PROJECT WORK

ETH Prediction

Team 6

Mancusi	Emanuele	0622702062
Rabasca	Dario	0622702003

Contents

Machine Learning

Introduction

The first phase is concentrated on the exploration, training, and deployment of a machine learning model, emphasizing both Classification and Time Series Data forecasting. Three distinct machine learning models were trained to identify the most suitable one for accurate predictions. The following chapters carefully detail each step of the process, providing a detailed account of the exploration, training, and deployment processes.

Database

The project utilizes a database for Ethereum price data, making it convenient to fetch information using the yfinance library in Python. This ensures organized data management and quick access to historical Ethereum prices. Additionally, it allows for model training on regularly updated data, providing a seamless integration of yfinance with the database for efficient handling of Ethereum prices.

Linear Regression

The first model trained is a linear regression model and, despite it may not capture more complicated patterns in how Ethereum prices move, it's easy to understand and gives a basic overview of trends in the data. For these reasons, the choice of the best model was made considering both the strengths and limitations of this model for predicting Ethereum prices.

Before starting the actual training process, the initial step involved preprocessing the data. This involved preparing and organizing the raw data to render it suitable for model training. One crucial aspect of this preprocessing was the removal of redundant columns, such as "Adj Close" and "Volume", as they exhibited no significant correlation with other features. Especially, "Adj Close" had the same information present in the "Close" column.

```python
def transform_data(df):

    features = ["Open", "High", "Low"]

    transformer = ColumnTransformer(
        transformers=[
            ('features', make_pipeline(
                SimpleImputer(missing_values=np.nan, strategy='median'),
                StandardScaler()
            ), features)
        ],
        remainder='passthrough'
    )

    X = transformer.fit_transform(df[features])

    dump(transformer, args.data)
    return X
```

Figure 1. Transform function for linear regression.

Then, a transform function was applied to the data to enhance the quality and usability of the dataset. The use of **SimpleImputer** helps handle missing values by filling them with a suitable strategy, in this case, the median. Standardizing the features using **StandardScaler** ensures that all variables contribute equally to the model, preventing the dominance of certain features due to differing scales. The **ColumnTransformer** encapsulates these operations and applies them selectively to the specified features.

```python
def linear_regression(args):

    data = None
    with open(args.data) as data_file:
        data = json.load(data_file)

    x_train, x_test, y_train, y_test = data['x_train'], data['x_test'], data['y_train'], data['y_test']

    param_grid={'fit_intercept':(True, False), 'copy_X':(True, False)}

    model = LinearRegression()
    grid_search = GridSearchCV(model, param_grid, n_jobs=-1, scoring='r2', cv=5)

    grid_search.fit(x_train, y_train)

    model = grid_search.best_estimator_

    dump(model, args.model)

    y_pred = model.predict(x_test)

    y_pred = y_pred.reshape(-1,1)
    y_test = np.array(y_test).reshape(-1,1)

    y_pred = StandardScaler().fit_transform(y_pred)
    y_test = StandardScaler().fit_transform(y_test)
    score = r2_score(y_test, y_pred)

    with open(args.r2, 'w') as score_file:
        score_file.write(str(score))
```

Figure 2. Linear regression model.

In the training phase, after loading the dataset from a specified file and splitting it into training and testing sets, was conducted the hyperparameter tuning using GridSearchCV to optimize the linear regression model. The best-performing model is then saved. After that, the function predicts the target values on the test set and evaluates its performance using the R-squared (r2) score.

Random Forest Regressor

The choice of a Random Forest Regressor is based on its ability to handle complex relationships within the data, capture non-linear patterns, and mitigate overfitting. The model was trained on historical ETH price data using the same preprocessing and transform functions used for the previous model, leveraging the features such as opening and closing prices.

```python
def random_forest_regressor(args):

    with open(args.data) as data_file:
        data = json.load(data_file)

    x_train, x_test, y_train, y_test = data['x_train'], data['x_test'], data['y_train'], data['y_test']

    parameter_grid={'n_estimators':[64, 128, 256],
                    'max_depth':[2, 4, 8, 16, 36, 64]}

    model = RandomForestRegressor(random_state=42)

    # Create GridSearchCV
    grid_search = GridSearchCV(model, parameter_grid, n_jobs=-1, scoring='r2', cv=5)

    grid_search.fit(x_train, y_train)

    model = grid_search.best_estimator_
    dump(model, args.model)

    y_pred = model.predict(x_test)

    y_pred = model.predict(x_test)

    y_pred = y_pred.reshape(-1,1)
    y_test = np.array(y_test).reshape(-1,1)

    y_pred = StandardScaler().fit_transform(y_pred)
    y_test = StandardScaler().fit_transform(y_test)
    score = r2_score(y_test, y_pred)

    with open(args.r2, 'w') as score_file:
        score_file.write(str(score))
```

Figure 3. Random Forest Regressor.

