

Università degli Studi di Salerno

Dipartimento di Ingegneria dell'Informazione ed Elettrica e Matematica Applicata



SECURE CLOUD COMPUTING

**Optimizing Application Lifecycle: An Investigation and Implementation of Kubeflow and
NGINX DoS Protect App Integration for Web Application Security on Kubernetes**

Prof. Angelo Marcelli
Prof. Antonio Della Cioppa

Pizzulo Rocco Gerardo
0622701990

Russo Luigi
0622702071

Anno Accademico 2023/2024

Contents

1	Introduction	3
1.1	Malicious URL Detection	3
2	Hardware Configuration and Technological Stack	5
2.1	Hardware specification	5
2.2	Technological Frameworks and Tools	5
2.3	Libraries	6
3	Machine Learning models development	7
3.1	Task	7
3.2	Experience	8
3.2.1	Features extraction	8
3.3	Models	10
3.4	Training and Validation	11
4	Cluster Configuration	12
4.1	Metric Server	14
4.2	Kubernetes Autoscaling	15
4.2.1	VPA	17
4.2.2	HPA	18
4.3	Kubeflow installation and setup	20
5	Kubeflow Pipeline	21
5.1	Securl pipeline	22
5.2	Pipeline building process	23
6	Securl	24
6.1	Web Application GUI	24
6.2	Deployment	25
7	Benchmark	26
8	Security Assessment	30
8.1	Threat Identification	30
8.2	NGINX App Protect DoS	31

CONTENTS

8.2.1 Configuration	32
9 Conclusions	33

Introduction

This report aims to document the process of developing, training and deploying a machine learning model that classifies URLs (Uniform Resource Location) with the goal of detecting malicious URLs. This task, commonly known as **Malicious URL detection**, will be performed by leveraging Kubeflow's main components.

A Kubernetes cluster comprises a group of worker machines, known as nodes, which execute containerized applications. Each cluster includes at least one worker node hosting the Pods, which represent the components of the application workload. The control plane oversees the worker nodes and the Pods within the cluster. Whether a node is virtual or physical depends on the specific cluster. In production environments, the control plane typically operates across multiple computers, and a cluster generally consists of multiple nodes, ensuring fault tolerance and high availability.

Note that the main objective of this work is not to develop a machine learning model that is as accurate as possible, but to create a scalable and resilient architecture that guarantees a **continuous integration and continuous deployment (CI/CD)** of machine learning workflows.

Definitively, the main goals of this work can be summarised in the following points:

- Demonstrate an End-to-End kubeflow example
- Present the ML model
- Build a web app able to perform the forecasting supposing that should serve a total number of 50.000 customers and, during the serving time, we can have a maximum of 5000 service requests at the same time.

1.1 Malicious URL Detection

Thousands of new websites are created every day that collect user data through login functions. The large number of networks makes it challenging to determine which websites are safe and reliable. In this vast digital landscape, malicious URLs, which are hyperlinks, stand as a prime tool that is exploited by cybercriminals to manipulate Internet users to

give out sensitive and personal information. By interacting with these links, users expose themselves to consequences, from compromising sensitive information to becoming prime targets for cyberattacks.

In this setting, machine learning can play a key role. In fact, as will be seen later on, by training models on large amounts of data, it is possible to achieve very good results in the detection of urls that have been specifically constructed to cause harm to uninformed users.

Hardware Configuration and Technological Stack

This chapter focuses on the description of the instrumentation used, starting with the hardware and ending with the software libraries used to develop the machine learning models.

2.1 Hardware specification

The hardware used in the development of the project is a Dell PC, specifically an XPS 13-7390. Below are the specifications of the device.

- **Device:** XPS 13-7390
- **CPU:** Intel® Core™ i7-10510U \times 8
- **Memory:** 16 GiB
- **OS:** Ubuntu 23.10

2.2 Technological Frameworks and Tools

- **kind:** a tool for running local Kubernetes clusters using Docker container “nodes”. kind was primarily designed for testing Kubernetes itself, but may be used for local development or CI.
- **Docker:** a platform designed to help developers build, share, and run container applications.
- **kubectl:** a command line tool for communicating with a Kubernetes cluster’s control plane, using the Kubernetes API.

2.3 Libraries

The following libraries were used in the development of the machine learning models and in the serving of the application. Note that they represent the requirements for the correct use of the application.

- **streamlit 1.31.0:** an open-source Python library that makes it easy to create and share beautiful, custom web apps for machine learning and data science. In just a few minutes you can build and deploy powerful data apps.
- **tld 0.13:** python library that extract the top level domain (TLD) from the URL given. List of TLD names is taken from Public Suffix [\[1\]](#).
- **kfp 1.8.22:** python library to compose a multi-step workflow (pipeline) as a graph of containerized tasks using Python code and/or YAML.
- **scikit-learn 1.3.2:** an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection, model evaluation, and many other utilities.
- **pandas:** a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.
- **joblib:** a set of tools to provide lightweight pipelining in Python.

Machine Learning models development

This chapter will explain the various steps that were performed during the development of the three machine learning models.

3.1 Task

A URL consists of the top-level domain, hostname, paths, and port of the web address, as in the following diagram:

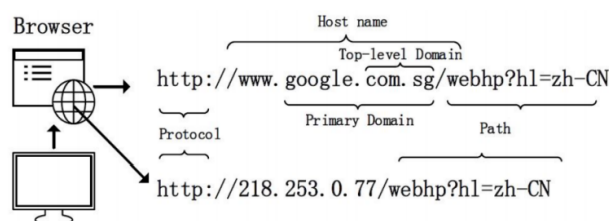


Figure 3.1: URLs structure

Malicious URL detection is an important task in the field of machine learning, particularly relevant in computer security and in the protection of computer networks and systems. This task consists of classifying URLs according to their nature, identifying those that could be malicious. In particular, the models developed propose to classify URLs into:

- **Benign or safe:** legitimate URL that has no malicious purpose.
- **Phishing:** associated with phishing practices, which are fraudulent attempts to obtain personal or sensitive information from users, such as usernames, passwords, credit card numbers or other financial information. Attackers usually use URLs that look legitimate to trick users into revealing their information. These URLs may direct users to fake web pages that mimic those of financial institutions, social media sites, or e-mail services in order to deceive users.
- **Defacement:** defacement is a practice in which a website is compromised and its content altered or damaged by an attacker. Attackers may alter the content of the site to promote political, social or personal messages, or simply to demonstrate their ability to breach the site's security.

- **Malware:** associated with the distribution of malicious software or malware. Attackers use URLs to direct users to compromised websites or to convey links to malicious files, scripts or other content that can infect users' devices with malware.

