

# Università degli Studi di Salerno

Dipartimento di ingegneria dell'informazione ed elettrica e matematica applicata

Corso di laurea magistrale in ingegneria informatica



**Project Work of Secure Cloud Computing**

*A.A. 2024/25*

## **WEATHER FORECASTING APPLICATION**

# ABSTRACT

---

This project describes the development of a **weather forecasting web application** that leverages a **machine learning model** to predict weather conditions based on **historical data**. The model was trained using an imbalanced dataset containing meteorological data from Seattle, with preprocessing steps including outlier removal and feature engineering. Among several classifiers tested, **XGBoost** was selected for deployment due to its superior performance in handling imbalanced data and key weather categories like rain and snow.

The application was containerized using **Docker** and deployed in a **Kubernetes cluster** to ensure scalability and high availability. A **Horizontal Pod Autoscaler (HPA)** was implemented, dynamically adjusting the number of replicas based on CPU utilization, achieving optimal resource allocation while maintaining system performance. Stress tests conducted using **JMeter** confirmed that the system can handle up to **50,000 daily requests** with peaks of **2,000 concurrent users**, maintaining an average response time below **3 seconds** and an error rate of **0%**.

Security assessments included a **container vulnerability analysis** using **Docker Scout**, identifying and mitigating issues such as **CVE-2024-5206**, **CVE-2024-6345**, and **CVE-2023-5752**. Additionally, an **NGINX Ingress Controller** was integrated to enhance security, implementing rate limiting and traffic shaping to mitigate potential **DoS attacks**.

Results indicate that the proposed system successfully balances **scalability, security, and efficiency**, providing a robust architecture for real-time weather prediction. Future improvements will focus on expanding the dataset, refining the prediction model, and integrating additional security measures.

## TABLE OF CONTENTS

Abstract .....	2
TABLE OF CONTENTS .....	3
INTRODUCTION.....	6
THE GOALS AND THE ROADMAP.....	6
PRIMARY GOALS.....	6
WHY DO WE CARE?.....	6
WHAT DO WE EXPECT FROM A WEATHER FORECASTING APP? .....	7
THE MODEL .....	8
ABOUT THE DATA .....	8
CHECKING NULL VALUES .....	9
DISTRIBUTION ANALYSIS OF THE TARGET VARIABLE .....	9
CORRELATION STUDY .....	11
MODEL DEVELOPMENT .....	13
DATA PREPROCESSING .....	13
CHOICE OF THE MODEL .....	14
TRAINING PHASE AND FINAL PIPELINE CONSTRUCTION.....	15
TESTING PHASE AND EVALUATION OF PERFORMANCES .....	16
LIMITATIONS AND AREAS FOR IMPROVEMENT .....	18
DOCKER CONTAINERIZATION.....	19
WEB APPLICATION GUI.....	19
IMAGE DEFINITION: DOCKER FILE.....	20
CONTAINER VULNERABILITIES ANALYSIS.....	23
CVE-2024-5206.....	23
CVE-2024-6345.....	23
CVE-2023-5752.....	24
CLUSTER INFRASTRUCTURE .....	27
THE ARCHITECTURE.....	27
DYNAMIC HORIZONTAL SCALABILITY ARCHITECTURE.....	27
COMPLEMENTAL ARCHITECTURES: WORKLOAD DISTRIBUTION ARCHITECTURE .....	28
ASSOCIATED MECHANISMS AND COMPONENTS.....	29
KUBERNETES CONFIGURATION .....	30
ARCHITECTURAL OPTIMIZATION .....	30
CLUSTER CONFIGURATION OVERVIEW .....	30
KUBERNETES CONFIGURATION FOR DEPLOYMENT AND SERVICE .....	31

DEPLOYMENT.....	31
SERVICE.....	32
METRIC SERVER .....	33
STEP BY STEP CONFIGURING PROCEDURE .....	34
CLUSTER CREATION .....	34
METRIC SERVER CONFIGURATION.....	34
KUBERNETES DASHBOARD.....	35
DEPLOYING THE WEATHER FORECASTING APP .....	35
BENCHMARK.....	37
HARDWARE SPECIFICATIONS AND PREMISES .....	37
TESTS SET UP .....	37
USAGE WITH DIFFERENT AMOUNTS OF REPLICAS .....	37
REPLICAS = 1 .....	38
REPLICAS = 2.....	38
REPLICAS = 5.....	38
REPLICAS = 10 .....	39
CONCLUSIONS .....	39
DEFINITION OF REQUESTS AND LIMITS VALUES.....	40
CPU .....	40
MEMORY .....	42
HORIZONTAL POD AUTOSCALER.....	42
MIN REPLICAS .....	44
MAX REPLICAS .....	46
PERCENTAGE UTILIZATION .....	46
WHY WE DIDN'T USE A VPA? .....	46
FINAL STRESS TEST .....	46
SECURITY ASSESSMENT .....	48
THREAT IDENTIFICATION.....	48
RISK EVALUATION .....	48
INGRESS CONTROLLER.....	49
NGINX APP PROTECT DDOS .....	50
CONFIGURATION FILES .....	50
CONCLUSIONS .....	53
FUTURE IMPROVEMENT .....	53
BIBLIOGRAPHY .....	54
FIGURES INDEX.....	54

TABLES INDEX..... 55

CODES INDEX..... 55

# INTRODUCTION

---

Weather forecasting is a process of **identifying and predicting** to a certain accuracy the **climatic conditions** using multiple technologies. Many real-time systems rely on accurate weather data to adjust their operations accordingly, so forecasting plays a crucial role in minimizing potential damage to life and property by allowing for timely precautions.

Although traditional methods of forecasting have improved over time, there is still room to make predictions more accurate and efficient. Recently, machine learning (ML) has become a promising approach for weather prediction. By using the large amounts of data collected from weather stations around the world, **ML models** can analyze complex patterns and make more accurate predictions.

## THE GOALS AND THE ROADMAP

The ultimate goal of this project is to **develop a predictive model for weather forecasting and deploy it in a Kubernetes cluster environment** for scalable, real-time weather prediction. The project will be executed with a focus on **ensuring continuous integration and continuous delivery (CI/CD)** through several fundamental steps:

1. **Develop a machine learning model** Design and train a machine learning model capable of accurately forecasting weather based on historical data. This process will be carried out through an **automated pipeline**, ensuring consistent model training and validation.
2. **Containerize the model using Docker:** Package the model into a *Docker* container to ensure consistency across different environments.
3. **Conduct security assessments:** Perform a comprehensive analysis of potential vulnerabilities in the system using tools like *Docker Scout* to ensure security and robustness.
4. **Deploy with Kubernetes:** Use *Kubernetes* to orchestrate the containerized model, allowing for efficient scaling, management, and monitoring of the application in a cloud environment.
5. **Ensure scalability and reliability:** Design the cloud deployment to automatically scale based on demand, allowing the system to handle varying loads and ensure high availability. Kubernetes' auto-scaling and monitoring capabilities will maintain continuous operation, embodying DevOps principles of reliability and responsiveness.
6. **Perform stress testing:** Execute stress tests to ensure the system can handle high traffic volumes without performance degradation, confirming its capability for production use.

## PRIMARY GOALS

There are **three primary goals** for our project:

- Demonstrate an End-to-End containerized application example on the Cloud.
- Present our application and ML model.
- Build a web app able to perform the forecasting supposing that the APP should **serve a total number of about 50.000 customers** and, during the serving time, the APP can have a **maximum of 2000 service requests simultaneously** during the day.

## WHY DO WE CARE?

Weather forecasting is far more than a daily routine—it's a vital tool that influences everything from individual choices to global industries. Accurate weather predictions help to **optimize resource**

**allocation, improve decision-making, minimize risks and costs.** In agriculture, it assists in planning harvests and protecting crops, while in transportation, it ensures safer travel conditions. Additionally, weather forecasts play a vital role in **energy demand prediction and disaster preparedness**, helping to mitigate the effects of extreme weather events.

If we don't want to think about the broader economic and environmental impacts of weather forecasts, we can consider their effects on a smaller scale. A study by the National Science Foundation (Dybas & Hosansky, 2015) found that about **90% of American adults** check the weather forecast an average of **3.8 times a day**, typically at peak times such as **early morning, late evening**, or even **late at night**. Other studies and user data show that people tend to check the weather a few hours before leaving home, especially for short-term forecasts. This behavior is usually driven by the need to prepare for immediate weather conditions, such as rain, snow, or sudden temperature changes, which could impact activities or plans. It's reasonable to assume that **we all rely on short-term weather predictions** to make last-minute decisions, like whether or not to bring an umbrella to work.

In a world increasingly affected by climate change, where both industries and individuals depend on accurate predictions, weather forecasting has become a cornerstone of safety, efficiency, and preparation — **something we all rely on, whether we realize it or not.**

## WHAT DO WE EXPECT FROM A WEATHER FORECASTING APP?

Given the importance of this field, as outlined in the previous paragraph, it is easy to see that the two primary characteristics a weather forecasting service must fulfill are **availability** and **reliability**. Our weather forecasting app aims to deliver **reasonably on time and reliable service** that users can depend on for informed decision-making. Whether it's planning daily activities, preparing for extreme weather events, or supporting critical sectors like aviation, logistics, and agriculture, the app must ensure **high availability and reliability**. A downtime or outdated forecast can lead to **loss of user trust**, impact professional operations, and even pose **safety risks**.

Therefore, we strive for **80% or higher availability** and seamless data processing to provide accurate and timely information, reinforcing user confidence and meeting the needs of both casual users and professionals in weather-dependent fields. Additionally, we anticipate the need for **relatively fast response times** to ensure that users receive critical updates promptly, allowing them to make informed decisions in dynamic situations.

## RECAP OF KEY QOS OBJECTIVES

To conclude this section, we provide a summary of the main **Quality of Service (QoS) objectives** that will guide our decision-making process during the design and implementation of the system:

- The app must be designed to serve a total user base of approximately **50,000 customers**.
- It should handle a maximum of **2,000 simultaneous service requests** during peak times.
- The service must ensure a **minimum availability and reliability of 99%**, maintaining consistent operation even under high demand.
- Response times should be relatively fast, with a target of processing each request and delivering a response within **an average of 3 seconds**.

# THE MODEL

This section of the report details the development process of the model, including the design decisions made and the data analysis conducted. It specifically explores the use of machine learning for weather prediction, with a focus on the use of supervised learning algorithms, such as XGBoost.

## ABOUT THE DATA

The dataset used contains daily weather measurements from weather stations in Seattle, and the goal is to train the ML models to predict future weather based on historical data. For simplicity, the study is limited to Seattle, but the approach can easily be generalized by using a larger dataset from other regions.

The dataset consists of **1461** entries with **6** columns:

- **date:** Day of the observation (Format: YYYY-MM-DD)
- **precipitation:** All forms in which water falls on the land surface and open water bodies as rain, sleet, snow, hail, or drizzle
- **max\_temp and min\_temp:** Max and min daily temperatures
- **wind:** Wind velocity recorded
- **weather (output):** Labels describing the weather (e.g., clear, cloudy, rainy).

#	Column Name	Non-Null Count	Data Type (Dtype)	Description
0	date	1461	object	Dates
1	precipitation	1461	float64	Amount of precipitation
2	temp_max	1461	float64	Maximum temperatures
3	temp_min	1461	float64	Minimum temperatures
4	wind	1461	float64	Wind speed
5	weather	1461	object	Description of weather conditions

Table 1. Dataset's columns datatypes

The dataset spans from **January 1, 2012, to December 31, 2015**, and contains daily weather data for Seattle. The average precipitation is around **3.03 mm**, with the highest recorded at **55.9 mm** while temperature ranges from **-7.1°C to 35.6°C**, and wind speeds vary between **0.4 m/s and 9.5 m/s**.

Statistica	date	precipitation	temp_max	temp_min	wind	weather_encoded
count	1461.000000	1461.000000	1461.000000	1461.000000	1461.000000	1461.000000
mean	2013-12-31	3.029432	16.439083	8.234771	3.241136	2.752225
std	NaN	6.680194	7.349758	5.023004	1.437825	1.191380
min	2012-01-01	0.000000	-1.600000	-7.100000	0.400000	0.000000
25%	2012-12-31	0.000000	10.600000	4.400000	2.200000	2.000000



<b>50%</b>	2013-12-31	0.000000	15.600000	8.300000	3.000000	2.000000
<b>75%</b>	2014-12-31	2.800000	22.200000	12.200000	4.000000	4.000000
<b>max</b>	2015-12-31	55.900000	35.600000	18.300000	9.500000	4.000000

Table 2. Dataset's statistic summary

## CHECKING NULL VALUES

Upon reviewing the dataset, it was found that there are no missing values in any of the columns. Specifically, the columns for **date**, **precipitation**, **temp\_max**, **temp\_min**, **wind**, and **weather** are all complete.

## DISTRIBUTION ANALYSIS OF THE TARGET VARIABLE

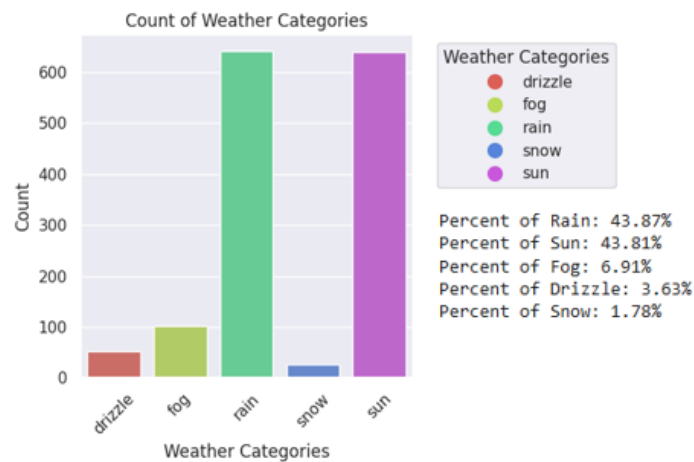
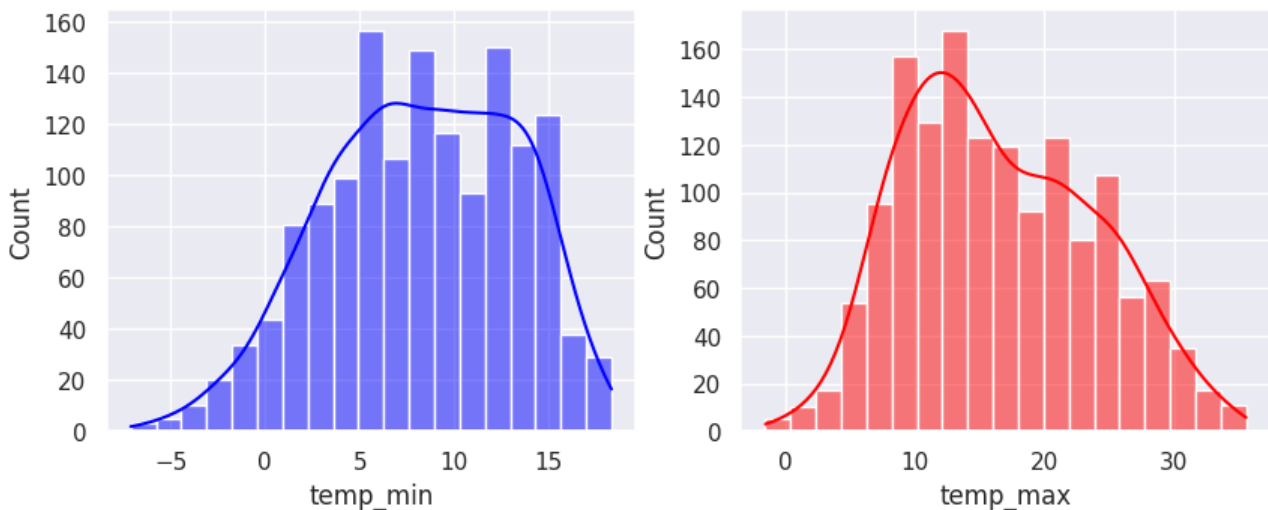


Figure 1. Output classes' distributions

A preliminary analysis of the dataset reveals an imbalanced distribution of weather conditions:

- The most frequent conditions are **rain** and **sun**, with 641 and 640 occurrences, respectively.
- **Fog** appears 101 times, while **drizzle** and **snow** are much less common, with only 53 and 26 occurrences.

We expect that this disparity in the data **could affect model performance**, suggesting the need for techniques such as oversampling or class weighting to improve predictions for less frequent weather events.



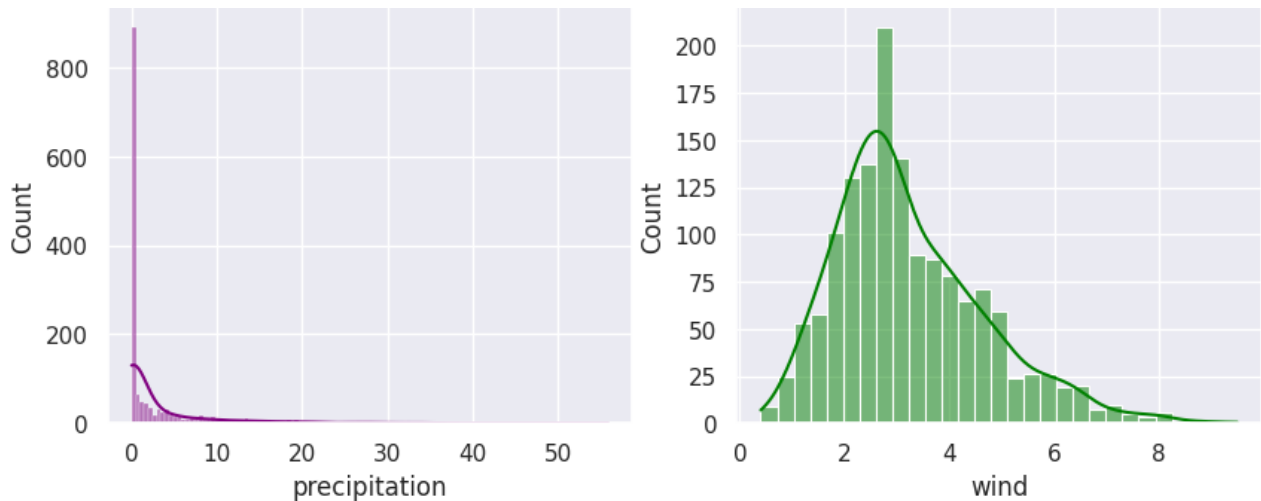
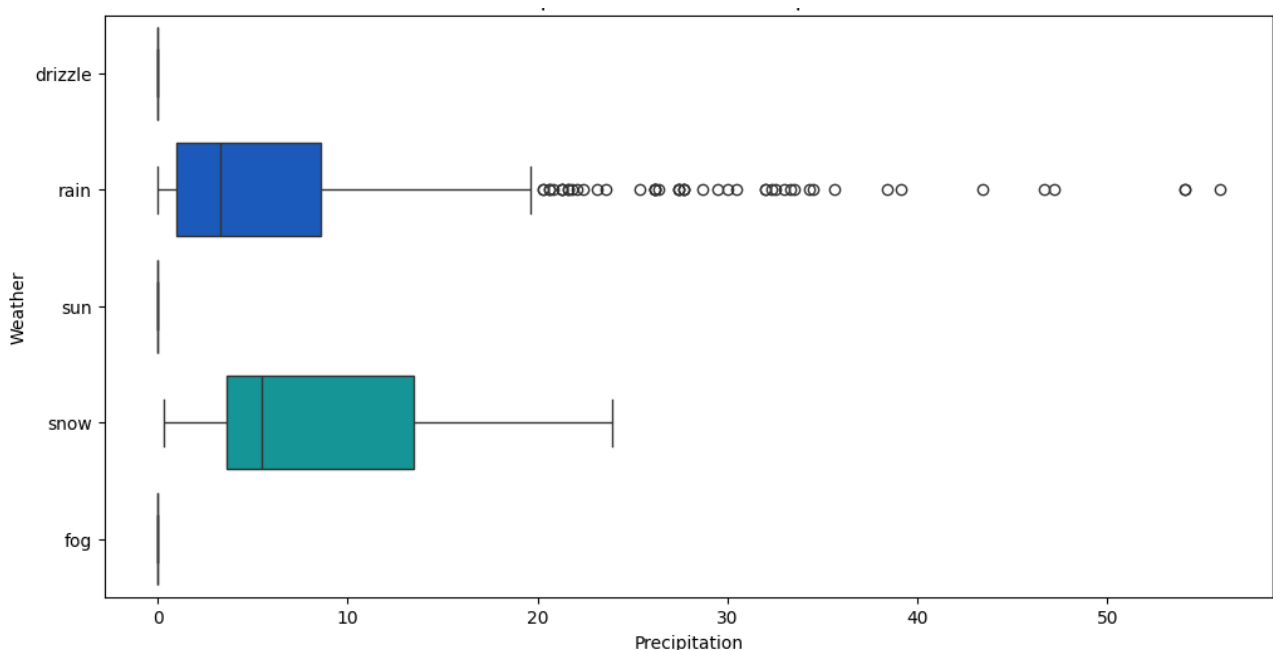


Figure 2. Dataset's columns KDE curves

The KDE curves of the dataset's columns show that:

- For the **maximum temperature**, the distribution is nearly **normal** with a slight negative skew, meaning most days fall within the 10°C to 25°C range. Similarly, the **minimum temperature** also exhibits a nearly normal distribution but with a mild positive skew, with most values between 0°C and 15°C. They appear well-distributed and balanced, requiring less preprocessing.
- The precipitation values are highly **positively skewed**, with most days having low or no precipitation and only a few instances of extreme rainfall.
- Wind speed data follows a **positive skew**, similar to precipitation, with most values being low and a few extreme highs.