The hyperparameter tuning is carried out through GridSearchCV, exploring combinations for the number of estimators and maximum depth as shown in the Figure 3. After the training, the performance of the model is evaluated using the R-squared score, providing a measure of how well the predictions align with the actual Ethereum prices.

LSTM

The last model used for predicting Ethereum prices is an LSTM (Long Short-Term Memory) neural network model. Unlike traditional machine learning models, LSTMs are a type of recurrent neural network (RNN) specifically designed to capture long-term dependencies in sequential data, making them well-suited for time series forecasting tasks such as predicting cryptocurrency prices.

In the preprocessing phase, the 'Date' column was converted to the datetime format, and the data were arranged in chronological order to ensure a proper time series sequence. Following that, in the transform function, a MinMaxScaler was employed to standardize the 'Close' column of the DataFrame.

```python
def lstm():

    window_size = 60
    input1 = Input(shape=(window_size,1))
    x = LSTM(units = 64, return_sequences=True)(input1)
    x = Dropout(0.2)(x)

    x = LSTM(units = 64, return_sequences=True)(x)
    x = Dropout(0.2)(x)

    x = LSTM(units = 64)(x)
    x = Dropout(0.2)(x)
    x = Dense(32, activation='linear')(x)
    dnn_output = Dense(30)(x)

    model = Model(inputs=input1, outputs=[dnn_output])
    model.compile(loss='mean_squared_error', optimizer='Adam', metrics=[r2_score], run_eagerly=True)
    model.summary()

    return model
```

Figure 4. LSTM model.

The model consists of three LSTM layers, each followed by a dropout layer to mitigate overfitting. The input layer expects sequences of length 'window_size' with one feature. The LSTM layers progressively capture temporal dependencies, the final Dense layer with linear activation refines the representation, and the output layer provides predictions for the next 30 values. This architecture is configured to minimize the mean squared error loss using the Adam optimizer.

The LSTM model, defined by the 'lstm()' function, is then trained on the specified number of epochs, with a designated batch size and validation split. Subsequently, the trained model is saved using the 'dump()' function from the joblib library and a performance evaluation is conducted on the test data, including the calculation of the r2-squared.

Kubeflow

Kubeflow is an open-source, Kubernetes-native framework for developing, managing, and executing machine learning (ML) workloads. It addresses many challenges associated with orchestrating ML pipelines by providing a set of tools and APIs that simplify the training and deployment of ML models at scale. In essence, Kubeflow streamlines ML operations (MLOPS) by managing projects and harnessing the benefits of cloud computing. Its comprehensive features make it easier to standardize ML operations while leveraging the advantages of the cloud. Some key use cases of Kubeflow include data preparation, training, evaluation, optimization, and model deployment.

Figure 5. Kubeflow logo.

Pipelines

A pipeline within the context of machine learning serves as a comprehensive plan for executing a series of steps, and in this scenario, the Kubeflow platform is employed for orchestration. Utilizing the code derived from the Machine Learning chapter, seven specific pipelines were meticulously crafted and subsequently integrated into Kubeflow for streamlined execution:

- Data Download: This initial step is crucial for obtaining an updated database, achieved through the utilization of the **yfinance** library. It ensures that the model is trained on the latest and relevant data.

- Preprocessing for Linear Regression and Random Forest Models: This phase involves preparing the data to suit the requirements of the Linear Regression and Random Forest models, addressing tasks such as handling missing values, scaling features, and organizing data structures.

- Preprocessing for LSTM Model: Similar to the previous step, this preprocessing stage is tailored specifically for the Long Short-Term Memory (LSTM) model, considering its unique architecture and data input requirements.

- Training the Linear Regression Model: The pipeline incorporates a dedicated stage for training the Linear Regression model, enabling it to learn patterns and relationships within the preprocessed data.

- Training the Random Forest Model: Similar to the Linear Regression model, this step involves training the Random Forest Regressor model, leveraging an ensemble of decision trees to enhance predictive accuracy.

- Training the LSTM Model: The LSTM model, known for its effectiveness in handling sequential data, undergoes specialized training in this phase, ensuring it captures temporal dependencies within the dataset.

- Show Result: This final step is important for checking and understanding how well the trained models performed. It helps in making decisions about which model is best for future predictions. Looking at the results and analysing them provides valuable information for choosing the most effective model.

After the creation of those pipelines, using a Python script made it possible to generate the corresponding .yaml file to be uploaded on Kubeflow for streamlined execution and management of machine learning workflows.

