
Cross-Site Request Forgery (CSRF) protection is enforced using Flask-WTF's CSRFProtect, which ensures that forms and state-changing actions cannot be submitted by malicious third-party sites. Combined with strong session management, this ensures that user actions remain trusted.

The application also includes carefully configured HTTP security headers to harden it against common web vulnerabilities. These include:

- Content Security Policy (CSP) to restrict where resources can be loaded from.
- X-Frame-Options to prevent clickjacking.
- X-Content-Type-Options to stop MIME type sniffing.
- Strict-Transport-Security (HSTS) to enforce HTTPS communication.
- Referrer-Policy to control how much referrer information is shared.

In terms of data handling, the application deliberately avoids handling sensitive patient data. The database stores only metadata about machine learning models, which limits the risk associated with data breaches.

Large files, such as plots and model artifacts, are kept out of the database and stored in the file system using persistent volumes, reducing the attack surface of the database.

On top of that, PDF reports and plots that are generated with each prediction are not stored as temporary files. These can contain sensitive information and would pose a security risk if left behind on disk. To mitigate this, the PDF report is generated entirely in memory using the WeasyPrint library. The report is first rendered as HTML, converted into a PDF byte stream on the fly, and then streamed directly to the user as a downloadable response—without ever touching the file system. This ensures that no sensitive artifacts remain on disk and that user data remains protected throughout the process. This same approach is applied to any plot images that may contain sensitive prediction explanations.

Additionally, the feature to be able to add new machine learning models by zip file upload was removed in order to mitigate file upload vulnerabilities. The trade-off between risk and reward was not worth having this feature.

For future scalability, the application architecture supports the separation of concerns, meaning sensitive services like the machine learning container and database are isolated from the web application itself. This not only improves maintainability but also reduces the risk of lateral movement in the event of a compromise.

5. Pipeline

This chapter describes the two main pipelines developed for the application: one for inference and one for training. Together, they form the core logic behind prediction, explanation, and model development. (Attachment F)

The inference pipeline handles user input in real time, transforming submitted form data into model-ready features, sending them for prediction, and returning both results and interpretability insights to the user. While it is not implemented as a formal pipeline class, it follows a structured and modular design. The logic is split across services, such as the FeatureService and PredictionService, which ensures each component has a clear responsibility. This setup makes the pipeline easy to understand, extend, and maintain.

The training pipeline is built with future scalability in mind. It allows different machine learning workflows — from traditional classification to advanced causal inference — to be implemented within a shared structure. Each training pipeline is defined as a class that inherits from a common BasePipeline, ensuring consistency while allowing for flexibility. This system supports a wide range of modelling approaches without requiring changes to the core application logic.

Together, these pipelines provide a complete, modular, and extensible foundation for deploying, explaining, and retraining machine learning models within the application. The following sections describe how each pipeline works, and how new components can be added with minimal effort.

5.1. Inference Pipeline

The inference pipeline in this project is responsible for turning user input into a model prediction, and then presenting both the result and an explanation. While it is not built with a formal pipeline framework, the structure follows a clear, step-by-step logic that reflects how a typical ML inference pipeline works in production. A sequence diagram of the prediction data flow is provided in Attachment G which outlines the entire flow.

It starts when a user fills out the input form in the web interface and submits it. This request is handled by a Flask route (/input), which takes care of user authentication and rate limiting. After that, the form is passed to the PredictionService, which coordinates the rest of the process.

The first part of the logic happens in the FeatureService. It reads the values from the form and builds a dictionary of input features. To make sure these inputs match what the selected model expects, it retrieves the feature mapping from the database — this is a mapping that was stored when the model was created or registered. It translates human-readable feature names (like "age" or "sex") to the internal feature names used by the model. If anything is missing or doesn't match the model's expectations, an error is raised and shown to the user.

Once the features are validated and mapped correctly, the PredictionService sends a request to the external prediction API. This request includes the structured features and the selected model name. As described in the API section, the response includes the prediction and the feature contributions (which help explain the model's decision).

After receiving the result, the contributions are passed back to the FeatureService, which handles explanation. First, it adjusts the values by including the bias term (if available), and then it generates a horizontal bar plot using Matplotlib. This plot shows which features had the biggest impact on the prediction and in which direction. Alongside the plot, a short explanation text is generated, describing the top positive and negative contributors. Both are returned to the frontend and shown to the user.

Even though the logic is split across different services, the whole flow — from form input to prediction and explanation — can be seen as a single pipeline. Each part has its own responsibility, which makes it easy to test, debug, and extend. For example, if new models are added later, or if causal inference or other analysis is introduced, the pipeline can be reused with minimal changes.

This inference pipeline plays an important role in the application. It connects the user interface to the machine learning models in a structured way, ensures that data is validated, and helps make the predictions more understandable.

5.2. Training Pipeline

This pipeline system is designed to be flexible and easy to extend. It allows developers to build and run machine learning workflows with minimal changes to the core code. The main idea behind this system is that each pipeline follows the same structure and communicates in the same way, no matter what kind of model it uses.

At the centre of this design is a base class called BasePipeline. Every pipeline you create must inherit from this class and implement a method called run(), which is the method the service uses to run the pipeline. This method is where the pipeline does its main job — loading data, training a model, calculating metrics, and returning results. Because every pipeline follows this same format, the rest of the system can treat all pipelines the same way. It does not need to know which exact model is being used, or how the pipeline works inside. Other helper functions can be used when implementing the BasePipeline, as long as the steps of the pipeline are defined in the run() function.

