

Secure Cloud Computing Project Work

*Kubeflow for ML CI/CD and*

*Web App Deployment on Kubernetes Cluster*

**Gruppo 09**

|  |  |  |  |
| --- | --- | --- | --- |
| **Nome** | **Cognome** | **Matricola** | **E-mail** |
| Giorgio | Galasso | 0622702063 | g.galasso31@studenti.unisa.it |
| Alessandro | Ambrosone | 0622702104 | a.ambrosone8@studenti.unisa.it |

Sommario

[1 Introduction 3](#_Toc155603408)

[1.1 Overview of the project 3](#_Toc155603409)

[1.2 Predictive Car Price in Automative Industry 3](#_Toc155603410)

[2 Project technology setup 4](#_Toc155603411)

[2.1 System Requirements and Specifications 4](#_Toc155603412)

[2.1.1 Hardware Specification 5](#_Toc155603413)

[2.1.2 Used tools 5](#_Toc155603414)

[2.1.3 Libraries and frameworks 5](#_Toc155603415)

[2.2 Kubernetes Cluster Configuration 7](#_Toc155603416)

[2.2.1 Design of our cluster 7](#_Toc155603417)

[2.3 Kubeflow Setup 9](#_Toc155603418)

[3 Machine Learning Development 10](#_Toc155603419)

[3.1 Dataset Overview 10](#_Toc155603420)

[3.2 Data Preprocessing 11](#_Toc155603421)

[3.3 Training and validation 12](#_Toc155603422)

[4 KubeFlow Pipeline 14](#_Toc155603423)

[4.1 Our Pipeline 14](#_Toc155603424)

[5 Application Development and Deployment 17](#_Toc155603425)

[5.1 Serving the model 17](#_Toc155603426)

[5.2 Deployment on Kubernetes 19](#_Toc155603427)

[5.2.1 Consideration on Deployment 20](#_Toc155603428)

[6 Webinar 20](#_Toc155603429)

[7 Conclusion 20](#_Toc155603430)

[Riferimenti: 22](#_Toc155603431)

[Indice Figure 22](#_Toc155603432)

# 1 Introduction

## 1.1 Overview of the project

The project revolves around the development and deployment of a car price prediction model. This model aims to estimate the market value of cars based on various attributes.

There are lots of individuals who are interested in the used car market at some point in their life because they wanted to sell their car or buy a used car. In this process, it’s a big corner to pay too much or sell less than its market value.

(Such a model is crucial in the automotive industry and for various stakeholders including car dealerships, buyers, sellers, and automotive enthusiasts.)

Integrating Kubernetes cluster with Kubeflow's main components, our goal is to achieve a scalable, resilient architecture capable of continuous integration and continuous deployment (CI/CD) for machine learning workflows. The project focuses on creating a Kubernetes cluster, composed of worker machines (nodes) executing containerized applications.

The application itself, designed as a containerized service, has a specific purpose: processing customer data to generate accurate car prices. Unlike a traditional emphasis on deploying a machine learning model, our project prioritizes building an architecture that ensures the model's responsiveness and accuracy.

## 1.2 Predictive Car Price in Automative Industry

Predictive analytics is vital in the automotive industry, particularly for developing car price prediction models. These models leverage extensive data to generate valuable insights, enabling accurate estimation of used car values. By understanding factors that influence car prices, such as make, model, age, mileage, and condition, predictive analytics facilitates more informed decision-making. This capability is essential for dealerships and individual sellers, as it helps in setting competitive yet profitable prices.

For buyers, a car price prediction model provides a benchmark, helping them to negotiate better deals and avoid overpaying. In the broader automotive market, these models assist in trend analysis, understanding how various factors affect car values over time. This insight is crucial for inventory management, ensuring dealerships maintain a balanced stock of vehicles that align with market demand.

Furthermore, predictive analytics in car pricing can inform dynamic pricing strategies. Dealerships and online platforms can adjust prices in real time based on market trends, demand fluctuations, and other relevant factors, optimizing profitability and market competitiveness.

Integrating predictive analytics into automotive sales operations and strategies results in a more transparent market, efficient business operations, and improved financial outcomes. In an industry increasingly reliant on data, predictive analysis is not merely a tool but a fundamental component for success and evolution in the automotive sector.

# 2 Project technology setup

This project includes various tools and technologies to ensure the efficiency and accuracy of the machine learning model to be implemented and eventually deployed in a cloud context.

