Project Report

Kubeflow deployment for time series forecasting

Authors	Ferrara Luigina	0622900144
	Gargiulo Anna	0622900139
	Maruotto Ida	0622900135

Summary

Project description

The aim of the project is the exploration, training, and serving of a machine learning model for Time Series Data forecasting by exploiting Kubeflow's main components. The Time Series Data forecasting task consists in predicting accurate blood glucose levels in six patients affected by Type 1 Diabetes.

The main tasks of the project are:

- Create a Kubernetes cluster and set up it by installing Kubeflow,
- Train a time-series model using TensorFlow on the cluster,
- Serve the model using TensorFlow Serving,
- Save and deploy the final model and build a web app able to perform the prevision.

Introduction

The project exploits Kubeflow's main functionalities. The Kubeflow project is dedicated to making deployments of machine learning (ML) workflows on Kubernetes simple, portable, and scalable. The main tool that has been used is Arrikto, which makes it easy to get started with Kubeflow by providing a Kubeflow cluster and JupyterLab. This allows training TensorFlow models on the cluster automatically.

Kubeflow pipelines are one of the most important features of Kubeflow, making AI experiments reproducible, and composable, i.e. made of interchangeable components, scalable, and easily shareable.

Each pipeline component, represented as a block, is a self-contained piece of code, packaged as a Docker image. It contains inputs (arguments) and outputs and performs one step in the pipeline.

Finally, when running the pipeline, each container will be executed throughout the cluster, according to Kubernetes scheduling, considering dependencies.

Application architecture

In general, the deployment can be done using Docker or Kubernetes. In this project, both are implemented.

The selected architecture is microservices since each model is defined for an individual patient. They are deployed in Kubernetes using nginx reverse proxy to allow inference and can be updated, modified, or substituted at any moment. Fig. 1 on the right displays the architecture.

The web application can also be deployed in Kubernetes, using the same proxy of the served models or a dedicated one, but it can also be deployed using Docker, which is the selected manner.

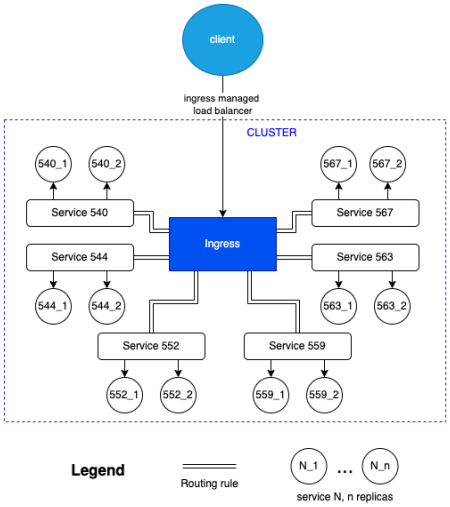

Figure 1: Application architecture

In a real case, the models are located on a cloud that is accessible through HTTP (or secure HTTP, HTTPS) post requests. Several clients can have the docker container running on their local hosts, or they can access such docker container through the network if located elsewhere.

Machine Learning architecture

Dataset

The dataset is composed of six different patients, identified by a number: 540, 544, 552, 559, 563, 567. For each of them, there are three measures (insulin, glucose, and carbohydrates) taken every 5 minutes for 1 hour and 15 minutes; from *t-14* to *t*. The desired output value is the glucose level 30 minutes (t+6) after the last measurement. Having the desired value enables building supervised machine learning training.

Although training and testing sets are already separated, only the training set has been used for machine learning purposes, while the testing set has been reserved for application demos.

Preprocessing

When dealing with health data, a common occurrence is that information is missing. This results in a preprocessing step that must fill in the missing values if there are any. In this specific case, checking the dataset proved that no missing values are present; so the analysis can continue without any change.

As a preprocessing step, dataset division has been done using 30% of the original training set for what has been called the *test set*, 20% of the remaining part to build the *validation set*, and the remaining part for the *train set*. The purpose of the *"test set"* was to have a measure on which the pipeline can be optimized.

Architecture

The architecture chosen for this task is a shallow multi-step dense network normally used to solve simple time series forecasting problems. It is composed of a *flatten* layer, two *dense* layers with *relu* activation function, and the output *dense* layer with a single neuron. The optimizer is ADAM.

