UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Ingegneria dell'Informazione ed Elettrica e Matematica Applicata

ANNO ACCADEMICO 2024/25

Prof. A. Marcelli & A. Della Cioppa

Secure Cloud Computing

Project Work Report

Nome	Cognome	Matricola	E-mail
Emanuele	Relmi	0622	e.relmi@studenti.unisa.it
Francesco	Quagliuolo	0622	f.quagliuolo@studenti.unisa.it

Introduction and Project Overview

In recent years, cloud computing has emerged as a transformative force in the digital landscape, revolutionizing how businesses and individuals access, store, and manage data. Its primary advantage lies in offering scalable, on-demand computing resources that eliminate the need for costly on-premise infrastructure. Cloud services enable companies to deploy applications rapidly, ensuring high availability, flexibility, and cost efficiency. This has led to widespread adoption across various industries, from finance and healthcare to artificial intelligence and big data analytics.

The growing reliance on cloud platforms is driven by key factors such as remote accessibility, automatic updates, and seamless integration with modern development frameworks. Moreover, cloud environments support powerful computing capabilities, facilitating the deployment of complex machine learning models and high-performance applications. The ability to scale infrastructure dynamically in response to user demand further enhances the appeal of cloud computing.

However, despite its numerous benefits, cloud computing is not without challenges. Security remains a major concern, as data stored in the cloud is vulnerable to cyber threats, unauthorized access, and compliance risks. Dependence on third-party cloud providers also raises concerns regarding data sovereignty, potential service outages, and vendor lock-in. Furthermore, managing cloud costs efficiently can be complex, as organizations must optimize resource allocation to prevent unnecessary expenditures. Addressing these challenges requires robust security measures, proactive risk management, and a strategic approach to cloud adoption to ensure reliability, efficiency, and long-term sustainability.

The aim of this project is to design, train, and deploy a machine learning model for Dogecoin (cryptocurrency) forecasting, based on collected data and some other inputs given by the user, within a scalable cloud-based infrastructure, and to analyse the containerization process and the deployment on a Kubernetes cluster. Major focus will be put on security measures and best practises to reduce the possibility to run into vulnerabilities, so to protect the application in the production environment.

Project Goals

As said before, the project objectives are about security techniques to ensure a protection while containerizing the application and deploying it on K8s cluster.

To achieve this, the following tasks will be performed:

1. **Training an ML model offline and saving the final model for inference**
2. **Developing a web application capable of performing real-time predictions on new data**
3. **Containerizing the application using Docker and ensuring security by analysing vulnerabilities**. This step involves:
    a. Building a Docker image for the Dogecoin forecasting web-app, following security best practices such as using trusted base images, reducing unnecessary layers, and applying the principle of least privilege to limit exposure to threats.
    b. Utilizing security scanning tools (*Docker Scout*) to detect vulnerabilities within the containerized application.
    c. Applying security patches and mitigation techniques, including dependency updates and proactive security enhancements.
4. **Setting up a Kubernetes cluster**
    a. Deploy a Kubernetes cluster with at least one control plane node and multiple worker nodes to manage the application workload.
    b. Securely configure and manage essential Kubernetes components such as Deployments, Services, Ingress controllers, ConfigMaps.

5. **Implementing Application and Cluster Security**
    a. Adopt security recommendations from industry guidelines and relevant webinars (such as "*Protecting Apps from Hacks in Kubernetes with NGINX*"), implementing features like rate limiting, Web Application Firewalls (*WAF*s), and fine-tuned network access controls.
    b. Enforce Kubernetes-native security policies, including Role-Based Access Control (*RBAC*), Pod Security Policies and network segmentation to safeguard the cluster from potential threats.
6. **Conducting stress tests** to evaluate deployment performance, including horizontal and vertical scaling, load balancing, and fault tolerance.

By the end of the project, a fully functional and secure machine learning web application will be deployed on a robust cloud infrastructure, demonstrating best practices in cloud-native application development and deployment.

Application Description

The developed application is a Dogecoin forecasting application, which, given some inputs (as opening, high and low prices, volume of exchange and a date) by the user, can predict the closing price of the cryptocurrency for the specified date.

Nonetheless the ML model is not the focus for this project, its structure will be outlined below:

- **Inputs**:
    - High price
    - Opening price
    - Low price
    - Volume of exchange
    - Date
- **Output**:
    - Closing price

The model was trained offline and saved in a format suitable for inference (a .joblib file for model and transformer). From a DevOps and security standpoint, this project focuses on integrating the forecasting model into a web microservice, containerizing it, and ensuring its scalability and security within a Kubernetes environment.

Cryptocurrency forecasting plays a significant role in cybersecurity, as blockchain networks and digital assets are frequent targets of fraud, market manipulation, and cyber threats. Predicting Dogecoin price trends can help detect unusual patterns that may indicate potential risks, such as pump-and-dump schemes or suspicious trading activity.

However, the primary reason for selecting a Machine Learning-based web app for Dogecoin forecasting in this project is to create a scalable use case that can be deployed, tested under high traffic conditions (up to 50,000 users), and stress-tested in a Kubernetes environment. This ensures a comprehensive evaluation of the system's resilience, security, and performance when handling large-scale requests.

1. Model Training

1.1. Introduction

The code in the *Jupyter Notebook*, which can be found in ***./notebook/dogecoin-forecasting-rf-pipeline.ipynb***, implements a machine learning system for forecasting Dogecoin prices, specifically focusing on predicting the closing price based on historical market data. The approach involves data preprocessing, feature engineering, and a machine learning pipeline built around the **Random Forest Regressor**, an ensemble learning method known for its robustness in handling complex datasets.

1.1.1. Random Forest for Time Series Forecasting

The **Random Forest Regressor** is an ensemble learning method that constructs multiple decision trees during training and averages their outputs to make predictions. It is particularly useful in time series forecasting because:

- It **handles non-linearity well**, making it effective for financial markets where price movements do not follow a simple trend.
- It is **resistant to overfitting**, as it averages multiple decision trees rather than relying on a single one.
- It can **capture complex interactions** between multiple market indicators, improving predictive performance.

1.1.2. Pipeline for Efficient Model Deployment

To streamline data transformation and model training, the project uses an ML **pipeline**. The pipeline consists of several preprocessing steps, including:

- **Feature Scaling**: Standardizing numerical variables to ensure consistency across different price ranges.
- **Imputation**: Filling missing values to prevent gaps in training data.
- **Feature Engineering**: Extracting meaningful predictors from raw data, such as rolling averages and volatility indicators.

There are a lot of advantages of using a pipeline, such as:

- **Automates preprocessing**, ensuring consistent transformations across training and testing datasets.
- **Improves reproducibility**, allowing easy deployment and retraining with updated data.
- **Facilitates hyperparameter tuning**, enabling adjustments at multiple stages in a structured way.

1.1.3. Reasons behind this approach for Dogecoin Price Prediction

The Random Forest model combined with a structured pipeline is particularly useful for forecasting Dogecoin closing prices due to:

- **Feature-rich data representation**: The pipeline ensures the inclusion of multiple predictive indicators, from historical prices to market trends.
- **Robust performance**: Random Forest can handle noisy and non-linear financial data, making it suitable for cryptocurrency markets.
- **Scalability**: The model can be retrained efficiently with new data, enabling continuous adaptation to market changes.

By leveraging ensemble learning and an automated preprocessing pipeline, this project aims to build a scalable, accurate, and efficient forecasting system for Dogecoin price movements. The following sections provide a detailed breakdown of the implemented functions and methodologies.

1.2. Code Analysis

1.2.1. Loading CSV file and Data Preprocessing

```python
# Load the data from the CSV file
doge_data = pd.read_csv('../data/dogecoin_data.csv')

# Ensure the 'Date' column is present and set it as the index
doge_data.reset_index(inplace=True)

# Feature engineering: create additional features
doge_data['Date'] = pd.to_datetime(doge_data['Date'])
doge_data['Day'] = doge_data['Date'].dt.day
doge_data['Month'] = doge_data['Date'].dt.month
doge_data['Year'] = doge_data['Date'].dt.year
doge_data['DayOfWeek'] = doge_data['Date'].dt.dayofweek
doge_data.set_index('Date', inplace=True)
doge_data.drop(columns=['index'], inplace=True)
doge_data.ffill(inplace=True)

# Verify that columns are unique
if doge_data.columns.duplicated().any():
    raise ValueError(f"Duplicate columns found: {doge_data.columns[doge_data.columns.duplicated()]}")

# Print the first few rows of the DataFrame to verify the data
print(doge_data.head())
```

Figure 1 - Loading CSV file and Data Preprocessing

The first function loads a dataset from the CSV file `dogecoin_data.csv`. Then, it is ensured that the '*Date*' column is set as the index and if it is converted into a datetime format. Done that, there is the creation of additional time-based features (such as *Day*, *Month*, *Year*, and *Day of the Week)* to capture periodic trends date. After that, a check about columns uniqueness is done and finally, to have a look at the table, a printing of the head is made, which returns the following output:

Date	Close	High	Low	Open	Volume	Day	Month	\
2020-01-01	0.001812	0.001829	0.001802	0.001808	45619467	1	1	
2020-01-02	0.001798	0.001889	0.001775	0.001813	58247425	2	1	
2020-01-03	0.001922	0.001951	0.001782	0.001797	56113646	3	1	
2020-01-04	0.002008	0.002231	0.001837	0.001921	84437147	4	1	
2020-01-05	0.002168	0.002231	0.001897	0.002007	47158934	5	1	

Date	Year	DayOfWeek
2020-01-01	2020	2
2020-01-02	2020	3
2020-01-03	2020	4
2020-01-04	2020	5
2020-01-05	2020	6

Figure 2 - Head of dogecoin_data

1.2.2. Transformer and Pipeline definitions

```python
# Define the columns transformer
transformer = ColumnTransformer([
    ('date', Pipeline([
        ('imputer', SimpleImputer(missing_values=np.nan, strategy='median')),
        ('scaler', StandardScaler())
    ]), make_column_selector(pattern='date')),
    ('high', Pipeline([
        ('imputer', SimpleImputer(missing_values=np.nan, strategy='median')),
        ('scaler', StandardScaler())
    ]), make_column_selector(pattern='high')),
    ('low', Pipeline([
        ('imputer', SimpleImputer(missing_values=np.nan, strategy='median')),
        ('scaler', StandardScaler())
    ]), make_column_selector(pattern='low')),
    ('open', Pipeline([
        ('imputer', SimpleImputer(missing_values=np.nan, strategy='median')),
        ('scaler', StandardScaler())
    ]), make_column_selector(pattern='open')),
    ('volume', Pipeline([
        ('imputer', SimpleImputer(missing_values=np.nan, strategy='median')),
        ('scaler', StandardScaler())
    ]), make_column_selector(pattern='volume')),
    ('day', Pipeline([
        ('imputer', SimpleImputer(missing_values=np.nan, strategy='median')),
        ('scaler', StandardScaler())
    ]), make_column_selector(pattern='day')),
    ('month', Pipeline([
        ('imputer', SimpleImputer(missing_values=np.nan, strategy='median')),
        ('scaler', StandardScaler())
    ]), make_column_selector(pattern='month')),
    ('year', Pipeline([
        ('imputer', SimpleImputer(missing_values=np.nan, strategy='median')),
        ('scaler', StandardScaler())
    ]), make_column_selector(pattern='year')),
    ('dayofweek', Pipeline([
        ('imputer', SimpleImputer(missing_values=np.nan, strategy='median')),
        ('scaler', StandardScaler())
    ]), make_column_selector(pattern='dayofweek')),
    ('cat', Pipeline([
        ('imputer', SimpleImputer(strategy='most_frequent')),
        ('encoder', OneHotEncoder(categories='auto', drop='first', handle_unknown='ignore'))
    ]), make_column_selector(dtype_include='category'))
], remainder='passthrough', verbose_feature_names_out=True, sparse_threshold=0)

# Define the steps for the pipeline
steps = [
    ('transformer', transformer),
    ('regressor', RandomForestRegressor())
]

# Define the RandomForest pipeline
rf_pipeline = Pipeline(steps=steps)
```

Figure 3 – Transformer and Model definitions

This step involves, in the first section, the implementation of a *ColumnTransformer* to preprocess numerical and categorical features efficiently. It includes:

- **Numerical transformations**: Standardization using *StandardScaler* and imputation of missing values with *SimpleImputer*.
- **Categorical transformations**: Encoding categorical variables (if any) with *OneHotEncoder*.

The second section revolves around the building of a *Pipeline* to automate these preprocessing steps and integrate them with the model training process.