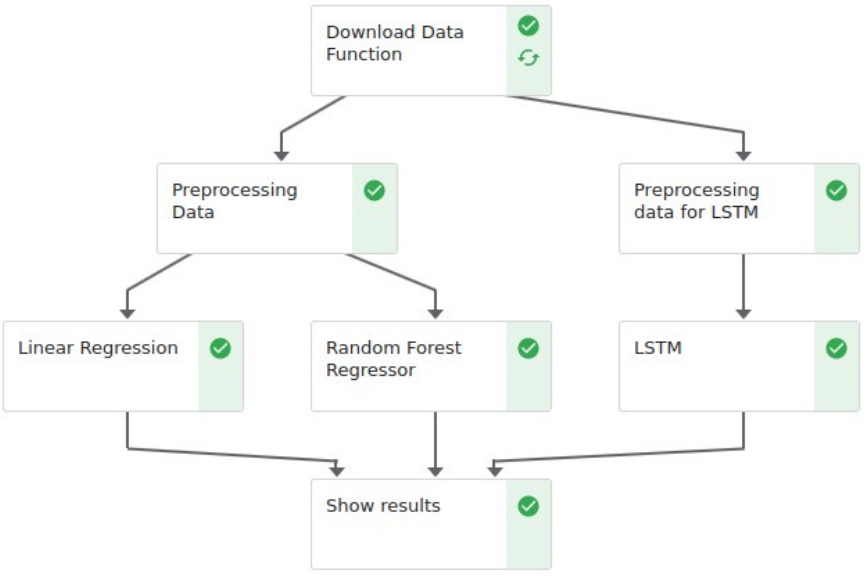

Figure 6. Kubeflow pipelines.

Optimizing Performance

Vineyard is a cloud-native, in-memory data manager designed for efficient sharing of intermediate data through the use of shared memory, the most effective medium. As a CNCF sandbox project, Vineyard serves as an in-memory object manager, utilizing shared memory to facilitate copyless data exchange between diverse tasks. As the size of intermediate data grows significantly, managing the heavy load of input/output (IO) operations can become challenging, leading to a decline in the efficiency of read and write operations.

In such scenarios, adopting more efficient approaches, such as those provided by Vineyard, becomes crucial. By leveraging Vineyard, the cost of data sharing can be substantially optimized, resulting in a significant end-to-end performance improvement. This improvement becomes even more evident as the volume of intermediate data scales up.[1]

In the context of training the models presented in the previous chapter, this solution may not be necessary as the data passed through the pipelines is not large enough to gain the described benefits. However, in a future scenario where the data for training grows, adopting Vineyard could become a very useful strategy to enhance performance.

[1] https://medium.com/cncf-vineyard/optimizing-data-sharing-in-kubeflow-pipelines-358be85bc00e

Kubernetes

Kubernetes, often abbreviated as K8s, is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. By utilizing Kubernetes as the foundation, Kubeflow inherits the scalability, fault tolerance, and container orchestration capabilities of Kubernetes. This integration enables users to deploy and scale machine learning applications in a consistent and scalable manner, leveraging the power and flexibility of Kubernetes to manage the entire machine learning lifecycle. In essence, Kubernetes provides the robust infrastructure on which Kubeflow builds and extends its capabilities for ML workflow management.

Figure 7. Kubernetes logo.

Target of the Application

The application is designed for investment management agencies, aiming to enhance investment strategies for their clients. This is particularly relevant in the context of cryptocurrencies, such as Ethereum (ETH), where predictive tools and indicators can be helpful for anticipating market trends and formulating effective investment strategies.

Regarding the user base, it's reasonable to anticipate a relatively low number of regular users, as the application is specifically designed for investment management agencies, which are typically fewer in number compared to individual users. However, given the inherent volatility of the cryptocurrency market, notable spikes in usage may occur at specific times of the day or during particular periods, such as during an upgrade of ETH protocol.

This indicator serves as a tool to assist individuals in making investment decisions. However, it's important to note that investment decisions should be based on a comprehensive analysis that takes into account various factors beyond those included in the application. Additionally, it doesn’t give you financial advice, we decline any responsibility for any investment made.

Deployment on Kubernetes

Deploying the application on Kubernetes with multiple replicas has enhanced its reliability, availability, scalability, and, notably, fault tolerance. This is particularly crucial as the failure of a single server will not result in the unavailability of the entire application. This configuration enables the system to handle increased loads, ensuring a robust and responsive performance. Furthermore, deploying the app on Kubernetes with appropriate load balancing configurations can efficiently handle the anticipated service requests from a large user base.

Selecting the right number of replicas for the web application is fundamental in achieving the goal of serving all customers, especially during high-traffic peaks. The replica count determines the simultaneous instances of the application running. In a Kubernetes deployment, dynamically adjusting the replica count based on demand is a key strategy for ensuring scalability and responsiveness. To achieve this, the number of replicas will be dynamically managed using KEDA, a Kubernetes-based Event-Driven Autoscaler designed to scale any container in Kubernetes based on the number of events that need processing.

Specifically, the replica count will vary between 2 and 10 to handle fluctuations in HTTP requests, thereby enhancing the availability and resilience of the application by creating new instances when there are at least 100 requests in pending state. It is chosen the threshold of 100, because setting the threshold too low may result in unnecessary scaling events, leading to additional resource consumption and potentially increased costs. On the other hand, setting it too high may result in delayed scaling responses, impacting the responsiveness of the application during periods of increased demand.