3.2 Experience

Since this was a supervised learning task, it was necessary to use a dataset that associates each sample with a label indicating the type of URL according to those mentioned above. The dataset used comprises a total of 651.191 URLs, out of which 428.103 benign or safe URLs, 96.457 defacement URLs, 94.111 phishing URLs, and 32.520 malware URLs. Since the most crucial task is to curate the dataset for a machine learning project, it was curated using various sources. The URL dataset ISCX-URL-2016 [2] was utilized for collecting benign, phishing, malware, and defacement URLs. To augment the phishing and malware URLs, the Malware Domain Blacklist dataset was employed. Additional benign URLs were obtained from the Faizan GitHub repository [3]. Moreover, an increased number of phishing URLs were sourced from the Phishtank dataset and PhishStorm dataset [4]. Initially, URLs were collected from diverse sources and organized into separate data frames. Subsequently, these data frames were merged to retain only the URLs and their corresponding class types.

3.2.1 Features extraction

In the realm of data analysis and machine learning, features extraction plays a pivotal role in processing and preparing data for model training. This phase focuses on identifying and extracting relevant information from raw data, transforming it into a suitable format for analysis and machine learning algorithms. The careful selection of features directly impacts the model's ability to learn patterns in the data and make accurate predictions. The following is a list of the identified features:

- **URL length:** the length of the given URL.
- **Domain:** the top-level domain of the given URL.
- **Special characters:** the number of occurrences of special characters such as '@', '?', '-', '=', ':', '#', '%', '+', '\$', '!', '*', ',', '/', ' '.

- **Abnormal URL:** a boolean value representing the presence of the hostname in the given URL.
- **HTTPS:** a boolean value representing the usage of the HTTPS protocol.
- **Digits:** the number of digits in the given URL.
- **Letters:** the number of letters in the given URL.
- **Shortening service:** a boolean value representing the association of the given URL with a shortening service.
- **IP address:** a boolean value representing the presence of an IP address in the given URL.

3.3 Models

In order to make the application as high-performance as possible, three different models were trained, and then the one with the best performance was chosen, but still leaving the user the option of using the other two. Specifically, the three models that were chosen are:

- **SDGClassifier:** Stochastic Gradient Descent is a simple yet very efficient approach to fitting linear classifiers and regressors under convex loss functions such as (linear) Support Vector Machines and Logistic Regression. Even though SGD has been around in the machine learning community for a long time, it has received a considerable amount of attention just recently in the context of large-scale learning.
- **RandomForestClassifier:** A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.
- **GaussianNB:** the Gaussian Naive Bayes (GaussianNB) classifier is a powerful algorithm that assumes each class follows a normal distribution. It assumes each parameter has an independent capacity of predicting the output variable.

After training and testing each mentioned model, the evaluation metric used was the F1-score, calculated as follows:

$$F1 = \frac{1}{n} \sum_{i=0}^{n-1} \frac{2 \times precision_i \times recall_i}{precision_i + recall_i} \quad (3.1)$$

where:

- **$precision_i$:** the ratio of the number of instances correctly classified as belonging to class i (true positives) and the total number of instances classified as belonging to that class (true positives plus false positives).
- **$recall_i$:** the ratio between the number of instances correctly classified as belonging to class i (true positives) and the total number of instances actually belonging to that class (true positives plus false negatives).
- **n :** number of total classes.

The F1-score balances precision and recall, providing a single measure that accounts for both false positives and false negatives. This makes it particularly useful when both metrics are important.

3.4 Training and Validation

The first operation tackled before the training phase was the division of the dataset into training and validation sets. To do this, the ‘train_test_split’ method of scikit-learn was used, in order to obtain 20% of the dataset as validation set and the remaining 80% as training set. Before proceeding to the training of the various models, there was a phase in which a grid search was carried out in order to find the model parameters that provided the best results. After this, the actual training of the models was performed, and then their behaviour was validated.

At this point, based on the F1-scores of each model, the one with the best performance was selected as the default model for the application.

Cluster Configuration

A Kubernetes cluster comprises a group of worker machines known as nodes. These nodes are responsible for running containerized applications. At minimum, every cluster has one worker node. These worker nodes host the Pods, which are the fundamental units of the application workload.

The control plane oversees the management of the worker nodes and the Pods within the cluster. Nodes can be either virtual or physical machines, depending on the setup of the cluster. In production settings, the control plane typically operates across multiple computers, and a cluster generally encompasses multiple nodes. This setup ensures fault tolerance and high availability for the applications running on Kubernetes.

In this project, the cluster was configured with a master node and 2 workers consisting of virtual machines as they offer greater advantages in terms of cost, scalability and management.

- **Cost:** virtual machines tend to be cheaper than physical machines.
- **Scalability:** easier addition or removal of virtual instances according to workload needs.
- **Management:** using virtual machines simplifies infrastructure management, as there is no need to worry about managing physical hardware or machine maintenance

In accordance with the above, the cluster configuration file used is:

```
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
```

All that remains now is to create the cluster. This is possible using the following command:

```
kind create cluster --config=cluster_config.yaml
```

4.1 Metric Server

One of the greatest advantages of the cloud is being able to monitor the resources in use and to scale as required. To achieve this functionality, it was necessary to deploy a **metric server** within the kubernetes cluster.

A Metric Server is a Kubernetes component designed to collect, store and expose metrics about the resources of nodes and pods within a Kubernetes cluster. These metrics include information such as CPU utilisation and memory usage. It provides cluster administrators and developers with a detailed view of cluster workload and resource utilisation trends, enabling them to make informed decisions on resource management and performance optimisation.

The manifest file used for the metric server was downloaded from the official kubernetes repository [\[5\]](#), and modified, adding the following line in the args:

```
--kubelet-insecure-tls
```

so that certificate validation could be disabled.

After that, the metric server was created with the following command:

```
kubect1 create --filename metric-server.yaml
```

Once the server has been created within the cluster, the resource consumption of the nodes and pods can be monitored using the command:

```
kubect1 top <nodes/pods>
```

4.2 Kubernetes Autoscaling

Scalability is a key characteristic of Kubernetes, and it's made even more powerful by automation. This automation eliminates the need for labor-intensive manual scaling.

Autoscaling is a feature that dynamically adapts the quantity of resources based on fluctuating needs, ensuring resources are used efficiently and the performance of the cluster is optimized. This leads to a more efficient system overall.

Autoscaling can help prevent capacity-related failure and also prevent users from paying extra for resources that they don't even need 24/7.

This functionality can be achieved mainly in two ways:

- *Horizontal Pods Autoscaling*: it changes the number of pods that are available for the cluster, in case there are any sudden changes in demand. This technique is mostly used to avoid any resource deficits since it scales pods instead of available resources. With horizontal scaling, it is possible to create different rules for stopping or starting instances assigned to a resource, when they end up reaching their lower or upper thresholds.