1.2.3. Splitting Data

```
X = doge_data.drop(columns=['Close'], axis=1)
y = doge_data['Close']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Figure 4 – Train-test splitting

In this section, the first thing done is a split of the dataset into training (80%) and testing (20%) subsets. This ensures the test set is not used during training to prevent data leakage. The closing price of Dogecoin is defined as the target variable (y), while the other features are grouped as the dependent variable (x).

1.2.4. Hyperparameters Tuning

```
# Define the seeds for reproducibility
cl_seeds = (42, 123, 643, 1337, 543, 1, 99, 885, 23, 77)

rf_param_grid = {
    'regressor__random_state': cl_seeds,
    'regressor__n_estimators': [200, 300, 400],
    'regressor__max_depth': [None, 2, 5, 10, 20, 50],
    'regressor__max_features': ['sqrt'],
    'regressor__bootstrap': [True],
}

model_list = [('rf', rf_pipeline)]
model_params_grid = [rf_param_grid]
```

Figure 5 - Grid search optimization

Here, ***GridSearchCV*** is used to find the optimal hyperparameters for the Random Forest Regressor. It tests different values for:

- **n_estimators:** number of trees in the forest.
- **max_depth:** maximum depth of each tree.
- **min_samples_split** and **min_samples_leaf:** to control tree splitting.

This method makes use of cross-validation to assess model generalization.

1.2.5. Best model

```python
# Get the best model from grid search
best_models_gs = []
for (model_name, model), hp in zip(model_list, model_params_grid):
    grid = GridSearchCV(model, hp, cv=3, scoring='neg_mean_squared_error', n_jobs=-1)
    grid.fit(X_train, y_train)
    print(f"Model: {model_name}")
    print(f"Best params: {grid.best_params_}, Best score: {grid.best_score_}")
    best_models_gs.append((model_name, grid.best_estimator_, grid.best_score_, grid.best_params_))

hof_model_gs = max(best_models_gs, key=lambda item:item[2])
print(f"HoF Model: {hof_model_gs}")
```

Figure 6 - Selecting the best model

This snippet of code has the task of retrieving the best-performing Random Forest model found during hyperparameter tuning. It prints the best parameter values and evaluates initial model performance.

1.2.6. Model Evaluation

```python
# Best model prediction and scores
hof_gs = hof_model_gs[1]
y_pred_train = hof_gs.predict(X_train)
y_pred_test = hof_gs.predict(X_test)
print(f'Mean Squared Error on training set: {mean_squared_error(y_train, y_pred_train)}')
print(f'Mean Squared Error on test set: {mean_squared_error(y_test, y_pred_test)}')
```

```
[7]
Mean Squared Error on training set: 8.347484636243886e-06
Mean Squared Error on test set: 2.9353643991832407e-05
```

Figure 7 - Model evaluation

Here is done the computing of key performance metrics, as:

- **Mean Squared Error (*MSE*)**, which measures overall prediction error.
- **Root Mean Squared Error (*RMSE*)**, which evaluates the average magnitude of prediction errors.

1.2.7. Saving Model

```
# Re-fit the model on the entire dataset
y = doge_data['Close']
x = doge_data.drop('Close', axis = 1)
columns = x.columns

scaler = StandardScaler()
scaler = scaler.fit(x)
X = scaler.transform(x)
features = pd.DataFrame(X, columns = columns)

# Save the transformer and model
dump(transformer, '../models/doge_transformer.joblib')
dump(hof_gs, '../models/doge_model.joblib')
```

Figure 8 - Saving the trained model

This section makes a refitting of the model and then it saves the trained Random Forest model using *joblib* for later use to ensure that the model can be loaded and used for predictions without retraining.

1.2.8. Inference

```
# Load new data for prediction
high = 0.06
low = 0.04
open = 0.05
volume = 1000000
day = 1
month = 1
year = 2023
dayofweek = 0
feat_cols = features.columns

row = [high, low, open, volume, day, month, year, dayofweek]

# Load the transformer and model
transformer = load('../models/doge_transformer.joblib')
model = load('../models/doge_model.joblib')

# Check the feature columns
print(feat_cols)

# Transform the new data
df = pd.DataFrame([row], columns = feat_cols)
tr = transformer.fit(df)
X = tr.transform(df)
features = pd.DataFrame(X, columns = feat_cols)

# Making a prediction
try:
    prediction = model.predict(features)
    print(f'Predicted Close Price: {prediction[0]}')
except Exception as e:
    print(f'Error during prediction: {e}')
```

Figure 9 - Making predictions

Finally, the saved model is loaded, and it is applied to new data. This part uses the trained model to forecast Dogecoin's future closing price based on unseen data to demonstrate how the model can be used in a real-time environment for price prediction.

2. Web App

The web app developed for this project is designed to predict Dogecoin's closing price based on inputs (opening, high and low prices, volume of exchange and a date) entered by the user, leveraging machine learning algorithms. The app can be used via an intuitive user interface (UI) built using **Streamlit**, enabling users to input data and receive real-time predictions. This chapter outlines the design, features, and implementation details of the web application.

2.1. Technology Used

The Dogecoin price prediction application leverages a combination of modern technologies to ensure interactivity, efficiency, and scalability.

The frontend is built using **Streamlit**, a lightweight Python framework that allows for the rapid development of interactive web applications.

The backend is implemented in **Python**, integrating a pre-trained machine learning model stored and loaded using *Joblib*. The model, along with a feature transformer, processes user inputs such as historical price data and trading volume to generate forecasts.

To enhance usability, the application incorporates **PIL** (**Pillow**) for displaying an image within the *Streamlit* interface.

Docker is used to containerize the application, making deployment more manageable.

For the deployment in a production environment, **Kubernetes** is used because it can provide scalability and fault tolerance, ensuring the system remains responsive even under high traffic loads.

This combination of technologies enables real-time inference, allowing users to predict Dogecoin's closing price dynamically based on input data.

2.2. Functionalities

The Dogecoin price prediction web app allows users to input market data such as the high, low, open price, and trading volume, along with a specific date. Once the data is entered, users can click the "P*redict Dogecoin Price*" button to receive a predicted closing price for Dogecoin, calculated by the ML model. The predicted price is displayed on the same page, and if any errors occur during the process, an informative message is shown to guide the user.

So, to resume in few steps what is done:

- **Inputs**: user enter data via an interactive interface with fields for high, low, open prices and the date for which the prediction is wanted.
- **Output**: the predicted closing price of Dogecoin is shown above all, after the model performs the calculation. If an error occurs, a message of explanation is shown to the user.

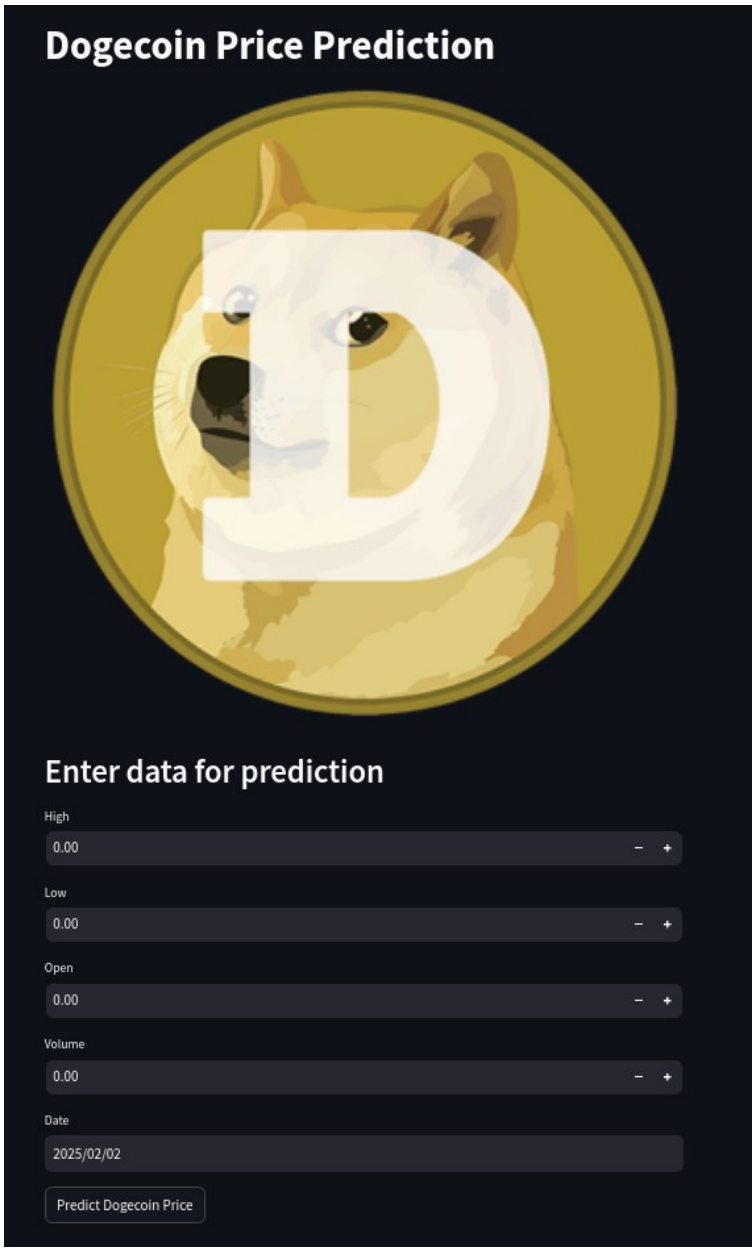


Figure 10 - Web App UI

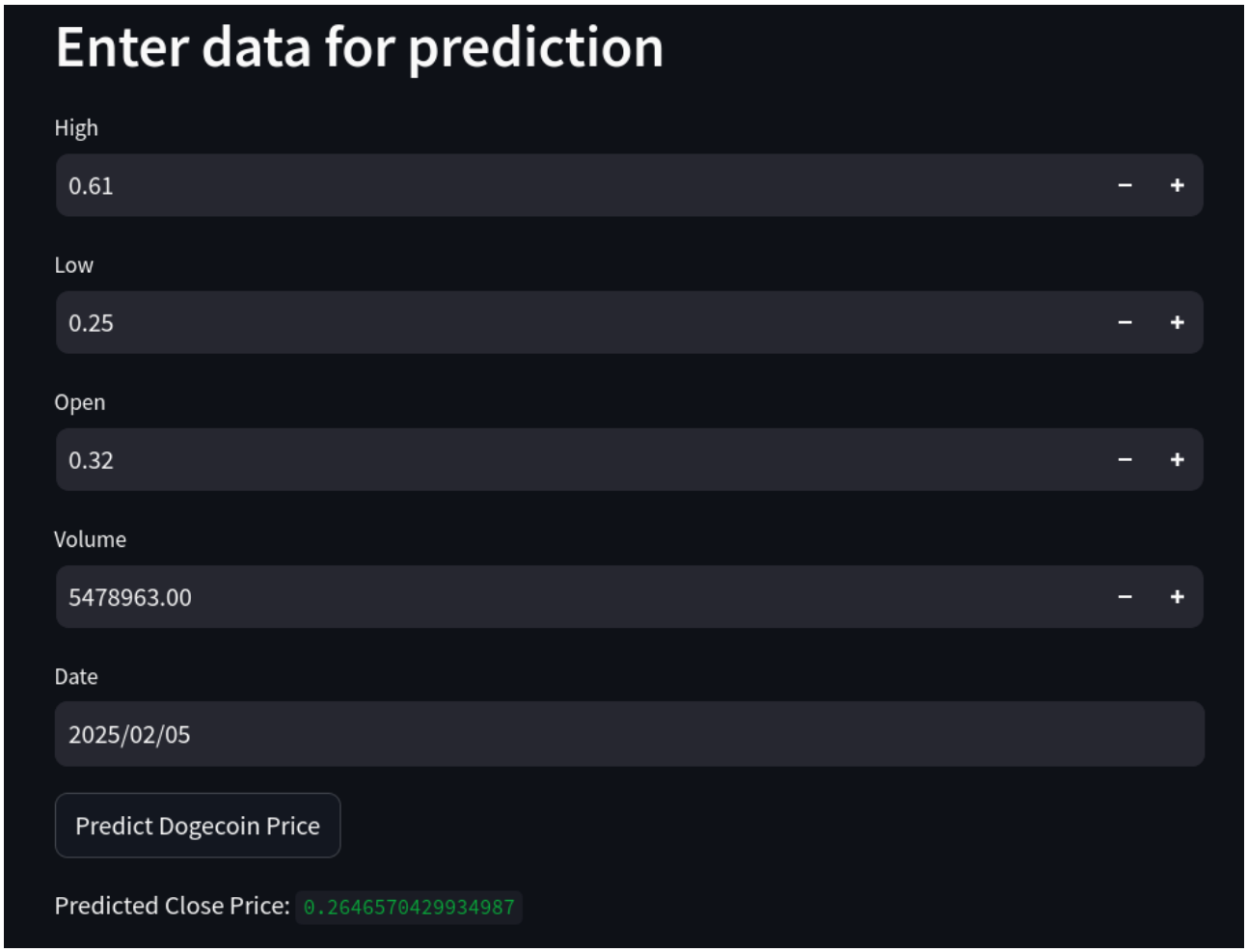


Figure 11 - Web App prediction with user feedback

2.3. Backend

The web app integrates a trained **Random Forest Regressor** model to perform real-time predictions on Dogecoin's closing price based on inputs. The feature extraction process applies a **preprocessing pipeline** that includes *feature engineering* techniques to transform the inputs, ensuring they are in the right format for regression. Additionally, the backend is built for scalability, handling concurrent requests through **Kubernetes** replicas and load balancing to ensure optimal performance even under high traffic.

Key aspects include:

- Integration of a *Random Forest Regressor* for prediction.
- Feature extraction through a *preprocessing pipeline* to transform inputs.
- Scalability with *Kubernetes* for handling concurrent requests and load balancing.

3. Docker containerization and Vulnerabilities Scan

In recent years, **containerization** has become a fundamental approach for efficiently developing, deploying, and scaling applications. Unlike traditional virtual machines, containers share the host operating system's kernel while packaging only the necessary libraries and dependencies required to run an application. This lightweight nature enables containers to start and stop within seconds, making them highly efficient for cloud environments.

Within the scope of **secure cloud computing**, containerization offers several advantages:

- **Application isolation**: Each container runs in its own separate environment, minimizing the potential impact of security breaches.
- **Reduced attack surface**: A well-structured container includes only essential components, limiting exposure to vulnerabilities.
- **Easier patching and updates**: Unlike traditional VMs, updating a containerized application is streamlined by regenerating the image and restarting containers with the latest patches.

3.1. Containerizing the Dogecoin Forecasting Web App

To ensure a **secure and efficient deployment** of the Dogecoin price prediction application, the containerization process followed some best practices, including:

1. **Building a Secure Docker Image**
    - Using **trusted base images** to minimize security risks.
    - Eliminating unnecessary dependencies to reduce the attack surface.
    - Applying the **principle of least privilege**, ensuring the container does not run with excessive permissions, obtained through the creation of a non-root user called '*streamlit*', whose purpose is only to run the web app.
2. **Vulnerability Scanning with Docker Scout**
    - Running automated scans to detect security flaws within the containerized application.
    - Identifying outdated or vulnerable dependencies and applying necessary fixes.
3. **Mitigation Strategies and Security Enhancements**
    - **Regular dependency updates** to prevent security exploits.
    - Implementing **container security best practices**, such as image signing and integrity verification.