**Outliers** in variables such as **precipitation** and **wind** can significantly distort data analysis or affect the performance of predictive models. Removing extreme values ensures that the dataset reflects typical patterns rather than anomalies.



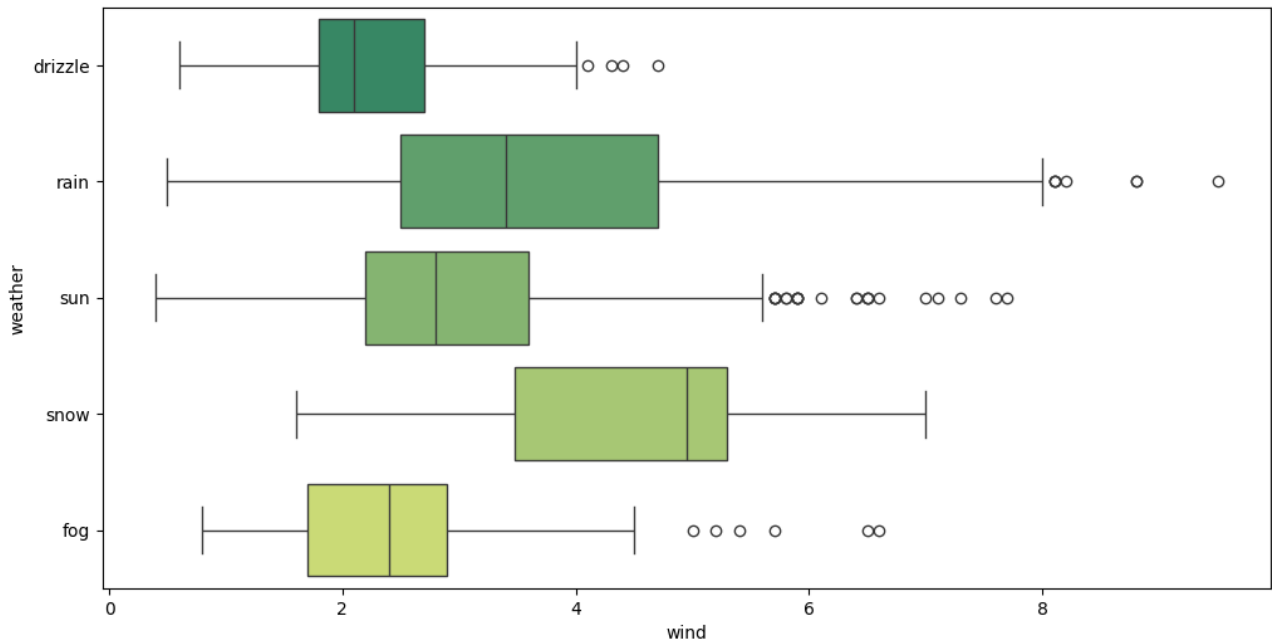


Figure 3. Precipitation and wind boxplots

Given the significant impact of outliers in the dataset, we opted to first identify and remove them using the interquartile range (IQR) method to enhance the dataset's representativeness. Prior to this, we conducted an initial training without eliminating the outliers. However, the presence of these anomalies led to a noticeable decrease in model performance.

### CORRELATION STUDY

The correlation matrix in Figure 5 reveals several interesting relationships between the meteorological variables.

- Temperature maximum and minimum show a **strong positive correlation**, indicating that days with high maximum temperatures tend to have higher minimum temperatures as well.
- There is a slight **negative correlation** between precipitation and temperature, suggesting that lower temperatures are slightly more common on days with precipitation.

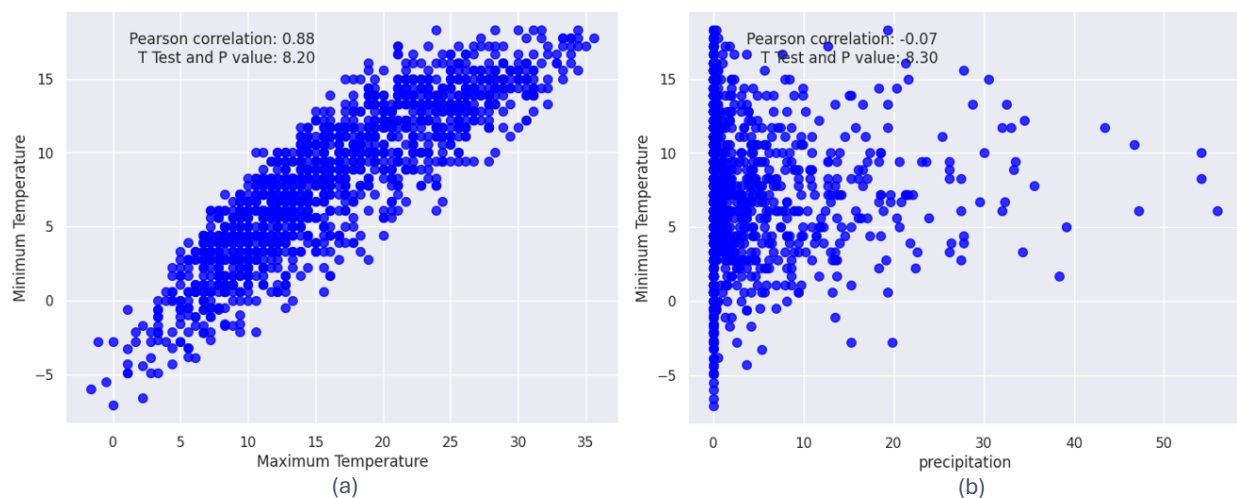


Figure 4. (a) Minimum and maximum temperature scatter plot (b) Minimum and precipitation scatter plot

- The correlation between wind speed and other variables is generally weak, indicating a less pronounced relationship.

- The low correlation between date\_num and the output suggests a weak or non-existent linear relationship. Removing "date\_num" can simplify the analysis and reduce multicollinearity, but it can also lead to a loss of information if the temporal component is crucial for understanding the data.

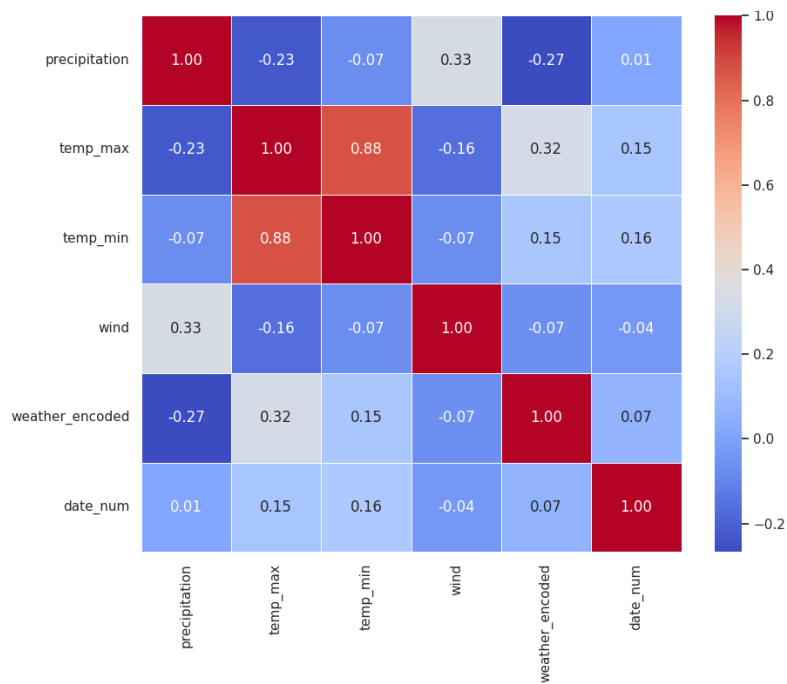


Figure 5. Correlation Matrix without date decomposition

## MODEL DEVELOPMENT

In this subsection, after outlining the necessary technical and contextual premises, we will explain the tools and libraries used, the objectives of the model, the decision-making process behind the choice of the classifier, and the phases of **training** and **testing**.

### PREMISES

Given that the dataset at our disposal is not sufficiently large and that the machines and environment used for training are not optimal for robust training capable of producing a highly reliable model, we do not expect the model to achieve outstanding performance. As discussed in the section **About the Data**, some classes are significantly underrepresented compared to others, which inevitably introduces precision issues in the model.

We acknowledge that the initial conditions do not allow for exceptional results, and therefore, we aim to achieve general performance that exceeds accuracy in at least half of the cases. Furthermore, due to the evident imbalance in the dataset, we will prioritize solutions that favor the most practically relevant output classes—rain, snow, and sun—as these are likely to have a greater impact on individuals or specific activities for which the model is being consulted.

### ENVIRONMENTS AND LIBRARIES USED

The model was developed using the cloud-based platform **Google Colaboratory**, which enabled training by leveraging advanced hardware resources, such as GPUs and a high number of CPUs. This approach allowed us to overcome the limitations of local hardware resources, ensuring a smooth, optimized, and computationally efficient development process.

The project utilized the following libraries: *NumPy*, *Pandas*, *Matplotlib*, *Seaborn*, *Scikit-learn*, *Imbalanced-learn*, *Joblib*, *Feature-engine*, *Psutil*, as well as system modules like *os*.

## DATA PREPROCESSING

Based on the analysis (Figure 6), it is clear that the **date-derived columns** (month, day, and year) exhibit **very low correlations** with other numerical variables in the dataset. Additionally, the trends and patterns observed in the data are already well-represented through other features such as *temp\_max*, *temp\_min* and *precipitation*. Including these date-related columns would likely introduce **redundancy** and **add complexity** to the model without providing significant additional predictive value. Furthermore, while the date columns capture temporal information, their weak correlation with target variables and other predictors suggests that they are not critical for capturing the underlying patterns in the data.

Initially, we considered transforming this information into a single column representing the **month** to retain some temporal context. However, after a series of tests and comparisons of the model's performance, we decided to **completely exclude** this column. Thus, excluding the date-related columns helps streamline the model, reduce dimensionality, and avoid the risk of overfitting, while focusing on the variables that truly drive predictive performance.

In the same preliminary offline data preprocessing phase, another critical decision was made regarding the **removal of outliers**. The excessive presence of outliers, as observed during initial experiments, significantly hindered the model's ability to learn effectively. These anomalies distorted the learning process and negatively impacted classification performance, as reflected in the evaluation metrics and test set results. Despite reducing the dataset size even further—thereby reinforcing expectations of a less-than-perfect model—removing the outliers proved necessary. After numerous trials, it became

evident that the presence of these outliers prevented the model from achieving meaningful learning and reliable classification.

While these operations were handled separately to maintain the simplicity of the pipeline, they could have easily been incorporated into it without any issues.

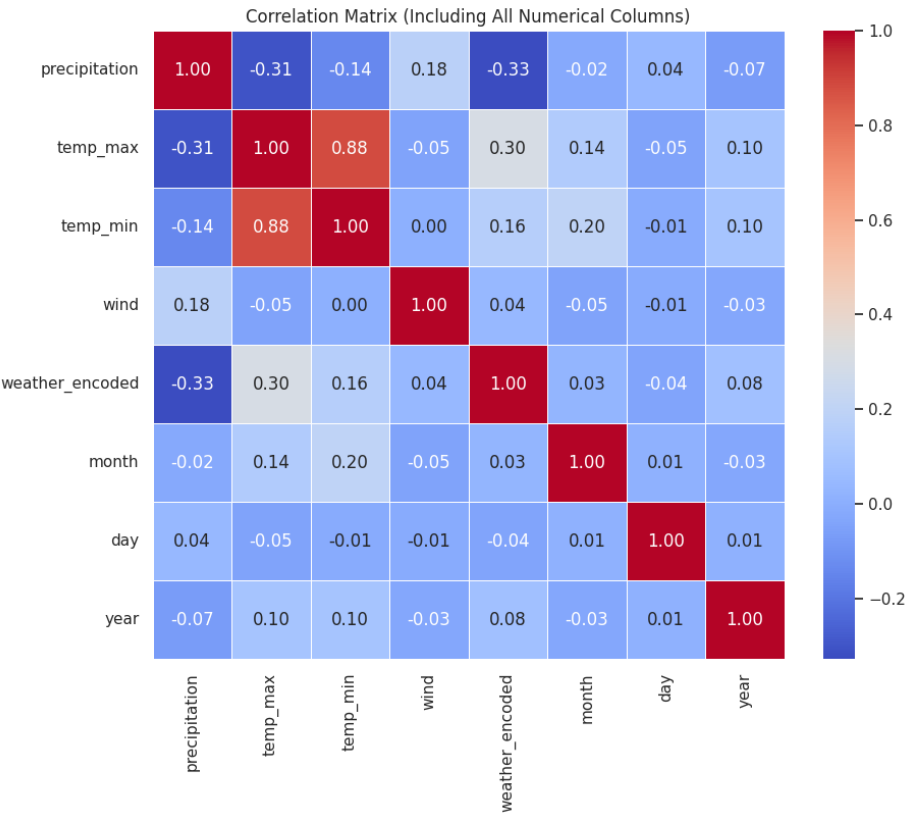


Figure 6. Correlation matrix with date decomposition

CHOICE OF THE MODEL

We examined various classification models and methodologies to identify the **most effective approach** for predicting weather conditions. Following the data preprocessing steps described earlier, several models were trained and evaluated, including *Logistic Regression*, *Decision Tree*, *Random Forest*, *XGBoost*, and *LightGBM*. The evaluation focused on key performance metrics such as:

- **F1 Score:** A measure balancing precision and recall.
- **ROC-AUC:** An indicator of the classifier’s ability to distinguish between classes.

Given the imbalanced nature of the dataset, **accuracy was not used** as an evaluation metric in this phase, as it could have been misleading. Instead, **F1 Score** and **ROC-AUC** were prioritized to ensure fair performance assessment across all classes. A **grid search technique** was utilized to fine-tune the **hyperparameters** for each model under consideration.

Simpler models, such as Logistic Regression and Decision Tree, **struggled to capture the complexity** of the data, leading to lower performance metrics. In contrast, **more sophisticated models** like XGBoost and LightGBM **effectively addressed data imbalances** and non-linear relationships, resulting in significantly improved classification performance.

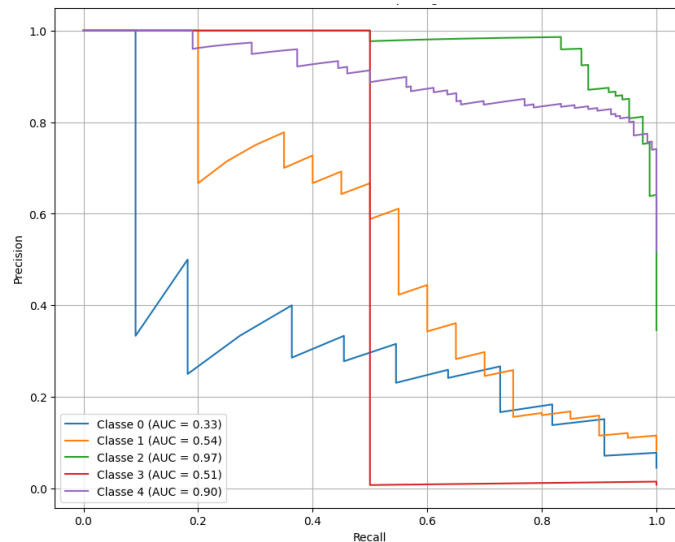


Figure 7. XGBoost model's AUC curve for each class

The **XGBoost classifier** emerged as the best-performing model, achieving an optimal balance between F1 Score and generalization. Its hyperparameters were fine-tuned to ensure stable predictions across the test set. XGBoost demonstrated good performance across key classes such as **rain, snow, and sun**, which are particularly valuable for users. While other models, such as LightGBM and Random Forest, achieved comparable results, XGBoost excelled in handling these critical classes, making it the **most suitable choice** for practical applications.

The selection of XGBoost as the final model was based on its superior overall performance and its ability to perform well on high-priority output classes. This ensured reliable predictions for the most critical weather conditions, making it the ideal choice for deployment.

## TRAINING PHASE AND FINAL PIPELINE CONSTRUCTION

The entire training process was underpinned by a **multi-step pipeline** integrating data preprocessing, feature transformation, and some machine learning techniques, supported by **systematic cross-validation** and **hyperparameter optimization**.

### TRAINING PHASE

The training started off with some structured preprocessing that ensured the dataset was as clean as possible to suit modeling. First, we separated the **target variable feature**, which is weather in the dataset, from all others, which were regarded as **input features**. We used a coercion of the features to **NaN** for **non-numeric data values**, and checks for verification that there were **no records missing or infinite values**. The target variable was then encoded into numeric labels using **LabelEncoder**, which prepared it for the machine learning algorithm.

For keeping the distribution of classes across datasets, **stratified splitting** was adopted in the train-test split, where the data was divided into **80% for training and 20% for testing** (Pareto's law). Where Stratified splitting means splitting the dataset into different subsets while preserving certain **distribution of classes or categories** for the target variable in both train and test set. This maintained the balance of the categories of weather and made the evaluation of the model fair.