There are already two examples of pipelines in this system, displaying the flexibility of the implementation. One is a standard machine learning pipeline using scikit-learn and XGBoost. It trains a classifier to make predictions and calculates common metrics like accuracy and AUC. The other pipeline is more advanced: it uses the EconML library to estimate causal effects, such as the Average Treatment Effect (ATE). This pipeline also introduces the love and propensity plot, seen in Figure 28 and 29 respectively. Even though these two pipelines do very different things, they both follow the same structure. They both inherit from the base class, and they both return their results in the same format. This shows that the system can support many kinds of models, from simple to complex.

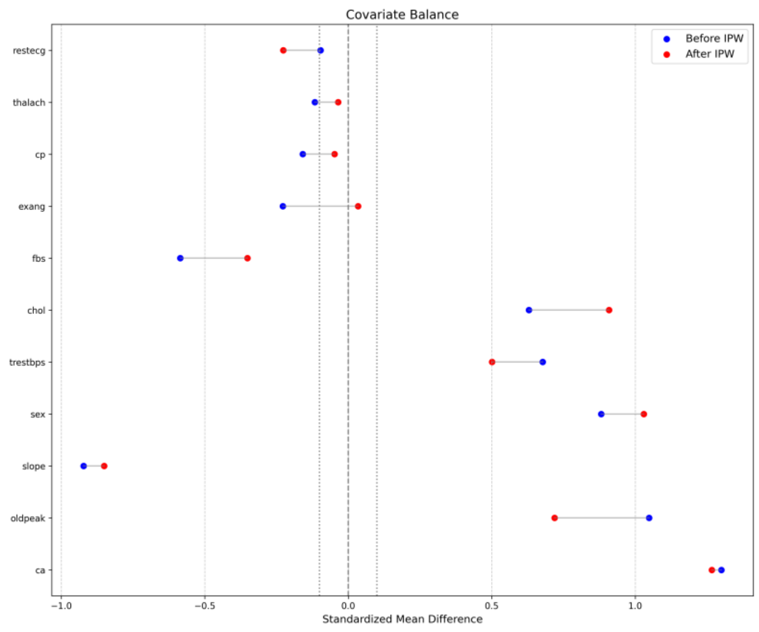


Figure 28. Love plot showing the Standardized Mean Differences (SMD) of each feature before and after applying Inverse Probability Weighting (IPW). Each line represents the imbalance of a covariate between treatment groups. Points closer to zero indicate better covariate balance. The plot illustrates how IPW improves balance across features, which is essential for valid causal inference.

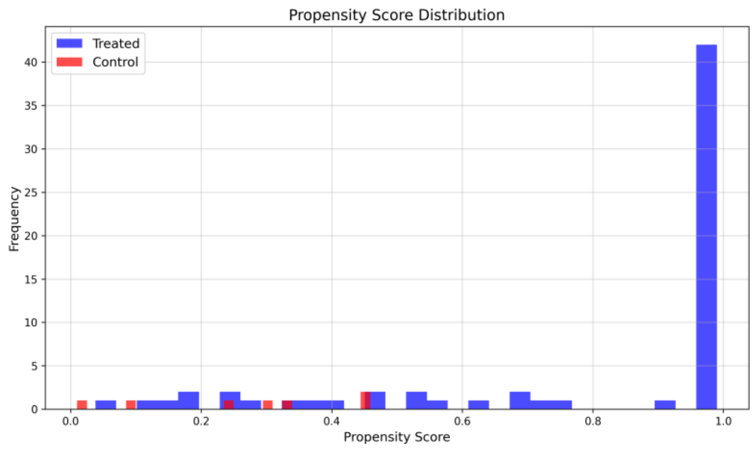


Figure 29. Histogram showing the distribution of propensity scores for the treated and control groups. The plot illustrates how well the treatment and control populations overlap in terms of their estimated probability of receiving the treatment. Good overlap (common support) is essential to ensure that comparisons between groups are valid for causal inference.

One key reason the system is easy to extend is that it separates different responsibilities. For example, pipelines do not load or save data by themselves. Instead, they use a helper called PipelineService to do that. This means that if you want to add a new kind of plot that can be implemented by other pipelines, you only have to add a new function that handles plotting logic to the PipelineService and add it to the dictionary of plot functions. This makes the new plot available to all pipelines without having to repeat the code for making the plot. Below is an example of the plot functions dictionary.

```
self.plot_functions = {
            "generate_love_plot": self._generate_love_plot,
            "generate_shap_plot": self._generate_shap_plot,
            "generate_propensity_plot": self._generate_propensity_plot,
            "generate_auc_plot": self._generate_auc_plot,
            "generate_aucpr_plot": self._generate_aucpr_plot
        }
```

Another reason the system is flexible is that it uses a pipeline registry. When a new pipeline is added in the pipeline directory, the system will automatically detect this new pipeline and make it available on the application. When the pipeline is selected, the service makes a new instance of that pipeline in order to run it. For this it's important that the naming conventions must be followed by the developer. That is, the class name must start with a capital letter and must be appended with the word 'Pipeline'. For example, 'CausalPipeline'.