```dockerfile
FROM python:3.10.9-slim-buster

# Ensures that Python outputs are sent straight to terminal (stdout) without buffering
ENV PYTHONUNBUFFERED=1

WORKDIR /app

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

# Install dependencies
RUN apt-get update && apt-get install -y \
    libgl1-mesa-glx \
    libglib2.0-0 \
    libsm6 \
    libxext6 \
    libxrender-dev \
    libxrender1 \
    gcc \
    g++ \
    build-essential
RUN pip3 install --no-cache-dir -r requirements.txt

# Expose port
EXPOSE 8501

# Copy all the files needes for the application
ADD data /app/data
COPY app.py /app
ADD models /app/models

# Creates a non-root user (streamlit) to enhance security
RUN useradd -m streamlit && chown -R streamlit /app
USER streamlit

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

Figure 12 - Dockerfile

3.2. Vulnerability Scanning

After creating a Docker image, it is essential to check for known vulnerabilities in the system libraries and application dependencies. Various tools are available for this purpose, both open-source and commercial. In our project, we opted for **Docker Scout**, as it is integrated into *Docker Desktop* and easily accessible from the command line. This tool analyses the image to detect known **CVEs** (*Common Vulnerabilities and Exposures*). The scanning process can be performed locally or on an image uploaded to *Docker Hub* or a private registry.

3.2.1. Scanning the App

The typical scanning process with Docker Scout involves:

1. Building the image: `docker build -t dogecoin-forecasting .`
2. Analyzing the image: `docker scout cves dogecoin-forecasting`

The last command analyses the image layers and compares the installed packages with an updated CVEs database. The returned vulnerability report lists all vulnerable packages, with details on the severity level (Low, Medium, High, Critical) and the vulnerability identifier (e.g. CVE-2024-xxxx). Furthermore, the report can also suggest patched version of the package, if available, or the base image that can be updated.

3.2.2. Vulnerability Report

After the Docker image has been built, the focus must shift on performing a vulnerability scan to identify packages or libraries affected by known CVEs. Docker Scout, after scanning the app, detected several vulnerabilities, among which:

- **setuptools 65.5.1 - CVE-2024-6345 (CWE-94) - 7.5 H (High severity)**
    - This vulnerability allows for remote code execution due to a code injection issue in the package_index module
- **pip 22.3.1 - CVE-2023-5752 (CWE-77) - 6.8 M (Medium severity)**
    - A vulnerability in the pip package manager that can be exploited when installing a package from a Mercurial VCS URL
- debian/binutils 2.31.1-16
    - **CVE-2023-25587 - N/A L (Low severity)**
        - A vulnerability in binutils may allow an attacker to cause a denial of service or potentially execute arbitrary code via specially crafted ELF files
    - **CVE-2023-22608 - N/A L (Low severity)**
        - A vulnerability in binutils could lead to a buffer overflow when processing corrupted ELF files, potentially allowing arbitrary code execution
    - **CVE-2023-2222 - N/A L (Low severity)**
        - A vulnerability in binutils may cause an application crash when analyzing specially crafted binary files
    - **CVE-2021-3487 - N/A L (Low severity)**
        - A vulnerability in binutils could enable an attacker to cause a denial of service through specially crafted binary files
- **debian/sqlite3 3.27.2-3+deb10u2 - CVE-2023-36191 - N/A L (Low severity)**
    - A vulnerability in sqlite3 may allow an attacker to execute arbitrary code or cause a denial of service via specially crafted SQL queries
- **debian/openssl 1.1.1n-0+deb10u3 - CVE-2010-0928 - N/A L (Low severity)**
    - A vulnerability in openssl may allow an attacker to cause a denial of service through specially crafted SSL/TLS requests
- **debian/glib2.0 2.58.3-2+deb10u6**
    - **CVE-2023-25180 - N/A U (Unknown severity)**
        - A vulnerability in glib2.0 could allow an attacker to execute arbitrary code or cause a denial of service via specially crafted input
    - **CVE-2023-24593 - N/A U (Unknown severity)**
        - A vulnerability in glib2.0 could enable an attacker to cause a denial of service or potentially execute arbitrary code through specially crafted input

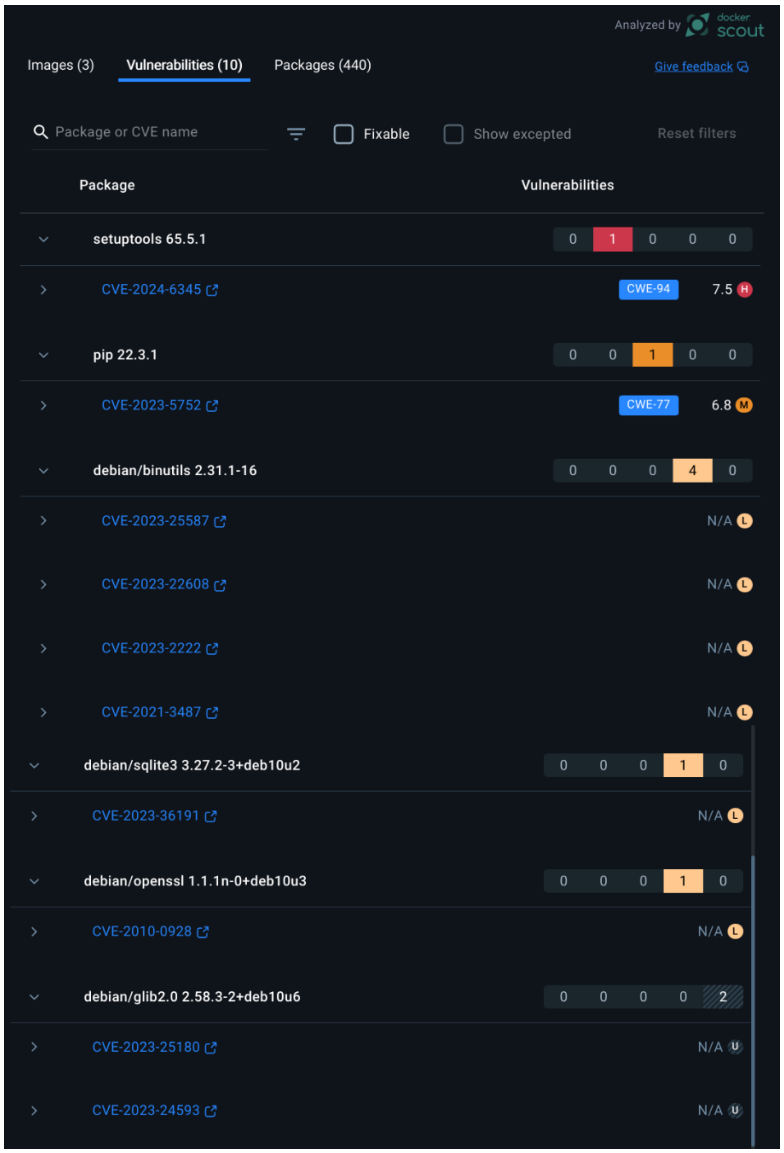


Figure 13 - Vulnerabilities found inside app image

There are a few important considerations to keep in mind:

- Some packages, like *binutils*, may not be strictly necessary, especially if binary files are not being processed. However, if they remain installed, they can expand the attack surface and introduce potential security risks.
- The image is built on **Debian Buster** (*slim-buster*). Some vulnerabilities may have already been addressed through security backports or mitigations, but the scanner might not always detect these fixes, particularly if the package version appears unchanged from the vulnerable one. Upgrading to a more recent base image, such as *Bullseye* or *Bookworm*, can often resolve many CVEs automatically due to more up-to-date security patches.

3.3. Fixing Vulnerabilities: Patches and Mitigations

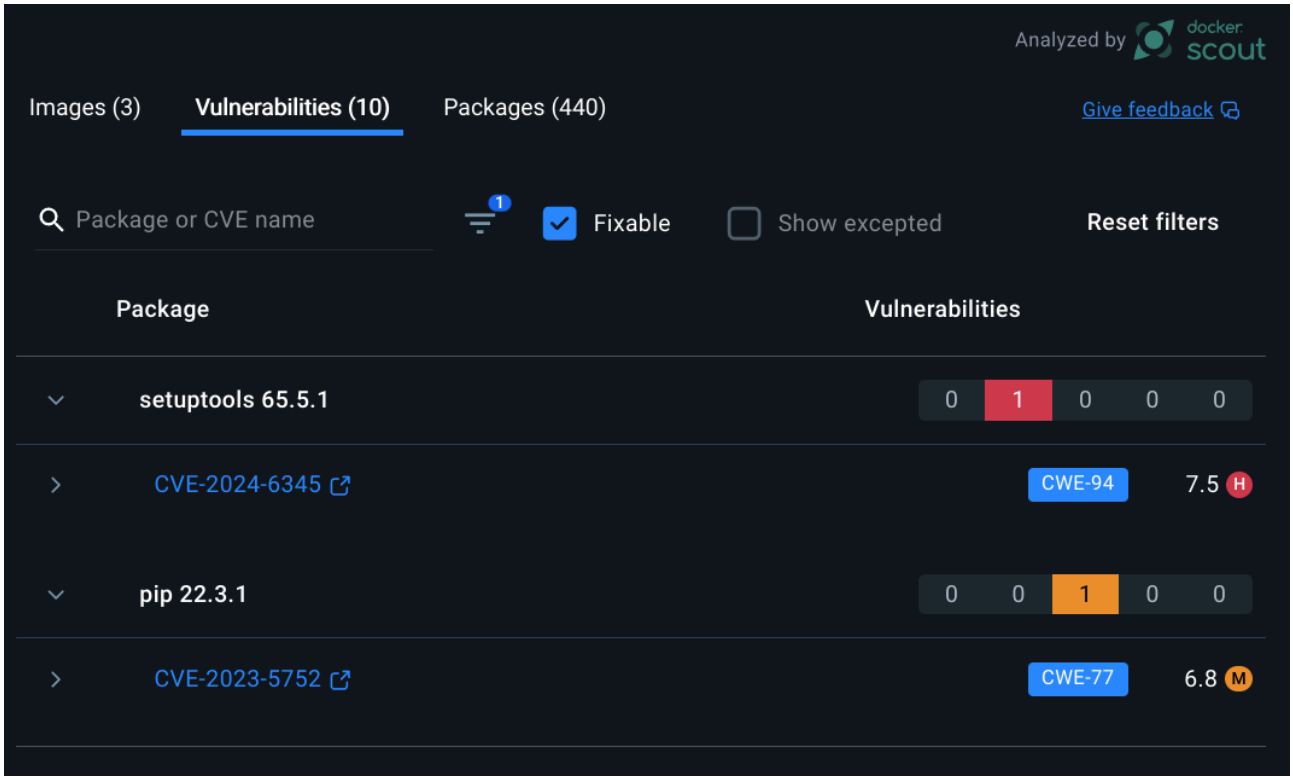


Figure 14 - Fixable vulnerabilities

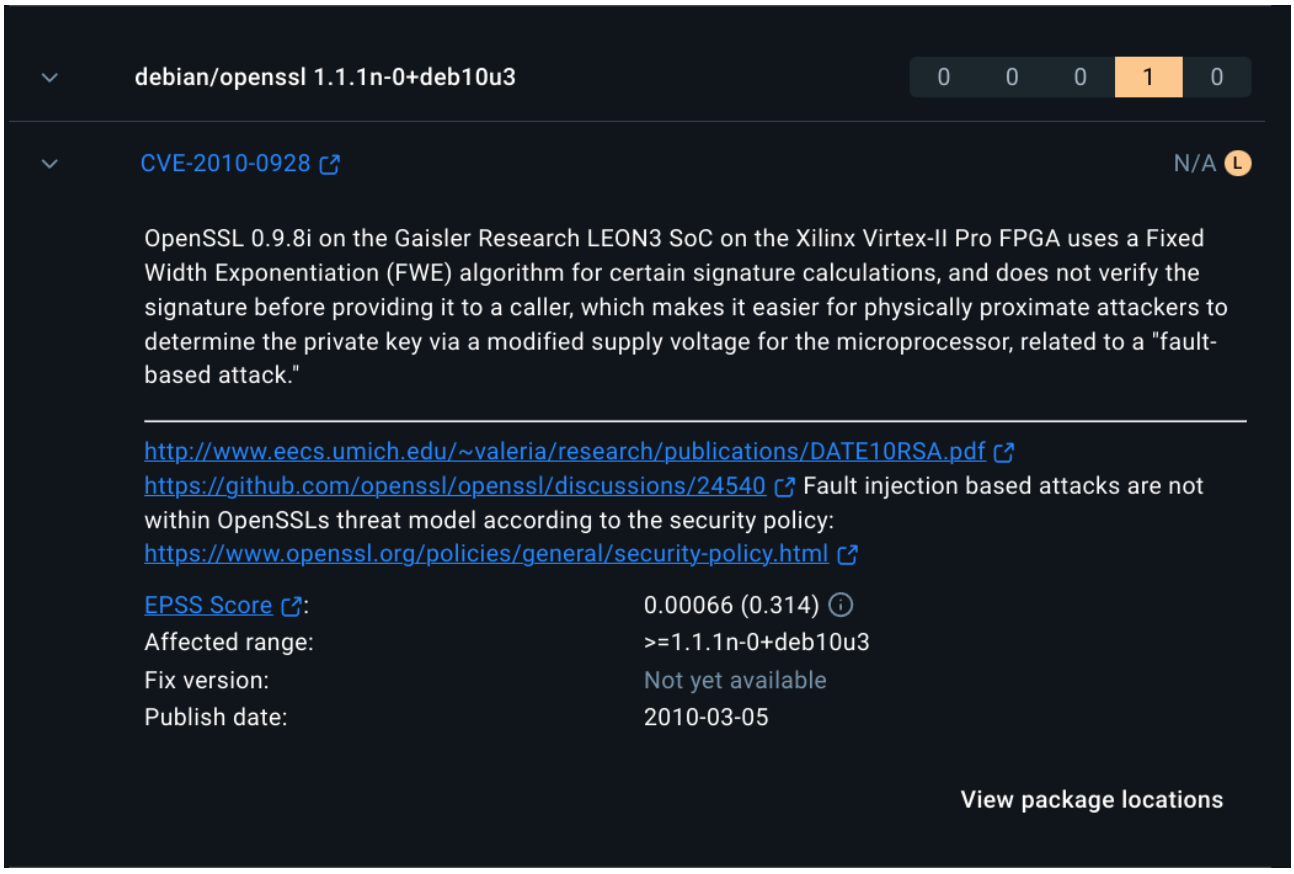


Figure 15 - Vulnerability description - Docker Scout

3.3.1. Updating setuptools

- **Vulnerability detected**: CVE-2024-6345 (High, CWE-94) on setuptools 65.5.1.
- **Possible solution:**
  - Update the *setuptools* package in your *requirements.txt* or explicitly install a later version where the bug is fixed (e.g., *setuptools >= 69.1.1* if the patch has been released).
  - Rebuild the Docker image and re-run the scanning to verify that the vulnerability has been resolved.