## 2.1 System Requirements and Specifications

Typically a kubernetes cluster is deployed across multiple machines dedicating each to a specific node of the Kubernetes cluster.  
However, due to the constraints of our current project setup, we find ourselves restricted to a single PC that virtualizes the entire Kubernetes cluster. In this configuration, both the control node and worker nodes are simulated on a single physical machine.

Despite this limitation, the setup offers advantages: it enables us to develop, test, and demonstrate the capabilities of our application in a controlled environment without the need for extensive and expensive hardware.

### 2.1.1 Hardware Specification

For the project we used an **Acer Swift SF314-56** device, in which there is a ubuntu partition with 70Gb disk.

**Processor** : Intel® Core™ i7-8565U CPU @ 1.80GHz × 8

**RAM** : 12 Gb

**OS** : Ubuntu 22.04.3 LTS - type: 64-bit

### 2.1.2 Used tools

* **Docker**: Docker is an open-source platform used for developing, shipping, and running applications. It enables the separation of applications from the infrastructure facilitating the swift delivery of software. Docker packages software into standardized units called containers that have everything the software needs to run including libraries, system tools, code, and runtime. This ensures that the application will run the same way, regardless of where it is deployed.
* **KIND (Kubernetes IN Docker)**: KIND is a tool for running local Kubernetes clusters using Docker container "nodes". KIND is primarily used for testing Kubernetes itself, but may also be used for local development or CI. It allows developers to easily spin up a Kubernetes cluster on their local machine, where each Kubernetes cluster node is represented by a Docker container.

### 2.1.3 Libraries and frameworks

* **Streamlit**: Streamlit is an open-source Python library used for creating web apps for data science and machine learning projects. It allows data scientists and engineers to quickly turn data scripts into interactive web applications without requiring extensive knowledge of web development.
* **Pandas**: Pandas is an open-source data analysis and manipulation library for Python. It offers data structures and operations for manipulating numerical tables and time series, making it an indispensable tool for data cleaning, analysis, and visualization in Python.
* **Joblib**: Joblib is a set of tools in Python for lightweight pipelining and parallel computing. It is often used for saving and loading Python objects that contain large data, particularly useful in machine learning for caching models and datasets.
* **Scikit-Learn**: Scikit-Learn is a popular open-source machine learning library for Python. It provides a range of supervised and unsupervised learning algorithms and tools for model fitting, data preprocessing, model selection, and evaluation.
* **Kubeflow Pipelines (KFP)**: KFP is a platform for building and deploying scalable machine learning workflows based on Docker containers. It’s part of Kubeflow, an open-source project dedicated to making deployments of machine learning workflows on Kubernetes simple, portable, and scalable. KFP enables the orchestration of complex machine learning pipelines and provides tools for monitoring and managing them.

## 2.2 Kubernetes Cluster Configuration

A Kubernetes cluster is a set of machines, called nodes, that run containers managed by Kubernetes. A cluster has, by definition, at least one Worker Node. The Worker Node(s) host the Pods that execute user workloads. The Control Plane Node(s) manage the Worker Nodes and everything that happens within the cluster. Its role involves efficiently managing and orchestrating the operational requirements of our machine learning model.

In our case due to hardware restrictions, our cluster is configured in a very simple way, with only one control plane and one worker node.

In a real scenario for a production environment, especially one serving 50,000 customers with 5,000 concurrent requests, a more robust setup is needed. Here are some key considerations:

1. **Hardware Resources**: Determine the resource requirements (CPU, memory, storage) based on the application's needs. For high traffic, we need powerful machines, especially for worker nodes where the actual applications run.
2. **Number of Nodes**: For high availability, we typically want multiple control-plane nodes (at least **three** for redundancy). The number of worker nodes depends on the resource requirements of your applications.
3. **Load Balancing**: load balancing is a strategy to distribute traffic evenly across the nodes. This is crucial for handling high numbers of concurrent requests.
4. **Scaling Strategy**: Implement both horizontal and vertical scaling strategies. Horizontal scaling (adding more nodes) is effective for handling increased loads, while vertical scaling (adding resources to existing nodes) can improve the capacity of existing infrastructure.
5. **Monitoring and Logging**: Implement robust monitoring and logging to keep track of the cluster’s health and performance.

### 2.2.1 Design of our cluster

The Kubernetes cluster has been configured primarily for local development and testing, utilizing KIND. This configuration facilitates the simulation of a multi-node Kubernetes cluster within a Docker environment.