### PIPELINE COMPOSITION

The pipeline utilized consists of the following key steps:

1. **Data Preprocessing:** The raw weather data underwent rigorous cleaning and imputation. Missing values in key columns like precipitation, temp\_max, temp\_min, and wind were

addressed using a **median imputation strategy**. This approach preserved the intrinsic patterns of the data.

2. **Column Transformation:** A **ColumnTransformer** pipeline was employed to handle both numerical and categorical features. It included:

- **Numerical Features:** Standard scaling and imputation to normalize data and handle missing values.
- **Categorical Features:** One-hot encoding to effectively transform and utilize categorical variables such as temporal attributes.

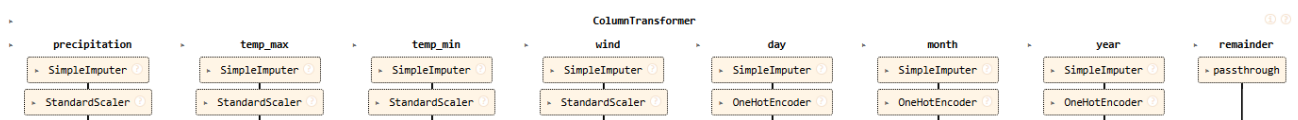


Figure 8. Pipeline's column transformer

3. **Class Balancing and Noise Reduction:**

- **SMOTE (Synthetic Minority Oversampling Technique)** was employed to generate synthetic samples for **minority classes**, addressing class imbalances effectively and improving model fairness.
- **Tomek Links** were applied to **remove noisy samples from overlapping classes**, enhancing the dataset's quality and reducing classification ambiguity.

4. **Model Selection and Hyperparameter Tuning:** The pipeline integrated an **XGBoost classifier**, leveraging its efficiency and robustness for classification tasks. A grid search using GridSearchCV and a **Stratified K-Fold Cross-Validation** strategy optimized hyperparameters such as learning rate, max depth, and the number of estimators, ensuring good model performance.
5. **Validation:** Stratified K-Fold Cross-Validation ensured consistent evaluation, particularly in the presence of class imbalances, focusing on metrics like the F1-weighted score to balance precision and recall.
6. **Deployment Preparation:** The pipeline's transformer, scaler, and best-performing model were exported using joblib, enabling seamless integration into a production environment.

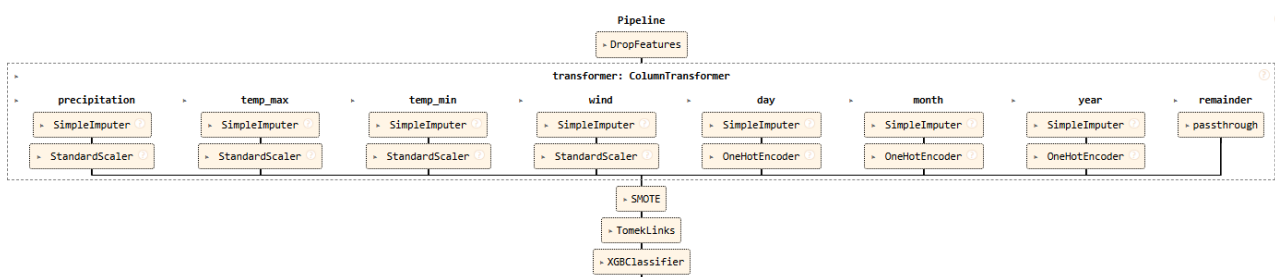


Figure 9. Pipeline

## TESTING PHASE AND EVALUATION OF PERFORMANCES

After the initial training phase, the model was evaluated using 20% of the data that had been previously separated from the rest of the dataset. This split was designed to ensure an unbiased assessment of the model's performance on unseen data, providing an accurate measure of its generalization capability.

## TRAINING PHASE

During the testing phase, the XGBoost model was **evaluated** using a previously unseen test dataset. The **predictions** generated by the model were **compared with the actual labels** to calculate performance metrics, such as **overall accuracy**, and to analyze the **distribution of predictions**.



**against the actual outcomes.** To simplify interpretation and analysis, a mapping was applied to convert numerical label values into comprehensible categories, such as rain, sun, and snow, making the representation and evaluation of the results more intuitive.

**ANALYSIS OF PERFORMANCES**

The classification report provides a detailed breakdown of the model's performance across the weather categories. The **overall accuracy** on the test set is **75%**, with a **weighted average F1-score** of **0.78**, indicating a well-balanced performance. Notably, the **Rain** class achieved very high **precision (0.95)** and **recall (0.88)**, reflecting the model's strong ability to correctly predict rainfall while minimizing both false positives and false negatives. Similarly, the **Snow** class, although having limited representation in the dataset, maintains **balanced performance** with a **precision and recall of 0.50**.

Class	Precision	Recall	F1-Score	Support
Drizzle	0.24	0.64	0.35	11
Fog	0.35	0.60	0.44	20
Rain	0.95	0.88	0.91	84
Snow	0.50	0.50	0.50	2
Sun	0.88	0.70	0.78	126
Accuracy			0.75	243
Macro Avg	0.58	0.66	0.60	243
Weighted Avg	0.83	0.75	0.78	243

Table 3. Classification report on test set

These results are deemed satisfactory because **we prioritize recall over precision** for critical output classes like Rain and Snow, where **false negatives pose a greater risk**. In the case of rain or snow, a false negative (failing to predict these conditions) could lead to severe **economic losses** and potential safety hazards, such as accidents due to a lack of appropriate preparations. On the other hand, a false positive for these conditions, while potentially leading to unnecessary expenditures on equipment, is far less costly in comparison. The model's tendency to **produce false positives in critical categories aligns with this priority**, making it the most suitable final model for our use case.

However, talking about **Drizzle**, it shows a **precision of 24%**, **recall of 64%**, and an **F1-score of 0.35**, based on 11 total samples in the test set. This indicates a bias in the model, which might be attributing **borderline cases** to Drizzle instead of other similar categories.

Weather	Correct	Total	Accuracy (%)
Drizzle	7	11	64.04
Fog	12	20	60.00
Rain	74	84	88.10
Snow	1	2	50.00
Sun	88	126	70.24

Table 4. Predictions summary for each class on test set

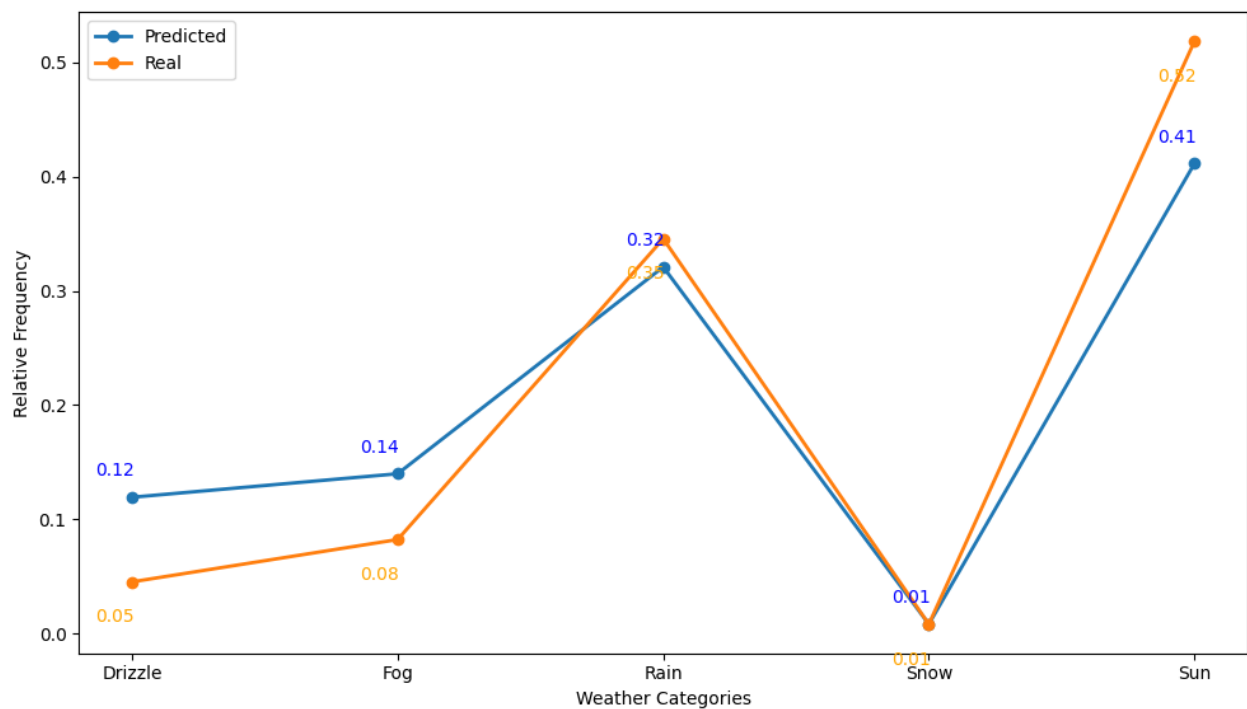


Figure 10. Discrepancy between predictions and real results

## LIMITATIONS AND AREAS FOR IMPROVEMENT

While the model achieved a satisfactory overall accuracy of **74.90%** on the test set, its **performance varied** significantly **across different weather categories**, exposing certain limitations. The Snow class, which was **severely underrepresented** in the dataset with **only thirteen samples**, achieved a correct prediction rate of **50%**. This highlights a **key weakness** in the model's ability to **handle rare events** effectively, suggesting the need for more balanced class distributions to improve its robustness.

Similarly, the Fog class, with a prediction accuracy of **60%**, indicates room for improvement in identifying situations where **visibility is a critical factor**. This limitation might stem from **insufficient or suboptimal features** in the dataset that could better capture the specific conditions associated with fog.

In addition for Drizzle class, given that Drizzle is not considered a **critical event** compared to categories like Snow or Rain, this limitation is acceptable within the context of the **model's primary objective**. While further improvements could enhance precision for this category, the focus remains on optimizing the performance for high-risk weather conditions to minimize potential safety and economic impacts.

A clear avenue for improvement involves **expanding the dataset with more samples**, especially for **underrepresented classes** like Snow and Fog, as well as increasing the diversity of features used in training. Additional features, such as atmospheric pressure, humidity gradients, or wind direction, could provide richer information to enhance the model's predictive performance. By addressing these limitations, future iterations of the model could achieve greater accuracy and reliability across all weather categories.

# DOCKER CONTAINERIZATION

In this section, we will explain step by step the creation of the **web application** that allows users to interact with the model, as well as its **containerization** and the **development of the Dockerfile**. This comprehensive walkthrough will highlight the technical decisions and tools used to streamline the deployment process.

## WEB APPLICATION GUI

The **graphical user interface (GUI)** of the **Weather Forecasting App** is designed to provide a user-friendly and interactive experience for predicting weather conditions. Built using **Streamlit**, the interface features a **sidebar** where users can input **weather-related parameters** such as maximum and minimum temperatures, precipitation levels, wind speed, and the date (day, month, year). By default, the date fields are **prefilled with the current date** to streamline the input process. Users can adjust these parameters using sliders or numeric input boxes, making the app intuitive and accessible.

Once the inputs are set, users can click the "**Predict Weather**" button to initiate the prediction process. The app loads a pre-trained machine learning model and its corresponding transformer to process the input data. After preprocessing and aligning the user inputs with the model's expected features, the model generates a weather prediction. **The result is displayed directly on the interface**, providing immediate feedback.

The mockup shows a web application interface for weather forecasting. On the left is a sidebar with input controls: four sliders for Maximum Temperature (°C), Minimum Temperature (°C), Precipitation (mm), and Wind Speed (m/s), each ranging from -50 to +50; and three dropdown menus for Day Of the Month (set to 1), Month (set to 1), and Year (set to 2025). The main content area has a title 'Weather Forecasting App', a disclaimer about data source and model use, a large blue box with weather icons (sun, cloud with rain, snowflake) and the text 'Weather Forecast PREDICTION TOOL', a prompt to fill in details, and a 'Predict weather' button.

Figure 11. GUI's mockup

The application is designed with a **single-page layout** to maintain a clear focus on its primary objective: delivering weather predictions. Adding multiple pages to such a **straightforward prototype** would introduce **unnecessary complexity without providing significant value**. Furthermore, the decision was made to **exclude a login page**, as authentication is not essential for this service. Since the application **does not handle sensitive user data**, the absence of authentication mechanisms simplifies the user experience while keeping the system efficient and streamlined.

## IMAGE DEFINITION: DOCKER FILE

The Dockerfile is designed to **containerize the Weather Forecasting application**, enabling seamless deployment and execution in any environment.

It begins with a **lightweight Python base image** (*python:3.10.9-slim-buster*) to ensure efficiency while supporting the required Python runtime. **Essential system libraries** and **Python dependencies** are installed, including those necessary for image processing and data transformation. The **application's files**, including the main script (*app\_pipeline.py*), model files, data, images, and logs, are organized into a dedicated **/weather working directory**.

The Dockerfile also **exposes port 8501**, which is used by **Streamlit**, the framework running the web interface of the application. An **entry point** and **command** ensure that the container **launches the application as soon as it starts**, providing a streamlined and portable solution for hosting the Weather Forecasting app.

```
FROM python:3.10.9-slim-buster
LABEL authors="Utente"

WORKDIR /weather

# Copy requirements
COPY requirements.txt ./requirements.txt

# Install dependencies and clean up to reduce image size
RUN apt-get update && apt-get install -y \
    libgl1-mesa-glx \
    libglu1-mesa-dev \
    libsm6 \
    libxext6 \
    libxrender-dev \
    libxrender1 \
    gcc \
    g++ \
    build-essential && \
    apt-get clean && rm -rf /var/lib/apt/lists/* && \
    pip3 install -r requirements.txt

# Expose port
EXPOSE 8501

# Copy all the files needed for the application
ADD data /weather/data
COPY app_pipeline.py /weather
ADD models /weather/models
ADD imgs /weather/imgs
ADD logs /weather/logs

# Create an entry point to make the image executable
ENTRYPOINT ["streamlit", "run"]
CMD ["app_pipeline.py"]
```

Codes 1. Dockerfile

```
imbalanced-learn
imblearn
streamlit
watchdog
pandas
pandas-profiling
joblib
pillow
numpy
feature_engine
dask[dataframe]
scikit-learn==1.5.0
threadpoolctl
xgboost
setuptools>=70.0
```

*Codes 2. requirements.txt*

## BUILDING AND RUNNING THE DOCKER IMAGE FOR THE WEATHER FORECASTING APPLICATION

As part of developing the weather forecasting application, two key actions were carried out using Docker.

First, the Docker image was built and named **weather\_forecasting\_image** using the **docker build** command. This step leveraged the specified **Dockerfile** to install all required dependencies, configure the Python environment, and prepare the application for execution in an isolated container. The process completed successfully, producing a ready-to-use image.

```
PS C:\Users\Utente\Desktop\WeatherForecastingApp> docker build --tag weather_forecasting_image .
[+] Building 352.6s (18/18) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 971B
=> [internal] load metadata for docker.io/library/python:3.10.9-slim-buster
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [ 1/12] FROM docker.io/library/python:3.10.9-slim-buster@sha256:ffcccf17ca6bd1ea2f2743fad16e1209d634f1e51a88dbfdc589e1dd57634299
=> => resolve docker.io/library/python:3.10.9-slim-buster@sha256:ffcccf17ca6bd1ea2f2743fad16e1209d634f1e51a88dbfdc589e1dd57634299
=> [internal] load build context
=> => transferring context: 5.34kB
=> CACHED [ 2/12] WORKDIR /weather
=> CACHED [ 3/12] COPY requirements.txt ./requirements.txt
=> CACHED [ 4/12] RUN apt-get update && apt-get install -y 11bg11-mesa-glx 11bg11b2.0-0 11bsm6 11bxtxt6 11bxrender-dev 11bxren
=> [ 5/12] RUN pip3 install -r requirements.txt 236.5s
=> [ 6/12] RUN pip install --upgrade "setuptools>=70.0" 5.7s
=> [ 7/12] RUN pip install --upgrade pip 5.1s
=> [ 8/12] ADD data /weather/data 0.1s
=> [ 9/12] COPY app_pipeline.py /weather 0.1s
=> [10/12] ADD models /weather/models 0.1s
=> [11/12] ADD imgs /weather/imgs 0.1s
=> [12/12] ADD logs /weather/logs 0.1s
=> exporting to image 102.5s
=> => exporting layers 84.0s
=> => exporting manifest sha256:db1fb6b33137e49f649395879abe0a08a69faec188b432b197c53d3383d6cc2f 0.0s
=> => exporting config sha256:54aef4b19ec03beeb8171cb53acf07f94f6800be1b0cb8ef915c8c510c47a8c9 0.0s
=> => exporting attestation manifest sha256:7b8790c9f16c219eac90b2a9b19d056ceb4c2f4121f0775b6385ade053c531c6 0.0s
=> => exporting manifest list sha256:d8b6db66b0d16d73f9fa1d93e9de94c62abf89d0aced31ee648327c70a825d6 0.0s
=> => naming to docker.io/library/weather_forecasting_image:latest 0.0s
=> => unpacking to docker.io/library/weather_forecasting_image:latest 18.4s
View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/ykwyjt22sgrb38h4hvbbsq24o
```

Next, the application was run within a **Docker container**. The built image was launched using the **docker run** command, **mapping port 8501** to enable access to the **Streamlit application**.

```
PS C:\Users\Utente\Desktop\WeatherForecastingApp> docker run -p 8501:8501 --name weather_forecasting_container weather_forecasting_image
collecting usage statistics. To deactivate, set browser.gatherUsageStats to false.

You can now view your Streamlit app in your browser.

Local URL: http://localhost:8501
Network URL: http://172.17.0.2:8501
External URL: http://77.47.0.202:8501
```

As a result, the application became available through the local address: <http://localhost:8501>.