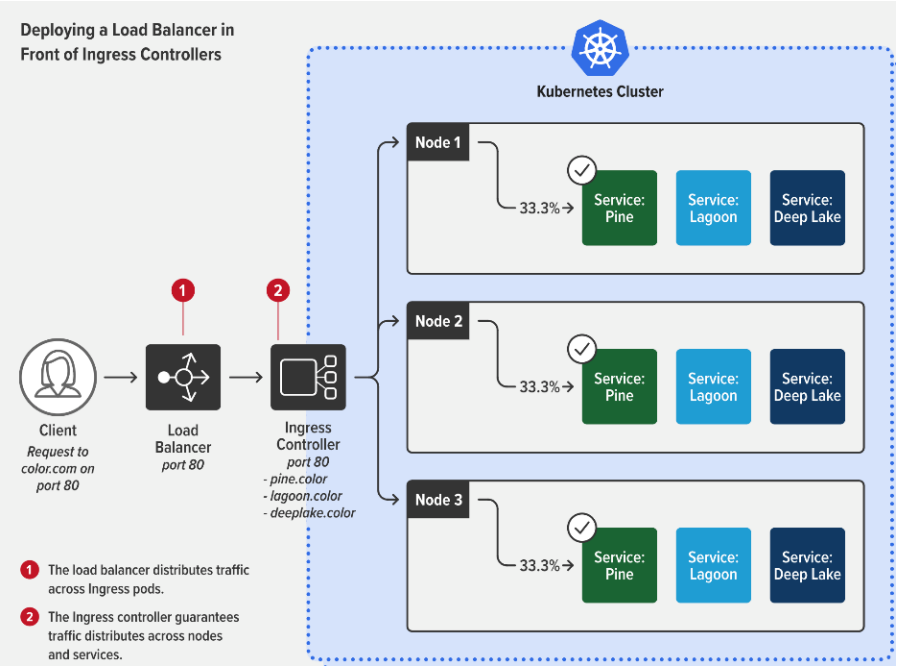

Figure 8. Load Balancer and Ingress Controller.

To expose the application, it has been utilized a NodePort, a Kubernetes service type that opens a specific port on every node in a cluster. This allows external traffic to reach a pod running on any of those nodes by associating the IP of the pod with a designated port, such as HTTP.

The problem with this type of Service is that the available ports are in the 30000 range which is not appropriate for Internet traffic usually arriving on ports 80 or 443 because firewalls would easily block this port on the Internet.

By doing so, it also creates a SPOF (Single Point of Failure) because it relies on a single node. To avoid this behaviour, there is the need to have a public IP address for each pod, as well as dealing with balancing traffic and configuring health checks. In the end, there is the need of building a Load Balancer in front of the Kubernetes nodes. This is why a "Load Balancer" service is useful to expose pods to the outside world. It must be configured in front of a NodePort service, and it becomes an entry point to the Kubernetes cluster that forwards traffic to the pods.

Unlike Load Balancers, Ingress Controllers handle SSL/TLS certificates directly. They allow to define routing rules, including specifying which certificates to use for different domains. The ingress Controller will be dealing with in the Security chapter.

The chosen number of worker nodes is 3 to enhance system reliability and ensure high availability, and the cluster will be set up on **virtual machines** for the following reasons: greater flexibility in resource allocation and scaling, isolation between workloads, which is advantageous for security as issues with one virtual machine would not affect others, and elasticity. Additionally, the use of virtual machines simplifies system maintenance and allows easy management of updates, patches, and configurations on each virtual machine independently.