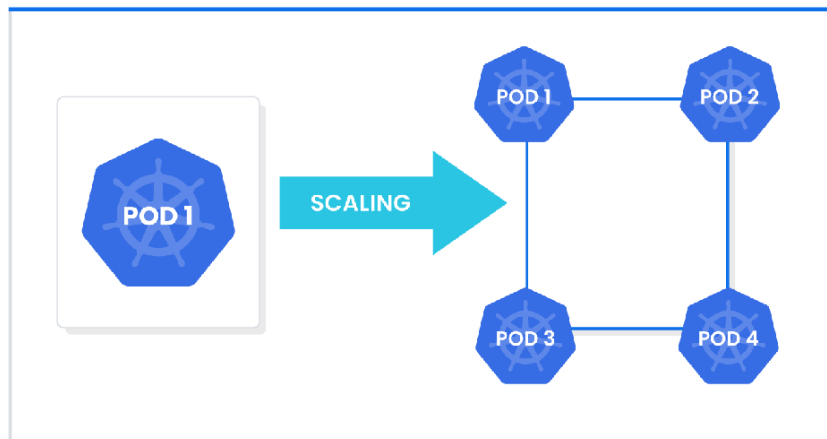


Figure 4.1: Horizontal Pods Autoscaling

- *Vertical Pods Autoscaling*: involves dynamically provisioning different resources like CPU and memory usage in order to meet the changing requirements of cluster nodes. VPA is responsible for modifying pod resource request parameters according to the consumption metrics of the workload. By adjusting pod resources according to usage over time, it becomes possible to optimize cluster resource utilization and minimize resource wastage. With vertical autoscaling, it is possible to establish a set of rules that in turn affect the memory utilization or CPU usage allocated to existing instances.

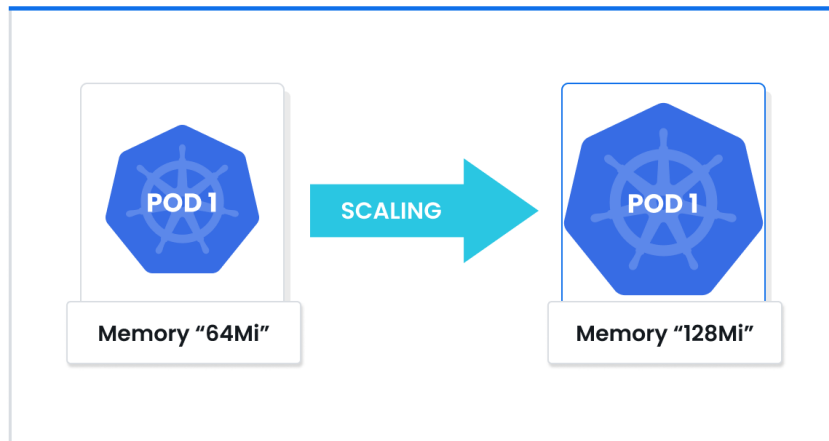


Figure 4.2: Vertical Pods Autoscaling

In order to obtain the advantages of both approaches, it was first considered to include both autoscalers. However, the official github repository of kubernetes points out that these may clash when considering the same resources (CPU and memory) [6]. For this reason, it was decided to introduce a VPA that adds resources to existing pods based on CPU and memory, and an HPA that adds pods based on the number of HTTP requests.

4.2.1 VPA

First, to deploy a VPA it is necessary to define a manifest file of type VerticalPodAutoscaler, which describes which deployment it refers to and the minimum and maximum values of the resources to be assigned to the pods:

```
apiVersion: "autoscaling.k8s.io/v1"
kind: VerticalPodAutoscaler
metadata:
  name: malicious-detection
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: malicious-detection
  resourcePolicy:
    containerPolicies:
      - containerName: '*'
        minAllowed:
          cpu: 100m
          memory: 50Mi
        maxAllowed:
          cpu: 1
          memory: 500Mi
        controlledResources: ["cpu", "memory"]
```

At this point, it was sufficient to create the vpa with the following command:

```
kubect1 create --filename vpa.yaml
```

4.2.2 HPA

Regarding HPA, things were slightly more complicated as it was first necessary to install KEDA (Kubernetes Event-driven Autoscaling), and all its dependencies. **KEDA** is a single-purpose and lightweight component that can be added into any Kubernetes cluster. It performs three key roles within Kubernetes:

- **Agent:** KEDA activates and deactivates Kubernetes Deployments to scale to and from zero on no events. This is one of the primary roles of the keda-operator container that runs when KEDA is installed.
- **Metrics:** KEDA acts as a Kubernetes metrics server that exposes rich event data like queue length or stream lag to the Horizontal Pod Autoscaler to drive scale out. It is up to the Deployment to consume the events directly from the source. This preserves rich event integration and enables gestures like completing or abandoning queue messages to work out of the box. The metric serving is the primary role of the keda-operator-metrics-apiserver container that runs when KEDA is installed.
- **Admission webhooks:** Automatically validate resource changes to prevent mis-configuration and enforce best practices by using an admission controller.

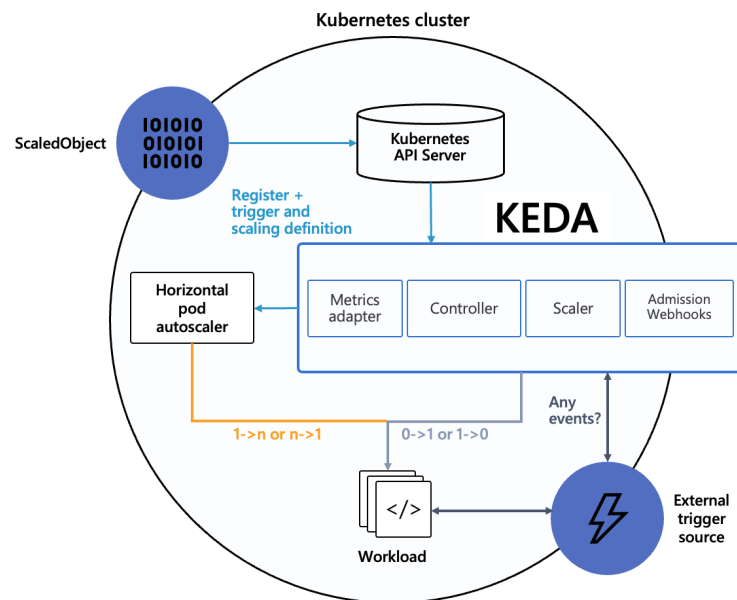


Figure 4.3: KEDA architecture

Once this was done, it was once again necessary to create a manifest file describing the scaler, which deployment it refers to, and the external metrics on which it bases its decisions.

```
kind: HTTPScalableObject
apiVersion: http.keda.sh/v1alpha1
metadata:
  name: malicious-detection-hpa
spec:
  hosts:
    - 127.0.0.1
  scaleTargetRef:
    name: malicious-detection
    kind: Deployment
    apiVersion: apps/v1
    service: malicious-detection
    port: 80
  replicas:
    min: 2
    max: 15
  targetPendingRequests: 100
```

To create the scaler, simply execute the command:

```
kubectl apply -f hpa.yaml
```

4.3 Kubeflow installation and setup

Once the cluster has been configured, Kubeflow must be installed in it. To do this, the following commands [7] were executed, which install Kubeflow Pipelines on a Kubernetes cluster using manifest resources from the Kubeflow Pipelines GitHub repository.