3.3.2. Updating pip

- **Vulnerability detected**: CVE-2023-5752 (CWE-77) - 6.8 M (Medium severity) on pip 22.3.1.
- **Possible solution:**
  - Upgrade *pip* in your *requirements.txt* or install a later version where the bug is fixed (e.g. *pip >= 23.3* if the patch has been released)
  - Rebuild the Docker image and re-run the scanning to verify that the vulnerability has been resolved.

3.3.3. Fix binutils

- **Detected vulnerabilities**: CVE-2023-25587, CVE-2023-22608, CVE-2023-2222, CVE-2021-3487.
- **Possible solutions**:
  - Remove the package (if not strictly necessary) from the Dockerfile, for example, by avoiding installing it in the apt-get install lines.
  - If needed during the build phase, it is possible to install it only in a "builder" stage and then copy only the necessary files into the "final" container (multi-stage build pattern).
  - Update to a patched version (if available on Buster) using the commands *apt-get update && apt-get upgrade binutils*, and then check if the Debian release has a backported patch.

3.3.4. Fix other vulnerabilities by updating Base Image

Many of the packages (*sqlite3*, *openssl*, *glib2.0*) have versions with known CVEs in *Debian Buster* (10). Switching to a *Debian Bullseye* (11) or even *Bookworm* (12) base image can resolve many CVEs at once, as the packages are patched in newer releases.

4. Cloud Architecture

Cloud architecture is a fundamental component in the design, deployment, and scalability of modern applications, especially those based on machine learning and artificial intelligence. In this project, our goal is to develop and deploy a Dogecoin price prediction application, ensuring high performance, security, availability, and cost optimization.

The Dogecoin forecasting application is designed to provide real-time predictions of cryptocurrency trends. Given the fluctuating nature of financial data and the varying load of user requests, a cloud-based architecture ensures scalability, resilience, and high availability. This chapter explores the key design decisions and architectural choices that underpin the system's robustness and efficiency.

To achieve this, we leverage Kubernetes for container orchestration, ensuring that the application can dynamically scale based on demand while maintaining security and redundancy. The architecture has been designed to minimize downtime and provide a seamless experience for users accessing the forecasting service.

4.1. Cloud Architectural Models

In our Dogecoin price prediction project, we leveraged various cloud architectures to ensure scalability, reliability, and security. The following principles guided our architectural choices:

- **Workload Distribution**

  It was used to allocate computational tasks across multiple worker nodes within the Kubernetes cluster. This allowed the machine learning model to run efficiently, distributing processing tasks among available nodes to reduce computation time and maintain stable performance, even when handling many simultaneous requests.

- **Service Load Balancing**

  This architectural model was implemented to evenly distribute incoming requests across multiple instances of the application, preventing overload on a single node and improving system responsiveness. By utilizing Kubernetes techniques to handle traffic, user requests are optimally routed, ensuring smooth access to the web app even under high traffic conditions.

- **Dynamic Scalability**

  It enabled the system to automatically adjust resources based on application demands. Thanks to the *Horizontal Pod Autoscaler (HPA)*, the number of web app instances increases or decreases according to traffic load, ensuring optimal resource utilization. This ensures that during peak usage, the system scales horizontally to maintain high performance without unnecessary resource consumption.

- **Redundant Storage**

  It was used to achieve data integrity and availability. The application logs were stored using *Persistent Volumes (PV)* and *Persistent Volume Claims (PVCs)*, ensuring that data remains accessible and protected against potential failures or losses.

In the next paragraph, we will analyse all of them in a deeper way, talking about how these models fit with our ML Dogecoin application.

4.2. Workload Distribution

In our Dogecoin price prediction application, **Workload Distribution** is crucial for ensuring that computational tasks are efficiently allocated across multiple identical instances (*pods*) of the app in the

Kubernetes cluster. Since our application needs to handle up to *50,000 users* with peaks of *5,000 simultaneous requests*, a well-designed and efficient distribution of workloads prevents system slowdowns and improves responsiveness. Indeed, the pods operate independently and are distributed across multiple worker nodes. This prevents any single instance from becoming a bottleneck, thereby enhancing reliability and throughput.

Workload Distribution is Important because, due to it, it is possible to get the following benefits:

- **Efficient Machine Learning Inference**

  Running the prediction model requires computing power, and Kubernetes ensures that inference requests are evenly distributed across multiple nodes to reduce response time.

- **High Performance Under Heavy Load**

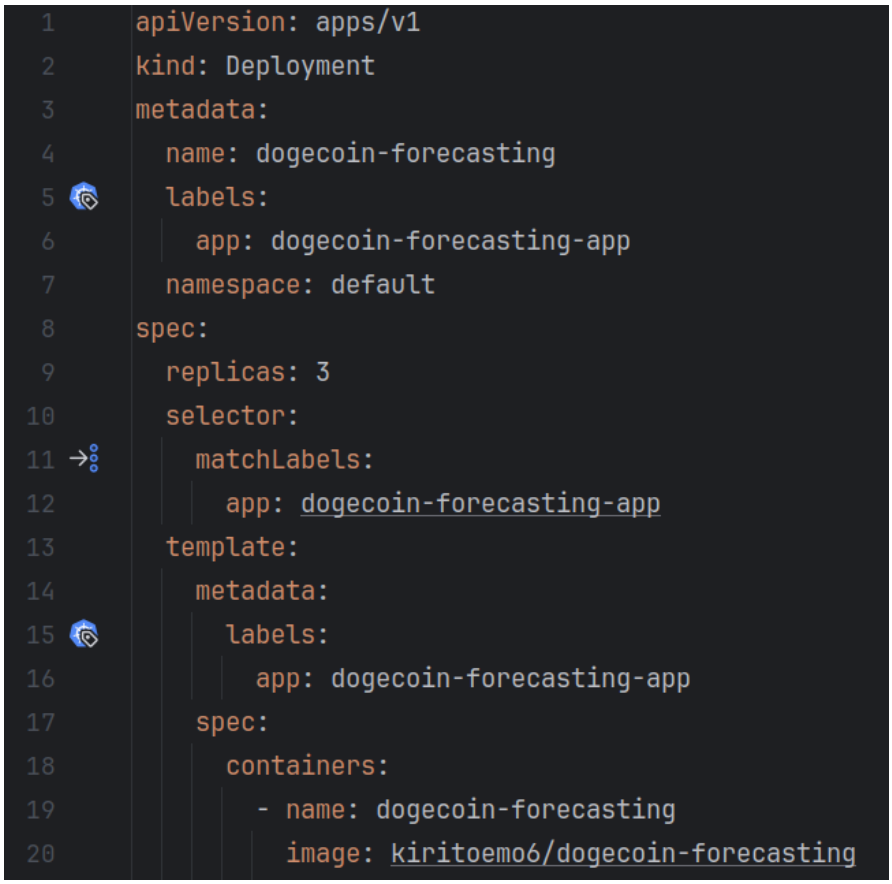
  By distributing workloads properly, our application can handle high-traffic scenarios smoothly without performance degradation.

- **Fault Tolerance and Redundancy**

  If a node fails, Kubernetes redistributes the workload to healthy nodes, ensuring that the service remains available.

- **Cost Optimization**

  Instead of over-provisioning resources permanently, workload distribution combined with autoscaling ensures we only use what is needed at any given moment.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dogecoin-forecasting
  labels:
    app: dogecoin-forecasting-app
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: dogecoin-forecasting-app
  template:
    metadata:
      labels:
        app: dogecoin-forecasting-app
    spec:
      containers:
        - name: dogecoin-forecasting
          image: kiritoemo6/dogecoin-forecasting
```

Figure 16 - Workload Distribution

4.3. Service Load Balancing

As regards the **Service Load Balancing**, it is essential to use it to distribute traffic in an efficient way across multiple instances of the application running in a Kubernetes cluster. Given the requirements to be met (50,000 customers and a maximum of 5,000 simultaneous service requests), load balancing ensures smooth performance and prevents bottlenecks.

4.3.1. First resolution: NodePort

In Kubernetes, without proper load balancing, traffic could be routed unevenly, causing some pods to become overloaded while others remain idle. To prevent this, we initially implement Service Load Balancing using the Kubernetes Service with NodePort configuration, which exposes services externally on a fixed port (we used the port 30070 for the container and the port 30070 on the host machine).

Nevertheless, if on one hand *NodePort* is great for quick testing, on the other hand it can be adequate just for uncomplicated clusters (e.g. single node), so it is not ideal for big projects, for security and management reasons. Plus, there are many constraints regarding which port numbers can be utilized, and as the cluster becomes more complex—potentially housing numerous microservices—the use of NodePort becomes increasingly impractical.

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

Figure 17 - NodePort Service

4.3.2. Second resolution: NGINX Ingress Controller

However, instead of using a standard **NodePort** service (which we initially tested anyway), we opted for a **NGINX Ingress Controller** to manage traffic efficiently, changing to a *ClusterIP* Service, which manages internal load balancing between pods.

The Ingress Controller provides several advantages:

- **SSL/TLS termination**, ensuring secure communication.
- **Path-based routing**, allowing for structured API calls.
- **Rate limiting and DDoS protection**, adding a layer of security.
- **Protection against various threats (SQL Injection, Command Injection, XSS etc.)**, we even redacted a custom rules file to better handle this scenario.

This change influenced in a beneficial way the cluster, because NGINX ensures traffic is redirected to healthy instances, letting us achieve a high availability for the web application. Furthermore, this new Ingress Controller works in sync with Kubernetes Horizontal Pod Autoscaler (HPA) to dynamically balance traffic as pods scale up or down, ensuring an optimal scalability. In addition to that, we found performance optimized, thanks to a reduction to response times and an efficient handling of large requests loads. Finally, talking about security, the WAF, DDoS protection and rate-limiting features of NGINX make it an excellent choice for this concern.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: dogecoin-forecasting
  namespace: default
spec:
  type: ClusterIP
  selector:
    app: dogecoin-forecasting-app
  ports:
  -   protocol: TCP
      port: 8501
      targetPort: 8501
```

Figure 18 - ClusterIP Service

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: dogecoin-ingress
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/ssl-redirect: "false"   # Disable SSL redirect (just for testing purposes)
    nginx.ingress.kubernetes.io/backend-protocol: "HTTP" # Set backend protocol to HTTP (just for testing purposes)
spec:
  ingressClassName: nginx
  rules:
  - host: localhost # Change to your domain name
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: dogecoin-forecasting
            port:
              number: 8501
```

Figure 19 - NGINX Ingress Controller (dogecoin-ingress)

4.4. Dynamic Scalability

If the prediction app faces considerable variations in traffic (such as surges during certain time periods), it's crucial not to keep the number of replicas constant. On one hand, we could end up wasting resources during off-peak times, and on the other hand, we might experience performance issues if there aren't enough pods to handle the increased load during peak times.

4.4.1. Setting Horizontal Pod Autoscaler (HPA)

To handle fluctuations in demand, **Horizontal Pod Autoscaler (HPA)** is employed. The task of a HPA in a Kubernetes cluster is to automatically adjusts the number of pod replicas based on certain metrics, such as CPU utilization, memory consumption, or custom application metrics. When demand rises, HPA increases the number of replicas to maintain application performance. While, when demand drops, HPA reduces the number of replicas, helping to optimize resource usage and lower costs. This dynamic scaling ensures that the application can effectively respond to changing demand while balancing performance and cost efficiency.

An HPA can be configured like showed in the following image:

```
56  # Horizontal scaling
57  apply_horizontal_scaling() {
58      printf "\n>>> Applying horizontal scaling... \n"
59      kubectl autoscale deployment dogecoin-forecasting --cpu-percent=80 --min=3 --max=8 # Horizontal scaling with autoscaler
60      echo "Horizontal scaling applied."
61  }
```

Figure 20 - Horizontal Pod Autoscaler (HPA)

As shown, if the pods exceed 80% CPU usage, the HPA creates an additional pod up to a maximum of 8 pods; while, if the load is low, the HPA reduces the replicas down to a minimum of 3.

The benefits drawn from this approach are multiple, but in particular we can count on a cost/performance optimization, better response times (due to the improved average latency after a replica creation) and a high adaptability to different situations.

4.5. Redundant Storage

The application generates logs that need to persist across pods restarts. To avoid any corruption by crashing pods, or disk failures on the nodes, the **Redundant Storage** architectural model is used. So, the **Persistent Volumes (PV)** and **Persistent Volume Claims (PVCs)** are used to store logs into `` `/logs/predictions.log` `` file.

This action allows you to count on:

- *High availability*: redundancy ensures that data is replicated to multiple nodes or disks, reducing the risk of data loss in the event of hardware failure or outages
- *Operational continuity*: applications can continue to operate without interruption even if one or more storage components fail
- *Fault-tolerance*: in a redundant system, the data is replicated on multiple devices or zones. If a disk or node fails, the data remains accessible from replicas
- *Downtime reduction*: in the event of failure, redundant systems can automatically restore access to data from replicas, minimizing downtime

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: dogecoin-logs-pv
  labels:
    type: local
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  hostPath:
    path: "/mnt/data/dogecoin-logs"
```

Figure 21 - Persistent Volume (PV) configuration

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dogecoin-logs-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: local-storage
```

Figure 22 - Persistent Volume Claim (PVC) configuration

5. Kubernetes Deployment

Kubernetes is an open-source container orchestration platform that automates the deployment, management, and scaling of containerized applications. It enables applications to be deployed in clustered environments, ensuring high availability, load balancing, and efficient resource management.

The system is based on an architecture consisting of a **control plane**, which manages the cluster state and container lifecycle, and **worker nodes**, which run the application containers. The fundamental units in Kubernetes are **pods**, which are groups of one or more containers that share networking and storage resources.

In our project, which leverages cloud computing to predict Dogecoin closing price, we used **Docker Desktop** combined with **Kind (Kubernetes IN Docker)**. Kind allowed us to create and manage Kubernetes clusters in local development environments in a lightweight and efficient way, simplifying testing and configuration without requiring a full cloud infrastructure.

5.1. Cluster Structure

The **cluster architecture** is designed to balance simplicity, functionality, and resource efficiency. A single control node (master) is used to minimize complexity and reduce hardware requirements, making it easier to manage the cluster. While high-availability setups in critical production environments often employ 3 or more master nodes, a single master is a practical choice for our project, offering a good compromise between simplicity and operational functionality.

The inclusion of three worker nodes provides several key benefits:

- **Load Balancing**
  The pods for the Dogecoin closing price application are distributed across the three worker nodes, ensuring that no single node becomes a performance bottleneck. This distribution optimizes resource utilization and improves overall system efficiency.
- **Resilience Simulation**
  Although on a small scale, this setup allows us to simulate a resilient environment. If one worker node fails, the pods can be automatically rescheduled to the remaining worker, ensuring continuous operation and minimal downtime.
- **Horizontal Scaling Testing**
  The three-worker configuration enables us to test horizontal scaling by increasing or decreasing the number of pod replicas. This allows us to observe how Kubernetes dynamically allocates resources and balances workloads across the available nodes.

Additionally, this 1 master + 3 workers configuration provides a realistic environment for testing Kubernetes features such as self-healing, automated scheduling, and scaling policies. It also offers a cost-effective solution for development and testing purposes, as it requires fewer resources compared to larger clusters while still providing a robust platform for experimentation.

In summary, this cluster structure strikes a balance between low complexity (reducing management overhead) and a realistic enough configuration to evaluate resilience, scalability, and operational efficiency. It serves as an ideal foundation for testing and development before potentially scaling up to a more complex production environment.

The web application can be replicated across the 3 worker nodes, significantly enhancing the parallel processing capacity for handling classification requests. This replication ensures that the system can efficiently manage higher workloads by distributing tasks evenly between the three nodes.

5.2. Port Mapping

5.2.1. Initial Configuration with NodePort Service

To test the web application - which by default runs on port 8501 - directly from localhost, we configure the the YAML file (`multinode-config-port-mapping.yaml`) with the following settings:

- `containerPort`: this refers to the internal port within the Kubernetes node where the web app is running.
- `hostPort`: this is the port on the host machine that is mapped to `localhost`, allowing external access to the application.

In this case, the web app is exposed as a **NodePort** service on port 30080, so it is accessible directly from the browser or via cURL at http://localhost:30070. This setup simplifies the testing process by eliminating the need to configure more complex components like ingress controllers or internal DNS.

Additionally, the cluster name, which defaults to "`kind`", can be customized using the `--name` flag when creating the cluster. This flexibility allows you to tailor the environment to your specific testing or development needs while maintaining a straightforward and efficient workflow.

```yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 30080
    hostPort: 30070
  - containerPort: 30020
    hostPort: 30010
- role: worker
- role: worker
- role: worker
```

Figure 23 - Nodes configuration and port mapping with NodePort

5.2.2. Latest configuration with NGINX Ingress Controller

However, NodePort is not the best choice when it comes to production environments, so we decided to rewrite the application using **NGINX Ingress Controller**. When using an *Ingress Controller* (like *NGINX* in this case), there is no need to expose the app with a NodePort. The Ingress controller will handle routing the external traffic to your app via the Ingress resource, and Kubernetes will manage the underlying services.

So, the configuration file has undergone the changes shown in the image below.

```yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
  - containerPort: 30080
    hostPort: 30070
  #- containerPort: 443 # Use this to enable HTTPS
  #  hostPort: 443
  labels:
      ingress-ready: "true"
- role: worker
  labels:
    ingress-ready: "true"
- role: worker
  labels:
    ingress-ready: "true"
- role: worker
  labels:
    ingress-ready: "true"
```

Figure 24 - Nodes configuration and port mapping with NGINX Ingress Controller

5.3. Metric Server

The gathering of metrics about CPU and memory usage at both the pod and node levels is essential to properly size the resources for the web app.

Since Kind does not include a **Metrics Server** by default, we manually install it to enable monitoring and resource optimization. The YAML configuration file (`metrics-server.yaml`) defines the necessary components for deploying the Metrics Server, including:

- **ServiceAccount**: creates a dedicated service account (metrics-server) with the required permissions to access metrics.
- **ClusterRole and RoleBinding**: grants the Metrics Server the necessary permissions to read metrics from pods and nodes, as well as access to the Kubernetes API.
- **Service**: exposes the Metrics Server internally within the cluster on port 443.
- **Deployment**: configures the Metrics Server with specific arguments, such as `--kubelet-insecure-tls`, which is justified for local testing environments where formal certificates are unnecessary. This simplifies setup while maintaining functionality.
- **APIService**: registers the Metrics Server API (v1beta1.metrics.k8s.io) to integrate it with the Kubernetes API server.

The Metrics Server collects resource usage data at a configurable resolution (e.g., every 15 seconds) and provides this information to Kubernetes components like the *Horizontal Pod Autoscaler (HPA)*. This capability is crucial for ensuring the web app can dynamically adapt to varying workloads.

Additionally, the configuration of the *Metrics Server* includes security best practices, such as running as a non-root user and restricting privilege escalation, ensuring a secure deployment even in a local testing environment. By monitoring resource consumption, we can determine whether optimizations are necessary or if scaling up replicas is warranted.

5.4. Kubernetes Dashboard

Even if the command-line interface (*kubectl*) is highly powerful and versatile, a **web dashboard** provides several advantages that enhance usability and efficiency:

- **Intuitive Visualization**: the dashboard offers a graphical representation of Kubernetes objects such as Pods, Deployments and Services, showing how they are interconnected. This makes it easier to understand the cluster's structure and relationships between components.
- **Quick Debugging**: by simply clicking on a pod, a user can access detailed information such as logs, events, and readiness status. This simplifies troubleshooting and reduces the time needed to identify and resolve issues.
- **Training and Education**: the dashboard serves as an excellent tool for demonstrating cluster functionality in an immediate and accessible way, making it ideal for training purposes or educational environments.

To enable access to the Kubernetes dashboard, we created a *ServiceAccount* with cluster-admin privileges in the *kubernetes-dashboard* namespace. While this approach is not recommended for production environments due to the risk of privilege escalation, it is justified in a lab or testing context as it simplifies setup and usage.

Access to the dashboard is granted via a token generated using the following commands:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml
kubectl apply -f dashboard-adminuser.yaml
kubectl apply -f cluster_rolebinding.yaml
kubectl -n kubernetes-dashboard create token admin-user #Token ServiceAccount
kubectl proxy
```

Figure 25 - Kubernetes Dashboard commands

In a real-world scenario, it would be advisable to implement more restrictive roles and secure authentication mechanisms, such as *role-based access control (RBAC)* or integration with external identity providers, to ensure proper security and compliance. However, for local testing and development, this streamlined approach provides a practical and efficient way to interact with the cluster.

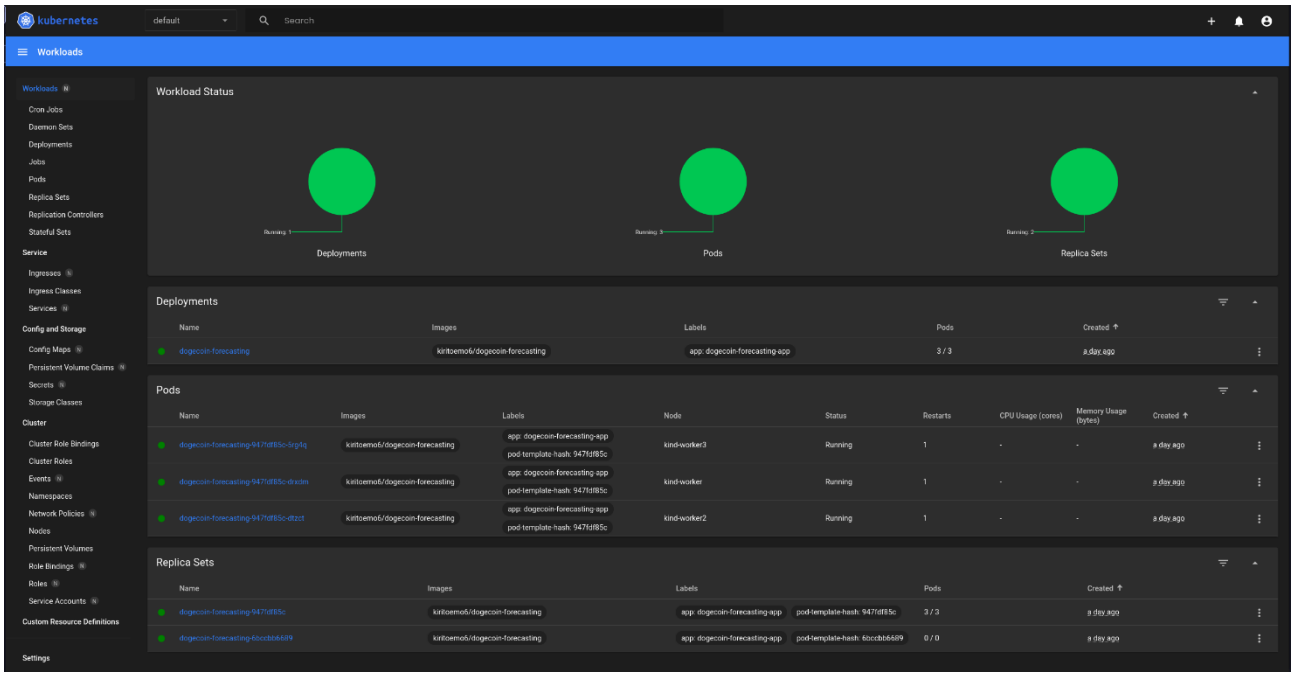


Figure 26 - Kubernetes Dashboard

5.5. NGINX Ingress Controller

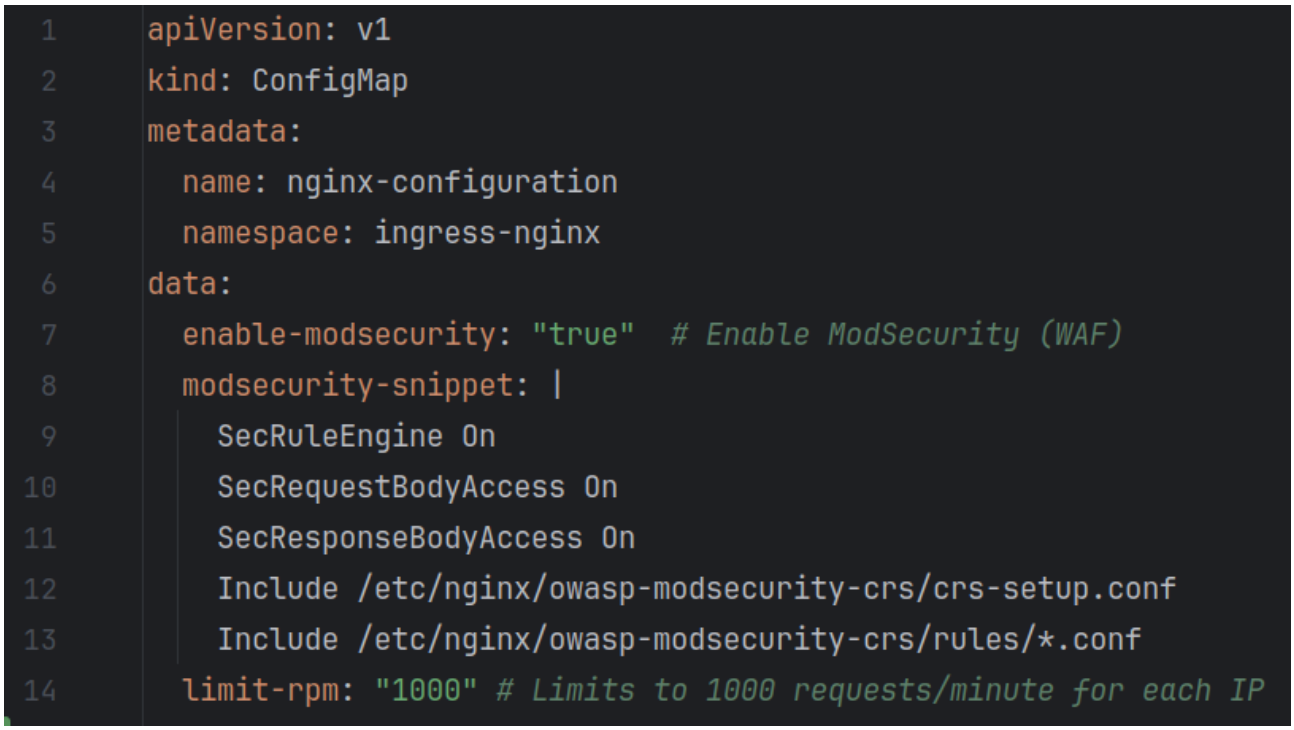
To manage external access to the URL classification web app, we deployed the **NGINX Ingress Controller** using the following commands:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/kind/deploy.yaml
kubectl apply -f nginx-config.yaml
```

Figure 27 - NGINX Ingress Controller commands

The NGINX Ingress Controller was configured with a **ConfigMap** (`nginx-config.yaml`) to enable *ModSecurity*, a *Web Application Firewall (WAF)*, and to limit requests to 1000 per minute per IP address. This provides essential security features such as protection against SQL injection, cross-site scripting (XSS), and command injection attacks. Additionally, custom rules were applied via a second **ConfigMap** (`nginx-custom-rules.yaml`) to further enhance security, including rate limiting for DDoS protection and blocking suspicious user agents.

The deployment (`nginx-deployment.yaml`) scales the NGINX Ingress Controller to 3 replicas for fault tolerance and mounts the custom configuration file. The **Ingress** resource (`dogecoin-ingress.yaml`) routes traffic to the dogecoin-forecasting service on port 8501, with annotations to disable SSL redirects and set the backend protocol to HTTP for testing purposes. This setup ensures secure, scalable, and efficient traffic management for the web app.

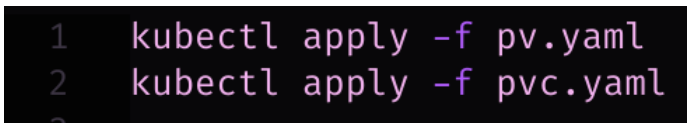