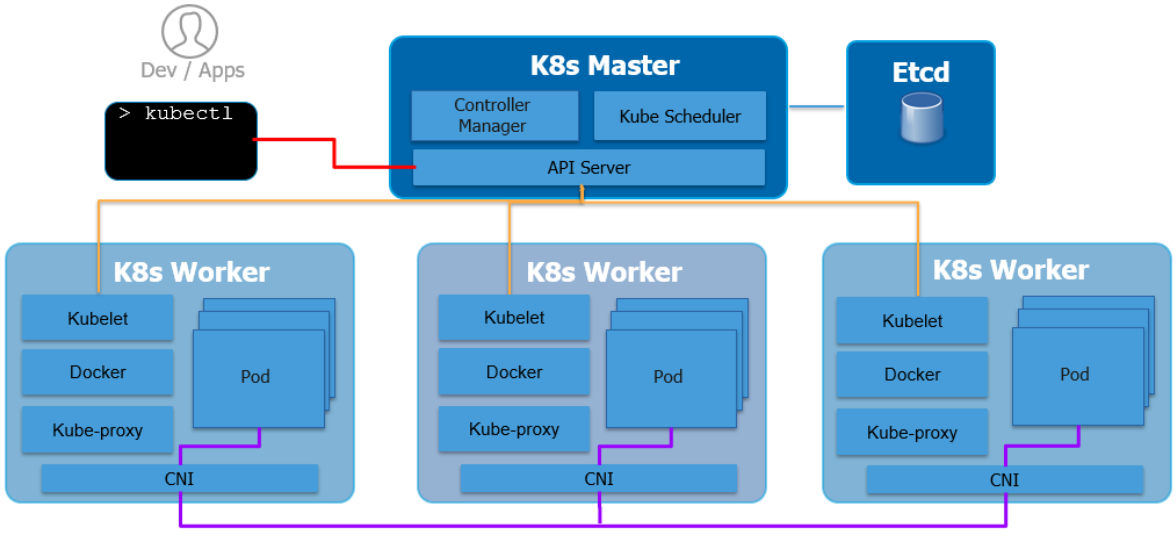

Figure 9. Kubernetes architecture.

Leveraging virtual machines for inference provides a cost-effective solution as it allows payment solely for the resources utilized during the inference process. This flexibility in cost management is particularly advantageous, ensuring financial efficiency based on actual resource usage. Moreover,

using VMs allows for easy adjustments to handle increases in requests without having to invest a lot in fixed resources. On the other hand, opting for physical hardware involves upfront fixed costs and demands meticulous planning concerning workload fluctuations and, even if they can deliver optimal performance during inference tasks, they might prove to be overprovisioned if the demand is highly variable as in this case.

Figure 10. Vertical Scaling.

If traffic to a web application is low during off-peak hours, it may be more cost-effective to add more resources to a single pod than to keep multiple pods running. By adding more resources (such as CPU or memory) to a single pod, it is possible to handle increased demand without the overhead of maintaining multiple pods. So, in this scenario, vertical scaling during off-peak hours can help manage traffic fluctuations while also keeping costs down. For this reasons, vertical scaling establishes a balance between resource availability and cost efficiency.

Vertical Pod Autoscaler (VPA) frees users from the necessity of setting up-to-date resource limits and requests for the containers in their pods. When configured, it will set the requests automatically based on usage and thus allow proper scheduling onto nodes so that appropriate resource amount is available for each pod. It will also maintain ratios between limits and requests that were specified in initial containers configuration.

It can both down-scale pods that are over-requesting resources, and also up-scale pods that are under-requesting resources based on their usage over time. The Kubernetes Vertical Pod Autoscaler automatically adjusts the CPU and memory reservations for Pods to help "right size" the application. This adjustment can improve cluster resource utilization and free up CPU and memory for other Pods.[2] Since the application performs inference using the LSTM model, it's essential in this case to

[2] https://www.giantswarm.io/blog/vertical-autoscaling-in-kubernetes

control both CPU and memory resources to effectively manage the growing demand for Ethereum predictions. This ensures that all requests are handled correctly, thereby guaranteeing the reliability of the application.

Inference

```python
def inference(scaler, model):

    end = datetime.now()
    start = end - timedelta(days=365)  # or any other suitable time range

    # Download data
    df = yf.download("ETH-USD", start, end)
    window_size = 60

    test_data = df.Close.values.reshape(-1, 1)
    test_data = scaler.fit_transform(test_data)

    # Prepare input sequence for prediction
    input_sequence = test_data[-window_size:].reshape(1, window_size, 1)
    # Get the model's predicted price values
    predictions = model.predict(input_sequence)

    # Inverse transform the predictions to the original scale
    predictions = scaler.inverse_transform(predictions)


    return predictions
```

Figure 11. Inference code.

The inference function is crafted to predict future Ethereum (ETH) prices utilizing a pre-trained LSTM model selected in the preceding step. The LSTM model was chosen due to its effectiveness in capturing long-term dependencies in sequential data, making it the optimal choice among the three models evaluated. It begins by downloading historical ETH price data within the last year in order to let the model continues it to predict future prices. The data is then transformed, with the 'Close' prices being reshaped and scaled using the provided scaler object. A window of the last 60 data points is prepared as an input sequence for the model. The pre-trained model predicts future prices based on this sequence, and the results are inversely transformed to their original scale using the scaler. The final output consists of predicted ETH prices, making this function a convenient tool for forecasting cryptocurrency prices based on historical data and a pre-existing machine learning model.