5.2.1. Adding a new pipeline

Adding a new pipeline is straightforward. First, you create a new file in the pipeline directory of the application and define a new class according to the naming conventions which inherits from the BasePipeline class. Then you define a function run(), which is the method called by the service for executing the pipeline, that defines all steps of the pipeline and returns data to the service. Helper methods should be used to further abstract the logic in the pipeline class to maintain a clean run() function. The data that is returned to the service should be in the following format.

```
return {
            "metadata": metadata,
            "metrics": metrics,
            "plots": plots
        }
```

This dictionary contains three dictionaries inside. The metadata dictionary should follow the format below in order to work properly.

```
"metadata": {
        "name": "CausalForestDML_v1.0",
        "type": "Causal",
        "version": "1.0",
        "createdAt": "2025-05-15 10:30:00",
        "featureMapping": {
            0: "age",
            1: "income",
            2: "education_level",
            3: "employment_status",
            # etc...
        }
    },
```

The metrics dictionary has a required field report which can be defined with any metrics required. This field was deliberately setup this way because metrics can vary greatly between pipelines. This approach allows for very flexible metric fields on the front end of the application. The overall structure should follow this pattern.

```
     "metrics": {
        "accuracy": 0,  # Use 0 or other metric, for causal models
        "report": {
            "ate": 0.17,
            "att": 0.21,
            "confidence_interval": "[0.1, 0.3]"
        },
        "training_shape": {
            "rows": 1023,
            "columns": 12
        }
    },
```

To generate plots, no plotting logic should be defined in the pipeline itself. This logic resided inside the PipelineService to keep the plots consistent between models and make implementing the pipeline easier. If new plots are required, they should be defined in the PipelineService as a function and added to the dictionary as described in the previous section. To use existing plots, the necessary data for the plot should be passed in the following format to the service and the plot will be automatically generated and added to the database.

```
plot_inputs = {
            "love_plot": {
                "function": "generate_love_plot",
                "data": {"X": X, "T": T, "weights": weights}
            },
            "shap_plot": {
                "function": "generate_shap_plot",
                "data": {"model": model, "X": X}
            },
             "propensity_plot": {
                "function": "generate_propensity_plot",
                "data": {"propensity_scores": t_pred_safe, "T": T}
            }
        }

        # Let the service generate all standard plots for this pipeline.
        plots = service.generate_plots(metadata["name"], plot_inputs)
```

Adhering to this structure is important, otherwise, the data will not be stored in the database. This structure can be maintained regardless of the complexity of the pipeline and can be expanded by making changes in the PipelineService.

6. Deployment

To make the solution accessible for other parties, a deployment strategy was required. The final target environment was designed to follow a microservice architecture deployed using Kubernetes. However, due to time constraints and the absence of server hardware, a fully configured Kubernetes cluster was not realized during the project.

As an intermediate and practical alternative, Docker Compose was used. This provided a lightweight orchestration layer that was helpful during development and also served as a temporary deployment mechanism. It allowed for modular testing, simplified service management, and established a strong foundation for a future transition to Kubernetes.

6.1. Docker Compose

As stated above, this setup acted as an intermediate step before making the transition to Kubernetes. The architecture of this setup is divided into three separate containers and can be seen on Figure 19: the web application, the machine learning API, and the database. This was done to resemble the final Kubernetes microservice architecture as closely as possible while maintaining full separation of concerns.

The web application container is responsible for handling the frontend, backend, and user interactions. It is built from a dedicated Dockerfile and exposes its services on port 5000. This container requires the database to be running for the best user experience. To configure the application securely, an external .env file is loaded at runtime, as shown in the snippet below. While this method is practical during development, it is not recommended for production environments. In such cases, sensitive configuration values like credentials or API keys should be passed as system environment variables or managed through a dedicated secrets management solution. This approach minimizes the risk of accidental exposure and aligns with best practices for secure deployments.

```
app:
    build:
      context: ./application
      dockerfile: Dockerfile
    ports:
      - "5000:5000"
    env_file:
      - ./application/.env
    depends_on:
      db:
        condition: service_healthy
    restart: always
```

The machine learning container acts as an independent API that serves machine learning models for inference tasks. This container is also built using a dedicated Dockerfile and is exposed on port 5001. The machine learning API is lightweight and fully decoupled from the web application, allowing each service to scale independently if needed. Its service configuration in Docker Compose is:

```
api:
    build:
      context: ./api
      dockerfile: Dockerfile
    ports:
      - "5001:5001"
    restart: always
```

The database container runs a MySQL 8.0 instance and is responsible for the persistent storage of model metadata. Credentials such as database name, user, and password are injected via environment variables to avoid hardcoding sensitive information. A persistent volume is mounted to store database data across container restarts, and an additional volume is attached to allow initialization scripts to automatically provision the database schema and seed it with dummy data. The database configuration looks like this:

```yaml
db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: ${DB_PASS}
      MYSQL_HOST: ${DB_HOST}
      MYSQL_DATABASE: ${DB_NAME}
      MYSQL_USER: ${DB_USER}
      MYSQL_PASSWORD: ${DB_PASS}
    ports:
      - "3306:3306"
    volumes:
      - db_data:/var/lib/mysql
      - ./scripts:/docker-entrypoint-initdb.d
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      interval: 10s
      timeout: 5s
      retries: 5
    restart: always
```

To ensure reliable service orchestration, a health check has been configured on the database container. This mechanism prevents the web application from starting before the database is healthy and available, reducing the risk of runtime errors during startup. All containers also automatically restart when they are down. This ensures that the entire architecture remains online.

By introducing this Docker Compose setup, modular development and testing became possible, while laying a strong foundation for production-grade container orchestration in Kubernetes later on. Concepts such as service health monitoring, environment-based configuration, persistent storage, and strict separation of concerns have already been incorporated at this stage, facilitating an easy transition towards a scalable microservice architecture.