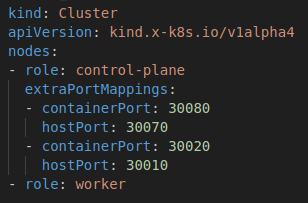


Figure 1 :node\_config.yaml

The cluster is composed of distinct roles, including at least one **control-plane** node and one **worker node**. Notably, the control-plane node is enhanced with extra port mappings, a setup which enables external access to services running within the cluster, thereby offering an efficient means for local testing and interaction.

In addition to the basic cluster setup, a comprehensive Role-Based Access Control (RBAC) system has been implemented.

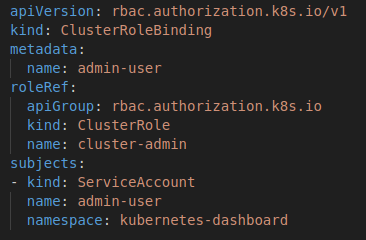


Figure 2:cluster\_role\_binding.yaml

This is evident from the establishment of a ClusterRoleBinding, specifically designated for a ServiceAccount named ‘admin-user’. This ClusterRoleBinding assigns the ‘cluster-admin’ role to the ‘admin-user’ ServiceAccount. The ‘cluster-admin’ role is a predefined role in Kubernetes, endowing the bearer with complete administrative access over the entire cluster and all its resources. This level of access **facilitates a broad range of administrative actions**, thereby streamlining cluster management processes.

The ‘admin-user’ ServiceAccount, crucial to this configuration, is created within the ‘kubernetes-dashboard’ namespace.

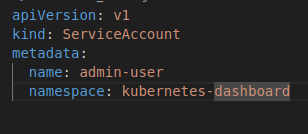


Figure : dashboard\_admin.yaml

The association of this ServiceAccount with the Kubernetes Dashboard namespace implies its intended use for administrative purposes within the Kubernetes Dashboard environment. This setup allows for a centralized and user-friendly interface for managing and operating the cluster, enhancing the efficacy of administrative tasks.

In summary, the configured Kubernetes cluster is tailored for a robust development and testing environment with advanced administrative capabilities via the Kubernetes Dashboard. The inclusion of a high-privileged ServiceAccount within the Kubernetes Dashboard namespace underscores a focus on efficient management and operation of the cluster.

## 2.3 Kubeflow Setup

Following this steps we will install Kubeflow on the Kubernetes cluster and set up the Kubeflow dashboard accessible on our local machine.

|  |
| --- |
| 1)export PIPELINE\_VERSION=2.0.3  2)-kubectl apply -k "github.com/kubeflow/pipelines/manifests/kustomize/cluster-scoped-resources?ref=$PIPELINE\_VERSION"-  3)-kubectl wait --for condition=established --timeout=60s crd/applications.app.k8s.io-   4)-kubectl apply -k "github.com/kubeflow/pipelines/manifests/kustomize/env/platform-agnostic-pns?ref=$PIPELINE\_VERSION"-  Kubeflow Pipelines UI is accessible by   5)-port-forwarding:kubectl port-forward -n kubeflow svc/ml-pipeline-ui 8080:80- |

# 3 Machine Learning Development

## 3.1 Dataset Overview

When building a machine learning application, we need to pay close attention to the various stages of development, especially in the choice of data on which to train the model, because a wrong choice at this stage could lead to significant difficulties later both during testing and even worse in production, with our model failing to work well on new data.

Therefore, analyzing and understanding the dataset well is vital. The following is a brief description of the dataset we used:

1. **Brand**: This denotes the manufacturer of the vehicle, providing an insight into the vehicle's make. The dataset includes brands like BMW, Mercedes-Benz, Audi, Toyota, Renault, Volkswagen, and Mitsubishi.
2. **Price**: The price column indicates the listed price of the vehicle. It is expressed in a numerical format, though some entries are marked as 'NA', indicating cases where the price is not available.
3. **Body**: This describes the body type of the vehicle, such as sedan, van, crossover, hatch, and others, offering a glimpse into the vehicle's structural design and intended use.
4. **Mileage**: Mileage, measured in kilometers, denotes the distance the vehicle has traveled. It serves as an indicator of the vehicle's usage and potential wear and tear.
5. **Engine Volume (EngineV)**: This represents the engine capacity of the vehicle, measured in liters. It provides an understanding of the vehicle's power and performance characteristics.
6. **Engine Type**: This column specifies the type of fuel the engine uses, such as Petrol, Diesel, or Gas, indicating the vehicle's fuel efficiency and environmental impact.
7. **Registration**: This indicates whether the vehicle is registered (yes or no), reflecting its readiness for legal operation on public roads.
8. **Year**: The year column refers to the year of manufacture of the vehicle, giving an idea of its age and potential technological advancements.
9. **Model**: This provides the specific model name of the vehicle, further detailing the exact variant within the brand's lineup.