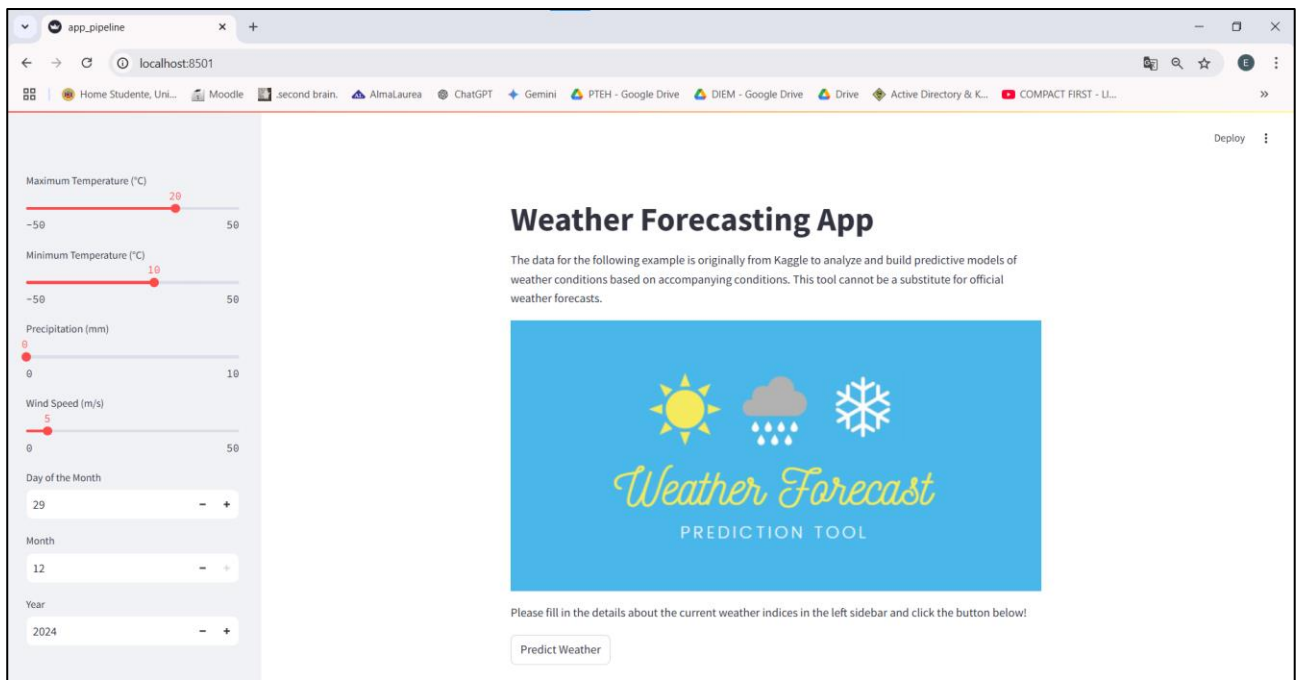


Figure 12. Final GUI

# CONTAINER VULNERABILITIES ANALYSIS

During the analysis of container images with **Docker Scout**, three critical vulnerabilities were identified that require immediate remediation to ensure system security and data integrity.

## CVE-2024-5206

The first vulnerability, **CVE-2024-5206**, is a **sensitive data leakage issue** affecting the **TfidfVectorizer** in the **scikit-learn** library. This vulnerability, present in versions up to **1.4.1.post1**, occurs because it **incorrectly stores unnecessary information**, including **sensitive data** such as **passwords or keys**, in its **stop\_words\_** attribute. This behavior can lead to **accidental data exposure**.

### CVE-2024-5206

A **sensitive data leakage vulnerability** was identified in the **TfidfVectorizer** module of the **scikit-learn** library. This issue, present in versions up to and including **1.4.1.post1**, was fixed in version **1.5.0**. The vulnerability arises from the unexpected storage of **all tokens** in the **stop\_words\_** attribute, rather than only storing the subset required for the **TF-IDF technique** to function. This behavior could result in the storage of sensitive information, such as **passwords or keys**, which were meant to be discarded. The extent of the vulnerability's impact depends on the nature of the data being processed by the vectorizer.

- **CVSS Score:** 5.3
- **EPSS Score:** 0.00043 (0.109)
- **CVSS Vector:** CVSS:3.0/AV:N/AC:H/PR:L/UI:N/S:U/C:H/I:N/A:N
- **Affected Range:** <1.5.0
- **Fix Version:** 1.5.0
- **Publish Date:** 2024-06-06

The issue has been fixed in **version 1.5.0**, and we have addressed the problem by **updating to this version during the model training process** to ensure compatibility and prevent potential data leaks.

### Weather Forecasting App

In this notebook, we will define the pipeline for the final model, incorporating the modifications and insights derived from the data analysis conducted in the notebook 'Weather\_Forecasting\_Offline\_Data\_Analysis\_and\_Preprocessing'. The pipeline will also leverage the binary classifier selected based on the evaluation performed in the notebook 'Weather\_Forecasting\_Model\_Selection'. Additionally, we will train and evaluate the model to ensure a certain level of performance.

```
!pip install feature_engine
!pip install "dask[dataframe]"
!pip uninstall -y scikit-learn
!pip install scikit-learn==1.5.0
!pip install --upgrade threadpoolctl
!pip list | grep scikit
```

## CVE-2024-6345

The second vulnerability, **CVE-2024-6345**, pertains to **remote code execution (RCE)** in the **package\_index** module of **setuptools**. Versions up to **69.1.1** allow malicious actors to **inject arbitrary commands into the system** through user-controlled inputs, such as package URLs used in the download functions. This vulnerability is particularly dangerous as it could be exploited to execute unauthorized commands on the host system, compromising its integrity.

### CVE-2024-6345

A critical vulnerability in the `package_index` module of `pypa/setuptools` (versions up to 69.1.1) allows for **remote code execution (RCE)** through its download functions. These functions, used to retrieve packages from user-provided URLs or package index servers, are vulnerable to **code injection**. If exposed to user-controlled inputs, such as manipulated package URLs, an attacker could exploit this vulnerability to execute arbitrary commands on the affected system.

The issue has been resolved in version 70.0.0, and immediate updates are strongly recommended for all affected systems.

- **CVSS Score:** 7.5
- **EPSS Score:** 0.00043 (0.109)
- **CVSS Vector:** CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:A/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N
- **Affected Range:** <70.0.0
- **Fix Version:** 70.0.0
- **Publish Date:** 2024-07-15

The issue has been mitigated in **version 70.0.0 of setuptools**, and we have resolved it by **updating to this version in the Dockerfile** to ensure compatibility and secure the environment.

```
Dockerfile x
1 FROM python:3.10.9-slim-buster
2 LABEL authors="Utente"
3
4 WORKDIR /weather
5
6 # Copy requirements
7 COPY requirements.txt ./requirements.txt
8
9 # Install dependencies
10 RUN apt-get update && apt-get install -y \
11     libgl1-mesa-glx \
12     libgl2.0-0 \
13     libsm6 \
14     libxext6 \
15     libxrender-dev \
16     libxrender1 \
17     gcc \
18     g++ \
19     build-essential
20 RUN pip3 install -r requirements.txt
21 RUN pip install --upgrade "setuptools>=70.0"
22 RUN pip install --upgrade pip
```

### CVE-2023-5752

The third vulnerability, **CVE-2023-5752**, impacts the **installation of packages from Mercurial VCS URLs** when using versions of **pip** prior to **22.3**. Specifically, the **Mercurial revision control system** could be manipulated to **inject malicious configurations** via the `hg clone` command or similar functions. This vulnerability could allow attackers to control repository settings and modify how configurations are applied, leading to potential compromises in the software supply chain.



## CVE-2023-5752

A **security vulnerability** was identified when using *pip* versions prior to **22.3** to install packages from Mercurial VCS URLs. This vulnerability arises because a **specified Mercurial revision** could be used to inject arbitrary configuration options into the hg clone call (e.g., `--config`). If exploited, this could allow attackers to manipulate the Mercurial configuration and control which repository is installed. However, this vulnerability does not affect users who are not installing from Mercurial.

- **CVSS Score:** 6.8
- **EPSS Score:** 0.00045 (0.178)
- **CVSS Vector:** CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:N/I:L/A:N
- **Affected Range:** <22.3
- **Fix Version:** 22.3
- **Publish Date:** 2023-10-25



We have resolved this issue by **upgrading to pip version 22.3 directly in the Dockerfile** to ensure the environment is secure and protected when installing packages from Mercurial VCS URLs.

```
Dockerfile
1 FROM python:3.10.9-slim-buster
2 LABEL authors="Utente"
3
4 WORKDIR /weather
5
6 # Copy requirements
7 COPY requirements.txt ./requirements.txt
8
9 # Install dependencies
10 RUN apt-get update && apt-get install -y \
11     libgl1-mesa-glx \
12     libglu1-mesa-dev \
13     libsm6 \
14     libxext6 \
15     libxrender-dev \
16     libxrender1 \
17     gcc \
18     g++ \
19     build-essential
20 RUN pip3 install -r requirements.txt
21 RUN pip install --upgrade pip
```

These findings underscore the critical importance of regular dependency updates, comprehensive vulnerability scanning, and proactive security practices in managing containerized environments. By addressing these vulnerabilities promptly, organizations can significantly reduce their exposure to potential threats and maintain the reliability and security of their systems.

Docker Scout [Give feedback](#)

## Advanced image analysis with Docker Scout

 Want to use Docker Scout on your remote repositories? [Set up your integrations now](#) 

Understand your application's dependencies, analyze the vulnerabilities, and act quickly with suggested remediation options. [Learn more](#) and [upgrade](#).

Sample image	Vulnerabilities	
weather_forecasting_image:latest	0 0 0 6 2	<a href="#">View packages and CVEs</a>

Figure 13. Docker Scout final results

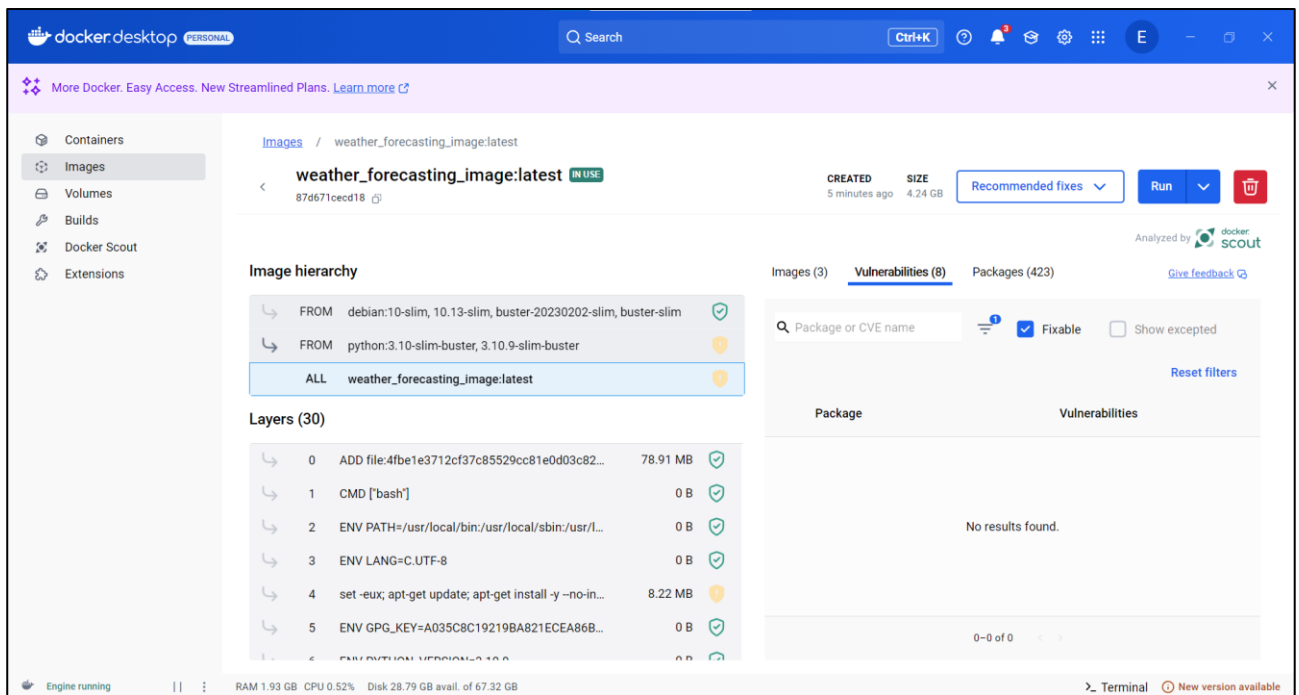


Figure 14. Docker scout final results - fixable vulnerabilities

# CLUSTER INFRASTRUCTURE

---

In this section, we will provide a **comprehensive overview of the cluster's architecture** and its key components, as well as discuss some **necessary mechanisms or enhancements** that could be integrated to improve functionality and scalability. This includes considerations for current requirements and potential future developments that may align with the **evolving needs** of the system. In the following section, we will **translate these plans into practical implementations** using Kubernetes, detailing the steps taken to bring the outlined concepts to fruition.

## THE ARCHITECTURE

A basic weather prediction application does **not require a highly complex base architecture**. Given that the prototype presented in this report uses an offline-trained model, the key focus of our architectural choices—as well as the majority of other design decisions, as detailed in the first chapter—has been **availability and speed** of the platform.

For an application like ours, availability and response speed are critical factors, as they underpin the primary objectives for delivering an **optimal Quality of Service (QoS)** that meets user expectations. For this reason, our architectural requirements can be summarized in two key attributes:

1. **Dynamic Adaptability:** The system's ability to automatically scale in response to fluctuating workloads is especially relevant in a weather forecasting system, where user demand can vary drastically during critical weather events like storms or floods. Scalability must ensure the system can handle request spikes without performance degradation while optimizing resource usage during normal periods.
2. **Modularity:** Every component must be designed as an independent module that can be updated, replaced, or scaled without disrupting the overall system's functionality.

Among the various architectures available, the choice narrowed down to two possibilities: the **Dynamic Scalability Architecture** and the **Elastic Resource Capacity Architecture**.

In the specific context of this weather forecasting project, the Elastic Resource Capacity Architecture proves unsuitable primarily due to the lack of sufficient machines or external resources required to significantly scale individual pods vertically. Vertical scaling also generally remains impractical in cloud environments, as it often involves downtime during the augmentation or replacement of resources, which is incompatible with the project's critical requirements for high availability and responsiveness.

Moreover, theoretical considerations indicate that extreme elasticity is not a primary driver for our design due to the specific main operative characteristics of the application:

- The **stateless nature** of the application, stemming from its offline-trained model, means there is no reliance on maintaining internal states across requests.
- The **generally uniform resource requirements** of requests reduce the need for frequent or extensive elasticity adjustments.

Considering these constraints, **dynamic scalability architecture** emerges as a far more effective and reliable solution.

## DYNAMIC HORIZONTAL SCALABILITY ARCHITECTURE

The **Dynamic Scalability Architecture** is an architectural model designed around a set of **predefined scaling conditions** that automatically trigger the dynamic allocation of IT resources from a cloud-based

compute resources. This model allows for variable utilization based on fluctuating usage demands, ensuring that resources are **allocated efficiently when needed** and **reclaimed when unnecessary**.

Dynamic **horizontal scalability**, in particular, involves dynamically adding or removing IT resource instances (e.g., Kubernetes pods) in response to fluctuating workloads. An automated scaling listener continuously monitors traffic and workload thresholds, triggering resource replication when needed to ensure the system can seamlessly adapt to variations in demand. This approach is a better solution compared to a version of the architecture relying **on vertical scaling or dynamic relocation**. Vertical scaling, which increases the processing capacity of a single resource, is **impractical** in this context due to its **downtime requirements and limited scalability**, especially given the project's constraints of operating with minimal nodes and resources. Similarly, **dynamic relocation**, which involves transferring IT resources to a host with greater capacity, is **not a viable option** here. It assumes the availability of **multiple hosts** with varying resource levels, which is not the case in this project. Therefore, **dynamic horizontal scalability** stands out as the most effective and reliable choice to address the specific demands of this weather forecasting application.

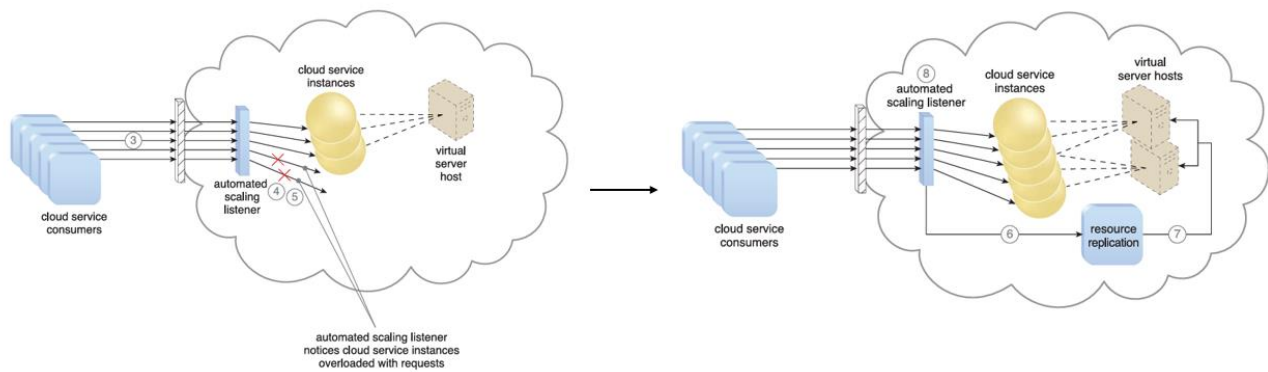


Figure 15. Dynamic scalability architecture

## COMPLEMENTAL ARCHITECTURES: WORKLOAD DISTRIBUTION ARCHITECTURE

In addition to the primary **Dynamic Scalability Architecture**, other complementary architectures can be integrated to further enhance scalability, speed, and availability of the weather forecasting application. Among these, the **Workload Distribution Architecture** stands out as a valuable option.

The Workload Distribution Architecture focuses on **efficiently distributing the incoming workload across multiple resources** or nodes. When combined with the Dynamic Scalability Architecture, it enhances the system's ability to **handle varying levels of demand effectively**:

- **Preventing Overload:** while dynamic scalability ensures that the number of pods increases or decreases in response to traffic, workload distribution ensures that the traffic is evenly **spread across available resources**.
- **Ensuring resource efficiency:** dynamic scalability can adjust the number of pods based on demand, but without a distribution mechanism, certain pods **may still become bottlenecks**. By implementing workload distribution, each pod operates closer to its **optimal capacity**, maximizing the efficiency of the scaled resources.
- **Guaranteeing failover resilience:** if a pod becomes unavailable, the workload distribution mechanism can **reroute traffic to other healthy resources**, ensuring continuous availability of the service even during partial failures.

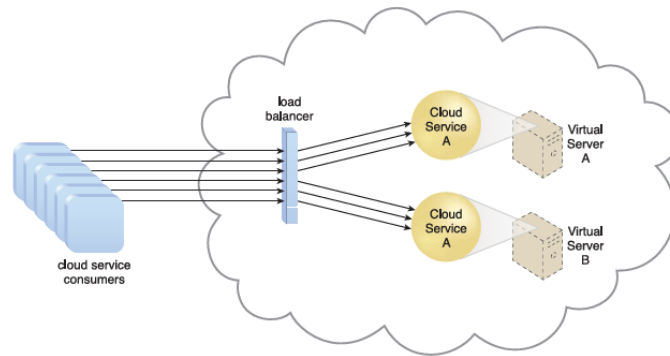


Figure 16. Workload distribution architecture

## ASSOCIATED MECHANISMS AND COMPONENTS

The architecture for the weather forecasting system must incorporate essential mechanisms such as **resource replication**, **automated scaling listeners** and **SLA monitoring**.

These mechanisms work in tandem to meet the specific requirements of the weather forecasting system:

- **Resource replication** and **automated scaling listeners** dynamically adjust resources to handle fluctuating demand.
- **Load balancer** ensures efficient distribution of incoming traffic across multiple instances, preventing bottlenecks and ensuring optimal resource utilization.
- **SLA monitoring** ensures compliance with QoS standards and triggers actions during performance issues.

This cohesive integration of mechanisms provides **the scalability, responsiveness, and quality required** for the project's success. Rather than delving into the detailed workings of each mechanism, we will limit ourselves to listing these mechanisms to establish their correspondence with the elements that will be configured and implemented in the following chapter.

# KUBERNETES CONFIGURATION

---

In this section, we outline the cluster configuration and how the theoretical concepts from the previous chapter were applied. For design choices (e.g., workers, pods, requests, limits), refer to the next chapter, except for minor exceptions.