6.2. Kubernetes

The Kubernetes deployment, shown in Figure 30, represents the intended production-ready architecture intended to replace the earlier Docker Compose setup. Although the deployment was ultimately not carried out due to hardware constraints and time limitations, the architecture itself demonstrates a modular, secure, and scalable system built with modern best practices in mind.

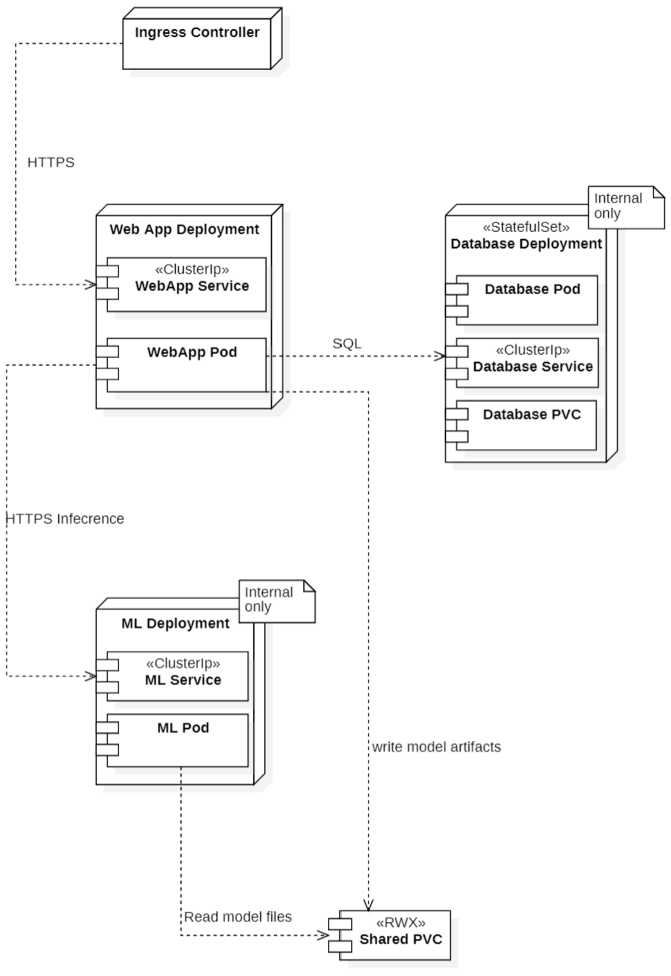


Figure 30. Kubernetes deployment diagram displaying how the micro service architecture could be deployed on a server.

The architecture is composed of three main components: the web application, the machine learning (ML) service, and the database. These components are deployed independently within the cluster using native Kubernetes resources such as Deployments, Services, and PersistentVolumeClaims (PVCs).

The web application is managed via a Deployment and is exposed to external users through an Ingress Controller. This ingress component handles HTTPS traffic and routes it to a ClusterIP service, which in turn directs requests to the web application pod. This separation of external and internal traffic provides a secure entry point while limiting exposure of internal services. The web app communicates with the internal database using SQL queries, fetching or updating metadata as needed. By using ClusterIP, access to the web service remains restricted to internal traffic unless explicitly routed through the ingress layer.

The machine learning deployment is a standalone Deployment that serves HTTPS inference requests from within the cluster. Like the web application, it is exposed internally using a ClusterIP service to prevent external access. This ensures that only trusted components, such as the web application, can interact with the machine learning API. The model files required for inference are loaded from a shared persistent volume, ensuring consistent access to the latest versions. The shared volume is declared using a PersistentVolumeClaim (PVC) adopting a ReadWriteMany (RWX) volume to allow simultaneous access by multiple pods if needed.

In the current implementation, the pipeline logic responsible for processing uploaded files and generating model artifacts resides within the web application service. As shown in the diagram, this service writes directly to the shared persistent volume (RWX) that also serves the machine learning component. While this tight coupling simplifies development and was appropriate for the project timeline, separating the pipeline into an independent service would be a logical next step in aligning more closely with microservice architecture principles. This would allow for better scalability, maintainability, and failure isolation.

The database is deployed as a StatefulSet, a Kubernetes construct designed to manage stateful applications like databases. It uses a dedicated PVC to persist its data and a ClusterIP service for internal communication. This setup ensures that the database can retain data across pod restarts and maintain a consistent network identity. As a security measure, the database service is only accessible from within the cluster and is not exposed to the public internet. Only the web application and machine learning service are authorized to communicate with it, which reduces the system's attack surface.

Although this Kubernetes deployment was not realized in practice, the architectural decisions were made with future scalability and security in mind. The use of internal-only services, persistent storage, secure ingress routing, and isolated concerns lays a solid foundation for future container orchestration in a production environment.

7. Courses

During my internship I did two complementary courses to aid me in my assignment. These courses helped me along the way and cleared up some uncertainties I had about my ability to complete the assignment.

7.1. Kubernetes Core Concepts

I started this learning path on the Kube Academy because I was unfamiliar working with microservices or designing microservices architectures. This course did not really go into the specifics about Kubernetes architecture or microservices, but it was a good way to refresh the Kubernetes skills I learned in school during the Linux Webservices course. I also learned some new things regarding volumes, workloads, and security in Kubernetes. (Schneider, n.d.)

More specifically, learning about volumes was crucial for my assignment. As model artifacts and plots need to be persistent and not ephemeral. Learning about this helped me to plan out the storage solution with persistent volume claims in the architecture described above. It became clear to me how Kubernetes separates the storage from the pod lifecycle, so even if a pod restarts or moves to a different node, the data can still be retained.