Kubeflow Pipeline

A *pipeline* is a description of a machine learning (ML) workflow, including all of the components in the workflow and how the components relate to each other in the form of a graph. The pipeline configuration includes the definition of the inputs (parameters) required to run the pipeline and the inputs and outputs of each component.

When a pipeline is executed, the system initiates one or more Kubernetes Pods that correspond to the components in the pipeline. The Pods start Docker containers, and these containers, in turn, launch the respective programs.

In this project, the python kfp library was used to create the pipeline. Kubeflow Pipelines (KFP) is a platform for building and deploying portable and scalable machine learning workflows using Docker containers. KFP enables data scientists and machine learning engineers to:

- Author end-to-end ML workflows natively in Python.
- Create fully custom ML components or leverage an ecosystem of existing components.
- Easily manage, track, and visualize pipeline definitions, runs, experiments, and ML artifacts.
- Efficiently use compute resources through parallel task execution and through caching to eliminating redundant executions.
- Maintain cross-platform pipeline portability through a platform-neutral IR YAML pipeline definition.

These benefits ultimately culminate in establishing a streamlined **Continuous Integration and Continuous Deployment (CI/CD)** process, enhancing the efficiency and agility of ML development and deployment pipelines.

5.1 Securl pipeline

During the design and development of the various machine learning models, several distinct phases were identified that can be interpreted as pipeline components, which are the building blocks of KFP pipelines. A component is a remote function definition; it specifies inputs, has user-defined logic in its body, and can create outputs.

- **Load Data Function:** this component reads the dataset from a csv file and pre-processes its contents. It then extracts the previously listed features and, using a scaler, ensures that each of them contributes equally to the analysis. Finally, it divides the dataset into training set and validation set by performing a split and saves the data in a json file.
- **Gaussian Naive Bayes classifier:** this components reads input data from a JSON file produced by the *Load Data Function* component. It then trains a GaussianNB model on the training data and evaluates its performance on the validation data. Finally, the F1-score and classification report are calculated using the `f1_score` and `classification_report` functions from the scikit-learn library.
- **Random Forest classifier:** this components reads input data from a JSON file produced by the *Load Data Function* component. It then trains a RandomForestClassifier model on the training data and evaluates its performance on the validation data. Finally, the F1-score and classification report are calculated using the `f1_score` and `classification_report` functions from the scikit-learn library.
- **Stochastic Gradient Descent classifier:** this components reads input data from a JSON file produced by the *Load Data Function* component. It then trains a SGDClassifier model on the training data and evaluates its performance on the validation data. Finally, the F1-score and classification report are calculated using the `f1_score` and `classification_report` functions from the scikit-learn library.
- **Best model:** this component takes as input the F1-scores produced by the three models and computes the maximum value among these. It finally print the best model to choose.
- **Show results:** this component is responsible for printing the classification reports computed by the three models.

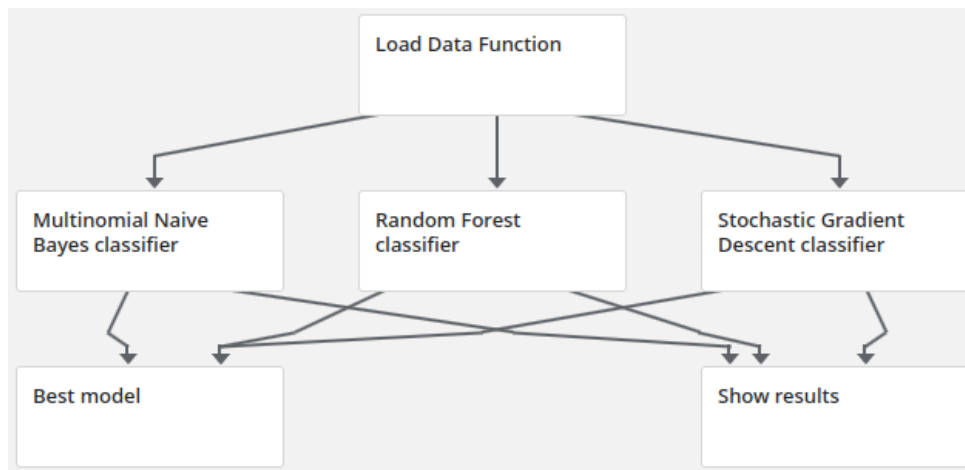


Figure 5.1: Securl pipeline

5.2 Pipeline building process

Each pipeline component is based on a python file describing its logic, a Dockerfile used to create the function's container, and a YAML file describing how to execute the function, what its inputs are, and what its outputs are. Once all the necessary files are available, the container image should be created and uploaded to the Docker Hub using the following commands:

```
docker build --tag <image_name> <path/to/Dockerfile>
docker tag <image_name> <username>/<image_name>
docker push docker.io/<username>/<image_name>
```

Next, a YAML file describing the complete pipeline [8] had to be created using the Kube-flow Pipelines (kfp) library. At this point, having obtained the pipeline YAML file, it was possible to upload it to the dashboard to verify that the pipeline was working properly.

Securl

6.1 Web Application GUI

The final phase of the project involves the distribution of the ML models via a web application. This application was realised using *streamlit*, a simple and intuitive framework for creating graphical interfaces.

The interface includes, first of all, the application logo, which is initially presented in its basic version, but which changes dynamically based on the evaluation of the URL provided by the user. This behaviour is designed to improve the user experience and provide graphical feedback to make the application more intuitive. In particular, the logo will turn red to indicate that the given URL is malicious, and green to indicate the opposite.

Proceeding, the graphical interface provides a text field for entering the URL to be verified and a sidebar where various informations are provided, including the dataset used, the statistics of each available model and information on the developers of the application. In addition, in this sidebar, it is also possible to select the model with which to perform the URL checks.