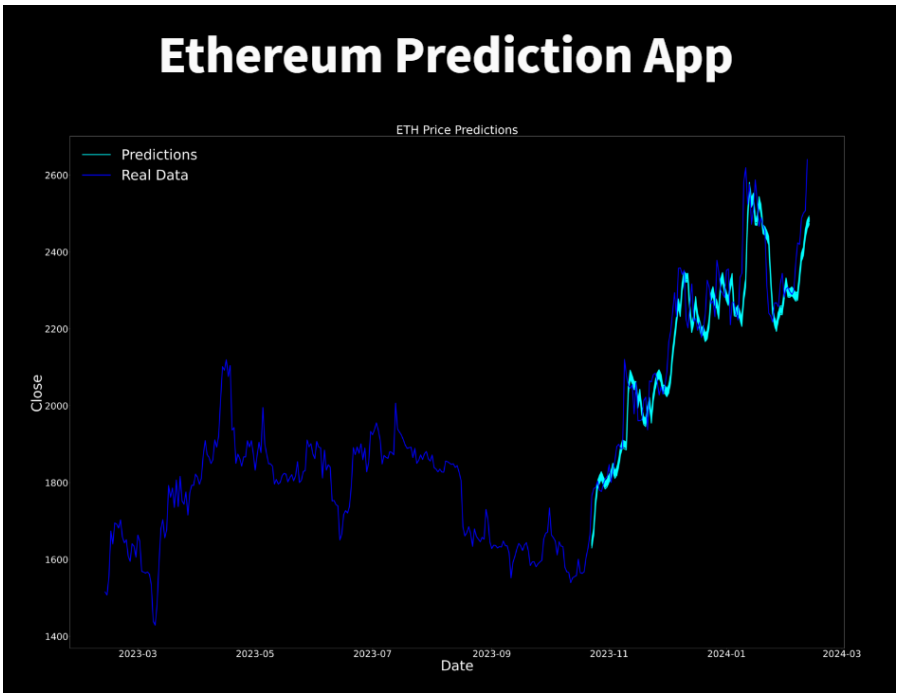

Figure 12. Test Accuracy.

To illustrate the model's effectiveness in predicting the ETH price, Figure 12 displays the variance between the actual data and the predicted values. Obviously, the actual app displays a live plot featuring future ETH price predictions, giving a sneak peek into upcoming trends as illustrated below.

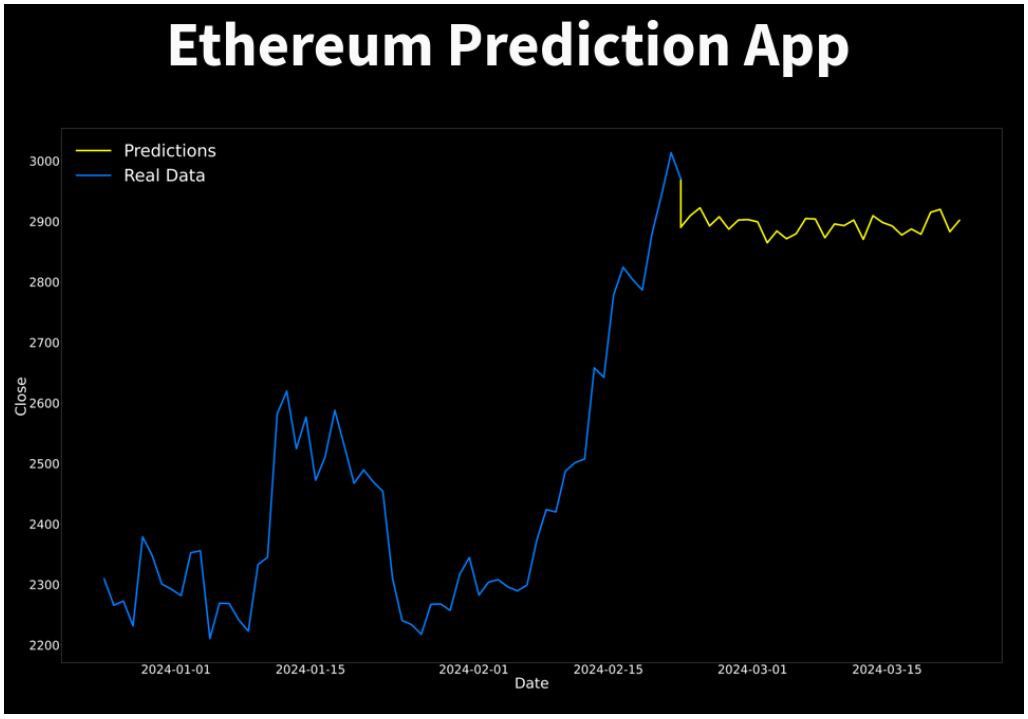

Figure 13. Ethereum Prediction App.

Security

Introduction

Most of the external attack are carried out from the Web applications, in particular, the Web application can suffer from SQL Injection, Cross-site Scripting or remote file inclusion, but also because of Software vulnerabilities. An increasing percentage is represented from DDoS, typically involving a ransom or to cover other attacks.

Both microservices based on applications in Kubernetes and monolithic applications running on a virtual machine suffer from application layer vulnerabilities exploit and application layer DDoS attacks. Moreover, Kubernetes can't directly detect malware inside container images or alert about anomalous behaviour. For this reason, it should be used external tools for protecting the system.

Protection

The protection phase starts from decrypting the packets, analysing both the Request and the Response and then, with the inspection of those, a decision is taken basing on the fact that something bad is happening or not. The last step is represented from the ACT, in which it is decided if to block the traffic, to trigger an alarm or to do nothing.