This architecture depends on four hyperparameters: learning rate, number of neurons in the first dense layer, number of neurons in the second dense layer, and number of epochs.

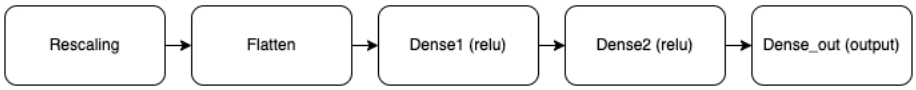

Figure 2: General model of multi-step network. Layer parameters are different for each patient.

An initial rescaling layer has been added to the architecture of the network to ensure that when the model is exported, the future data fed to the network will still be rescaled.

Pipeline automation with Kale and Katib

Kale pipeline

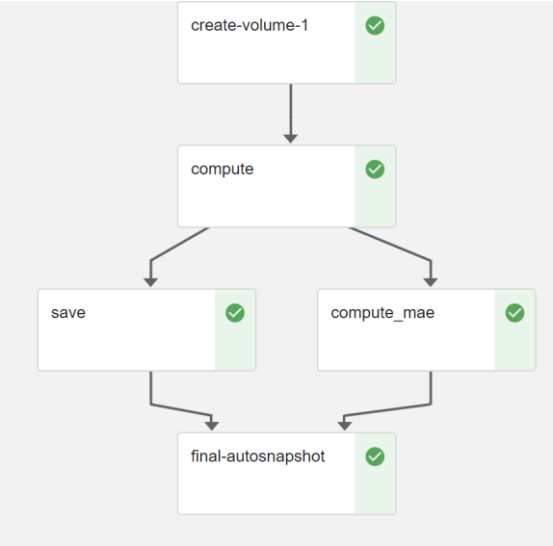

Figure 3: Pipeline

Kale is a tool for orchestrating end-to-end workflows. It provides a UI in the form of a JupyterLab extension thanks to whom cells can be annotated to define pipeline steps, hyperparameters, metrics imports, and functions.

Our pipeline is made of import cells, functions cells, and operational cells. Neglecting all the cells defined as *functions* and the cell meant for *imports*, there is one cell for hyperparameters tuning, and one for metrics (the sole metric in our experiments is MAE, on the *test set*).

This means that there are only three pipeline steps, described below. Every step is executed in its dedicated pod. Here it is the power of this tool: every pod is interested only in its inputs, so the user can change the way a step produces its output, but if the input and the output remain the same the new pod can substitute the old one, ideally changing the whole behavior of the pipeline.

These are the three steps of the pipeline:

- **Compute**: this step does two things: loads the data and trains the network. The two steps are not separated since loading data is just a memory access, and it would be useless to define a pod that loads data, then writes the same data on the shared volume that another pod will access to do its computation. The network is trained with hyperparameters selected using Katib later described.
- **Save**: this step is responsible for saving the trained network for future use.
- **Compute_mae**: this step computes the error score on the *test* set that the pipeline optimization will try to minimize. It is necessary because the *metric* cell cannot evaluate the selected metrics but can only look to previously calculated ones.

Hyperparameter tuning using Katib AutoML

Katib is a Kubernetes-native project for automated machine learning (AutoML). It can tune hyperparameters of applications and supports TensorFlow, which is the project framework. Arrikto provides a light version of Katib allowing only three of the AutoML algorithms supported by Katib: Bayesian optimization, grid search, and random search.

It is based on *experiments*. An experiment is a single training run, also known as an optimization run. The main configurations of an experiment are objective (train MAE), the search space, and the searching algorithm.

One iteration of the hyperparameters tuning process is called a *trial*. Each experiment runs several trials until it reaches either the objective or the configured maximum number of trials. It must be considered that each trial has its pipeline running. For efficiency reasons, in cloud architectures, each node can run pods belonging to one or more trials, parallelizing computations.

Katib provides a graphical description of the experiment when it ends, showing which hyperparameters have been tested and their relative metrics.

The selected search space is very small, consisting of only 9 different values total from which to choose from.