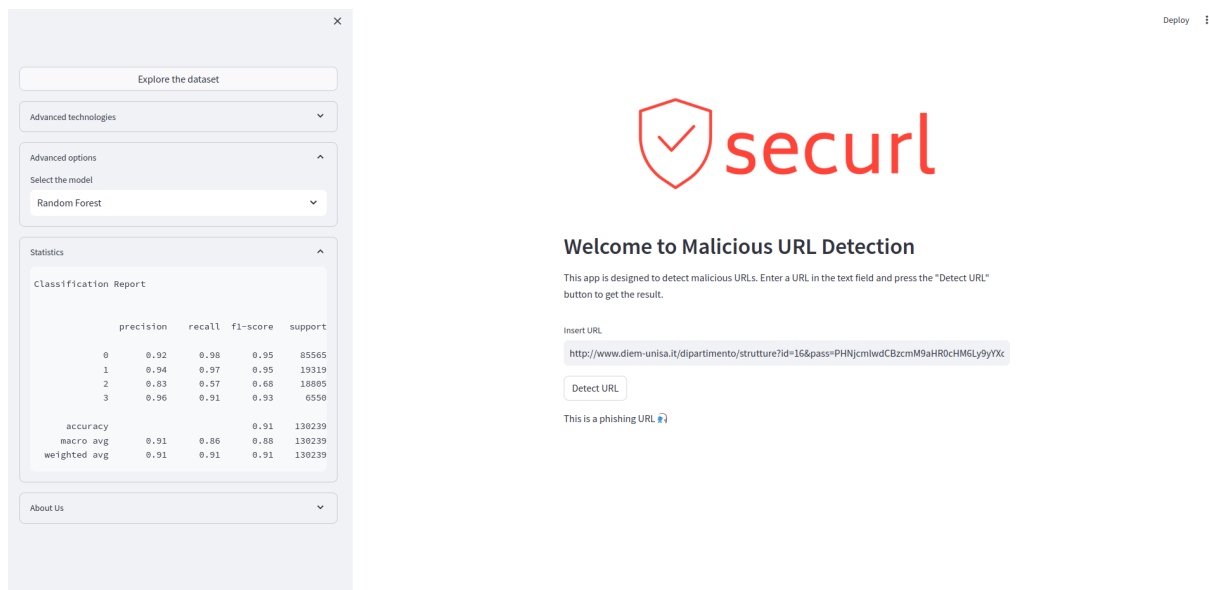


Figure 6.1: Securl web application GUI

6.2 Deployment

This phase represents an important moment in the development of the project because it is here that the application is deployed on the cluster, and thus made available to users. The deployment of an application involves several steps. First, the application must be containerised so that it is isolated and easily managed by Kubernetes. Next, it is necessary to define Kubernetes manifests that describe how the application should be deployed and managed within the Kubernetes cluster. In particular, the manifest file defined for the application presented in this report is **MUD_K8S.yaml** [9], which defines a Deployment that manages two replicas of the application, and specifies the Docker image that each replica must use. It also defines a Service of type LoadBalancer to expose the application externally to the cluster using an external load balancer [1], which distribute the workload over the available instances of the service. In this way, there will be no situations in which one instance has to handle an excessive load while another remains unused. Once the manifest file has been written, the resources specified in it must be created within the cluster. This is possible with the command:

```
kubectl create --filename MUD_K8S.yaml
```

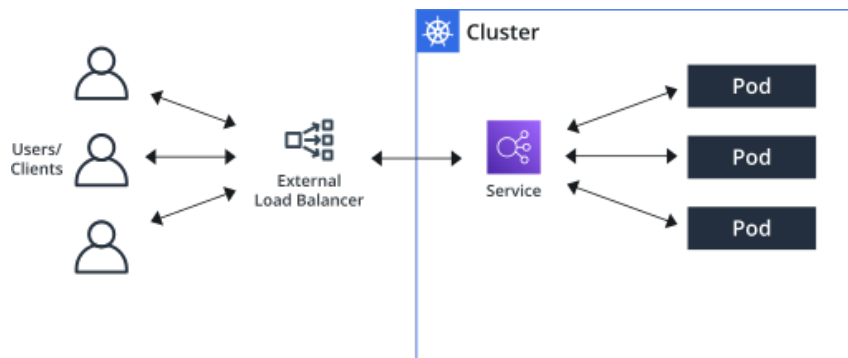


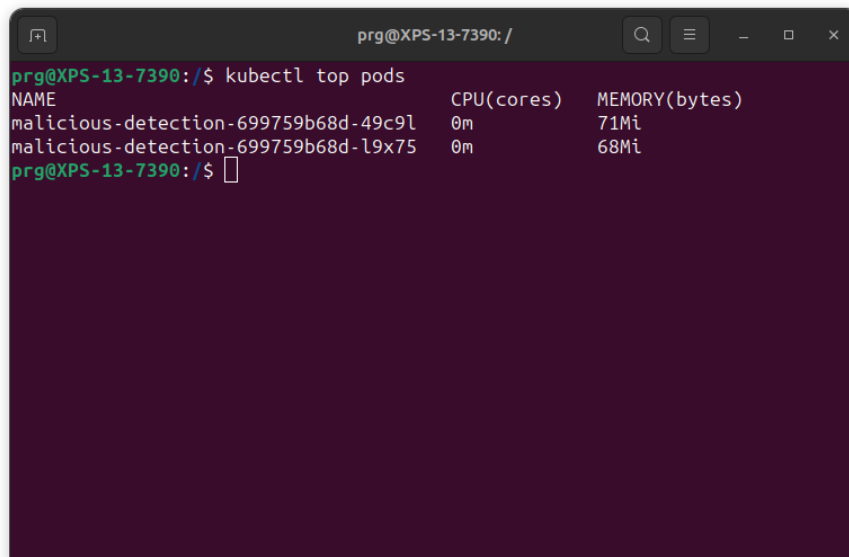
Figure 6.2: Load Balancer

¹Note that kubernetes does not directly offer a load balancing component; you must provide one, or you can integrate your Kubernetes cluster with a cloud provider.

Benchmark

Since the developed web application is intended for a total number of clients of 50000, but considering a maximum peak of 5000 simultaneous connections, tests were conducted on the application's response to this load, varying the number of workers and replicas. The tool used for the tests is *wrk*, a modern HTTP benchmarking tool capable of generating significant load when run on a single multi-core CPU. Thanks to the previously installed metric server, it is possible to check resource consumption with the command:

```
kubectl top pods
```

A terminal window titled 'prg@XPS-13-7390: /' showing the command 'kubectl top pods' and its output. The output is a table with three columns: NAME, CPU(cores), and MEMORY(bytes). It lists two pods, both with 0m CPU and 71Mi/68Mi memory usage.