The first thing to recognize a DDoS attack is to check for changes in the application behaviour, for example, during slow post attack, when a valid request is trying to upload a large quantity of data slowly. It is important in this circumstance to create a baseline of how things should be working, in order to be able to detect situations such as when the time response have gone down or the type of traffic has changed.

Solution

A solution can be used to mitigate these types of attacks is to use two different modules that can be added in Kubernetes such the app protect Web Application Firewall (WAF) and the app protect DoS. The first one allows to look for attacks such as Cross-site Scripting, Information Leakage, SQL Injections and other type of application layer attacks. Furthermore, the two things that kill application layer security solutions and that must be avoided are the complexity and false positives. When the solution becomes overly complex, it may lead to difficulties in implementation, maintenance, and understanding, potentially creating vulnerabilities. False positives, on the other hand, can erode trust in the system, as they may trigger unnecessary alerts, leading to wasted resources and decreased confidence in the accuracy of the security measures. Thus, it is crucial to design and implement application layer security solutions that strike a balance between robustness and simplicity, minimizing false positives and avoiding unnecessary complexity for optimal performance.

When an anomaly is found, it is necessary to examine the user behaviour, so the different traffic that is coming in, the patterns, where it is coming from etc. A good solution is to dynamically create the rules to spot the bad traffic and to choose a mitigation between do nothing, block bad IP's, block bad requests and global rate limit.

Once an anomaly is detected from the baseline model, the first thing to do is to try and block some known bad IP's, also to not stop the good traffic and then, a control of the application behaviour is made to understand if the fix was successful. If not, the next level mitigation is taken (block bad request) and so on. The last level is represented by some global rate limiting on various requests with the possibility of blocking a part of good traffic too.

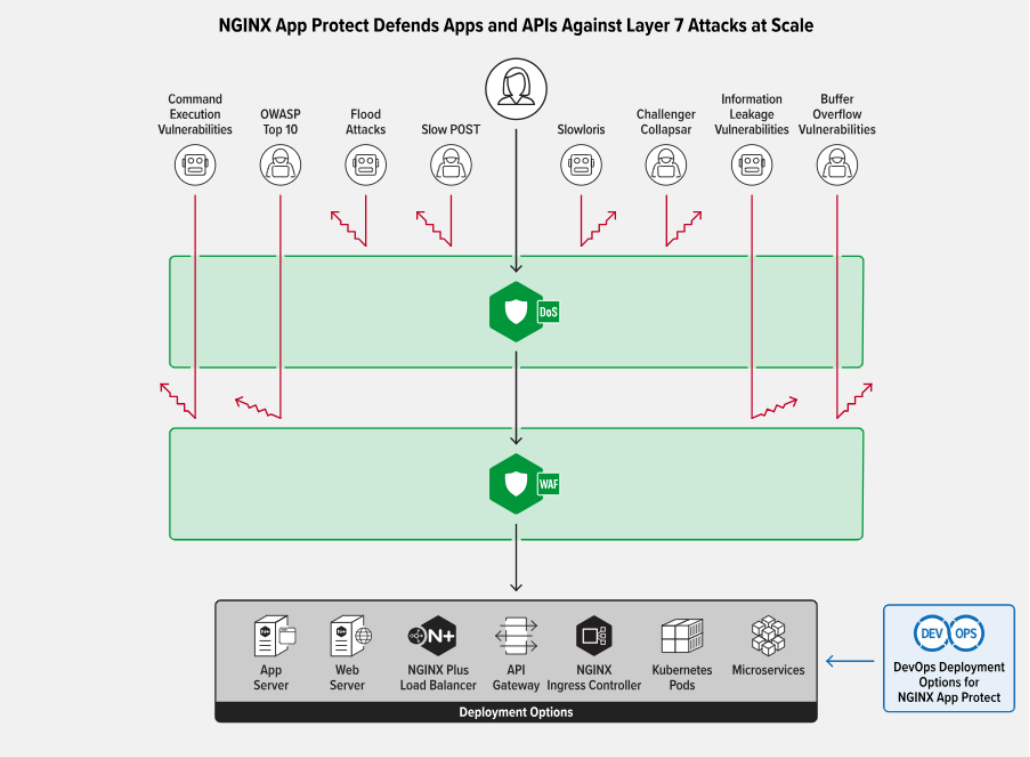

Figure 14. App protect DoS and WAF.

Protecting Apps from Hacks in Kubernetes

There are different ways to protect the application: the first one is the load balancer and by applying the NGINX app. It is possible to apply it also on the API gateway or in the Ingress Controller. An Ingress controller in Kubernetes acts as a traffic manager for incoming external requests. It helps route external traffic to the appropriate services within the Kubernetes cluster based on defined rules. The Ingress controller essentially acts as an entry point for external users or applications to access services inside the cluster.