I also gained a better understanding of how workloads are managed in Kubernetes. This includes the different types of controllers like Deployments, StatefulSets, and Jobs. Each of these has its own use case — for example, Deployments are good for stateless applications that can scale easily, while StatefulSets are used when each instance needs a stable identity and storage, which is often the case in data-heavy applications. Jobs and CronJobs, on the other hand, are useful for running one-off or scheduled tasks, like retraining a model or generating reports at regular intervals.

Another important aspect was security. The cluster must be secure as to prevent attacks. For this, it was important that I knew the best practises. The course showed me how to limit access through role-based access control (RBAC), so that users and services only get the permissions they really need. It also covered the importance of network policies, which control traffic between pods and can be used to isolate sensitive components. On top of that, I learned how Kubernetes secrets can be used to safely manage things like API keys and passwords, rather than hardcoding them into configuration files.

Overall, even though the course was fairly high-level, it helped me connect a lot of smaller pieces and apply them to the project I was working on. It was not so much about learning everything from scratch, but about understanding how different Kubernetes features work together in practice. Especially when it comes to managing data, workloads, and keeping things secure.

7.2. Machine Learning & Causal Inference

This online course provided by Stanford University on YouTube was challenging. It required knowledge about statistics and a deeper understanding of machine learning algorithms to be able to follow. Luckily, I was better able to follow due to completing another course on basic statistics before my internship. That does not mean, however, that I was able to understand everything taught in the course as it re (Schneider, n.d.)quired too much background information. (Stanford Graduate School of Buisiness, 2021)

The most important thing I learned in the course was the average treatment effect (ATE) and the causal average treatment effect (CATE). The first of which describes the average effect of a treatment or intervention in a population. It compares the expected outcome if everyone were treated with the expected outcome if no one were. The CATE, on the other hand, looks at how the treatment effect changes across different groups or individuals based on their characteristics. This is important when effects are not the same for everyone.

In observational data, people are not randomly assigned to treatments. This can lead to bias if people who get the treatment are systematically different from those who don’t. These differences are called confounding variables. For example, people with higher income might be more likely to get access to a medical treatment, but also more likely to have better health in general. If we don’t control for these variables, we might wrongly attribute the health differences to the treatment itself.

To deal with this problem, the course introduced the concept of the propensity score. This is the probability that a person receives the treatment, given their background variables. If we can estimate this score accurately, we can use it to make the treated and untreated groups more comparable.

One method that uses the propensity score is called inverse probability weighting (IPW). It tries to correct for the fact that people who receive treatment might be different from those who don’t. It does this by giving more weight to people who were less likely to receive the treatment they actually got, based on their propensity score. This creates a sort of balance, as if the data came from a randomized experiment.

An improved version of this is called augmented inverse probability weighting (AIPW). This method combines two different models: one that predicts how likely each person was to receive the treatment (the propensity model), and another that predicts what their outcome would be with and without the treatment (the outcome model). If at least one of these models is correct, the method still gives a good estimate. This is why it's called “doubly robust.”

The course also showed how machine learning can be used to build these prediction models more flexibly. Algorithms like random forests or gradient boosting can find complex patterns in the data. But using powerful models can also lead to overfitting, which happens when the model learns noise instead of signal. To avoid this, the course introduced regularisation techniques like Lasso. Lasso is a type of linear model that automatically leaves out variables that aren’t useful. It’s helpful when you have many variables and want to focus only on the most important ones.

Later in the course, the focus moved from estimating average effects to estimating effects for individuals or groups — the conditional average treatment effect (CATE). This helps answer the question “Who benefits most from the treatment?” instead of just “Does the treatment work on average?” This is useful in real-world decision-making, such as deciding who should receive a medical treatment or participate in a policy program.

Overall, this course helped me understand how machine learning can be used in causal inference, and how statistical thinking is still important when applying these tools. Even though some of the material was difficult, I came away with a better sense of how to estimate causal effects using modern methods.

8. Conclusion

Looking back, this internship has been an invaluable learning experience. I was exposed to a wide range of concepts, many of which were entirely new to me, such as microservice architectures and container orchestration. Working on a decision support system for clinical use, aimed at improving understanding of cardiovascular disease, was a unique and meaningful opportunity.

Throughout the project, I applied technologies and design patterns I encountered during my bachelor's degree, including API development and layered application design. Doing so helped me realize how broadly applicable my current skills are, but also how much there still is to learn, a realization that keeps me excited and motivated to grow further.

The solution I delivered lays a strong foundation, but it is not without room for improvement. Future enhancements could include a dedicated pipeline microservice and a fully realized Kubernetes deployment. These additions fell outside the scope of this internship, but I am confident they would have further deepened my understanding and strengthened the system as a whole.

Despite the constraints, I am proud to have delivered a working solution that aligns with the project goals. I hope my work contributes meaningfully to the ongoing efforts within the Mobilab&Care group and forms a solid stepping stone for future development.

9. Reference list

Amazon. (2025). *Amazon SageMaker*. Retrieved from https://aws.amazon.com/sagemaker/

Amazon. (n.d.). *Amazon SageMaker AI pricing*. Retrieved from AWS: https://aws.amazon.com/sagemaker-ai/pricing/?