NAME	CPU(cores)	MEMORY(bytes)
malicious-detection-699759b68d-49c9l	0m	71Mi
malicious-detection-699759b68d-l9x75	0m	68Mi

Figure 7.1: Initial situation

From the output received, it can be verified that the number of pods is as defined in the application deployment, and the resources consumed by them are almost zero, due to the currently non-existent load.

At this point, it is possible to start the test, using wrk with the following command:

```
wrk -t10 -c5000 -d5m --latency http://127.0.0.1:30070
```

Where:

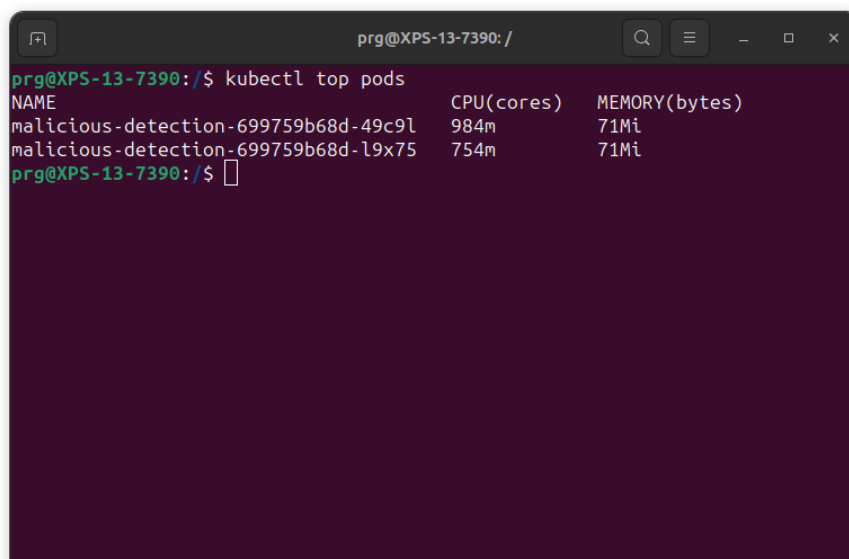
-c, --connections: total number of HTTP connections to keep open with
each thread handling $N = \text{connections/threads}$

-d, --duration: duration of the test, e.g. 2s, 2m, 2h

-t, --threads: total number of threads to use

--latency: print detailed latency statistics

After a while, you can see that the 2 pods originally created have up to 1000m CPU available thanks to the VPA:

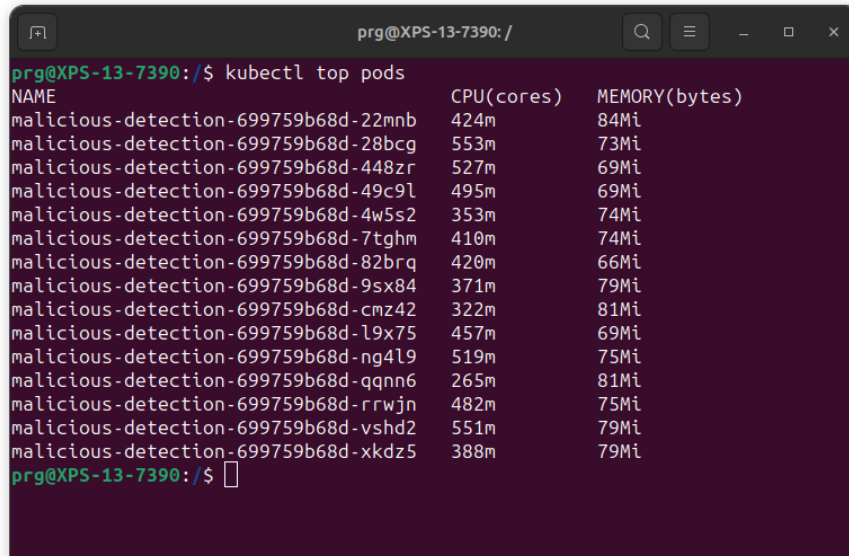
A terminal window titled 'prg@XPS-13-7390: /' showing the output of the command 'kubectl top pods'. The output is a table with three columns: NAME, CPU(cores), and MEMORY(bytes). There are two rows of pod data.

NAME	CPU(cores)	MEMORY(bytes)
malicious-detection-699759b68d-49c9l	984m	71Mi
malicious-detection-699759b68d-l9x75	754m	71Mi

Figure 7.2: Midterm situation

7. BENCHMARK

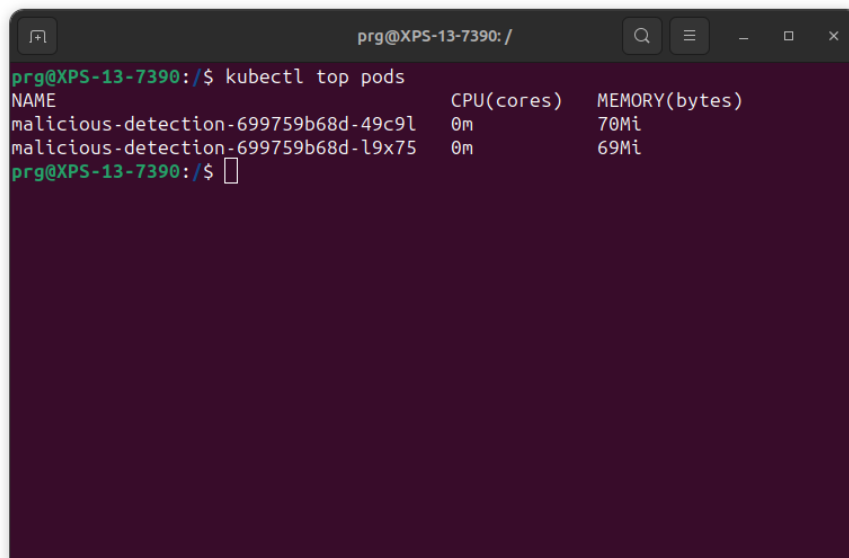
Subsequently, as the test and thus the HTTP requests progress, it is possible to see the increase in pods as predicted by the HPA:

A terminal window titled 'prg@XPS-13-7390: /' showing the output of 'kubectl top pods'. It lists 15 pods, all with names starting with 'malicious-detection-699759b68d-'. Each pod is shown with its CPU usage in millicores (m) and memory usage in Mi. The CPU usage varies between 265m and 553m, and memory usage varies between 69Mi and 84Mi.

```
prg@XPS-13-7390:/$ kubectl top pods
NAME                                CPU(cores)   MEMORY(bytes)
malicious-detection-699759b68d-22mnb 424m         84Mi
malicious-detection-699759b68d-28bcg 553m         73Mi
malicious-detection-699759b68d-448zr 527m         69Mi
malicious-detection-699759b68d-49c9l 495m         69Mi
malicious-detection-699759b68d-4w5s2 353m         74Mi
malicious-detection-699759b68d-7tghm 410m         74Mi
malicious-detection-699759b68d-82brq 420m         66Mi
malicious-detection-699759b68d-9sx84 371m         79Mi
malicious-detection-699759b68d-cmz42 322m         81Mi
malicious-detection-699759b68d-l9x75 457m         69Mi
malicious-detection-699759b68d-ng4l9 519m         75Mi
malicious-detection-699759b68d-qqnn6 265m         81Mi
malicious-detection-699759b68d-rrwjn 482m         75Mi
malicious-detection-699759b68d-vshd2 551m         79Mi
malicious-detection-699759b68d-xkdz5 388m         79Mi
prg@XPS-13-7390:/$
```

Figure 7.3: Workload situation

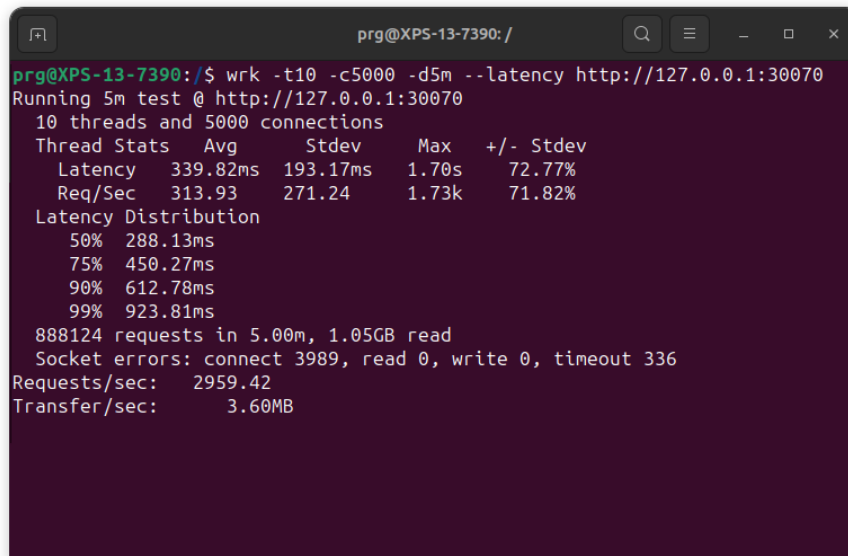
Once the test is over, the HPA will reset the number of pods:

A terminal window titled 'prg@XPS-13-7390: /' showing the output of 'kubectl top pods' after the workload has ended. Only two pods remain: 'malicious-detection-699759b68d-49c9l' with 0m CPU and 70Mi memory, and 'malicious-detection-699759b68d-l9x75' with 0m CPU and 69Mi memory.

```
prg@XPS-13-7390:/$ kubectl top pods
NAME                                CPU(cores)   MEMORY(bytes)
malicious-detection-699759b68d-49c9l 0m           70Mi
malicious-detection-699759b68d-l9x75 0m           69Mi
prg@XPS-13-7390:/$
```

Figure 7.4: Final situation

Finally, looking at the wrk report, it can be seen that the average latency is 339.82 ms and that the 99th percentile is below 1 second, which in maximum load situations is a quite acceptable result:

A terminal window titled 'prg@XPS-13-7390: /' showing the output of a 'wrk' benchmark. The command executed is 'wrk -t10 -c5000 -d5m --latency http://127.0.0.1:30070'. The output includes thread statistics, latency distribution, and overall performance metrics.

```
prg@XPS-13-7390:/$ wrk -t10 -c5000 -d5m --latency http://127.0.0.1:30070
Running 5m test @ http://127.0.0.1:30070
10 threads and 5000 connections
Thread Stats   Avg      Stdev     Max   +/-  Stdev
  Latency  339.82ms  193.17ms  1.70s   72.77%
  Req/Sec   313.93   271.24   1.73k   71.82%
Latency Distribution
  50%    288.13ms
  75%    450.27ms
  90%    612.78ms
  99%    923.81ms
888124 requests in 5.00m, 1.05GB read
Socket errors: connect 3989, read 0, write 0, timeout 336
Requests/sec: 2959.42
Transfer/sec: 3.60MB
```

Figure 7.5: Latency distribution

Security Assessment

8.1 Threat Identification

The application presented in this report does not have any login mechanisms, let alone a database. This makes the application vulnerable to far fewer attacks, which certainly include DDoS and the Virtualisation Attack. To mitigate the risks of a virtualisation attack, security mechanizations such as encryption and proper isolation of the cloud infrastructure must be put in place. As for DDoS attacks, on the other hand, the potential of NGINX can be exploited.

- **DoS:** overload IT resources to the point where they cannot function properly. In a DDoS attack, incoming data traffic flooding the resources comes from many different sources.
- **Virtualization Attack:** exploits vulnerabilities in the virtualization platform to jeopardize its confidentiality, integrity, and/or availability. In this attack, a trusted attacker can successfully accesses a virtual server to compromise its underlying physical server.

In addition, a secure protocol such as HTTPS can be used to mitigate threats such as traffic eavesdropping and malicious intermediary. It should be noted that these two attacks do not pose a primary threat in the context of the presented application as there is no sensitive data in transit.

8.2 NGINX App Protect DoS

Distributed denial-of-service (DDoS) attacks continue to grow in size and complexity, with application-layer attacks up by 165% in 2022 over the previous two years. Threat actors use multi-vector attacks that include targeting the application layer (Layer 7) to maximize damage, knowing that even a short service interruption can cause revenue loss, reputational damage, and exposure to other types of attacks. NGINX makes the protection of apps and APIs from hard-to-detect Layer 7 DDoS attacks easy with NGINX App Protect DoS, which provides a configurable, robust, multi-layered defense for a comprehensive mitigation strategy. Running across distributed architectures and environments, it delivers adaptive and consistent protection.