## 3.2 Data Preprocessing

To train our model we need to clean up our data so that we have only the features of our interest, also we need to avoid having null values or values too far from the mean to avoid biased training.

**Preprocessing steps** in our project:

* **Drop features** that are not required to build our model. In the ‘Model’ column There are three hundred and twelve unique models. That's something really hard to implement, so we are dropping the ‘Model’ column;
* Check for any **missing value** in the data set. We are simply dropping all the missing values, this is not always recommended, however we are removing less than 5% of our data, so it is acceptable;
* Dealing with **outliers**.To address this issue, we opted to handle outliers using percentiles on the ‘Price’ ,’Year’ and ‘Mileage’ columns. This approach helps ensure a more accurate and reliable analysis of the data, enhancing the overall robustness of the model;
* Problems with data acquisition. The ‘EngineV’ column presents a strange situation. Manually checking the data we noticed that the missing values ​​were indicated with 99, furthermore there were some incorrect entries such as 75. Because the engine volume of a car is usually <= 6.5, we have removed all the rows with an ‘EngineV’ > 6.5;
* We are transforming **categorical features** with the dummy variable operation in order to represent them in a format suitable for machine learning algorithms;
* ‘Price’ column. Seeing the plot of the 'price' columns we are noting that the patterns are quite exponentials, in this condition log transformation is a common way to deal with this issue;
* Applying a **standard scaler** on features is useful for several reasons and it helps improve the performance, stability, and interpretability of machine learning models. In our case we are applying the scale to ‘Mileage' and 'EngineV' columns;

## 3.3 Training and validation

In our car price prediction project, aiming to estimate the cost of cars for potential buyers, we've implemented a regression algorithm. Regression, tailored for estimating continuous values, is suitable for forecasting the monetary value of cars based on diverse input features such as specifications, market trends, and economic factors.

To enhance the robustness and precision of our model, we performed training and comparison using three distinct regression models: Linear Regression, XGBoost, and Random Forest. Each model brings unique strengths, addressing various aspects of our car pricing dataset.

To enhance the robustness and precision of our model, we performed training and comparison using three distinct regression models: **Ridge Regression**, **Gradient Boosting Regressor**, and **Random Forest Regressor**. Each model brings unique strengths, addressing various aspects of our car pricing dataset.

1. **Random Forest Regressor**

* **Model Training**: The Random Forest Regressor, known for its robustness and effectiveness in dealing with complex datasets, was trained on the provided data. This ensemble method operates by constructing multiple decision trees during training and outputting the average prediction of the individual trees.
* **Hyperparameter Tuning**: To optimize the model, a grid search approach was employed. Parameters like **n\_estimators** (number of trees), **max\_depth** (maximum depth of each tree), and **min\_samples\_split** (minimum number of samples required to split an internal node) were varied. This approach ensured the identification of the most effective combination of these parameters.
* **Model Evaluation**: The best model from the grid search was evaluated using R-squared and RMSE (Root Mean Squared Error) metrics on a separate test set, providing insights into the model's predictive accuracy and generalization ability.

**2. Ridge Regression**

* **Model Training**: Ridge Regression, an extension of linear regression that includes regularization, was utilized. This approach is beneficial in preventing overfitting, a common issue in high-dimensional datasets.
* **Hyperparameter Tuning**: The model was fine-tuned using grid search, focusing on the **alpha** parameter, which controls the strength of regularization. The optimal **alpha** was determined by evaluating various values, balancing model complexity and performance.
* **Model Evaluation**: The best-performing Ridge model, as per the grid search results, was assessed on the test set using R-squared and RMSE metrics. These metrics helped quantify the model's effectiveness in capturing and predicting the underlying trend in the data.

**3. Gradient Boosting Regressor**

* **Model Training**: The Gradient Boosting Regressor, a powerful and widely used ensemble learning method, was trained. It builds an additive model in a forward stage-wise fashion, allowing for the optimization of arbitrary differentiable loss functions.
* **Hyperparameter Tuning**: A grid search was conducted to optimize critical parameters, including **n\_estimators**
* (number of boosting stages), **learning\_rate**, and **max\_depth**. This tuning is essential for gradient boosting, as these parameters significantly impact model performance.
* **Model Evaluation**: Following the selection of the best configuration through grid search, the model's performance was evaluated using R-squared and RMSE on the test data. These metrics provided a clear indication of the model's predictive power and accuracy.