Auth0. (2025). *Start Building*. Retrieved from Auth0: https://auth0.com/docs

Auth0. (n.d.). *Authorization Code Flow with Proof Key for Code Exchange (PKCE)*. Retrieved from auth0: https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-pkce

Bootstrap. (n.d.). *Introduction*. Retrieved from Bootstrap: https://getbootstrap.com/docs/4.5/getting-started/introduction/

CausalWizard. (n.d.). *Covariate Balance*. Retrieved from CausalWizard: https://causalwizard.app/inference/article/covariate-balance

CauzalWizard. (n.d.). *Average Treatment Effect (ATE)*. Retrieved from CauzalWizard: https://causalwizard.app/inference/article/ate

deployKF. (2024). *deployKF*. Retrieved from https://www.deploykf.org/

Django. (n.d.). *Django Documentation*. Retrieved from Djangoproject: https://docs.djangoproject.com/en/5.2/

Flask. (n.d.). *Flaks documentation*. Retrieved from Flask: https://flask.palletsprojects.com/en/stable/

Flask-security. (2025). *Flask-security*. Retrieved from Flask-security: https://flask-security-too.readthedocs.io/en/stable/index.html

Geeks for Geeks. (2024, Septermber 25). *Flask vs Django – Which Framework Should You Choose in 2024*. Retrieved from Geeks for Geeks: https://www.geeksforgeeks.org/flask-vs-django/

Geeks, G. f. (2025, May 14). *Propensity Score Matching*. Retrieved from Geeks for Geeks: https://www.geeksforgeeks.org/propensity-score-matching/

Google. (2025, April 15). *Classification: ROC and AUC* . Retrieved from https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc

Gunicorn. (n.d.). *Gunicorn*. Retrieved from Gunicorn: https://gunicorn.org/

H2O.ai. (2025). *The World's Best Deep Research*. Retrieved from h2o: https://h2o.ai/

H2O.ai. (n.d.). *H2O Wave Documentation*. Retrieved from H20.ai: https://docs.h2o.ai/h2o_wave/index.html

Isaiah, A. (2025, March 24). *Docker Compose vs Kubernetes* . Retrieved from Better stack: https://betterstack.com/community/guides/scaling-docker/docker-compose-vs-kubernetes/#comparison-scalability

Joblib. (2021). *Joblib: running Python functions as pipeline jobs*. Retrieved from Joblib: https://joblib.readthedocs.io/en/stable/

Lapp, D. (2019). *Heart Disease Dataset*. Retrieved from Kaggle: https://www.kaggle.com/datasets/johnsmith88/heart-disease-dataset

Lundberg, S. (2018). *An introduction to explainable AI with Shapley values*. Retrieved from shap.readthedocs.io/en/latest/example_notebooks/overviews/An introduction to explainable AI with Shapley values.html

Microsoft. (2025). *Azure Machine Learning documentation*. Retrieved from https://learn.microsoft.com/en-us/azure/machine-learning/?view=azureml-api-2

Microsoft. (n.d.). *Azure Machine Learning pricing*. Retrieved from Microsoft: https://azure.microsoft.com/en-us/pricing/details/machine-learning/

MongoDB. (2024). *MonogDB Documentation*. Retrieved from MonogoDB: https://www.mongodb.com/docs/

MySQL. (2025). *Documentation*. Retrieved from MySQL: https://dev.mysql.com/doc/

Pearl, J. (2010, February 26). *An Introduction to Causal Inference*. Retrieved from PubMed Central: https://pmc.ncbi.nlm.nih.gov/articles/PMC2836213/

PostgreSQL. (2025). *Documentation*. Retrieved from PostgreSQL: https://www.postgresql.org/docs/

Restack. (2024, November). *Streamlit vs Flask vs Django comparison*. Retrieved from Restack: https://www.restack.io/docs/streamlit-knowledge-streamlit-vs-flask-vs-django

Ribeiro, M. T. (2016). *Local Interpretable Model-Agnostic Explanations (lime)*. Retrieved from https://lime-ml.readthedocs.io/en/latest/

Schneider, R. (n.d.). *Kubernetes Core Concepts*. Retrieved from Kube Academy: https://kube.academy/paths/kubernetes-core-concepts

Scikit-learn. (n.d.). *precision_recall_curve*. Retrieved from scikit-learn: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html

Scikit-learn. (n.d.). *scikit-learn machine learning in python*. Retrieved from scikit-learn: https://scikit-learn.org/stable/index.html

Stanford Graduate School of Buisiness. (2021, August). *Machine Learning & Causal Inference: A Short Course*. Retrieved from YouTube: https://www.youtube.com/playlist?list=PLxq_lXOUlvQAoWZEqhRqHNezS30lI49G-

Streamlit. (n.d.). *Streamlit Documentation*. Retrieved from Streamlit: https://docs.streamlit.io/

Toxigon. (2025, January 19). *Django vs Flask: Comparing Python Web Frameworks in 2025* . Retrieved from Toxigon: https://toxigon.com/django-vs-flask-a-comparison-of-python-web-frameworks

Vercel. (2023, November 8). *Comparing MySQL, PostgreSQL, and MongoDB* . Retrieved from Vercel: https://vercel.com/guides/mysql-vs-postgresql-vs-mongodb

WeasyPrint. (n.d.). *The Awesome Document Factory*. Retrieved from weasyprint: https://weasyprint.org/

Attachments

ATTACHMENT A

Weighted ranking matrix to determine the pipeline framework to be used.