```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-configuration
  namespace: ingress-nginx
data:
  enable-modsecurity: "true"  # Enable ModSecurity (WAF)
  modsecurity-snippet: |
    SecRuleEngine On
    SecRequestBodyAccess On
    SecResponseBodyAccess On
    Include /etc/nginx/owasp-modsecurity-crs/crs-setup.conf
    Include /etc/nginx/owasp-modsecurity-crs/rules/*.conf
  limit-rpm: "1000" # Limits to 1000 requests/minute for each IP
```

Figure 28 - NGINX configuration file

5.6. PV & PVC

To store logs and other persistent data for the application, we created a **PersistentVolume (PV)** and a **PersistentVolumeClaim (PVC)** using the following commands:



```
kubectl apply -f pv.yaml
kubectl apply -f pvc.yaml
```

Figure 29 - PV and PVC commands

The **PersistentVolume** (`pv.yaml`) is configured with a storage capacity of *1Gi* and uses a hostPath to map to a local directory (`/mnt/data/dogecoin-logs`). The *persistentVolumeReclaimPolicy* is set to *Retain*, ensuring that data is preserved even if the PVC is deleted.

The **PersistentVolumeClaim** (`pvc.yaml`) requests *1Gi* of storage with *ReadWriteOnce* access mode, binding to the PV.

This setup provides a reliable and persistent storage solution for the application's logs, ensuring data durability and availability across pod restarts or failures. By combining these components, we achieve a robust and scalable infrastructure that supports both secure external access and reliable data storage for the Dogecoin prediction app.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: dogecoin-logs-pv
  labels:
    type: local
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  hostPath:
    path: "/mnt/data/dogecoin-logs"
```

Figure 30 - PV configuration file

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dogecoin-logs-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: local-storage
```

Figure 31 - PVC configuration file

5.7. Application Deployment

To test the cluster previously created, we can deploy a containerized app.

The steps to obtain such result are the following:

1. Build the Docker image locally
   - `docker build -t dogecoin-forecasting .`
   - `docker run --publish 8501:8501 -it dogecoin-forecasting`
2. Push the image to a registry (in this case Docker Hub)
   - `docker login -u kiritoemo6 docker.io`

    - `docker tag dogecoin-forecasting kiritoemo6/dogecoin-forecasting`
    - `docker push docker.io/kiritoemo6/dogecoin-forecasting`
3. Create the deployment using the related file (in this case `k8s_dogecoin_deployment.yaml`)
    - `kubectl create --filename k8s_dogecoin_deployment.yaml`
4. Apply the Ingress Resource (`dogecoin-ingress.yaml`)
    - `kubectl apply -f dogecoin-ingress.yaml`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dogecoin-forecasting
  labels:
    app: dogecoin-forecasting-app
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: dogecoin-forecasting-app
  template:
    metadata:
      labels:
        app: dogecoin-forecasting-app
    spec:
      containers:
        - name: dogecoin-forecasting
          image: kiritoemo6/dogecoin-forecasting
          resources:
            requests:
              memory: "400Mi"
              cpu: "500m"
            limits:
              memory: "512Mi"
              cpu: "750m"
          volumeMounts:
            - mountPath: "/app/data"
              name: storage-volume
      volumes:
        - name: storage-volume
          persistentVolumeClaim:
            claimName: dogecoin-logs-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: dogecoin-forecasting
  namespace: default
spec:
  type: ClusterIP
  selector:
    app: dogecoin-forecasting-app
  ports:
  -   protocol: TCP
      port: 8501
      targetPort: 8501
```

Figure 32 - Application deployment

5.7.1. Deployment and Service

The **Deployment** resource is responsible for managing the number of replicas (pods) and ensuring a smooth update process. For instance, if a new version of the application is released (e.g., due to an update in the ML model), Kubernetes performs a *rolling update*. This means that pods are updated incrementally, ensuring zero downtime and continuous availability of the service.

The **Service** resource, configured as a *ClusterIP* (to connect to NGINX Ingress Controller), was chosen for its modularity and security enhancements in local testing environments.

5.7.2. Scaling

To ensure high availability and resilience, the **Deployment** is configured with **3 replicas** of the web application. This means that if one pod crashes or needs to be restarted, the other pods continue to handle requests, minimizing downtime and maintaining service continuity. Scaling can be done manually using the `kubectl scale` command or automatically by configuring the *Horizontal Pod Autoscaler (HPA)* with `kubectl autoscale`.

This setup is designed to handle sudden spikes in traffic, such as *5000 simultaneous requests*. If the CPU usage of the pods exceeds a defined threshold, the *HPA* automatically creates additional pods and distributes the load across the available worker nodes. This ensures that the application remains responsive and reliable, even under heavy load, while also optimizing resource utilization. By combining replication and autoscaling, the system achieves both resilience and scalability, meeting the demands of a production-grade environment.

6. Security and Protection from Attacks

Transitioning from a simple local Docker environment to a Kubernetes cluster introduces a significant expansion in the attack surface. This shift brings new challenges and vulnerabilities that must be addressed to ensure the security and integrity of your applications and infrastructure. In a Kubernetes environment, applications are no longer confined to a single machine; instead, they are distributed across multiple nodes, each running containerized workloads. This distributed nature, while beneficial for scalability and resilience, also opens up new avenues for potential attacks. Additionally, the Kubernetes control plane itself becomes a prime target for malicious actors, as compromising it could grant an attacker control over the entire cluster. Furthermore, the exposure of endpoints and services, which provide APIs and ports for communication, can serve as entry points for intrusions if not properly configured and secured.

In the context of secure cloud computing, it is imperative to protect both your applications and the Kubernetes cluster from a wide range of threats, including exploitation attempts, Distributed Denial of Service (DDoS) attacks, injection attacks, and more. This chapter delves into both theoretical concepts and practical security measures, with a focus on two primary areas:

1. **Application-level security**: this involves safeguarding the Dogecoin prediction application from common attacks such as injection, Cross-Site Scripting (XSS), and Denial of Service (DoS) attacks, particularly targeting machine learning endpoints. Application-level security ensures that the application itself is resilient to attacks that could compromise its functionality or data.
2. **Cluster-level security**: this encompasses the protection of the Kubernetes cluster as a whole. Key measures include implementing Role-Based Access Control (RBAC), enforcing Pod Security Standards, defining network policies, and defending against attacks on the control plane. Cluster-level security ensures that the underlying infrastructure is secure, preventing unauthorized access or tampering.

Additionally, this chapter incorporates best practices from the webinar *"Protecting Apps from Hacks in Kubernetes with NGINX"*, which covers essential strategies such as Ingress configurations, rate limiting, Web Application Firewall (WAF) implementation, and other mitigation techniques. These practices are crucial for building a robust defense against both common and sophisticated attacks.

6.1. Common Threats in Cloud and Kubernetes Environments

In a Kubernetes-based Dogecoin prediction application, several threats are prevalent, each posing unique risks to the security and stability of the system. Understanding these threats is the first step toward implementing effective countermeasures.

1. **Control Plane Attacks**: The Kubernetes control plane, which includes components like the API server and the scheduler, is a critical target for attackers. Compromising the control plane can grant an attacker full control over the cluster, allowing them to manipulate workloads, steal data, or disrupt services. Protecting the control plane involves securing communication channels, enforcing strict access controls, and regularly updating Kubernetes components to patch known vulnerabilities.
2. **Privilege Escalation**: Containers running with excessive privileges, such as those running as the root user, or Pods with overly permissive RBAC configurations, can be exploited by attackers to gain elevated access. Privilege escalation attacks can lead to unauthorized access to sensitive resources or even full control of the cluster. Mitigating this risk involves adhering to the principle of least privilege, ensuring that containers and service accounts have only the permissions they need to function.

3. **Secret Theft**: Poor management of sensitive information, such as passwords, tokens, or access keys, can lead to data breaches. Secrets stored in plaintext in ConfigMaps, public Dockerfiles, or exposed environment variables are particularly vulnerable. Kubernetes provides a Secrets mechanism for securely storing and managing sensitive data, but it must be used correctly to avoid exposure.
4. **DDoS and Resource Overload**: Distributed Denial of Service (DDoS) attacks aim to overwhelm the system by saturating CPU, memory, or network resources, rendering the application unavailable. In a Kubernetes environment, resource overload can also occur due to misconfigured autoscaling or excessive resource requests. Implementing rate limiting, network policies, and resource quotas can help mitigate these risks.
5. **Application Vulnerabilities**: Common application-level vulnerabilities, such as injection attacks (e.g., SQL injection, command injection), Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and input manipulation, can be exploited to compromise the application. For the Dogecoin prediction application, malicious user inputs could be used to manipulate prediction results or gain unauthorized access to the system.

6.2. Kubernetes Cluster Security

6.2.1. Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) is a Kubernetes mechanism that defines which users or service accounts can perform specific actions on resources. In a production cluster, RBAC is essential for enforcing security policies and minimizing the risk of unauthorized access.

- **Restricting Destructive Actions**: RBAC can be used to prevent developers from executing destructive *kubectl* commands, such as deleting Deployments or Secrets. This ensures that only authorized personnel can perform critical operations.
- **Limiting Service Account Permissions**: Service accounts used by applications, such as the Dogecoin prediction app, should have limited permissions. For example, the app should not be able to create Pods or access system Secrets. This reduces the risk of privilege escalation and unauthorized access.
- **Principle of Least Privilege**: RBAC enforces the principle of least privilege, ensuring that each role has only the minimum permissions necessary to perform its tasks. For the Dogecoin prediction application, roles could include:
    - **app-operator**: Manages Deployments and Services within the app's namespace.
    - **read-only-logs**: Allows analysts to view Pod logs without modification rights.

6.2.2. Pod Security Standards (PSS)

Pod Security Standards (PSS) define security rules at the Pod level, replacing the deprecated Pod Security Policies. These standards help enforce security best practices, such as:

- **Non-Root Containers**: Prohibiting containers from running as the root user reduces the risk of privilege escalation.
- **Restricting hostPath Mounts**: Limiting unnecessary hostPath filesystem mounts prevents containers from accessing sensitive host files.
- **Disabling Privileged Capabilities**: Disallowing privileged capabilities like SYS_ADMIN or SYS_PTRACE reduces the attack surface.

For the Dogecoin prediction app, ensuring that the Dockerfile uses a non-root user and enabling PSS with a restricted profile will help reject Pods that require excessive privileges.

6.2.3. Network Policies

Network Policies control communication between Pods and services, providing an additional layer of security by isolating workloads and restricting unnecessary network access.

- **Allowing Incoming Traffic Only on Specific Ports**: Incoming traffic should be allowed only on HTTP/HTTPS ports (e.g., NodePort 30080 or NGINX Ingress Controller on port 80 and 443) to ensure that only legitimate user interactions are permitted.
- **Isolating Frontend and Backend**: For the Dogecoin prediction app, frontend Pods should be isolated from database Pods to prevent unauthorized access to sensitive data.
- **Restricting Outbound Internet Access**: Pods that do not require external connectivity should have outbound internet access restricted to minimize the risk of data exfiltration.

6.3. Protecting the Dogecoin Prediction Application

While Kubernetes provides robust infrastructure-level security, application-level protection is equally critical. The following practices are essential for securing the Dogecoin prediction application:

- **Input Validation**: Validate all user inputs for Dogecoin predictions to prevent injection attacks or malicious payloads. Input validation ensures that only legitimate data is processed by the application.
- **Error Handling**: Avoid exposing detailed stack traces in error responses, as they may reveal sensitive application logic or vulnerabilities. Instead, provide generic error messages that do not disclose internal details.
- **Secure Logging**: Never log sensitive information, such as API keys or user credentials, in public logs. Secure logging practices help prevent data leaks and ensure compliance with privacy regulations.
- **Rate Limiting and WAF**: Implement rate limiting and Web Application Firewall (WAF) features to mitigate DDoS and injection attacks. Rate limiting restricts the number of requests a client can send within a defined time interval, while a WAF analyses request content to block malicious patterns.