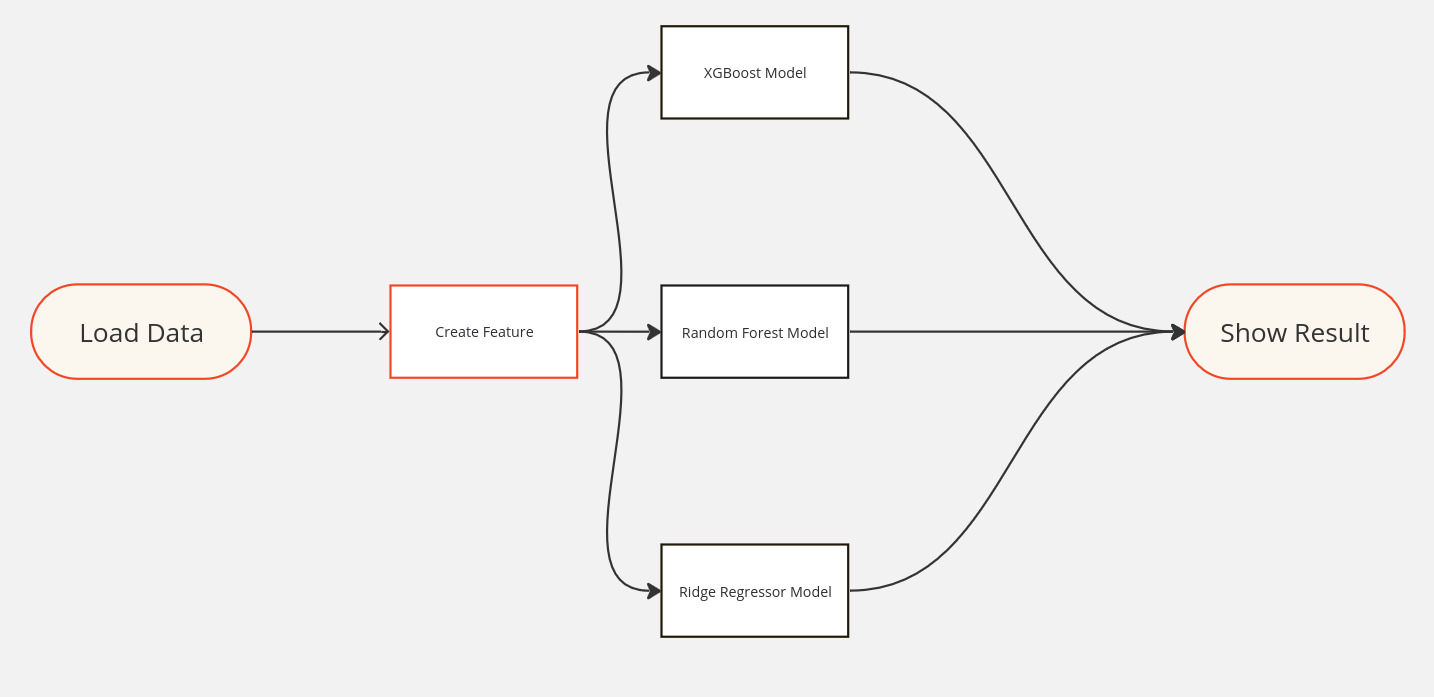


Figure : ML Pipeline

The best model is manually evaluated by us. The various tests done locally showed us that the best model is  **Gradient Boosting Regressor**

# 4 KubeFlow Pipeline

**Kubeflow** reveals itself to be the ideal choice for our **CI/CD** needs in the context of car price prediction. Its ability to orchestrate elaborate workflows on Kubernetes clusters allows us to define and manage pipelines that automate various phases of the machine learning lifecycle, from training to deployment. Integrating Kubeflow into our CI/CD pipeline **ensures greater efficiency**, repeatability and scalability of our machine learning processes. This strategic integration not only minimizes manual intervention, but also amplifies the overall agility and responsiveness of our machine learning operations.