Criteria	Weight	Scikit-learn	H2O	deployKF	Azure ML	AWS SageMaker
Own Experience	0,20	7	3	2	4	4
Scalability (large data sets)	0,20	5	9	9	8	8
Distributed computing build in	0,20	5	10	10	9	9
Model training speed	0,10	6	8	7	8	8
Integration	0,10	9	9	8	9	9
Flexibility	0,10	9	9	9	8	8
Cost/Licensing	0,10	10	9	10	4	4
Total	1,00	6.80	7.75	7.70	6.90	6.90

ATTACHMENT B

Weighted ranking matrix to determine the deployment framework to be used.

Criteria	Weight	Kubernetes	Docker-compose
Own Experience	0,10	5	7
Scalability	0,20	9	7
Complexity	0,10	6	4
Performance	0,20	8	5
Integration	0,30	9	9
Flexibility	0,10	8	7
Total	1,00	8	6.90

ATTACHMENT C

Weighted decision matrix to determine the backend framework to be used.

Criteria	Weight	Flask	Django	Streamlit	H2O Wave
Own Experience	0,20	5	0	6	2
Scalability	0,20	7	9	4	4
Performance	0,15	7	7	5	8
Security	0,20	7	9	5	2
Integration	0,15	7	7	9	10
Flexibility	0,10	9	7	5	4
Total	1,00	6.8	6.6	5.6	4.7

ATTACHMENT D

Weighted decision matrix to determine the authentication framework to be used.

Criteria	Weight	Flaks-Securty	Auth0
Own Experience	0,15	3	5
Scalability	0,20	6	9
Complexity	0,15	8	4
Integration	0,20	9	9
Security	0,30	6	9
Total	1,00	6	7.85

ATTACHMENT E

Weighted decision matrix to determine the database to be used.

Criteria	Weight	MySQL	PostgreSQL	MongoDB
Own Experience	0,20	9	5	3
Performance	0,25	8	8	7
Flexibility	0,20	7	8	9
Integration	0,20	9	8	6
Learning Curve	0,15	8	7	6
Total	1,00	8.15	7.25	6.40