## ARCHITECTURAL OPTIMIZATION

In the previous chapter, we outlined the system's basic architecture, which combines dynamic scalability and workload distribution architectures. However, as is often the case when moving from theory to practice, we were forced to **simplify the initial architecture** due to some challenges encountered during testing (discussed in the next chapter) and the limitations of the tools available to us.

The main architectural change concerns the **removal of the load balancer between the cluster nodes**. Although theoretically the load balancer seemed like the most appropriate solution, given that our system involves multiple nodes with different pods running the same service and needs to be accessible by external users, we could not implement it for two main reasons:

1. **Kubernetes does not support a native load balancer**, and while we could implement a Service as a load balancer within the cluster, it would still require an external cloud provider or infrastructure to manage the traffic across multiple nodes effectively. Since we don't have access to such a provider, an internal load balancer within the cluster would not be feasible or efficient.
2. Even if we had chosen to implement an **Ingress Controller** as a load balancer (in addition of what is already discussed in the last chapter), the tests showed that the optimal configuration for our scenario is a **single node**. In this configuration, a load balancer would be ineffective and would result in **resource waste**, without providing any benefits.

Therefore, considering that the presented prototype is a **simplified demo** where users interact with a primarily **static page**, we decided **not to implement a load balancer between the nodes**. Instead, we focused on the load balancing managed by the Kubernetes Service, which distributes traffic among the pods within the single node, thus avoiding unnecessary complications. This choice was also made to ensure more stable pod tests, as the load balancer would have been redundant and would have only slowed down the process.

## CLUSTER CONFIGURATION OVERVIEW

A Kubernetes cluster consists of a group of worker machines, referred to as nodes, which are responsible for running containerized applications. Each cluster must have at least one worker node. These nodes host **Pods**, the smallest and most basic units of application workload in Kubernetes, where containers are deployed and executed.

Our cluster consists of a **control plane** and a **worker node**, as defined in the *weather-app-config-with-port-mapping.yaml* file, where:

- **Control Plane Node:** serves as the brain of the Kubernetes cluster. It manages the state of the cluster, schedules workloads, and handles tasks such as pod orchestration, service discovery, and failover.
- **Worker Node:** These nodes handle the actual application workloads by running the pods.

```

kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
    extraPortMappings:
      - containerPort: 30080
        hostPort: 30070
  - role: worker

```

Codes 3. *weather-app-config-with-port-mapping.yaml*

### WHY JUST ONE WORKER NODE?

We chose to maintain a **single worker node** in our configuration to optimize **resource utilization** and ensure **stable performance**. During **load testing** (as detailed in the next chapter), we observed that increasing the number of nodes did **not lead to a significant improvement** in performance. As shown in the graph, the **two-node setup** initially had a **higher time response** compared to the **single-node configuration**, but as the load increased, the advantage **diminished and eventually stabilized**.

This behavior suggests that, given the **same overall resources**, **distributing the load across multiple nodes** did not provide **tangible scalability benefits**. Instead, it introduced **overhead** that **limited system responsiveness**. As a result, we decided to **concentrate resources** on a **single node**, maximizing its **utilization** to achieve **more consistent response times**.

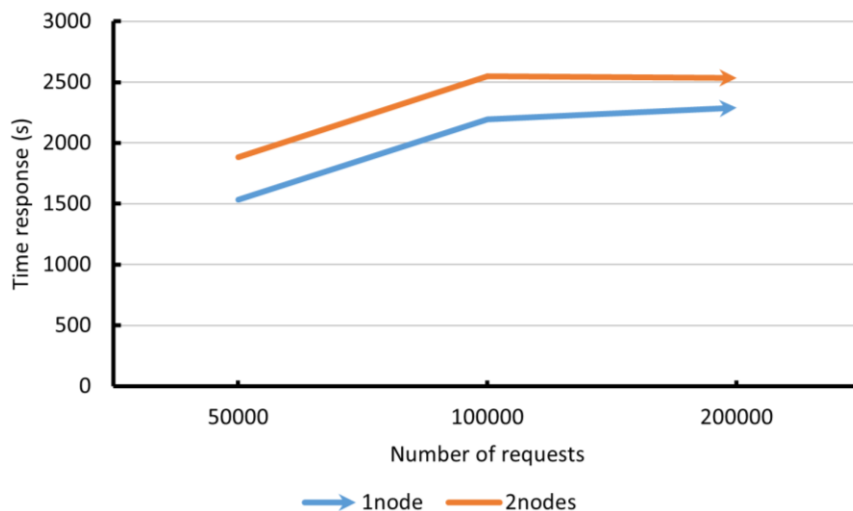


Figure 17. *Number of nodes vs number of requests (related to next chapter's tests results)*

## KUBERNETES CONFIGURATION FOR DEPLOYMENT AND SERVICE

Deploying a web app in a Kubernetes cluster involves packaging the application into a container, **creating Kubernetes manifest files**, and **applying them to the cluster**. This process ensures scalability, high availability, and efficient resource management for the application. The manifest file defined for the application is the *k8sweatherdeployment.yaml*. It includes two main resources: Deployment and Service, each serving a specific purpose to ensure the application's functionality within the cluster.

### DEPLOYMENT

This resource manages the Pods running the application by defining their configuration and behavior. The Deployment specifies the application name as **weather-forecasting-app** and uses our custom Docker image, **elenafalcone9/weather\_forecasting\_image**. It is configured to run a **two replicas**, ensuring a couple of instances of the application are active. **Resource requests and limits** for CPU

and memory are set to guarantee efficient and controlled utilization of cluster resources. The rationale behind the chosen values for resource requests and limits, as well as the decision to start with a single pod, will be detailed in the next chapter. The container exposes the application on port **8501**, allowing it to communicate with the Service and ensuring accessibility within the Kubernetes cluster.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: weather-forecasting
  labels:
    app: weather-forecasting-app
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: weather-forecasting-app
  template:
    metadata:
      labels:
        app: weather-forecasting-app
    spec:
      containers:
        - name: weather-forecasting
          image: elenafalcone9/weather_forecasting_image
          resources:
            requests:
              memory: "80Mi"
              cpu: "400m"
            limits:
              memory: "250Mi"
              cpu: "800m"
          ports:
            - containerPort: 8501
```

*Codes 4. k8sweatherdeployment.yaml-deployment*

## SERVICE

A Kubernetes Service of type **NodePort** exposes our **weather-forecasting application to external traffic** by mapping a high port on the node to the application running in the cluster. It is configured to forward traffic from **nodePort 30080** (the external entry point on the Kubernetes node) to the application listening on **port 8501** inside the container. The **selector** matches pods labeled **app: weather-forecasting-app**, ensuring that the Service forwards traffic only to the appropriate pods.

```
apiVersion: v1
kind: Service
metadata:
  name: weather-forecasting
  namespace: default
spec:
  type: NodePort
  selector:
    app: weather-forecasting-app
  ports:
    - port: 8501
      targetPort: 8501
      nodePort: 30080
```

*Codes 5. k8sweatherdeployment.yaml-service*

This configuration **enables external users to access the application via the node's IP address** and port 30080, while Kubernetes handles internal routing to the application.



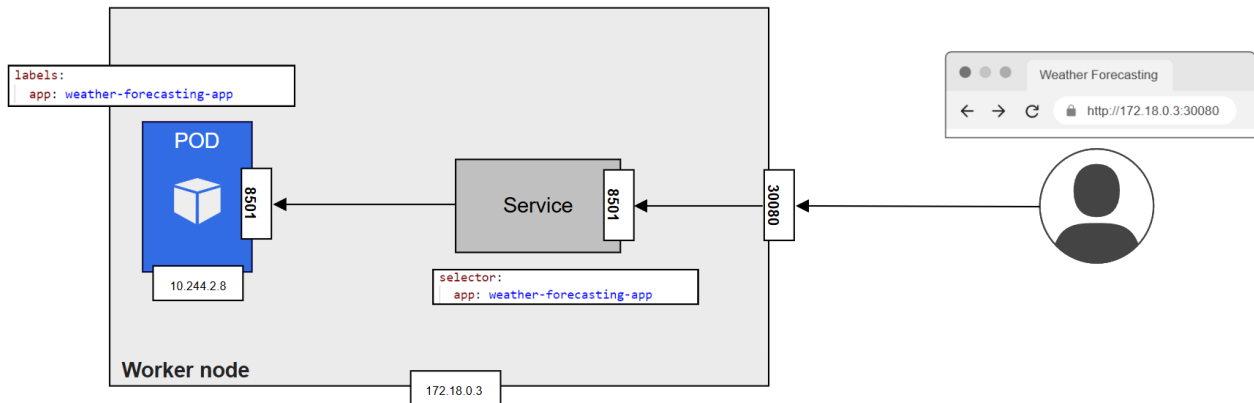


Figure 18. Cluster's initial configuration scheme

### PORT MAPPING FOR ACCESSIBILITY

Port mapping is a mechanism that **allows external traffic** (from the host machine or network) to be directed to a specific port inside a container or service running in a Kubernetes cluster (or a containerized environment). It creates a bridge between the external port of a host machine and the internal port of a container or application.

It works by binding a **host port** (the port on the physical machine or node) to a **container port** (the port on which the containerized application listens). In this specific case: when a user accesses port **30070** on the host, the traffic is routed through port mapping to port **30080** on the Kubernetes node. The Kubernetes Service then captures the traffic on port **30080** and automatically forwards it to the corresponding container, where the application listens on port **8501**. This flow enables the application to be exposed externally in a controlled manner, leveraging Kubernetes infrastructure to manage traffic effectively.

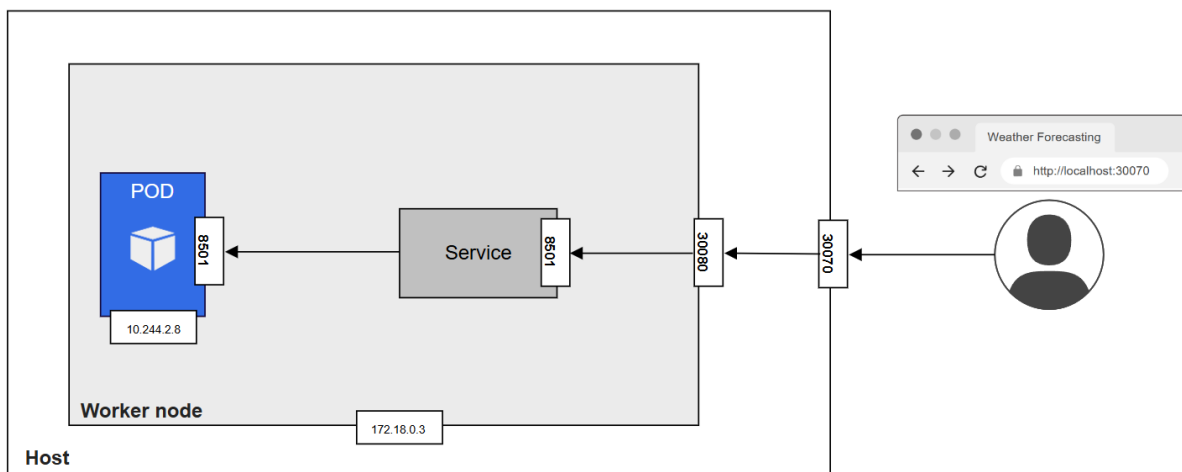


Figure 19. Port mapping scheme

### METRIC SERVER

To monitor the SLA, a **Metrics Server** has been deployed. This server aggregates and provides real-time performance metrics for Pods and nodes, enabling us to:

- Track resource utilization, such as CPU and memory, ensuring that requests and limits are appropriately set.
- Detect anomalies, such as increased resource usage or degraded performance, which could indicate potential failures.

- Lay the groundwork for implementing auto-scaling based on actual usage patterns.

The **Metrics Server** integrates with Kubernetes to provide actionable insights, helping maintain compliance with the SLA and allowing for timely adjustments to resources or configurations.

```
PS C:\Users\Utente\Desktop\WeatherForecastingApp\kubernetes-configuration> kubectl create --filename metrics-server.yaml
serviceaccount/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
service/metrics-server created
deployment.apps/metrics-server created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created
```

## STEP BY STEP CONFIGURING PROCEDURE

In this section, we will outline the detailed process of replicating the configuration of the cluster, deployment, and service. We will not repeat or restate the configuration files already provided in the previous section. Unless explicitly stated otherwise, please refer to the explanations provided earlier. The files, when fully detailed, have not been included here for brevity, as they are fairly standard. For any clarification or reference, they can be accessed within the dedicated folder.

### CLUSTER CREATION

First, we **create a Kubernetes cluster** named **weather-forecasting-cluster**, specifying a configuration file ([weather-app-config-with-port-mapping.yaml](#)).

```
PS C:\Users\Utente\Desktop\WeatherForecastingApp\kubernetes-configuration> .\kind.exe create cluster --name weather-forecasting-cluster --config=weather-app-config-with-port-mapping.yaml
Creating cluster "weather-forecasting-cluster" ...
  • Ensuring node image (kindest/node:v1.31.2) ...
    [x] Ensuring node image (kindest/node:v1.31.2) ...
  • Preparing nodes ...
    [x] Preparing nodes ...
  • Writing configuration ...
    [x] Writing configuration ...
  • Starting control-plane ...
    [x] Starting control-plane ...
  • Installing CNI ...
    [x] Installing CNI ...
  • Installing StorageClass ...
    [x] Installing StorageClass ...
  • Joining worker nodes ...
    [x] Joining worker nodes ...
Set kubectl context to "kind-weather-forecasting-cluster"
You can now use your cluster with:

kubectl cluster-info --context kind-weather-forecasting-cluster

Not sure what to do next? Check out https://kind.sigs.k8s.io/docs/user/quick-start/
```

Once the cluster is created, we can verify that the **cluster nodes have been successfully provisioned** and are operational.

```
PS C:\Users\Utente\Desktop\WeatherForecastingApp\kubernetes-configuration> kubectl get nodes
NAME                                STATUS    ROLES    AGE    VERSION
weather-forecasting-cluster-control-plane Ready    control-plane 30s    v1.31.2
weather-forecasting-cluster-worker  Ready    <none>      18s    v1.31.2
weather-forecasting-cluster-worker2 Ready    <none>      18s    v1.31.2
PS C:\Users\Utente\Desktop\WeatherForecastingApp\kubernetes-configuration> kubectl get nodes -o wide
NAME                                STATUS    CONTAINER_RUNTIME    STATUS    ROLES    AGE    VERSION    INTERNAL-IP    EXTERNAL-IP    OS-IMAGE                                KERNEL-VERSI
weather-forecasting-cluster-control-plane Ready    containerd://1.7.18   Ready    control-plane 42s    v1.31.2    172.18.0.4    <none>         Debian GNU/Linux 12 (bookworm)        6.10.14-linu
weather-forecasting-cluster-worker  Ready    containerd://1.7.18   Ready    <none>      30s    v1.31.2    172.18.0.2    <none>         Debian GNU/Linux 12 (bookworm)        6.10.14-linu
weather-forecasting-cluster-worker2 Ready    containerd://1.7.18   Ready    <none>      30s    v1.31.2    172.18.0.2    <none>         Debian GNU/Linux 12 (bookworm)        6.10.14-linu
```

### METRIC SERVER CONFIGURATION

The Metrics Server is a crucial component that **collects resource usage metrics** (such as CPU and memory) from nodes and pods within the cluster. These metrics are used for features like **Horizontal Pod Autoscaler** and **performance monitoring**. By specifying the [metrics-server.yaml](#) file, a predefined configuration is applied to install and set up the Metrics Server in the cluster.

```
PS C:\Users\Utente\Desktop\WeatherForecastingApp\kubernetes-configuration> kubectl create --filename metrics-server.yaml
serviceaccount/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
service/metrics-server created
deployment.apps/metrics-server created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created
```

## KUBERNETES DASHBOARD

We can configure and launch the **Kubernetes Dashboard** to manage our cluster. First, we use the `kubectl apply` command to **download and apply the necessary configuration** from the **official repository**.

```
PS C:\Users\Utente\Desktop\weatherForecastingApp\kubernetes-configuration> kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio-deploy/recommended.yaml
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrif created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
```

Next, we create an **admin user** using *dashboard-adminuser.yaml* and assign the required permissions with *cluster\_rolebinding.yaml*.

```
PS C:\Users\Utente\Desktop\WeatherForecastingApp\kubernetes-configuration> kubectl apply -f dashboard-adminuser.yaml
serviceaccount/admin-user created
clusterrolebinding.rbac.authorization.k8s.io/admin-user created
PS C:\Users\Utente\Desktop\WeatherForecastingApp\kubernetes-configuration> kubectl apply -f cluster_rolebinding.yaml
clusterrolebinding.rbac.authorization.k8s.io/admin-user unchanged
```

Then, we generate an **authentication token** for the admin user with the command `kubectl -n kubernetes-dashboard create token admin-user`. Finally, we **start a proxy server** using `kubectl proxy`, which allows us to access the dashboard through the local address 127.0.0.1:8001.

[illegible]

The dashboard launches successfully, but no workloads are currently present in the default namespace. However, once we create a deployment, the associated pods and other resources will appear in the dashboard, allowing us to monitor and manage them effectively.

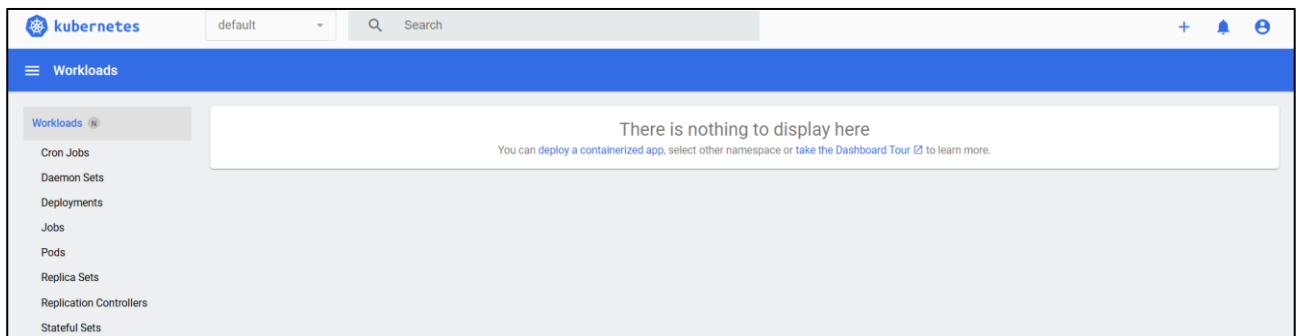


Figure 20. Kubernetes dashboard

## DEPLOYING THE WEATHER FORECASTING APP

Before deploying the Weather Forecasting App, we need to **containerize it using Docker**, push the container image to **Docker Hub**, and **create a Kubernetes deployment** for the app, following the procedure described in the chapter on Docker.