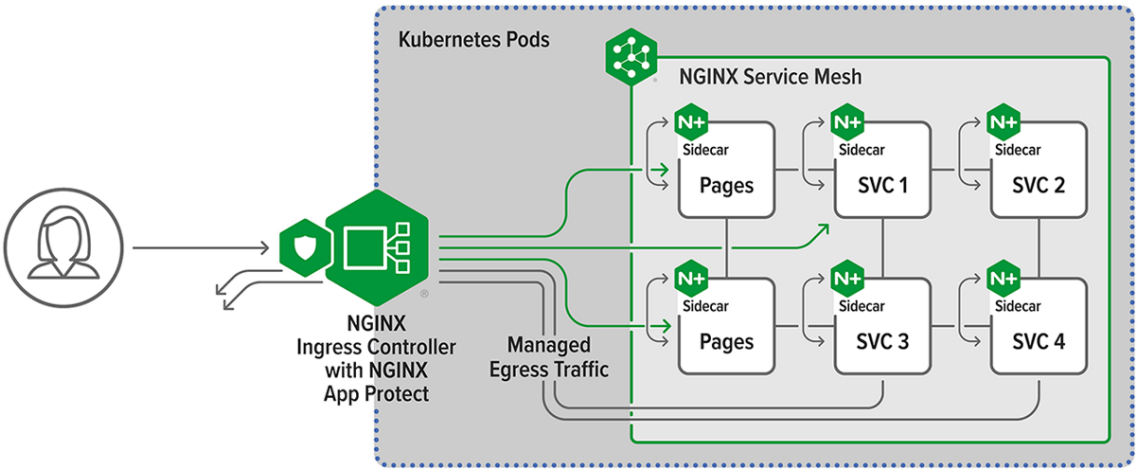

Figure 15. Ingress Controller with NGINX.

Having an Ingress controller, especially with NGINX, comes with several advantages. It helps simplify the setup, speeds up the time it takes to launch applications, improves resilience, provides better visibility, and enhances security. The use of NGINX Ingress Custom Resource Definitions (CRDs) brings additional benefits, such as modular and reusable objects. These objects enable the selection of policies like Access Control, Rate Limiting, and App Protect WAF, contributing to a more streamlined and less complex configuration.

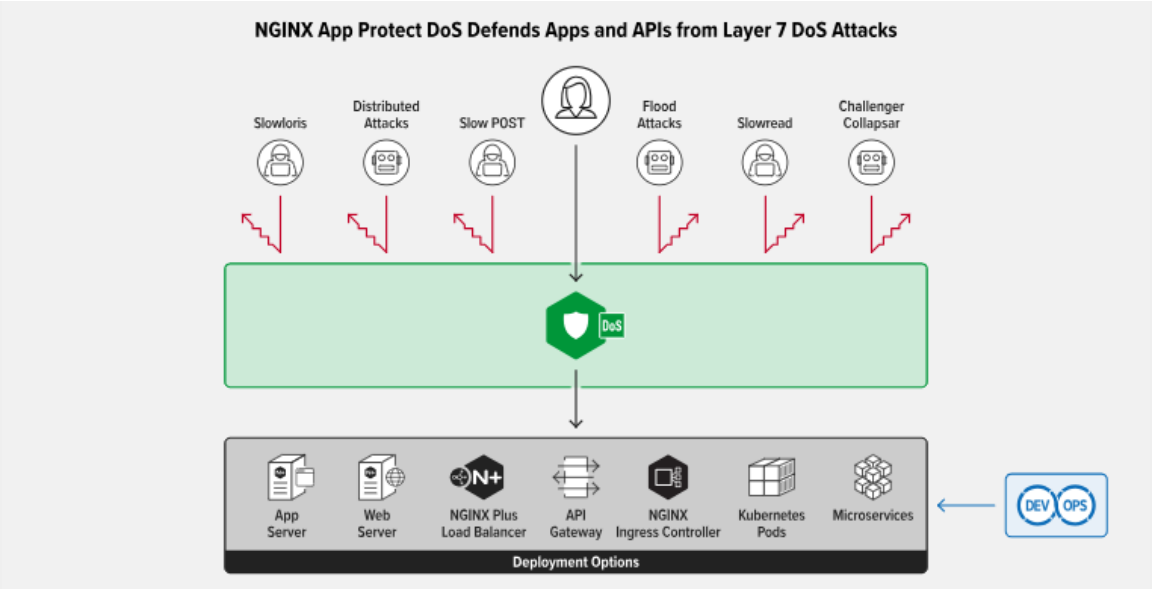

Figure 16. Protection of the application.

Due to the application's immunity to application layer attacks, such as SQL Injection or remote file inclusion, the primary concern shifts to potential Denial of Service (DoS) attacks. To address this threat, the selected solution involves implementing the NGINX Ingress Controller with App Protect DoS. Alternative solutions, such as WAF, are not beneficial in these situations. However, considering that the application may introduce new features that could pose additional threats, there might be a need to implement them for enhanced security.

Table of figures