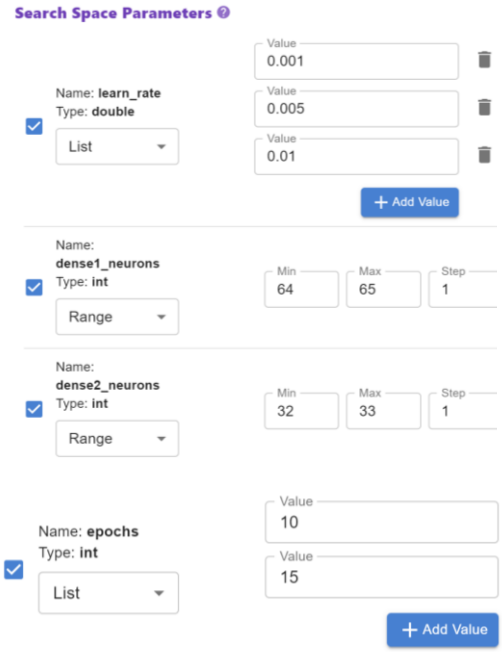

Figure 4: Search space parameters

Overall, six experiments have been done, one for each patient, with 10 maximum trials, Random Search algorithm to minimize MAE, and searching for a value lower than 25 as a goal.

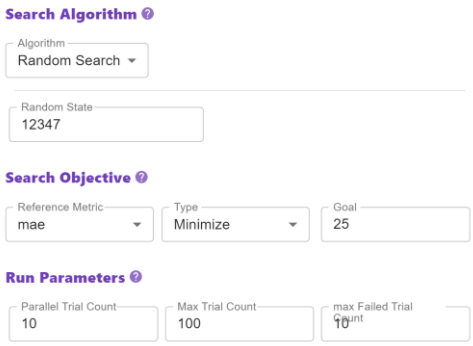

Figure 5: Algorithm details

This plot is an example of the *experiments* performed for patient 552 to find the best hyperparameters. The other experiments share the same behavior.

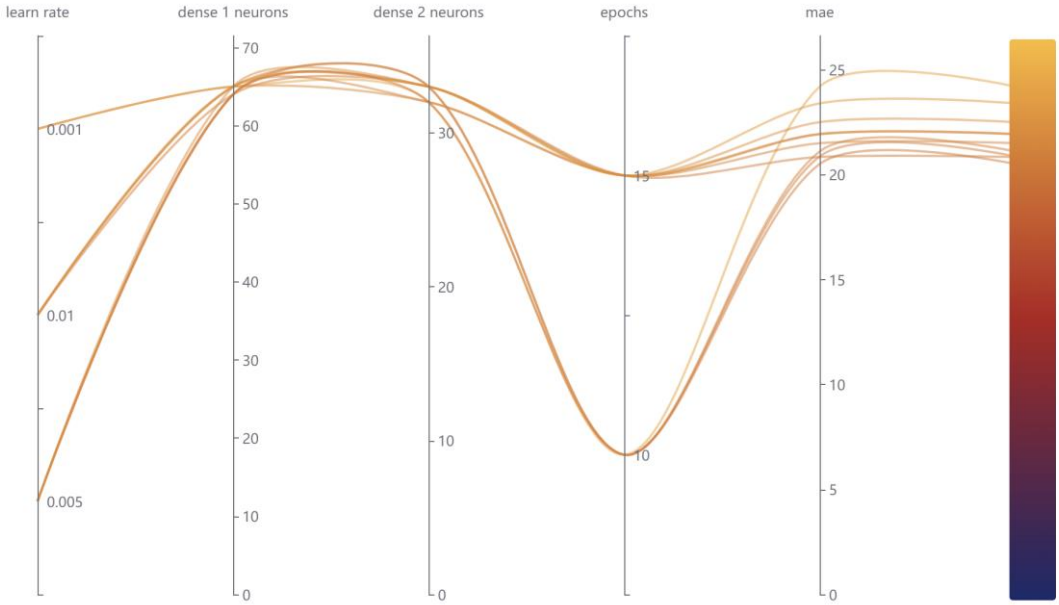

Figure 6:Experiments

These are the details of the experiments with the identification of the trial parameters and the best trial performance.