In the CI/CD domain of machine learning, automation includes the processes of data preprocessing and model training through to deployment. This automated approach ensures consistent quality and rapid iteration of models. During the integration phase, automatic merging and verification of code changes related to model updates or data processing scripts occur without interruption. This process ensures a smooth integration with the existing code base and plays a key role in the detection and early resolution of bugs during the development cycle.

In the deployment phase, Kubeflow simplifies the deployment of trained models in production environments. By effectively managing the complexities associated with deploying machine learning workflows, it allows models to be updated **without interruption**. This capability is particularly crucial in our car price prediction project, where accurate and timely car price prediction is of a high importance.

Although an automated distribution process would be preferable, our current project lacks the expertise and infrastructure for automated distribution. Therefore, unfortunately, the distribution phase of our project involves manual implementation of the trained models in the web application. This manual implementation allows us to monitor and ensure the accurate integration of the trained models into the application.

## 4.1 Our Pipeline

In the Kubeflow framework, each segment of the machine learning process is encapsulated in a single **Kubeflow component**. These components are independent blocks of code, each tasked with performing a specific function in the overall ML workflow. Operating within individual pods in the Kubernetes environment, they provide segregated execution spaces, which improve both scalability and efficient resource allocation.We have strategically organized our pipeline with a set of these components, each of which has been carefully designed to play a distinct role in the machine learning workflow:

* **load\_data:** this component is responsible for loading the dataset, removing irrelevant features and cleaning the data in order to be pre-processed;
* **create\_features:** this component is responsible for preprocessing the dataset (standard scaler, one hot encoding), dividing the dataset into train set and test set and for storing the scaler dump;
* **linear\_regression:** this component is responsible for performing training and tuning of the Ridge regressor model using a grid search technique and for storing the model dump;
* **g\_boost:** this component is responsible for performing training and tuning of the Gradient Boosting Regressor using a grid search technique and for storing the model dump;
* **random\_forest:** this component is responsible for performing training and tuning of the Random Forest model using a grid search technique and for storing the model dump;
* **show\_results:** this component is responsible for showing the results of the three models, in particular R-squared and MSE(mean squared error);

For the creation of each component we carried out the following steps:

* **Python code:** each component is based on a python file which defines its logic related to the specific ML task;
* **Container definition:** we created a dockerfile with the container definition and the dependencies expressed in the requirements files;
* **Component definition:** we created a yaml file that describes the component in terms of input and output and specifies how to run it;
* **Docker hub:** we also uploaded the images built from the dockerfiles to the docker hub. This was done to ensure that our cluster could easily pull the images necessary for deployment;

The entire pipeline was defined using the Kubeflow DSL (Domain Specific Language) to ensure efficient orchestration and execution on a Kubernetes cluster. The tasks are interconnected in a sequential manner, with each subsequent task depending on the output of the preceding one. To facilitate deployment and execution, the Kubeflow pipeline was compiled into a YAML file using the Kubeflow compiler. This file encapsulates the entire pipeline definition and can be easily deployed on a Kubernetes cluster.

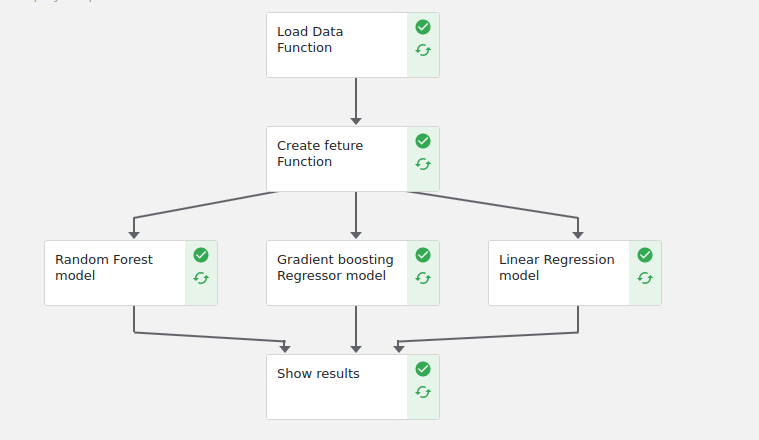


Figure 5: Kubeflow Pipeline

# 5 Application Development and Deployment

## 5.1 Serving the model

To serve the ML model, we build and deploy a web application on Kubernetes cluster. The application is a web interface for car price prediction, developed using **Streamlit** and integrated with the machine learning model for real-time forecasting.

The application includes functions to load the model and scaler, to prepare user-entered data via the interface (mileage, engine volume, year, make, body, engine type, registration), and a function to perform inference.