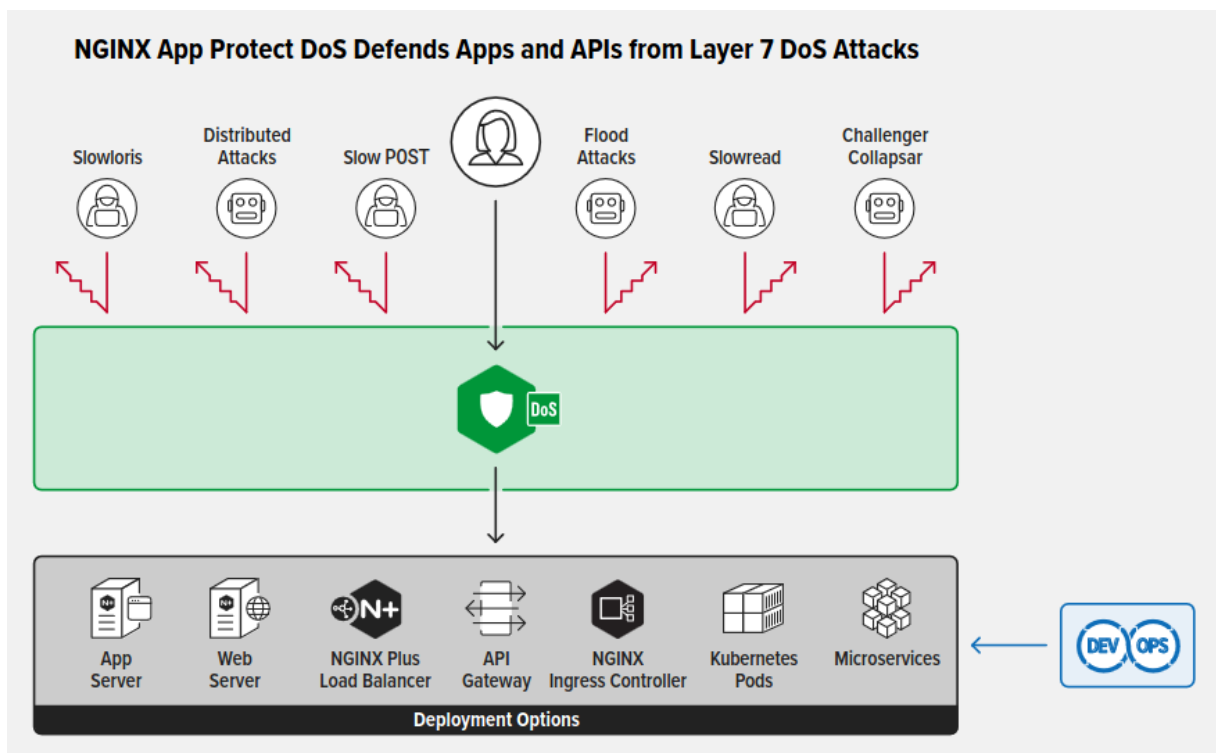


Figure 8.1: NGINX App Protect DoS

8.2.1 Configuration

In order to configure an NGINX App Protect Dos module, the first thing to do is to define a **DosProtectedResource**, a Custom Resource that holds the configuration of a collection of protected resources [10]. In this way, an ingress can be protected by specifying a reference to the **DosProtectedResource**. Furthermore it is possible to define a custom Kubernetes object(**APDosPolicy**) to configure a DoS attack mitigation policy. [11]

In addition, it is also possible to make a log service available to keep track of events occurring on protected resources. This can be made possible by defining an **APDosLogConf** Custom Resource [12] and specifying the qualified identifier (namespace/name) of the **APDosLogConf** in the **DosProtectedResource**.

At this point, the ingress can be defined in order to expose the application securely against DoS attacks. [13]

Conclusions

This project led to the realisation of a functioning kubernetes cluster in all its main aspects, emphasising the strengths of this system for orchestrating and managing containerised applications. Through it, it was possible to fully understand the importance of its automatic depolying and scaling features.

Regarding the use of Kubeflow, unfortunately, due to the physical limitations of the machine with which the project was carried out, it was not possible to run the pipeline on the cluster and thus to train the model on it as in a real scenario. However, when working on the creation of the pipeline, it was possible to see how these are a powerful tool for the creation of portable and scalable machine learning workflows, as they make it possible to create a modular structure whose components can be reused, thus achieving continuous integration and continuous deployment.

Future developments include the creation of a real cluster, without physical limits and also able to scale its structure thanks to an Autoscaler Cluster. This particular type of autoscaler can in fact be configured in such a way as to scale the cluster structure and add nodes on the basis of requests from running applications.

In addition, when the application is to be extended and thus provide further functionality, an NGINX app protection Web Application Firewall (WAF) must be introduced, so as to protect the web app from major threats such as SQL injection, buffer overflow and so on. In the hope that, through this project, all key aspects and benefits of cloud computing have been effectively highlighted, we would like to extend our sincere appreciation for your attention to this report.

List of Figures

3.1 URLs structure	7
4.1 Horizontal Pods Autoscaling	15
4.2 Vertical Pods Autoscaling	16
4.3 KEDA architecture	18
5.1 Securl pipeline	23
6.1 Securl web application GUI	24
6.2 Load Balancer	25
7.1 Initial situation	26
7.2 Midterm situation	27
7.3 Workload situation	28
7.4 Final situation	28
7.5 Latency distribution	29
8.1 NGINX App Protect DoS	31

Bibliography

- [1] *Public Suffix*. URL: https://publicsuffix.org/list/public_suffix_list.dat.
- [2] *ISCX-URL-2016*. URL: <https://www.unb.ca/cic/datasets/url-2016.html>.
- [3] *Faizan git repo*. URL: <https://github.com/faizann24/Using-machine-learning-to-detect-malicious-URLs/tree/master/data>.
- [4] *FishStorm Dataset*. URL: <https://research.aalto.fi/en/datasets/phishstorm-phishing-legitimate-url-dataset>.
- [5] *Metric Server Manifest*. URL: https://github.com/notprg/Securl/blob/main/cluster/metric_server.yaml.
- [6] *HPA and VPA conflict*. URL: <https://github.com/kedacore/http-add-on/issues/368>.
- [7] *Kubeflow deployment*. URL: <https://github.com/notprg/Securl/blob/main/kubeflow.sh>.
- [8] *Pipeline YAML file*. URL: https://github.com/notprg/Securl/blob/main/malicious_URL_pipeline.yaml.
- [9] *Application Deployment and Service*. URL: https://github.com/notprg/Securl/blob/main/app/MUD_K8S.yaml.
- [10] *DosProtectedResource*. URL: <https://github.com/notprg/Securl/blob/main/nginx/DosProtectedResource.yaml>.
- [11] *APDosPolicy*. URL: <https://github.com/notprg/Securl/blob/main/nginx/APDosPolicy.yaml>.
- [12] *APDosLogConf Custom Resource*. URL: <https://github.com/notprg/Securl/blob/main/nginx/APDosLogConf.yaml>.
- [13] *NGINX Ingress*. URL: https://github.com/notprg/Securl/blob/main/app/MUD_K8S.yaml.