ATTACHMENT F

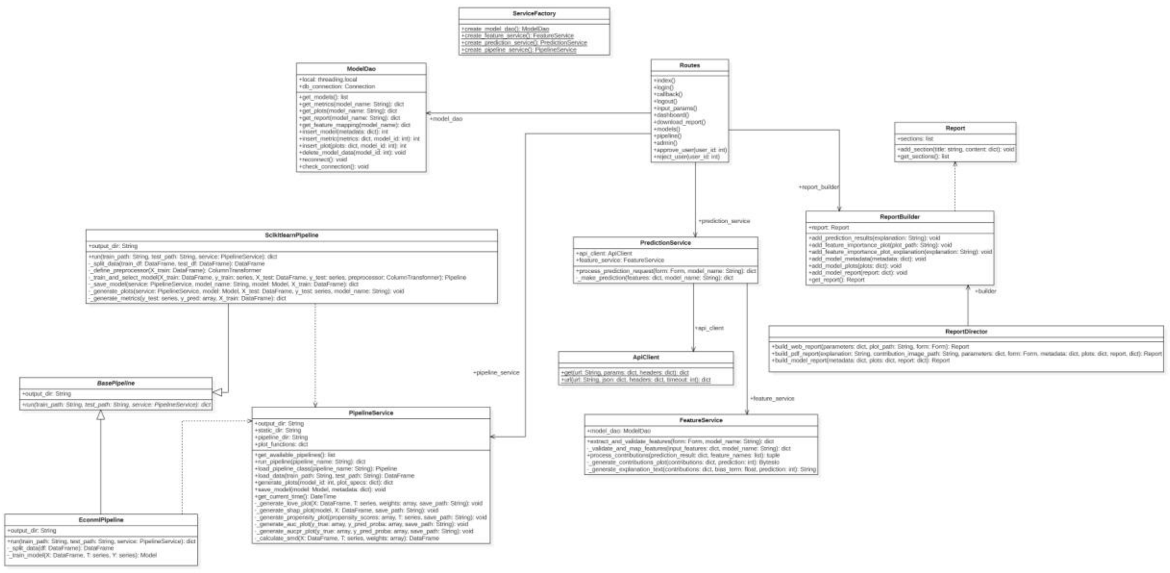

ServiceFactory
+create_model_dao(): ModelDao
+create_feature_service(): FeatureService
+create_prediction_service(): PredictionService
+create_pipeline_service(): PipelineService
ModelDao
+local: threading.local
+db_connection: Connection
+get_models(): list
+get_metrics(model_name: String): dict
+get_plots(model_name: String): dict
+get_report(model_name: String): dict
+get_feature_mapping(model_name): dict
+insert_model(metadata: dict): int
+insert_metric(metrics: dict, model_id: int): int
+insert_plot(plots: dict, model_id: int): int
+delete_model_data(model_id: int): void
+reconnect(): void
+check_connection(): void
+model_dao
Routes
+index()
+login()
+callback()
+logout()
+input_params()
+dashboard()
+download_report()
+models()
+pipeline()
+admin()
+approve_user(user_id: int)
+reject_user(user_id: int)
Report
+sections: list
+add_section(title: string, content: dict): void
+get_sections(): list
+report_builder
+prediction_service
ReportBuilder
+report: Report
+add_prediction_results(explanation: String): void
+add_feature_importance_plot(plot_path: String): void
+add_feature_importance_plot_explanation(explanation: String): void
+add_model_metadata(metadata: dict): void
+add_model_plots(plots: dict): void
+add_model_report(report: dict): void
+get_report(): Report
SciktlearnPipeline
+output_dir: String
+run(train_path: String, test_path: String, service: PipelineService): dict
-_split_data(train_df: DataFrame, test_df: DataFrame): DataFrame
-_define_preprocessor(X_train: DataFrame): ColumnTransformer
-_train_and_select_model(X_train: DataFrame, y_train: series, X_test: DataFrame, y_test: series, preprocessor: ColumnTransformer): Pipeline
-_save_model(service: PipelineService, model_name: String, model: Model, X_train: DataFrame): dict
-_generate_plots(service: PipelineService, model: Model, X_test: DataFrame, y_test: series, model_name: String): void
-_generate_metrics(y_test: series, y_pred: array, X_train: DataFrame): dict
PredictionService
+api_client: ApiClient
+feature_service: FeatureService
+process_prediction_request(form: Form, model_name: String): dict
-_make_prediction(features: dict, model_name: String): dict
+builder
ReportDirector
+build_web_report(parameters: dict, plot_path: String, form: Form): Report
+build_pdf_report(explanation: String, contribution_image_path: String, parameters: dict, form: Form, metadata: dict, plots: dict, report: dict): Report
+build_model_report(metadata: dict, plots: dict, report: dict): Report
+api_client
+pipeline_service
ApiClient
+get(url: String, params: dict, headers: dict): dict
+url(url: String, json: dict, headers: dict, timeout: int): dict
BasePipeline
+output_dir: String
+run(train_path: String, test_path: String, service: PipelineService): dict
+feature_service
PipelineService
+output_dir: String
+static_dir: String
+pipeline_dir: String
+plot_functions: dict
+get_available_pipelines(): list
+run_pipeline(pipeline_name: String): dict
+load_pipeline_class(pipeline_name: String): Pipeline
+load_data(train_path: String, test_path: String): DataFrame
+generate_plots(model_id: int, plot_specs: dict): dict
+save_model(model: Model, metadata: dict): void
+get_current_time(): DateTime
-_generate_love_plot(X: DataFrame, T: series, weights: array, save_path: String): void
-_generate_shap_plot(model, X: DataFrame, save_path: String): void
-_generate_propensity_plot(propensity_scores: array, T: series, save_path: String): void
-_generate_auc_plot(y_true: array, y_pred_proba: array, save_path: String): void
-_generate_aucpr_plot(y_true: array, y_pred_proba: array, save_path: String): void
-_calculate_smd(X: DataFrame, T: series, weights: array): DataFrame
FeatureService
+model_dao: ModelDao
+extract_and_validate_features(form: Form, model_name: String): dict
-_validate_and_map_features(input_features: dict, model_name: String): dict
+process_contributions(prediction_result: dict, feature_names: list): tuple
-_generate_contributions_plot(contributions: dict, prediction: int): BytesIo
-_generate_explanation_text(contributions: dict, bias_term: float, prediction: int): String
EconmlPipeline
+output_dir: String
+run(train_path: String, test_path: String, service: PipelineService): dict
-_split_data(df: DataFrame): DataFrame
-_train_model(X: DataFrame, T: series, Y: series): Model

Figure 31. Class diagram depicting the application backend architecture.

ATTACHMENT G

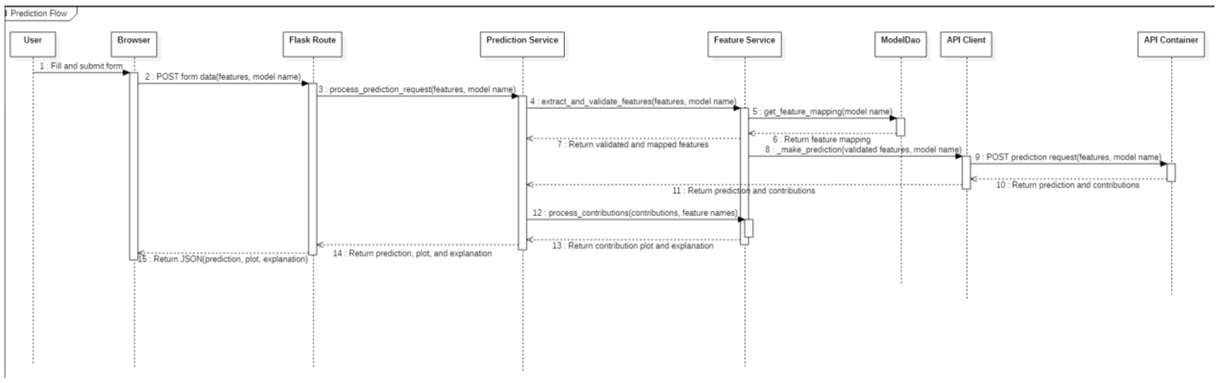

Prediction Flow
User
Browser
Flask Route
Prediction Service
Feature Service
ModelDao
API Client
API Container
1 : Fill and submit form
2 : POST form data(features, model name)
3 : process_prediction_request(features, model name)
4 : extract_and_validate_features(features, model name)
5 : get_feature_mapping(model name)
6 : Return feature mapping
7 : Return validated and mapped features
8 : _make_prediction(validated features, model name)
9 : POST prediction request(features, model name)
10 : Return prediction and contributions
11 : Return prediction and contributions
12 : process_contributions(contributions, feature names)
13 : Return contribution plot and explanation
14 : Return prediction, plot, and explanation
15 : Return JSON(prediction, plot, explanation)

Figure 32. Sequence diagram illustrating the data flow when inferencing.