Our Streamlit app is designed to be user-friendly and intuitive. It allows users to input data through a form for the features:

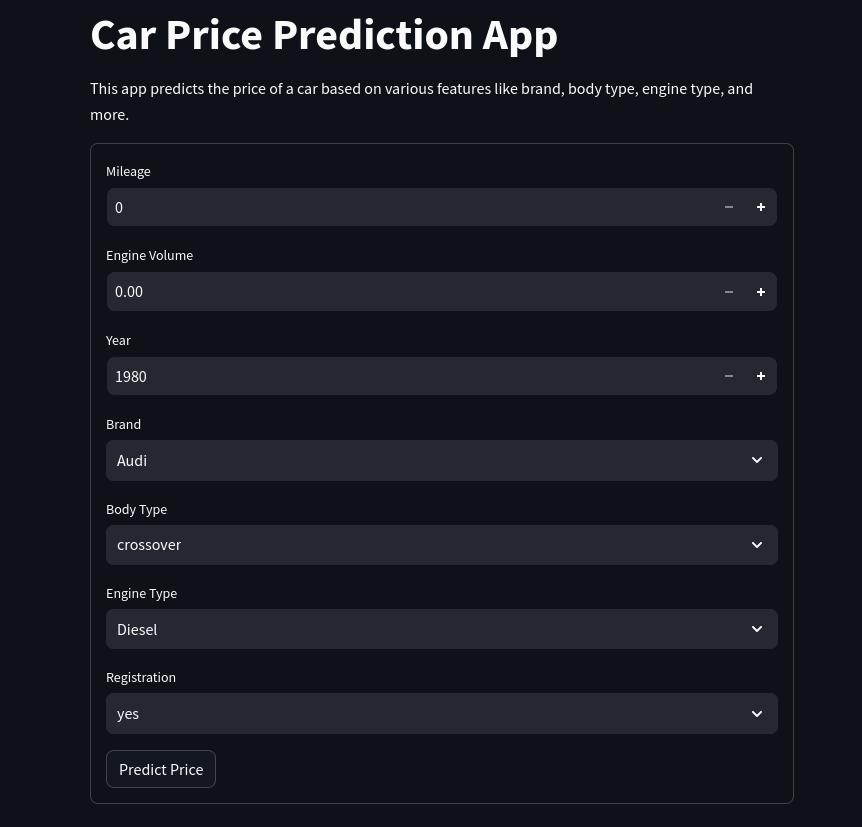


Figure 6: Streamlit app

These inputs match the required characteristics of our model to predict car prices. The application loads the trained model with its scaler to ensure that the input data are normalized appropriately prior to prediction.

For testing and validation purposes, the application is initially run on a local machine. This phase ensures that the application performs as expected.

## 5.2 Deployment on Kubernetes

Deployment of the application on the Kubernetes cluster occurs after the local testing phase has been successfully completed. Initially, this process involves building a Docker container specifically for the Streamlit application.

|  |
| --- |
| FROM python:3.10.9-slim-buster WORKDIR /app # Copy requirements COPY requirements\_app.txt ./requirements\_app.txt # Install dependencies RUN pip install -r requirements\_app.txt # Expose port EXPOSE 8501 COPY . /app # Create an entry point to make the image executable ENTRYPOINT ["streamlit", "run"] # Run the application: CMD ["app.py"] |

For application **deployment**, we created a YAML file to define the deployment and service on Kubernetes. The file provides the necessary instructions to ensure that the application runs reliably and is accessible within the cluster. The YAML file defines a Deployment with two replicas of the application. We configured the Deployment selector to handle pods labeled "app: car-price-prediction-app." This allows us to keep two replicas of the application running simultaneously to ensure availability and scalability. The pod model specifies the necessary features, including the container labeled "car\_price\_app," built with the Docker image of our previously created application. Finally, we configured a NodePort type service to expose the application to external traffic. The service was designated with the name "car-price" and placed in the default namespace. In the service definition, we specified the service port as 8501, the destination port within the pod as 8501, and the node port as 30080. This configuration allows the application to be exposed on every node in the Kubernetes cluster on port 30080, allowing external traffic to reach the application.