6.3.1. DDoS Attacks

Denial of Service (DoS) attacks aim to exhaust system resources, such as CPU, memory, or network bandwidth, to make the service unavailable. Distributed Denial of Service (DDoS) attacks involve multiple compromised hosts (botnets) and are more difficult to mitigate. Common types of DDoS attacks include:

- **Volumetric Attacks**: These attacks flood the network with massive traffic, such as UDP packets, overwhelming the available bandwidth.
- **Application-Layer Attacks**: These attacks target the application with excessive HTTP/HTTPS requests, exhausting server resources.
- **Protocol-Based Attacks**: These attacks exploit protocol vulnerabilities, such as SYN floods, to disrupt service.

For the Dogecoin prediction app, an attacker might flood the prediction endpoint with thousands of requests, exhausting resources and disrupting service. Implementing rate limiting, WAF, and network-level defences can help mitigate these attacks.

6.3.2. Rate Limiting

Rate limiting is a critical defence mechanism that restricts the number of requests a client can send within a defined time interval. Benefits of rate limiting include:

- **Preventing Resource Monopolization**: Rate limiting ensures that a single IP address cannot monopolize resources, ensuring fair access for all users.
- **Mitigating Application-Layer DDoS Attacks**: By limiting the number of requests, rate limiting helps mitigate application-layer DDoS attacks that aim to exhaust server resources.
- **Ensuring Fair Access**: Rate limiting ensures that legitimate users can access the service without being affected by malicious traffic.

However, advanced attackers may use techniques such as IP spoofing to bypass IP-based rate limiting. To address this, consider implementing additional layers of defence, such as WAF rules or behavioural analysis.

6.3.3. NGINX as an Ingress Controller

NGINX is a popular Ingress Controller in Kubernetes, offering a range of features that enhance security and performance. For the Dogecoin prediction app, NGINX can:

- **Route Incoming Requests**: NGINX routes incoming HTTP/HTTPS requests to the appropriate services, ensuring that traffic is directed correctly.
- **Apply Rate Limiting and WAF Rules**: NGINX can enforce rate limiting and WAF rules to block malicious traffic before it reaches the application.
- **Enable SSL/TLS**: NGINX supports SSL/TLS termination, ensuring that data is encrypted in transit.

6.3.4. Web Application Firewall (WAF)

A Web Application Firewall (WAF) analyses request content, including methods, headers, paths, and payloads, to block malicious patterns. For the Dogecoin prediction app, integrating ModSecurity with NGINX can provide robust protection against common threats such as SQL injection and XSS. Key considerations include:

- **OWASP Rules**: Apply OWASP rules to detect and block known attack patterns.
- **Blocking Suspicious Paths**: Block access to suspicious paths, such as /admin or /.git/, to prevent unauthorized access.
- **Parameter Inspection**: Inspect request parameters for injection syntax and block malicious payloads.

6.3.5. Integration with Autoscaling and Resilience

Rate limiting and autoscaling work together to handle traffic spikes and attacks. The Horizontal Pod Autoscaler (HPA) scales Pods to handle legitimate traffic spikes, while rate limiting and WAF block malicious traffic before it consumes resources. In a DDoS attack, autoscaling without rate limiting can lead to uncontrolled resource consumption. A balanced approach ensures scalability while mitigating attacks.

6.3.6. Additional Countermeasures

For large-scale production environments, additional countermeasures can further enhance security:

- **Content Delivery Network (CDN)**: Use a CDN, such as Cloudflare or Akamai, to filter volumetric DDoS traffic before it reaches the Kubernetes cluster.
- **Network-Level Anti-DDoS**: Deploy firewalls or IP traffic filters to block layer 3/4 attacks.
- **Layer 7 Inspection**: Use NGINX and ModSecurity to inspect and block malicious HTTP/HTTPS traffic at the application layer.

By implementing these measures, you can build a robust defense against a wide range of threats, ensuring the security, integrity, and availability of your Dogecoin prediction application in a Kubernetes environment.

7. Stress Tests

One of the primary requirements for our prediction web app is the capability to manage traffic spikes of up to 5,000 simultaneous requests and serve a total of 50,000 customers.

In this chapter, we will explore how we evaluated the system's robustness and scalability through a series of stress tests, focusing on the following aspects:

- **Stress Testing Tools**: the tools we selected and the rationale behind their choice.
- **Methodology**: the approach we followed to conduct the tests, including parameters, metrics collected, and the execution environment.
- **Results**: an analysis of key statistics such as throughput, latency, and error rates.
- **Optimization**: strategies implemented to enhance the cluster's performance, including autoscaling, resource limits, and network configuration.

The overarching objective is to demonstrate that our application, when deployed on Kubernetes with an appropriate number of replicas, can dynamically scale to handle increasing load while maintaining acceptable latency, minimizing errors, and reducing timeouts.

7.1. Tools Used

7.1.1. Wrk

In order to test the availability and fault tolerance of the service, we used *wrk*, a modern HTTP benchmarking tool capable of generating significant load of requests.

Wrk is:

- **Multi-threaded**: it can efficiently utilize multiple threads, enabling thousands of concurrent connections to simulate high traffic.
- **Easy to configure:** it is executed via the command line with straightforward parameters such as `-t` (threads), `-c` (connections), and `-d` (duration).
- **Essential statistics**: it provides critical metrics like throughput (requests/sec), average and maximum latency, transfer rate (MB/s), and the number of errors (e.g., socket errors or timeouts).

Wrk has proven to be a good choice thanks to its lightweight nature and ability to generate high volumes of requests from a single host, making it ideal for stress testing our application.

```
Running 30s test @ http://127.0.0.1:30070/
  8 threads and 5000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   348.71ms   54.26ms   1.45s    74.62%
    Req/Sec   363.40    196.85     1.01k    65.82%
  86655 requests in 30.08s, 183.71MB read
  Socket errors: connect 3987, read 0, write 0, timeout 0
Requests/sec:   2880.75
Transfer/sec:      6.11MB
```

Figure 33 - Wrk tool

7.1.2. Apache Benchmark (ab)

We also explored **Apache Benchmark (ab)**, another widely used tool for performance testing:

- **Single-threaded**: unlike wrk, *ab* operates on a single thread, which can limit its ability to simulate extremely high concurrency.
- **Simple to use**: it is launched from the command line with parameters such as `-n` (number of requests) and `-c` (concurrency level).
- **Key metrics**: it provides useful statistics like requests per second, time per request, and the percentage of requests served within a certain time.

While **ab** is user-friendly and effective for basic load testing, its single-threaded design makes it less suitable for scenarios requiring massive concurrency, which is why we prioritized **wrk** for our stress tests.

```
This is ApacheBench, Version 2.3 <$Revision: 1923142 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient)


Server Software:        TornadoServer/6.4.2
Server Hostname:        127.0.0.1
Server Port:            30070

Document Path:          /
Document Length:        1837 bytes

Concurrency Level:      500
Time taken for tests:   26.189 seconds
Complete requests:      50000
Failed requests:        0
Total transferred:      111150000 bytes
HTML transferred:       91850000 bytes
Requests per second:    1909.19 [#/sec] (mean)
Time per request:       261.891 [ms] (mean)
Time per request:       0.524 [ms] (mean, across all concurrent requests)
Transfer rate:          4144.66 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   2.9      0      43
Processing:     1  255 334.0    197    3484
Waiting:        1  255 333.9    197    3483
Total:          1  255 333.8    198    3485

Percentage of the requests served within a certain time (ms)
  50%    198
  66%    229
  75%    252
  80%    268
  90%    345
  95%   1151
  98%   1338
  99%   1608
 100%   3485 (longest request)

```

Figure 34 - Apache Benchmark tool

7.2. Test environment

Our testing environment is composed of:

- **Local Kubernetes Cluster**: Docker Desktop + Kind (1 control plane and 3 worker nodes).
- **Host machine**: same or different machine running *wrk* and *ab*. If the PC is the same, it can occur a resource contention. In order to perform a reliable test, it would be better to have a second PC available, although this choice may involve a more complex configuration of the network.
- **Endpoint**: the *NGINX Ingress Controller* evenly reroute network traffic through the port 80 and 443, where the app is listening.

7.3. Tests

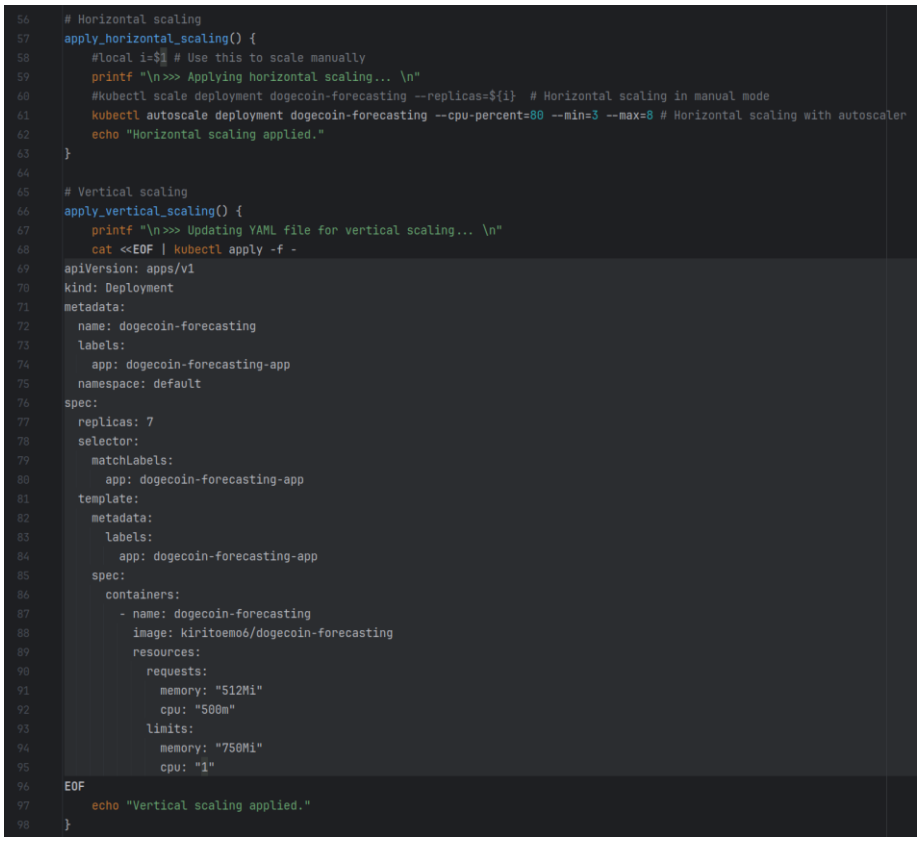
As for the tests, we performed several measurements in different conditions. First of all, we performed detections with tests performed on the NodePort Service and others with NGINX Ingress Controller. We also tested the various cases of horizontal scaling by examining, one by one, each available option starting from a minimum of 2 replicas and reaching a maximum of 15. The results highlighted better performance in the range of 6-8 replicas, with a peak optimization achieved with 7 replicas. After doing this, we imposed the range within the autoscaler, so as to have a measure of handle limited to the desired scope of use.

In order to optimize the execution times of the tests and the related measurements, we wrote a bash script so that it can be launched from the Linux terminal as soon as the final deployment is completed. The script first defines the initial variables such as: the URL of the service to be tested (`SERVICE_URL`), the duration of the test (`TEST_DURATION`), the maximum number of connections (`MAX_CONNECTIONS`), the total number of requests (`TOTAL_REQUESTS`), the number of threads for wrk (`WRK_THREADS`) and the output directory for the results (`OUTPUT_DIR`), which it will create (if non-existent) when it starts.