```
PS C:\Users\Ute\Deskto\WeatherForecastingApp> docker build --tag weather_forecasting_image .
[+] Building 2.2s (15/15) FINISHED
=> [internal] load build definition from Dockerfile
=> [internal] load metadata for docker.io/library/python:3.10.9-slim-buster
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load .dockerignore
```

We then proceed to **push the image** to Docker Hub:

```
PS C:\Users\Utente\Desktop\WeatherForecastingApp> docker tag weather_forecasting_image elenafalcone9/weather_forecasting_image
PS C:\Users\Utente\Desktop\WeatherForecastingApp> docker push docker.io/elenafalcone9/weather_forecasting_image
Using default tag: latest
The push refers to repository [docker.io/elenafalcone9/weather_forecasting_image]
aa79b8d69108: Pushing [>] 2.097MB/1.371GB
5aedb16535ea: Pushed
9b91180f0481: Pushed
636989d996d6: Pushing [=====] 3.345MB/3.345MB
b36b3eae5d3e: Pushing [=====] 2.097MB/4.141MB
73f2d4ecfb17: Pushing [=====] 2.097MB/2.781MB
1df92d2e332: Pushed
ac52e67086ee: Pushing [=====] 2.097MB/11.45MB
4090dfa0fe62: Pushed
0cf508b37688: Pushing [=>] 1.049MB/27.14MB
c67df8af2695: Pushed
4789cb1b2851: Pushed
ba3e5c6abe79: Pushed
92a1d342682d: Pushed
```

The **deployment phase** is crucial in ensuring that the application is **correctly served** within the cloud environment. Firstly the [k8sweatherdeployment.yaml](#) file is applied, successfully **creating the Deployment and Service** for the Weather Forecasting App. The `kubectl get deployments`, `kubectl get services`, and `kubectl get pods` commands confirm that the **Deployment, Service, and Pods are created**, with the Service exposing the app on a **NodePort (8501:30080/TCP)**.

```
PS C:\Users\Utente\Desktop\WeatherForecastingApp> kubectl create --filename k8sweatherdeployment.yaml
deployment.apps/weather-forecasting created
PS C:\Users\Utente\Desktop\WeatherForecastingApp> kubectl get deployments
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
weather-forecasting 0/2     2             0            4s
PS C:\Users\Utente\Desktop\WeatherForecastingApp> kubectl get services
NAME                TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
kubernetes          ClusterIP   10.96.0.1     <none>         443/TCP           22m
weather-forecasting NodePort    10.96.109.182 <none>         8501:30080/TCP    7s
PS C:\Users\Utente\Desktop\WeatherForecastingApp> kubectl get pods
NAME                READY   STATUS             RESTARTS   AGE
weather-forecasting-775f55f8c-fp9g1 0/1     ContainerCreating   0           10s
weather-forecasting-775f55f8c-rdd1f 0/1     ContainerCreating   0           10s
```

The Weather Forecasting App is now **running successfully** and accessible through the exposed Service.

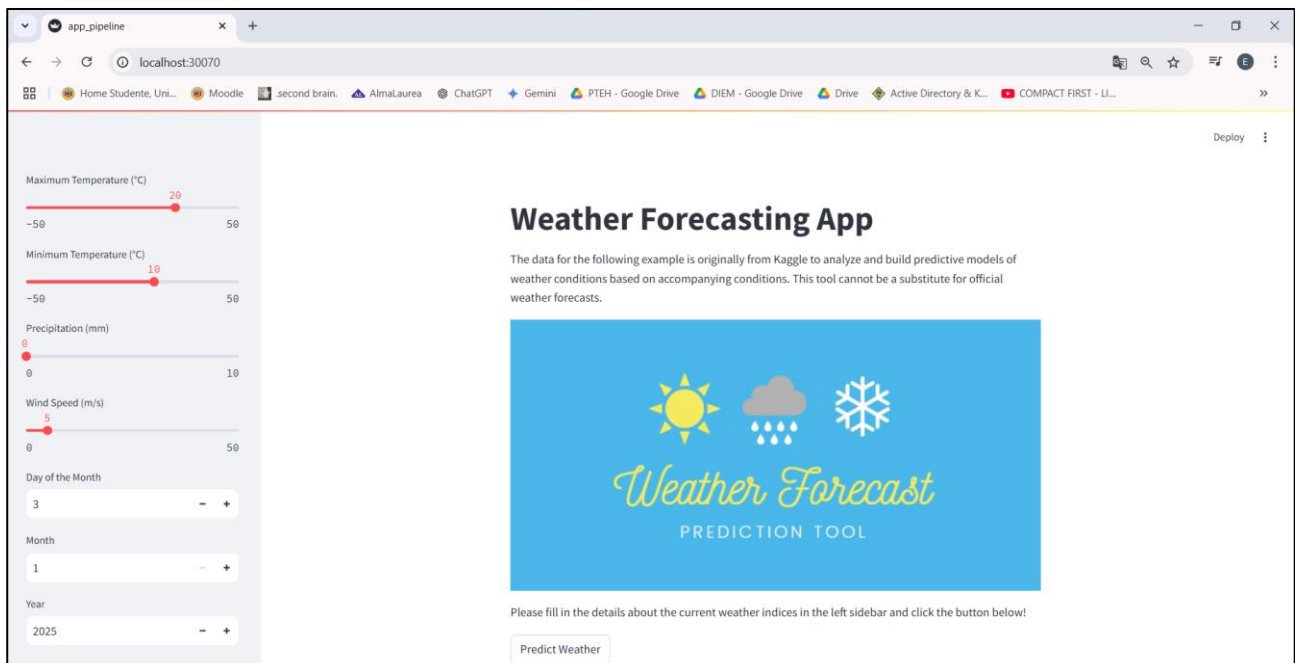


Figure 21. Final home page view after clusterization

# BENCHMARK

---

In this section, we will present the results of the stress tests that led to the final configuration of the cluster, pods, and the overall deployment. Additionally, we will explain the reasoning behind the chosen parameters for the horizontal pod autoscaler, whose inclusion is not only necessary but also aligned with the architecture we have selected. Finally, we will showcase the results of the final test, which will confirm whether the objectives we set at the beginning of this report have been achieved.

## HARDWARE SPECIFICATIONS AND PREMISES

The hardware used in the development and testing of this project consists of **two** identical Lenovo PCs to ensure the reliability and accuracy of the tests. Below are the specifications of the devices:

- **Device:** Lenovo IdeaPad (Model: 82TV)
- **CPU:** AMD Ryzen 5 3452U with Radeon Graphics, 2.70 GHz, 4 cores
- **Memory:** 8 GB RAM
- **OS:** Microsoft Windows 10 Pro (Version 10.0.19045)
- **IP Addresses:** 192.168.1.185 (server machine) and 192.168.1.58 (client machine)

We also specify that the tests presented in this chapter were conducted using the JMeter tool. This web application was developed with the goal of handling **50,000 requests per day**, with a peak of **2,000 simultaneous requests**. Therefore, all reported tests focused on the application's response to this load, varying the number of replicas, nodes, or resources.

## TESTS SET UP

In the test setup, **JMeter** was configured to simulate a **load of 2,000 simultaneous users** using the **Thread Group** element. The **ramp-up period** was set to **1 second**, ensuring that all threads (users) would start nearly instantaneously, replicating a peak traffic scenario. For the **loop count parameter**, it was decided to run multiple tests with different values (**10, 20, and 40**) in order to make the results as consistent as possible and minimize the impact of any outliers in tests results. This approach allowed for testing **total loads of 50,000, 100,000, and 200,000**. Specifically, the load was increased to **4 times the value required** by the specifications, as 50,000 was too small a value to provide an accurate analysis of the configurations' behavior over time.

Each user repeatedly sent HTTP requests to the target server located at 192.168.1.185 on port 30070, using the **HTTP Request** element with the GET method. The test was designed to loop infinitely, continuously generating requests until manually stopped, allowing for an extended evaluation of the system's stability and performance under high load. Results were monitored and aggregated using the **Summary Report** elements for detailed analysis.

## USAGE WITH DIFFERENT AMOUNTS OF REPLICAS

Starting with a total of 50,000, 100,000 e 200,000 requests, including 2,000 simultaneous ones, we initially focused on determining the **number of replicas required to handle these sudden traffic spikes**. Multiple tests were conducted with **varying numbers of replicas and varying requests amounts**. Thanks to the previously installed Metrics Server, we were able to **monitor resource consumption through the dashboard** we configured earlier.



### REPLICAS = 1

With **1 replica** handling the requests, the error rate remains impressively **low**, well below the **1% threshold**, with the highest value reaching only 0.12% at 200,000 requests. Although the error rate is low, the throughput **decreases significantly as the load increases**, indicating that a **single replica is not sufficient** to handle higher traffic efficiently. Additionally, the **response time** is too close to the limit of our **3-second target** for the number of requests specified in the test case, and it fails to meet the required time for higher load scenarios. This highlights that for higher traffic loads, additional replicas or scaling solutions are necessary to maintain both optimal performance and the desired **response time**.

	Average	Error%	Throughput
50,000	2671	0.00%	1135.4/sec
100,000	3570	0.09%	1025.7/sec
200,000	4374	0.12%	945.2/sec

Table 5. Stress test results - 1 pod

### REPLICAS = 2

With **2 replicas** handling the requests, the error rate remains at 0.00%, which is actually a very good result, and throughput remains consistent between different loads. These results meet the performance specifications, with **error rates staying well under the limit**. However, the **throughput remains relatively high**, even as the load increases. This could indicate that, while the two pods are sharing the load, they may **still be handling more traffic than ideal** for the current setup, potentially leading to resource inefficiencies. The high throughput suggests that additional pods or further optimization might be required to balance the load more effectively and prevent the system from reaching its maximum capacity.

	Average	Error%	Throughput
50,000	1531	0.00%	1974.3/sec
100,000	2192	0.00%	1941.6/sec
200,000	2288	0.00%	1952.9/sec

Table 6. Stress test results - 2 pods

### REPLICAS = 5

With **5 replicas**, the system has shown a **significant improvement** in performance. The **response time** is now reduced to **half** of the previous results, falling within the **lower half of our initial target range**, which is a strong indication of improved efficiency. The **error rate** is consistently at **0%**, and **throughput has met the requested load**, demonstrating that the system is handling traffic effectively without any issues. These results clearly show that the system is performing well and meeting the objectives.

To further explore the potential for better performance, we can test the system with **10 replicas** to see if additional scaling offers further improvements.

	Average	Error%	Throughput
50,000	545	0.00%	3186.3/sec
100,000	1334	0.00%	3041.8/sec
200,000	1386	0.00%	3175.6/sec

Table 7. Stress test results - 5 pods

## REPLICAS = 10

With **10 replicas**, the system maintains **high reliability**, with an **error rate of 0%**. However, the **average response time** does not always improve compared to the configuration with **5 replicas**, and the **throughput** experiences a slight decrease. This behavior can be attributed to the **additional overhead** caused by the need to coordinate a **larger number of replicas**, which partially offsets the benefits of scalability. Nevertheless, the **average response time** still falls within an **acceptable range** based on the objectives outlined in the task, making this configuration **valid** and not entirely **dismissible**.

	Average	Error%	Throughput
<b>50,000</b>	751	0.00%	2403.8/sec
<b>100,000</b>	1028	0.00%	2357.8/sec
<b>200,000</b>	1448	0.00%	2371.5/sec

Table 8. Stress test results - 10 pods

## CONCLUSIONS

Given the key Quality of Service (QoS) objectives established at the beginning of the project, both 5 and 10 replicas are viable options, as they meet the defined performance criterias. Both configurations ensure **high availability** and **fast response times**, fulfilling the primary goals of the system.

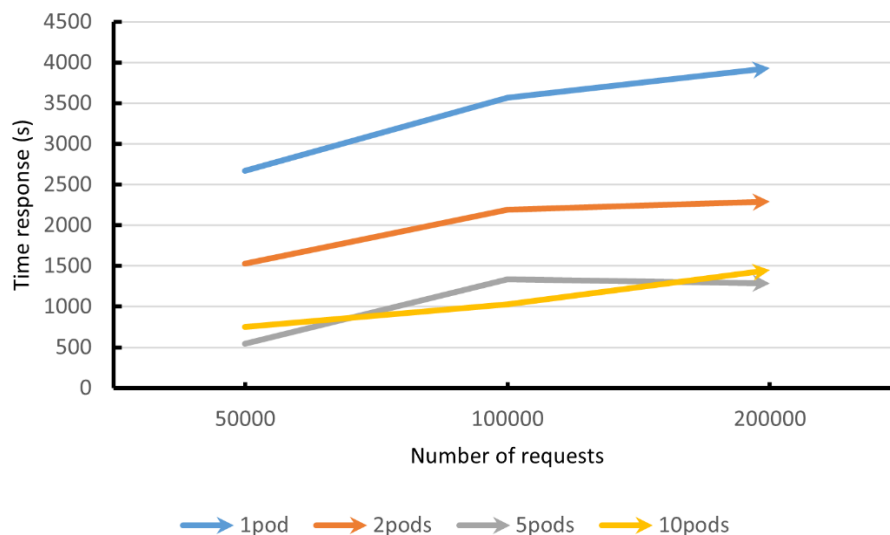


Figure 22. Number of pods vs number of requests vs response time

However, while 10 replicas offer a slight improvement in performance, they come with a significant increase in **resource consumption**, which could lead to inefficiencies. This is particularly concerning given the project's goal to **minimize resource waste**. Therefore, **5 replicas** strike the optimal balance, providing adequate performance while maintaining **resource efficiency**. This setup aligns well with the app's core objectives of **reliability** and **minimal resource overhead**, making it the preferred choice..

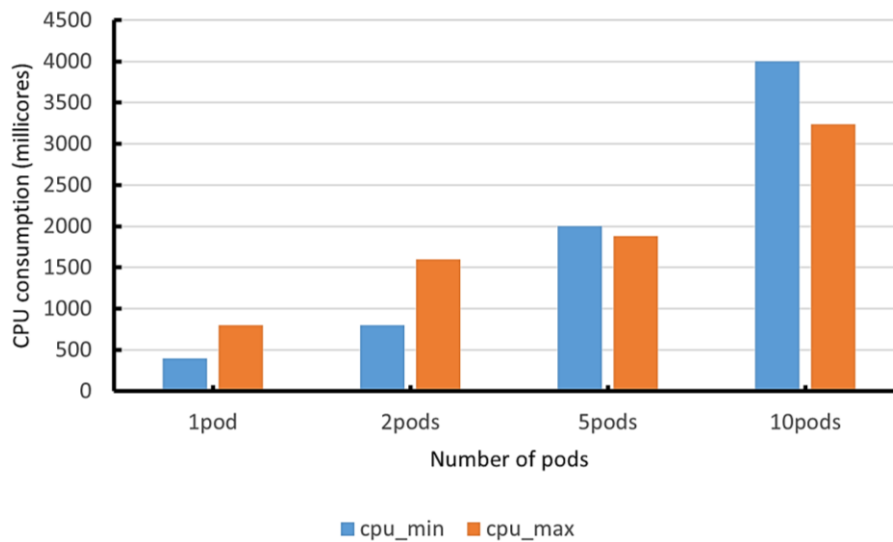


Figure 23. Total CPU consumption for pods - for 50000 requests

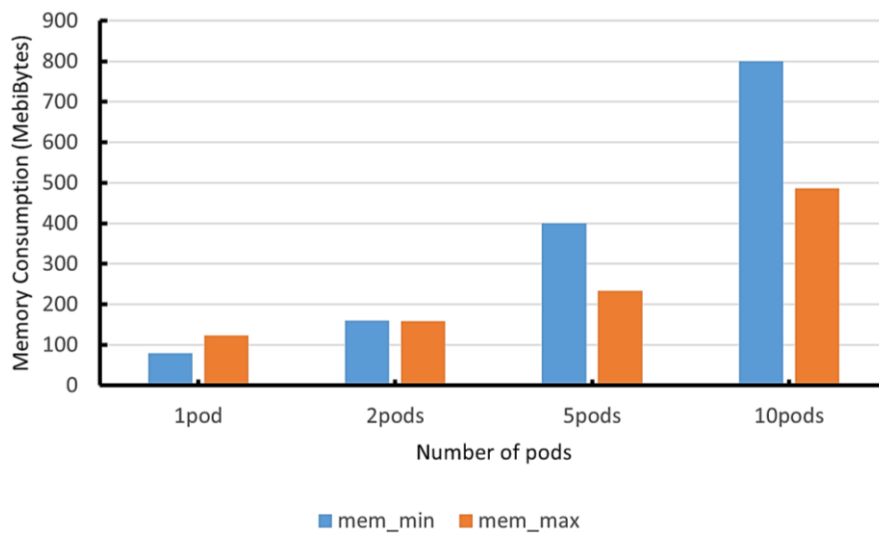


Figure 24. Total memory consumption for pods - for 50000 requests

To further optimize the system, we decided to implement a **horizontal autoscaler**. This autoscaler will dynamically calculate the **optimal number of replicas** based on **real-time load** and **resource usage**, ensuring the system maintains **high performance** and **efficiency** across varying conditions. We will detail this choice in the **dedicated section** later on, where we will discuss how the autoscaler adjusts the system's resources to meet the required **Quality of Service (QoS) objectives**.

## DEFINITION OF REQUESTS AND LIMITS VALUES

Based on various initial trial tests, we evaluated different CPU and memory usage scenarios before finalizing the values, which were later validated through the tests presented in the previous section.

### CPU

During our tests, we set the CPU **requests** for each pod to **400m**. This ensures that each pod can handle a significant workload from the start without immediately needing to scale up. This is crucial to avoid the **"thrashing" phenomenon**, a scenario where pods are created unnecessarily because the resource usage **quickly reaches the scaling threshold**, even for moderate loads. This can lead to an **unstable state** where new pods are constantly being created and terminated without effectively improving performance.



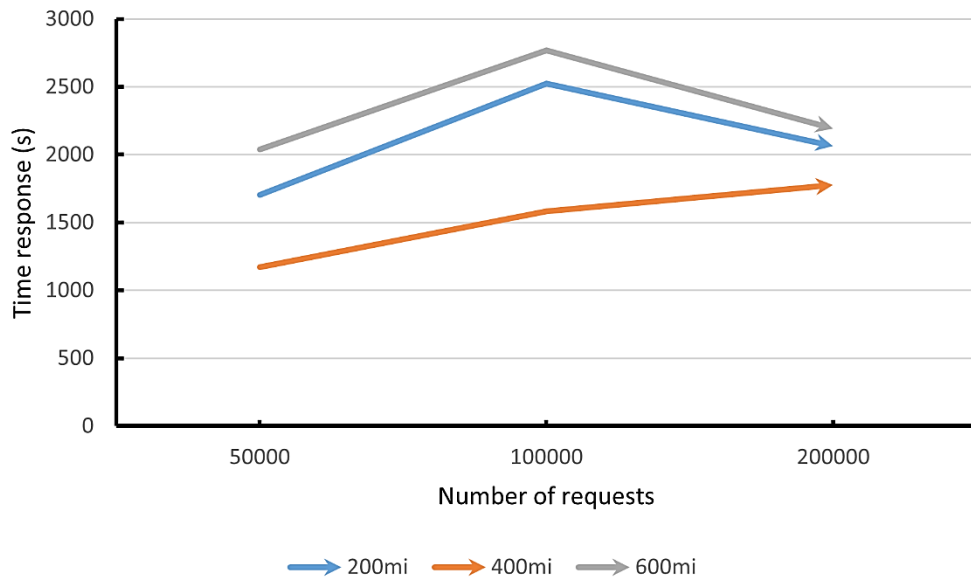


Figure 25. CPU request vs number of requests vs time response

- **Why not lower?** A lower request value would cause pods to struggle with even moderate traffic, quickly leading to resource saturation. This would trigger premature scaling, with new pods potentially failing to serve requests in a reasonable time due to propagation delays.
- **Why not higher?** A higher number of requests leads to a slower scaling process, which in turn causes the system to slow down. This is evident from the graph (Figure 26. CPU limit vs number of requests vs time response), where the response time increases as the request load rises.

The CPU **limit** was set to **800m**, which is twice the request value, to allow for temporary bursts in usage during peak traffic without affecting normal operations.

- **Why not lower?** A lower limit (e.g., 600m) would restrict the pod's ability to handle traffic spikes, potentially causing performance bottlenecks.
- **Why not higher?** A much higher limit (e.g., 1000m or more) could lead to resource contention on the node if multiple pods compete for CPU resources.

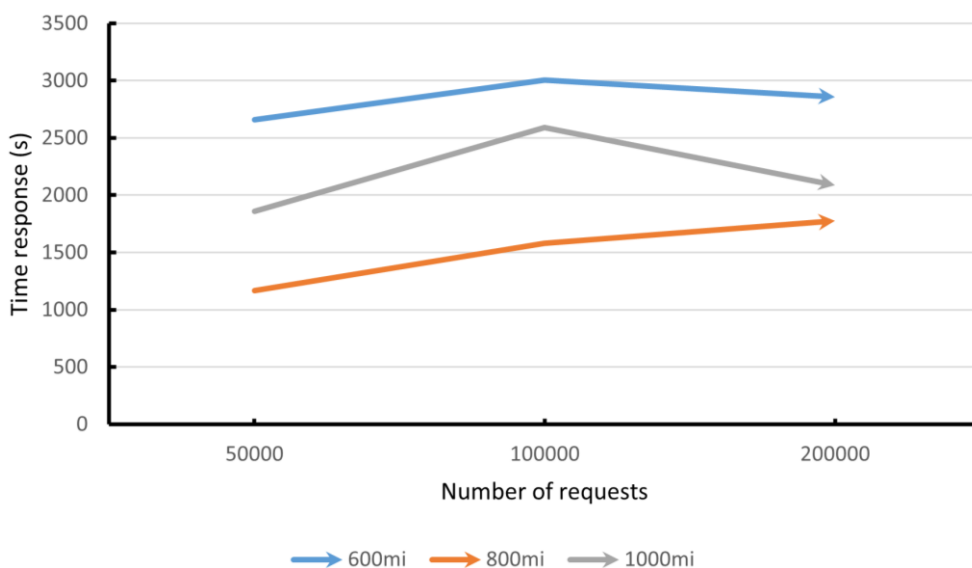


Figure 26. CPU limit vs number of requests vs time response

## MEMORY

Based on our tests, we observed that each pod consistently used on **50Mi** of memory under normal conditions (Figure 27. Mean memory utilization for each pods). We decided to set it up to **50Mi** to ensure sufficient memory allocation for loading the machine learning model and handling incoming requests.

- **Why not lower?** Allocating less than **50Mi** would have been counterproductive, as the pod would not have enough memory to operate effectively.
- **Why not higher?** Allocating more than **50Mi** would waste resources since the pod never utilized more than this value during standard operations.

For memory **limits**, we observed during a stress test with **2,000 simultaneous requests** that memory usage peaked at around **90Mi**. Based on this observation, we rounded the memory limit to **100Mi**, as it accommodates peak memory usage without over-allocating resources unnecessarily.

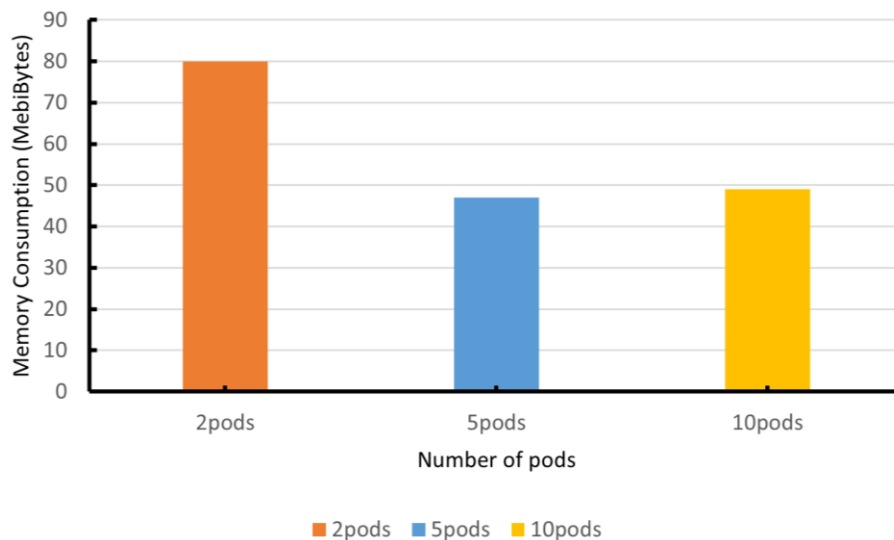


Figure 27. Mean memory utilization for each pods

## HORIZONTAL POD AUTOSCALER

In addition to aligning with the architecture we've designed for our application, the experiments conducted in the previous section have shown how **horizontal scaling**—by increasing the number of pods—can significantly improve overall performance. We decided to **test the addition of a Horizontal Pod Autoscaler (HPA)** to see if it could further improve them.

Tests revealed that **the HPA leads to better response times compared to the baseline case** with 2 pods, ensuring a more robust service. However, we observed lower performance compared to the cases with **5 and 10 static pods**. This could be due to the **HPA's overhead** in monitoring metrics and managing scaling, especially with **variable workloads**. We have ruled out the HPA percentage as a contributing factor to the performance gap, because our analysis indicates that **75% is the optimal HPA percentage** for achieving the **fastest response times**.

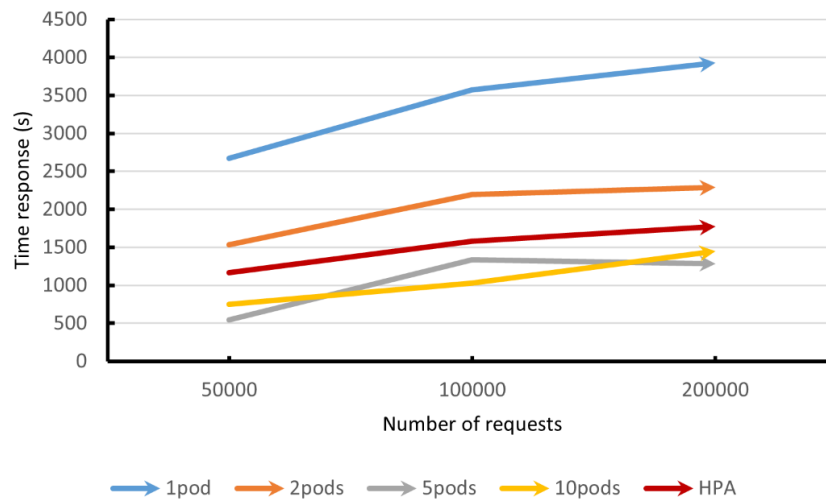


Figure 28. Static Pods/HPA vs number of request vs time response

Although one might expect the HPA to use fewer resources, its CPU consumption was only marginally lower than a static deployment of 10 pods. While it's tempting to focus on peak resource usage, the crucial point is the **minimum resource allocation**. Deploying 5 pods, for example, means reserving resources that often go **unused during periods of low demand**, leading to significant waste. With 5 pods, we might reserve **2 CPU cores even when they're not needed**.

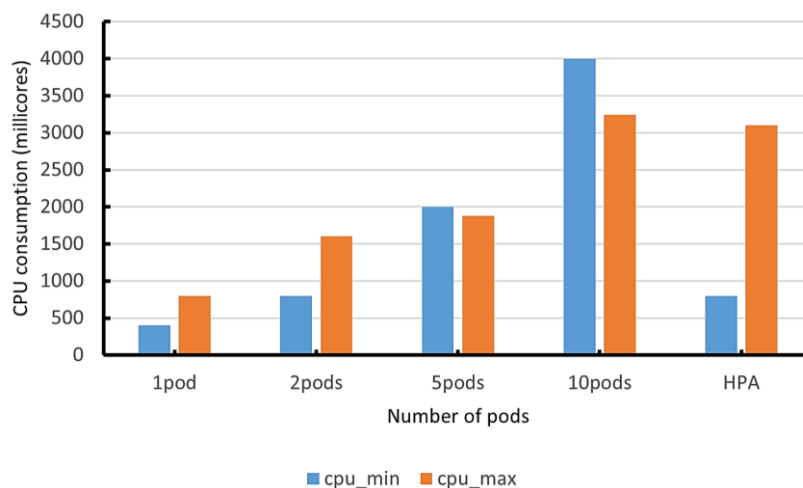


Figure 29. CPU consumed vs number of pods - for 50000 requests

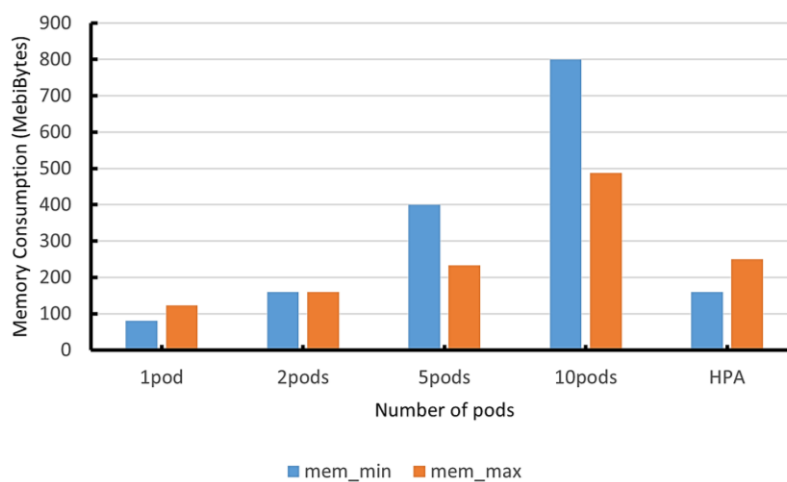


Figure 30. Memory consumed vs number of pods - for 50000 requests

With the HPA, the number of pods adjusts dynamically based on the **actual workload**. During **short-lived peaks**, such as a 10-second surge in demand, the system temporarily scales up, using more CPU to handle the load. When demand decreases, the HPA reduces the number of pods, conserving resources and consuming **less CPU** compared to a static configuration with 5 pods. This dynamic scaling allows the system to **optimize resource usage**, balancing **responsiveness** during peak periods with **efficiency** during low-demand periods.

Therefore, while the HPA might introduce a **slight increase in response time** compared to the 5 and 10 pod static scenarios, the **substantial reduction** in allocated resources is a **worthwhile trade-off**, provided it stays within our established performance thresholds. In our tests, the HPA remained well **below half of our acceptable performance** limit. For these reasons, it makes sense to integrate a **Horizontal Pod Autoscaler**, which uses Kubernetes' built-in metrics and scaling mechanisms to adjust the number of pods dynamically based on user demand.

Kubernetes implements this scaling through the **Metrics Server** we previously defined, which **continuously collects resource usage data**, such as CPU and memory, from the cluster. The autoscaler acts as a **automated scaling listener**, monitoring these metrics and comparing them to the **target thresholds** defined in the deployment configuration. If resource usage exceeds or falls below these thresholds, the autoscaler automatically **increases or decreases the number of pods** to maintain optimal performance while efficiently utilizing resources.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: weather-forecasting-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: weather-forecasting
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 75
```