### 5.2.1 Consideration on Deployment

Our configuration is very simple and does not reflect the application load in a real scenario, this is due, as mentioned before to the limitations of our hardware. However, to handle a higher workload (such as that required by the trace) with a higher peak of requests, some changes should be made to the Kubernetes configuration file (YAML). The following are some recommendations:

* **Deployment Replications**:We should increase the number of deployment replicas to handle a higher load by distributing requests among multiple pods. A reasonable number of replicas could be between 5 and 10
* **Resource limits for containers**:We should specify resource limits for containers to prevent a pod from consuming all available resources. This can be done within the configuration files
* **Load balancer**: The current service type is NodePort. To handle high traffic,we consider using a LoadBalancer-type service, to distribute the load among the cluster nodes
* **Horizontal Pod Autoscaler (HPA)**:Use a Horizontal Pod Autoscaler to automatically scale the number of replicas based on the workload

Managing a consistent workload effectively requires a thoughtful approach to scalability and resource optimization. The above considerations, including adding replicas, using Horizontal Pod Autoscaler (HPA), and improving service configurations, can help make sure that the application reliably handles a large number of users with simultaneous peak requests. In addition, it is advisable to closely monitor system performance during heavy load situations to implement additional optimizations, if necessary.

# 6 Protecting app

Among the main challenges in using cloud-native technologies such as containers, kubernetes, etc., certainly security issues are among them. Looking at the main causes of cyber attacks, application layer vulnerabilities, such as web application and software vulnerabilities, still rank first. So how can we try to mitigate these vulnerabilities?

One solution is to get in between the flow of requests and responses from our application and inspect it to try to understand what kind of traffic we are looking at, so that we can make decisions and take actions congruent with them.

### 6.1

The main solution NGINX offers consists of NGINX App Protect WAF and DoS, these are two add-on modules of NGINX plus. These offer protections against sql injection, cross site scripting and other classic attacks related to the application layer

Immagine che contiene testo, schermata, diagramma, linea

Descrizione generata automaticamente

NGINX WAF (Web Applications Firewall) is a reverse proxy that acts as an intermediary by protecting the web app server from malicious clients by filtering, monitoring, and blocking any malicious HTTP/S traffic traveling to the web application, and prevents any unauthorized data from leaving the app. It does this by adhering to a set of policies that help determine what traffic is malicious and what traffic is safe.

The main objective of NGINX App Protect DoS is to detect, by monitoring traffic behavior and comparing it to a baseline, and mitigate DoS attacks, which seek to overload a web application or online service by making it inaccessible to legitimate users.

There are several ways to apply NGINX App Protect, the most common being as an Ingress Controller before Kubernetes Pods, because is lightweight and much easier to manage

# 7 Conclusion

AGGIUNGERE RIFERIMENTI ALLE IMMAGINI, AL NOSTRO GITHUB, DOCKER HUB E RICONTROLLARE SE ABBIAMO SALTATO QUALCOSA, INOLTRE AGGIUNGERE UN README PER ESEGUIRE IL PROGETTO (compreso di come buildare immagine ecc..)

Our project has been a journey through the core principles of cloud architecture, intertwining these fundamentals with advanced machine learning techniques. In the context of today's technological sphere, our investigation into cloud architecture has shed light on the synergy between these two fields.

Direct effort with this project has deepened our knowledge of cloud-based infrastructures and highlighted the cloud's vital role in the smoothie integration of machine learning into applied settings.

Throughout the project, we faced a range of obstacles, especially due to the limitations of our hardware. These constraints limited our ability to create a fully representative cloud environment. The challenges of working with a single-node infrastructure were significant, yet they contributed to our enhanced understanding of the project's real-world relevance and scope.

The utilization of the Kubeflow framework for continuous integration and continuous deployment (CI/CD) was a key focus of our efforts. This component of the project underlined the necessity of automating integration processes to streamline development and reduce errors. Implementing CI/CD in Kubeflow not only reaffirmed its essential nature but also revealed its role in improving workflow efficiency, facilitating team collaboration, and improving the overall project's productivity.

# Riferimenti:

-Github Project Repository:

<https://github.com/AlessandroAmbrosone/scc/tree/main/esame>

-Docker Hub Repository:

<https://hub.docker.com/repositories/ossalag00>

# Indice Figure

[Figure 1 :node\_config.yaml 8](#_Toc155602773)

[Figure 2:cluster\_role\_binding.yaml 8](#_Toc155602774)

[Figure 3: dashboard\_admin.yaml 9](#_Toc155602775)

[Figure 4: ML Pipeline 13](#_Toc155602776)

[Figure 5: Kubeflow Pipeline 16](#_Toc155602777)

[Figure 6: Streamlit app 18](#_Toc155602778)