Secondly, the script runs metrics collection functions for Kubernetes. After that, the *ab* and *wrk* tests are run. Finally, the scaling strategies are applied, first the horizontal one via *HPA* and then the vertical one with the *resources updated* to the new values. Each of these phases is subject to testing, in order to identify the best possible scenario.

```
# Initial configurations
SERVICE_URL="http://localhost:80/"
TEST_DURATION="30s"
MAX_CONNECTIONS=5000
TOTAL_REQUESTS=50000
WRK_THREADS=8
OUTPUT_DIR="stress_test_results"

# Create directory for results
mkdir -p $OUTPUT_DIR

# Function to collect Kubernetes metrics
collect_metrics() {
    local i=$1
    printf "\n>>> Collecting Kubernetes metrics..."
    kubectl top pods >> "$OUTPUT_DIR/pods_metrics_${i}.txt"
    kubectl top nodes >> "$OUTPUT_DIR/nodes_metrics_${i}.txt"
}

# Apache Benchmark (ab) test
run_ab_test() {
    local i=$1
    printf "\n>>> Running Apache Benchmark (ab)... \n"
    ab -n $TOTAL_REQUESTS -c 500 $SERVICE_URL >> "$OUTPUT_DIR/ab_test_${i}.txt"
    echo "Apache Benchmark completed. Results saved."
}

# Function to collect Kubernetes metrics after ab test
collect_metrics_ab() {
    local i=$1
    printf "\n>>> Collecting Kubernetes metrics after ab test..."
    kubectl top pods >> "$OUTPUT_DIR/pods_metrics_ab_${i}.txt"
    kubectl top nodes >> "$OUTPUT_DIR/nodes_metrics_ab_${i}.txt"
}

# wrk test with different configurations
run_wrk_test() {
    local i=$1
    printf "\n>>> Running wrk with $MAX_CONNECTIONS connections for $TEST_DURATION... \n"
    wrk -c $MAX_CONNECTIONS -d $TEST_DURATION -t $WRK_THREADS $SERVICE_URL >> "$OUTPUT_DIR/wrk_test_${i}.txt"
    echo "wrk completed. Results saved."
}

# Function to collect Kubernetes metrics after wrk test
collect_metrics_wrk() {
    local i=$1
    printf "\n>>> Collecting Kubernetes metrics after wrk test..."
    kubectl top pods >> "$OUTPUT_DIR/pods_metrics_wrk_${i}.txt"
    kubectl top nodes >> "$OUTPUT_DIR/nodes_metrics_wrk_${i}.txt"
}
```

Figure 35 - Tests with metrics detection



```
# Horizontal scaling
apply_horizontal_scaling() {
    #local i=$1 # Use this to scale manually
    printf "\n>>> Applying horizontal scaling... \n"
    #kubectl scale deployment dogecoin-forecasting --replicas=${i}  # Horizontal scaling in manual mode
    kubectl autoscale deployment dogecoin-forecasting --cpu-percent=80 --min=3 --max=8 # Horizontal scaling with autoscaler
    echo "Horizontal scaling applied."
}

# Vertical scaling
apply_vertical_scaling() {
    printf "\n>>> Updating YAML file for vertical scaling... \n"
    cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dogecoin-forecasting
  labels:
    app: dogecoin-forecasting-app
  namespace: default
spec:
  replicas: 7
  selector:
    matchLabels:
      app: dogecoin-forecasting-app
  template:
    metadata:
      labels:
        app: dogecoin-forecasting-app
    spec:
      containers:
        - name: dogecoin-forecasting
          image: kiritoemo6/dogecoin-forecasting
          resources:
            requests:
              memory: "512Mi"
              cpu: "500m"
            limits:
              memory: "750Mi"
              cpu: "1"
EOF
    echo "Vertical scaling applied."
}
```

Figure 36 - Testing scaling

7.4. Test Execution

After performing the steps to configure the entire Kubernetes cluster, you need to make sure that the cluster is running with 3 worker nodes and 3 pods (due to the 3 replicas set in the deployment YAML file) via the following commands:

- `kubectl get pods`
- `kubectl get services`

After doing this, it is possible to launch the bash script to automate the tests both for the base configuration of the cluster and for the scaled version of it, and to collect the metrics related to each step of the testing phase.

```
# Full execution of tests
main() {
    printf "\n>>> Starting stress tests and data collection..."

    # Phase 1: Collect initial metrics
    collect_metrics 0

    # Phase 2: Test with Apache Benchmark
    run_ab_test 0
    collect_metrics_ab 0

    # Phase 3: Test with wrk
    run_wrk_test 0
    collect_metrics_wrk 0

    # Phase 4: Horizontal scaling
    apply_horizontal_scaling
    sleep 30  # Wait for pods to stabilize
    collect_metrics 1
    run_ab_test 1
    collect_metrics_ab 1
    run_wrk_test 1
    collect_metrics_wrk 1

    # Phase 5: Vertical scaling
    apply_vertical_scaling
    sleep 30  # Wait for pods to stabilize
    collect_metrics 2
    run_ab_test 2
    collect_metrics_ab 2
    run_wrk_test 2
    collect_metrics_wrk 2

    printf "\n>>> Stress tests completed. Results saved in $OUTPUT_DIR"
}

# Start of the script
main
```

Figure 37 - Main part of script for testing

7.5. Test Results

7.5.1. Test Results with NodePort Service

Regarding the tests carried out during this phase, good results were found in general, reaching however the best optimization with horizontal scaling with 7 replicas, while vertical scaling proved to be excessive compared to what was required by the assignment.

```
This is ApacheBench, Version 2.3 <$Revision: 1923142 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient)


Server Software:        TornadoServer/6.4.2
Server Hostname:        127.0.0.1
Server Port:            30070

Document Path:          /
Document Length:        1837 bytes

Concurrency Level:      500
Time taken for tests:   16.108 seconds
Complete requests:      50000
Failed requests:        0
Total transferred:      111150000 bytes
HTML transferred:       91850000 bytes
Requests per second:    3104.07 [#/sec] (mean)
Time per request:       161.079 [ms] (mean)
Time per request:       0.322 [ms] (mean, across all concurrent requests)
Transfer rate:          6738.63 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    2   3.5      1      46
Processing:     2  158 182.4    106    3543
Waiting:        2  157 182.4    106    3541
Total:          2  159 182.3    109    3544

Percentage of the requests served within a certain time (ms)
  50%    109
  66%    166
  75%    210
  80%    241
  90%    328
  95%    416
  98%    521
  99%    701
 100%   3544 (longest request)
```

Figure 38 - ab test result after horizontal scaling to 7 replicas with NodePort Service

```
Running 30s test @ http://127.0.0.1:30070/
  8 threads and 5000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   201.32ms   30.93ms 379.12ms   75.52%
    Req/Sec   629.91    209.17     1.47k    69.83%
  150561 requests in 30.07s, 319.19MB read
  Socket errors: connect 3987, read 0, write 0, timeout 0
Requests/sec:   5006.98
Transfer/sec:     10.61MB
```

Figure 39 - wrk test result after horizontal scaling to 7 replicas with NodePort Service

Looking at the **test results** we see that:

- with *ab* we know that the server can handle 3,104 requests/sec with a concurrency level of 500 and that the average time per request is 161ms.
- with *wrk*, on the other hand, we see that with 5,000 concurrent connections, the server handled 5,007 requests/sec, an improvement over the ab test (this suggests that the server can scale better with a higher number of connections); the average latency is 201ms, with a maximum of 379ms. This is slightly higher than the ab test, but understandable given the much higher number of connections.

From these two tests, under these conditions, we can deduce:

- **Overall performance**: the cluster is able to handle a high load (up to 5,000 requests per second), but with increased latency and some connection errors (due to the large number of connections that the host PC cannot handle).
- **Scalability**: the server scales well up to a certain point but may need optimizations to handle a very large number of concurrent connections (e.g. increase the operating system limits or optimize the application).

7.5.2. Test Results with NGINX Ingress Controller

Regarding the adoption of NGINX, we performed two types of tests. The first was carried out in the same conditions as before (with NodePort as Service), so a fixed horizontal scaling at 7 replicas and then the vertical scaling combined with the horizontal. From this test we obtained the following results:

```
This is ApacheBench, Version 2.3 <$Revision: 1923142 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)


Server Software:
Server Hostname:        localhost
Server Port:            80

Document Path:          /
Document Length:        1837 bytes

Concurrency Level:      500
Time taken for tests:   16.846 seconds
Complete requests:      50000
Failed requests:        0
Total transferred:      110650000 bytes
HTML transferred:       91850000 bytes
Requests per second:    2968.01 [#/sec] (mean)
Time per request:       168.463 [ms] (mean)
Time per request:       0.337 [ms] (mean, across all concurrent requests)
Transfer rate:          6414.25 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    3   3.9      1      45
Processing:    17  165  46.3    156     532
Waiting:        5  163  46.0    155     532
Total:         27  167  45.8    159     534

Percentage of the requests served within a certain time (ms)
  50%    159
  66%    173
  75%    185
  80%    194
  90%    223
  95%    254
  98%    293
  99%    330
 100%    534 (longest request)
```

Figure 40 - ab test result after horizontal + vertical scaling combined

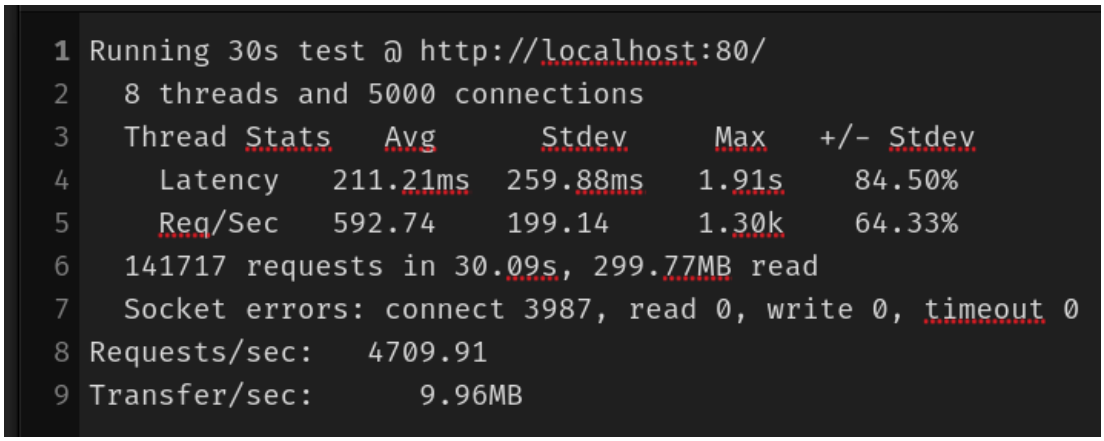


Figure 41 - wrk test result after horizontal + vertical scaling combined

Looking at the **test results** we see that:

- with *ab* we know that the server can handle 2968 requests/sec with a concurrency level of 500 and that the average time per request is ~168.5ms.
- with *wrk*, on the other hand, we see that with 5,000 concurrent connections, the server handled ~4710 requests/sec, an improvement over the ab test (this suggests that the server can scale better with a higher number of connections); the average latency is 211 ms.

From these tests, we can deduce:

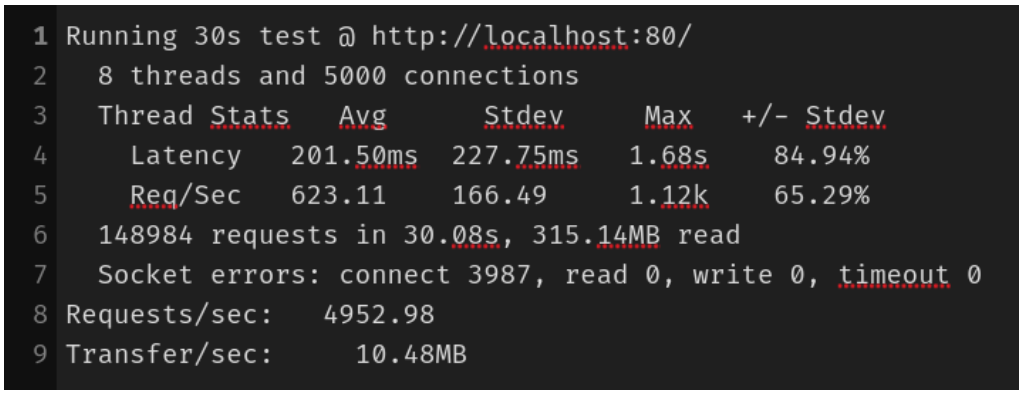
- **Overall Performance**: the cluster can handle a substantial load (up to ~4,700 requests/sec), but latency increases significantly under extreme concurrency, and connection errors occur due to system limitations.
- **Scalability**: the server scales well up to a point, but performance degrades with very high concurrency. Optimizations may be needed to handle such scenarios, such as increasing system limits or tuning the application.

7.5.3. Test Results with NGINX Ingress Controller and HPA

While the second set of tests was performed using the horizontal autoscaler (HPA) with a variable range from a minimum of 3 to a maximum of 8 replicas and then a vertical scaling, which did not undergo any changes, also this time combined with the horizontal. The obtained results are shown below.

```
This is ApacheBench, Version 2.3 <$Revision: 1923142 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)


Server Software:
Server Hostname:        localhost
Server Port:            80

Document Path:          /
Document Length:        1837 bytes

Concurrency Level:      500
Time taken for tests:   16.005 seconds
Complete requests:      50000
Failed requests:        0
Total transferred:      110650000 bytes
HTML transferred:       91850000 bytes
Requests per second:    3124.00 [#/sec] (mean)
Time per request:       160.051 [ms] (mean)
Time per request:       0.320 [ms] (mean, across all concurrent requests)
Transfer rate:          6751.38 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    3   4.0      2      45
Processing:    19  156  27.1    155     371
Waiting:        4  155  26.8    153     364
Total:         49  159  26.0    157     371

Percentage of the requests served within a certain time (ms)
  50%    157
  66%    166
  75%    172
  80%    177
  90%    191
  95%    205
  98%    222
  99%    237
 100%    371 (longest request)
```

Figure 42 - ab test result after HPA + vertical scaling combined



```
Running 30s test @ http://localhost:80/
  8 threads and 5000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   201.50ms  227.75ms   1.68s    84.94%
    Req/Sec   623.11    166.49     1.12k    65.29%
  148984 requests in 30.08s, 315.14MB read
  Socket errors: connect 3987, read 0, write 0, timeout 0
Requests/sec:   4952.98
Transfer/sec:     10.48MB
```

Figure 43 - wrk test result after HPA + vertical scaling combined

Looking at the test results, we can observe the following:

- With **ApacheBench (ab)**, the server handled **3,124 requests/sec** with a concurrency level of 500, and the average time per request was **160.05ms**. The longest request took **371ms**, and 99% of the requests were served within **237ms**. This indicates strong performance under moderate concurrency, with relatively low latency and no failed requests.
- With **wrk**, using **5,000 concurrent connections**, the server managed **4,952.98 requests/sec**, which is significantly higher than the ab test, demonstrating better scalability under extreme load. However, the average latency increased to **201.50ms**, with a maximum latency of **1.68 seconds**. This is expected given the much higher number of concurrent connections.

From these tests, we can deduce:

- **Overall Performance**: The cluster can handle a substantial load (up to ~4,950 requests/sec), but latency increases under extreme concurrency. The server performs well under moderate load, with no failed requests and consistent response times.
- **Scalability**: The server scales effectively under higher concurrency, as evidenced by the *wrk* test results. However, the connection errors and increased latency suggest that optimizations may be needed to handle such extreme scenarios, such as increasing system limits or tuning the application.

Figure Index