*Codes 6. k8sweatherdeployment.yaml-hpa*

## MIN REPLICAS

We chose to start with **2 pods** for the Horizontal Pod Autoscaler. During testing, we observed that with only **1 pod**, the system was unable to meet the initially defined QoS objectives, likely because a single pod was **not sufficient to handle potential initial traffic spikes**. This was also expected, as the tests with a static number of pods showed that with just 1 pod, we were either too close to or even exceeding the response time thresholds. As the number of requests increased, the **response time decreased**, which confirmed that additional pods were needed to better handle the load.

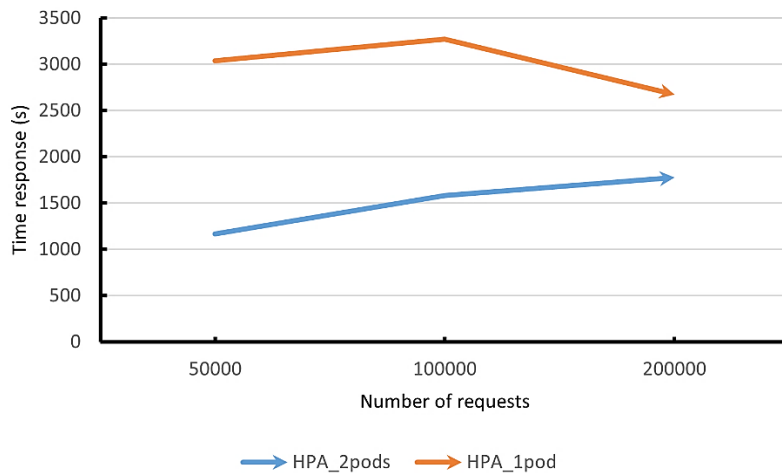


Figure 31. Number of HPA's min replicas vs number of requests vs time response

We also considered the fact that **scaling horizontally** by adding more pods leads to a higher **CPU consumption**, which is often the case for **CPU-bound workloads**. However, starting with more than 2 pods would have resulted in **excessive resource consumption**. The setup with 2 pods did use a certain amount of CPU, but it provided a reasonable **compromise**, ensuring a balance between **resilience** and **efficiency**. This approach minimized both the **economic and environmental impact** of the solution while still meeting the system's **performance requirements**.

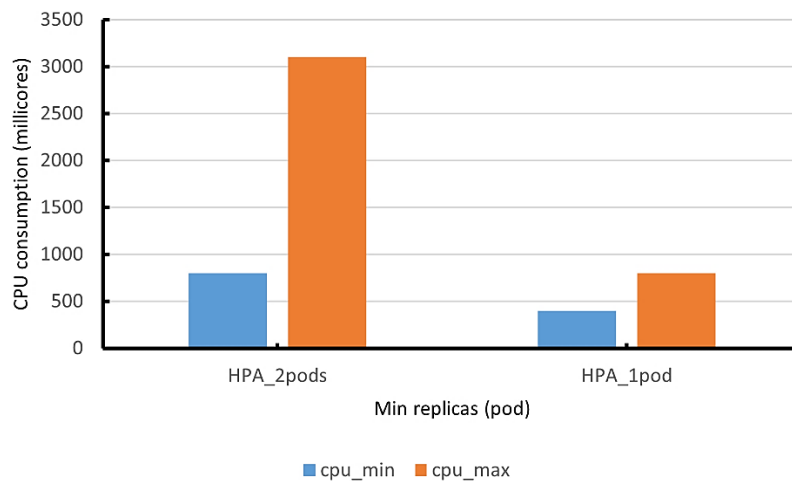


Figure 32. CPU consumed vs min replicas - for 50k requests

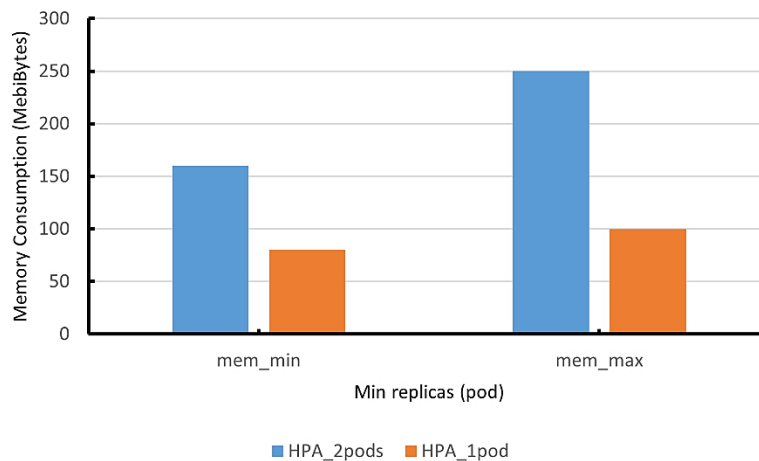


Figure 33. Memory consumed vs min replicas - for 50k requests

## MAX REPLICAS

The **maximum number of pods** was set to 10 because it **slightly exceeds the threshold** where pods begin to be **underutilized**, which is just over 5. This setup helps allocate **resources efficiently** while avoiding unnecessary **overhead**. Moreover, by setting the limit to 10, we avoid **resource waste**, as new pods are only created when the total **CPU usage** surpasses the predefined threshold, ensuring that additional pods are deployed only when **absolutely necessary**.

## PERCENTAGE UTILIZATION

The autoscaler was configured with a **CPU utilization threshold of 75%**.

- **Why not lower?** A lower threshold would trigger aggressive scaling, causing the system to create additional pods even for a small number of requests. This can lead to **thrashing**, where pods are continuously created and terminated, destabilizing the system and causing significant performance degradation due to the frequent resource reallocation.
- **Why not higher?** A higher threshold would fail to provide an adequate margin for CPU utilization to handle **traffic spikes** effectively. In this case the main issue is that there would be a significant drop in **performance**, with **response times** increasing considerably as the saturated resources would no longer be able to respond to users in a timely manner.

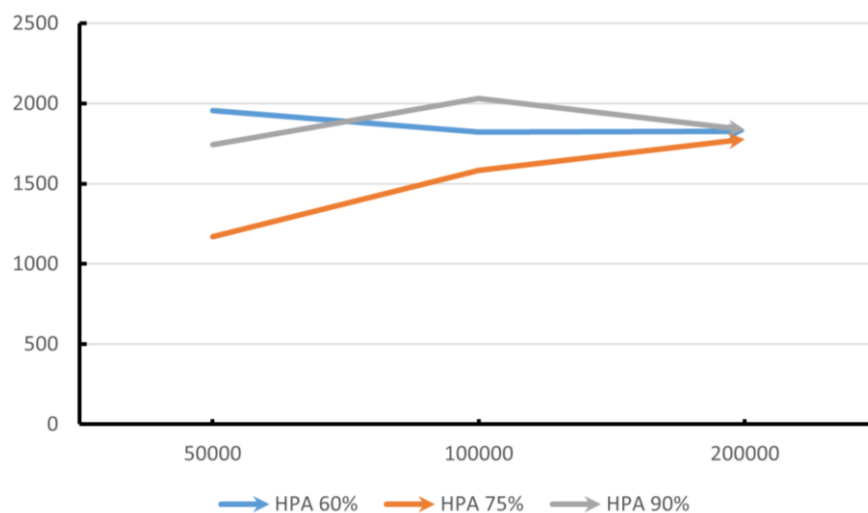


Figure 34. HPA percentage vs number of requests vs time response

## WHY WE DIDN'T USE A VPA?

We decided not to use the **Vertical Pod Autoscaler (VPA)** in conjunction with the **Horizontal Pod Autoscaler (HPA)** because both scale based on the same resource metrics, such as **CPU** and **memory usage**. When a metric (CPU or memory) reaches its defined threshold, both the VPA and HPA could trigger scaling events **simultaneously**. This overlap may result in **unknown side effects**, potentially **destabilizing the system** or leading to **resource allocation issues**. To avoid these complications, we opted for a configuration that solely relies on the **Horizontal Pod Autoscaler**, ensuring more **predictable** and **controlled scaling behavior**. (Kubecost, 2025)

## FINAL STRESS TEST

In the final test, we combined all the **chosen configurations** to verify that the final results met the **initial QoS specifications**. To do this, we conducted a **test with 50,000 requests**, obtaining the following results:

	Average	Error%	Throughput
50,000	1169	0.00%	2564.2/sec

Table 9. Final stress test results

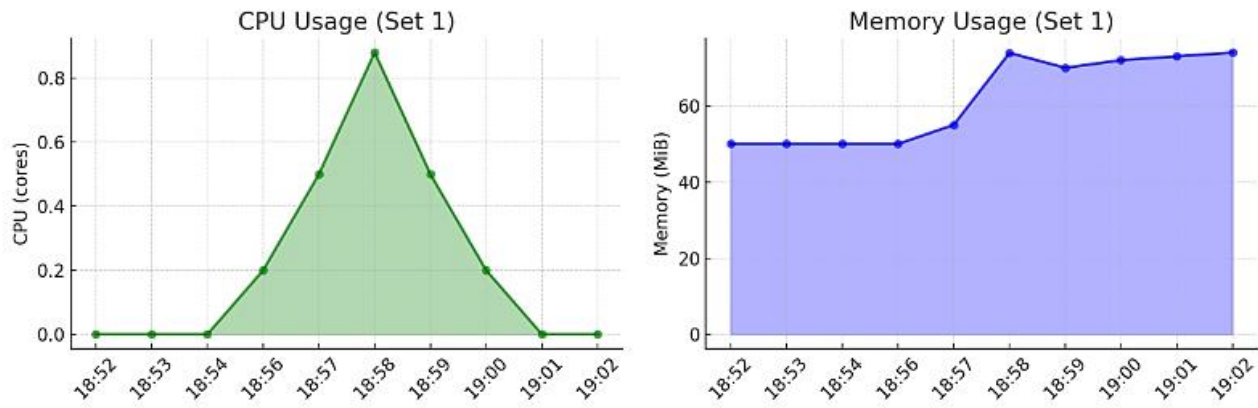


Figure 35. Final stress test CPU and Memory consumption for each pod

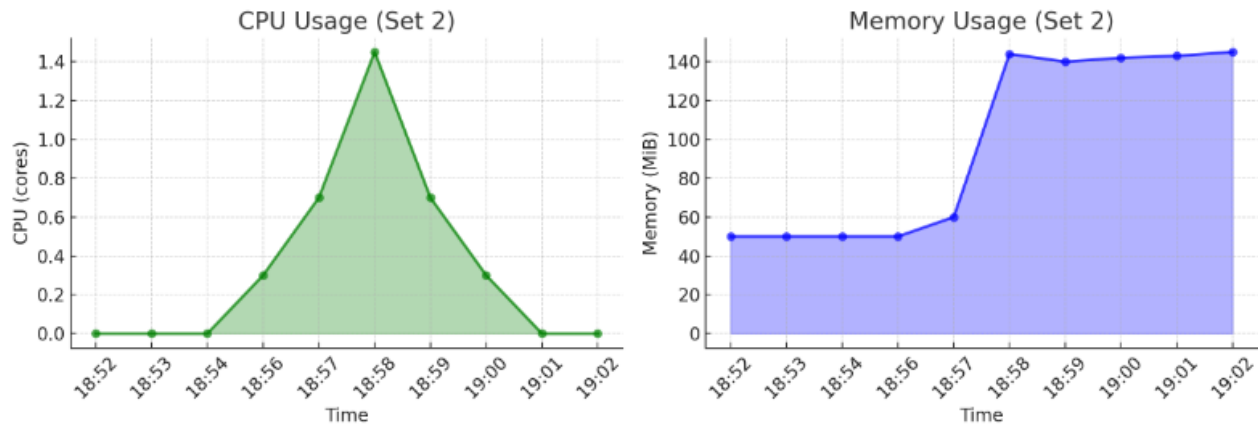


Figure 36. Final stress test CPU and Memory consumption for each node

The aggregate report showed an **average response time of 1169 ms**, which is within the SLA target of average 3 seconds per request. No errors were recorded, achieving **100% availability** and surpassing the SLA's minimum requirement of 99% reliability. With a **throughput of 2564.2 requests per second**, the system effectively handled high traffic loads without meaningful degradation.

These results confirm that the system **meets all SLA objectives**, demonstrating the ability to serve 50,000 customers and handle up to 2,000 simultaneous requests while maintaining good resource usage, reliability, and consistent performance under peak loads.

# SECURITY ASSESSMENT

---

In this section, we will **list the potential threats** to our web application and **implement an NGINX ingress controller** to mitigate those we consider the most pressing.

## THREAT IDENTIFICATION

The application is susceptible to various threats, including **Denial of Service (DoS) attacks**, **traffic eavesdropping**, **malicious intermediaries (Man-in-the-Middle)**. Here is an assessment of these threats:

### 1. Denial of Service (DoS)

- **Description:** a DoS attack overwhelms an application or service with excessive requests, rendering it unavailable by depleting critical resources such as CPU, memory, or bandwidth. This is particularly relevant for applications exposed to high traffic.
- **Mitigation:** approaches such as rate limiting, traffic shaping, and load balancing can effectively reduce the impact of this threat.

### 2. Traffic Eavesdropping

- **Description:** occurs when an attacker intercepts unencrypted network traffic to access sensitive information, such as user credentials or personal data.
- **Mitigation:** by enforcing the **HTTPS protocol**, traffic is encrypted, making it highly resilient to eavesdropping. Given that the application does not process sensitive data, this threat is considered low priority at this stage.

### 3. Malicious Intermediary (Man-in-the-Middle)

- **Description:** in this attack, an adversary intercepts or manipulates traffic between the client and the server to steal data or modify communications.
- **Mitigation:** the use of **HTTPS** effectively mitigates this threat by ensuring encrypted communication and verifying server authenticity.

## RISK EVALUATION

The most significant threat to this application is a **Denial of Service (DoS) attack**. While threats like **traffic eavesdropping** and **malicious intermediaries** are possible, they are **less concerning** in this context. The application **handles no sensitive data**; the worst-case scenario from such an attack would be an incorrect weather prediction, an outcome that could also occur naturally due to the inherent limitations of the prediction model. Furthermore, the lack of sensitive data makes the application a **less attractive target** for these kinds of attacks.

Although DoS attacks are **relatively common**, and the application could be targeted, it's not considered a high-profile target. However, a successful DoS attack could still **render the application unavailable**, disrupting service and preventing users from accessing weather information. Even though the data itself isn't highly sensitive, this unavailability remains a significant concern.



Threat	Impact (I)	Probability (P)	Risk Level (R = I x P)	Mitigation	Priority
Denial of Service (DoS)	High	Medium	High	Rate limiting, traffic shaping, load balancing, DDoS protection services.	High
Traffic Eavesdropping	Low	Low	Low	HTTPS	Low
Man-in-the-Middle (MitM)	Low	Low	Low	HTTPS	Low

Table 10. Risk evaluation

Therefore, our immediate focus is **mitigating the DoS threat**. We acknowledge that other potential risks exist and will need to be addressed as the application evolves and its functionality expands.

## INGRESS CONTROLLER

An **Ingress Controller** in Kubernetes acts as a layer that sits on top of these services (usually using **NodePort** or **LoadBalancer** under the hood) to provide more advanced routing, load balancing, and features like **SSL termination**, **HTTP routing** and virtual host support, enabling efficient and secure management of application traffic. By monitoring the status of pods and services, the Ingress Controller **dynamically updates its configuration** to ensure high availability and reliability of applications.

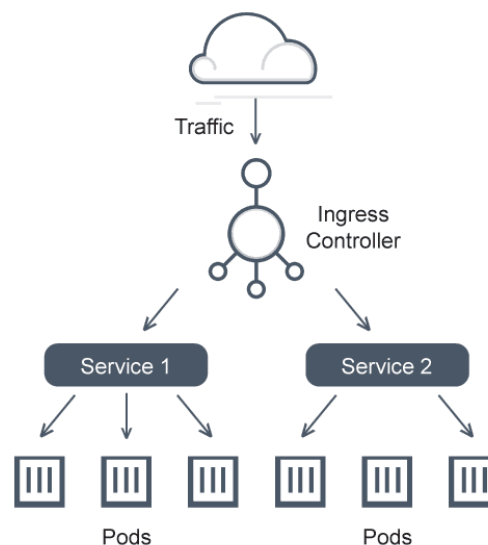


Figure 37. Ingress controller image

Implementing an Ingress Controller enhances security by **centralizing access control** and **reducing the attack surface**, as it serves as the single entry point for external traffic.

An Ingress Controller in Kubernetes also enhances the security of the application by mitigating various types of threats as it acts as the gateway for external traffic entering the cluster. Key examples include:

- **Denial of Service (DoS):**  
It reduces the risk of resource exhaustion by implementing **rate limiting**, setting **connection timeouts**, and enforcing **maximum payload sizes**. These measures help prevent the cluster from being overwhelmed by malicious or excessive requests.

- **Traffic Eavesdropping and Malicious Intermediaries:**

The NGINX Ingress Controller mitigates these threats by enforcing **HTTPS encryption**, ensuring secure communication between clients and the cluster.

While the NGINX Ingress Controller provides robust features for managing HTTP/HTTPS traffic and mitigating threats like DoS attacks, traffic eavesdropping, and malicious intermediaries, comprehensive protection requires integration with additional security solutions. These include **Web Application Firewalls (WAFs)**, **Content Delivery Networks (CDNs)**, **configuration management tools**, and **API security platforms**, which collectively enhance the system's ability to withstand more sophisticated and targeted attacks.

## NGINX APP PROTECT DDOS

**Distributed Denial of Service (DDoS)** attacks have been escalating in both **frequency and sophistication**. In 2022, application-layer attacks accounted for 78% of all DDoS incidents, marking a significant shift from previous years (StormWall, 2022).

These attacks, particularly those targeting the **application layer (Layer 7)**, are designed to overwhelm specific functions of a web application, rendering **services unavailable** to legitimate users.

To combat these sophisticated threats, solutions like **NGINX App Protect DoS** offer comprehensive protection. This tool provides **behavioral DoS detection and mitigation**, employing adaptive machine learning to identify and counteract malicious traffic patterns in real-time. By integrating seamlessly with *NGINX Plus* and *NGINX Ingress Controller*, it ensures robust defense mechanisms are in place to safeguard applications and APIs from Layer 7 DDoS attacks.

Implementing such advanced security measures is crucial in today's threat landscape, where even brief service interruptions can lead to **significant revenue loss, reputational damage, and increased vulnerability** to further attacks.

Since we do not have access to NGINX Plus, we are unable to implement the anti-DDoS solution directly. Instead, we will provide the configuration files and outline the steps required to deploy such a solution, allowing for its implementation when the necessary resources become available.

## CONFIGURATION FILES

Since our application primarily handles HTTP/HTTPS traffic, we have opted to use an **Ingress Controller** with a **ClusterIP** service. Unlike **LoadBalancer** or **NodePort**, which expose services externally, ClusterIP restricts access to within the cluster. The reason for this choice is that a **Load Balancer** requires additional tools for configuration and management and incurs extra costs associated with provisioning and maintaining its infrastructure. On the other hand, **NodePort** allows requests to be sent directly to the nodes, which introduces a security risk by exposing the cluster nodes to the outside world, increasing their vulnerability to potential attacks.

```

apiVersion: v1
kind: Service
metadata:
  name: weather-forecasting-app
  namespace: default
spec:
  type: ClusterIP
  selector:
    app: weather-forecasting-app
  ports:
    - port: 8501
      targetPort: 8501
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: weather-forecasting-ingress
  namespace: default
  annotations:
    appprotectdos.f5.com/app-protect-dos-resource: "default/dos-protected"
    nginx.ingress.kubernetes.io/rewrite-target: "/"
spec:
  ingressClassName: nginx
  rules:
    - host: localhost
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: weather-forecasting-app
                port:
                  number: 8501

```

*Codes 7. k8sweatherdeploymentconingress-ingress*

Continuing from the initial configuration of an ingress controller, the **NGINX App Protect DoS module** requires defining specific Kubernetes resources to strengthen application security. The configuration includes creating a **DosProtectedResource.yaml**, which acts as a central definition for securing multiple resources.

```

apiVersion: appprotectdos.f5.com/v1beta1
kind: DosProtectedResource
metadata:
  name: dos-protected
spec:
  enable: true
  name: "weather-forecasting"
  apDosMonitor:
    uri: 127.0.0.1:80
  apDosPolicy: "default/dospolicy"
  dosSecurityLog:
    enable: true
    apDosLogConf: "doslogconf"
    dosLogDest: "syslog-svc.default.svc.cluster.local:514"

```

*Codes 8. DosProtectedResource.yaml*

By integrating this into the ingress definition, the application gains protection against DoS attacks. Additionally, an **APDosPolicy.yaml** resource allows for precise customization of mitigation strategies, ensuring that the configuration aligns with the application's traffic patterns and resilience needs.

```
apiVersion: appprotectdos.f5.com/v1beta1
kind: APDosPolicy
metadata:
  name: dospolicy
spec:
  mitigation_mode: "standard"
  signatures: "on"
  bad_actors: "on"
  automation_tools_detection: "on"
  tls_fingerprint: "on"
```

*Codes 9. APDosPolicy.yaml*

To monitor the system, an **APDosLogConf.yaml** is introduced, enabling detailed event logging.

```
apiVersion: appprotectdos.f5.com/v1beta1
kind: APDosLogConf
metadata:
  name: doslogconf
spec:
  filter:
    traffic-mitigation-stats: all
    bad-actors: top 10
    attack-signatures: top 10
```

*Codes 10. APDosLogConf.yaml*

This logging resource is referenced within the DosProtectedResource.yaml, ensuring consistent tracking of events. With all these components implemented, the ingress is configured to securely expose the application, leveraging the defined defenses and monitoring capabilities to **maintain stability and resist potential DoS attacks effectively**.

# CONCLUSIONS

---

In this project, we **successfully developed and deployed a weather forecasting application** leveraging Kubernetes to ensure scalability, resilience, and efficient resource utilization. From designing the model to deploying the application in a containerized environment, every step was carefully planned and implemented to meet the **SLA requirements** we defined at the beginning of this paper. The system has proven capable of handling 50,000 requests per day with up to 2,000 simultaneous requests, maintaining reliable performance and adhering to the defined QoS objectives.

## FUTURE IMPROVEMENT

The current system has proven to be robust and effective, meeting its **SLA-defined requirements** while offering **scalability** and **reliability**. However, there are several opportunities for future improvement, such as:

- **Leverage internal cloud resources** by incorporating a database to store historical weather data and user inputs. This would allow for continuous retraining of the machine learning model, enhancing prediction accuracy and enabling advanced analytics.
- **Enhance redundancy and failover capabilities** by increasing the number of replicas and distributing them across multiple nodes. This would reduce downtime risks and improve system resilience.
- **Strengthen security** by integrating tools such as Web Application Firewalls (WAFs), API gateways, and enhanced anomaly detection. These measures would protect the system from evolving threats and ensure robust application defense.
- **Optimize the machine learning lifecycle** by integrating Kubeflow, enabling direct model training within the containerized environment. This would automate the process of model updates and simplify deployment, ensuring the system remains adaptive to new data.
- **Expand the system architecture** by adding new nodes and incorporating a load balancer. While the current setup operates on a single-node architecture without a load balancer, a multi-node deployment with a load balancer would utilize an architecture that serves as a hybrid between **workload distribution** and **dynamic scalability**.

## BIBLIOGRAPHY

- Dybas, & Hosansky. (2015, January 23). *National Science Foundation*. Tratto da [www.nsf.gov](http://www.nsf.gov): [www.nsf.gov/news/news\\_summ.jsp](http://www.nsf.gov/news/news_summ.jsp)
- Kubecost. (2025, 01). *Kubernetes autoscaling*. Tratto da Kubecost: <https://www.kubecost.com/kubernetes-autoscaling/kubernetes-vpa/>
- StormWall. (2022, February 3). *DDoS Attacks Report for 2022*. Tratto da StormWall: <https://stormwall.network/resources/blog/ddos-report-2022>

## FIGURES INDEX

Figure 1. Output classes' distributions .....	9
Figure 2. Dataset's columns KDE curves .....	10
Figure 3. Precipitation and wind boxplots .....	11
Figure 4. (a) Minimum and maximum temperature scatter plot (b) Minimum and precipitation scatter plot .....	11
Figure 5. Correlation Matrix without date decomposition .....	12
Figure 6. Correlation matrix with date decomposition .....	14
Figure 7. XGBoost model's AUC curve for each class .....	15
Figure 8. Pipeline's column transformer .....	16
Figure 9. Pipeline .....	16
Figure 10. Discrepancy between predictions and real results .....	18
Figure 11. GUI's mockup .....	19
Figure 12. Final GUI .....	22
Figure 13. Docker Scout final results .....	25
Figure 14. Docker scout final results - fixable vulnerabilities .....	26
Figure 15. Dynamic scalability architecture .....	28
Figure 16. Workload distribution architecture .....	29
Figure 17. Number of nodes vs number of requests (related to next chapter's tests results) .....	31
Figure 18. Cluster's initial configuration scheme .....	33
Figure 19. Port mapping scheme .....	33
Figure 20. Kubernetes dashboard .....	35
Figure 21. Final home page view after clusterization .....	36
Figure 22. Number of pods vs number of requests vs response time .....	39
Figure 23. Total CPU consumption for pods - for 50000 requests .....	40
Figure 24. Total memory consumption for pods - for 50000 requests .....	40
Figure 25. CPU request vs number of requests vs time response .....	41
Figure 26. CPU limit vs number of requests vs time response .....	41
Figure 27. Mean memory utilization for each pods .....	42
Figure 28. Static Pods/HPA vs number of request vs time response .....	43
Figure 29. CPU consumed vs number of pods - for 50000 requests .....	43
Figure 30. Memory consumed vs number of pods - for 50000 requests .....	43
Figure 31. Number of HPA's min replicas vs number of requests vs time response .....	45
Figure 32. CPU consumed vs min replicas - for 50k requests .....	45
Figure 33. Memory consumed vs min replicas - for 50k requests .....	45
Figure 34. HPA percentage vs number of requests vs time response .....	46
Figure 35. Final stress test CPU and Memory consumption for each pod .....	47

Figure 36. Final stress test CPU and Memory consumption for each node .....	47
Figure 37. Ingress controller image .....	49

## TABLES INDEX

Table 1. Dataset's columns datatypes .....	8
Table 2. Dataset's statistic summary .....	9
Table 3. Classification report on test set .....	17
Table 4. Predictions summary for each class on test set .....	17
Table 5. Stress test results - 1 pod .....	38
Table 6. Stress test results - 2 pods .....	38
Table 7. Stress test results - 5 pods .....	38
Table 8. Stress test results - 10 pods.....	39
Table 9. Final stress test results .....	47
Table 10. Risk evaluation .....	49

## CODES INDEX

Codes 1. Dockerfile .....	20
Codes 2. requirements.txt.....	21
Codes 3. weather-app-config-with-port-mapping.yaml .....	31
Codes 4. k8sweatherdeployment.yaml-deployment .....	32
Codes 5. k8sweatherdeployment.yaml-service.....	32
Codes 6. k8sweatherdeployment.yaml-hpa .....	44
Codes 7. k8sweatherdeploymentconingress-ingress .....	51
Codes 8. DosProtectedResource.yaml .....	51
Codes 9. APDosPolicy.yaml .....	52
Codes 10. APDosLogConf.yaml.....	52