Name	glucose-prediction552-tgb07
Status	Experiment has succeeded because Objective goal has reached
Best trial	glucose-prediction552-tgb07-d367694f
Best trial's params	learn_rate: 0.005 dense1_neurons: 64 dense2_neurons: 32 epochs: 10
Best trial performance	mae: 20.5203
User defined goal	mae < 25
Running trials	0
Failed trials	0
Succeeded trials	10

Figure 7: Experiments' details

Therefore, we chose, and saved, for each patient, the model with the best performances in relation to the set of hyperparameters shown.

Deployment with Kubernetes

Kubernetes components

Deployment and Pods

Kubernetes pods are the smallest deployable units of computing that you can create and manage in Kubernetes. They are groups of one or more containers, and they contain all the necessary information to run such containers (storage and network resources). In this application, each pod will contain only one container.

A Kubernetes deployment is a tool that allows managing pods. It needs some characteristics of the pod to spawn it, then it also needs information about the maintenance of the pod. For example, a parameter that can be set when declaring a deployment, or when updating its attributes, is the number of replicas. This attribute allows to keep several pods running in the same container always active, and whenever a pod is stopped another one immediately replaces it. For this reason, in this application, the number of replicas is set higher than one, to be sufficiently confident that at least one pod is active. Replicas are an optimal way not only to avoid inactive service but also to share the load of work among the pods. In fact, Kubernetes allows updating (or performing a 'rollout' to a previous version) deployment definitions, increasing the number of replicas, and changing other parameters. It is also possible to 'shut down' or 'pause' a service putting the number of replicas to zero, meaning that there are no active pods that run that service.

Services

A Kubernetes Service is a method for exposing a network application that is running as one or more Pods in a cluster. Each Service object defines a logical set of endpoints along with a policy about how to make those pods accessible. A Service can be used to make that set of Pods available on the network so that clients can interact with it.

The replicas have been exposed externally by a Node Port Kubernetes Service, each of them targeted by the specific service thanks to the selector chosen.

Ingresses

Ingress is an API object that manages external access to the services in a cluster. An Ingress controller is bootstrapped with some load-balancing policy settings. For example, when there are more replicas of the same pod, as in this case, it can share the load amongst pods. The entry point to services is done using an nginx reverse proxy that hides the true IPs of the services behind the rewriting of the URL. To enhance security Kubernetes provides 'secrets' resources that permit the implementation of HTTP requests.

Minikube

Minikube is local Kubernetes. When started, it creates a docker cluster named *'minikube'* and associates all the Kubernetes operations done using *kubectl* command to its container. Minikube creates a cluster with one slave node in which all pods will be spawned.

To route to services, we defined an ingress. Minikube has an addon to enable routing: `minikube addons enable ingress`. To create a routable IP for a 'balanced' deployment `minikube tunnel` can be used. In this case, the access is not to a balanced deployment but to an ingress that has a load-balancing policy. Then the ingress routes traffic to the service corresponding to the specific deployment inferred.

Workflow to serve models

To serve a TensorFlow model with TensorFlow Serving, the model needs to be saved as a bundle, where the name of the model and a version corresponding to it is explicated. Moreover, to deploy the served models with Kubernetes, they must be encapsulated into Docker Images.

Thus, we created a Docker image, by associating the TensorFlow Serving Image (*tensorflow/serving*) to each of the models, independently. The images have been pushed to DockerHub, where they are public and accessible in repositories named after the number of patients they refer to.

As an example, the steps for the creation of the served model associated with patient 540 are described:

1. Spawning a daemon container with the TensorFlow/serving image `docker run -d --name serving_540 tensorflow/serving`;
2. Copy the model in the container `docker cp /path/to/model/540 serving_540:/models/540`;
3. Set the *model_name* environment variable in the container by running `docker commit --change "ENV MODEL_NAME 540" serving_540 540`. It has been decided to not use the same name for all the models (although it is possible) to add differentiation between models' URLs.
4. Kill the daemon container `docker kill serving_540`;
5. Tag the image to the repository `docker tag serving_540:latest username/540:latest` and then push to the hub `docker push username/repo_name:latest`.

Workflow for deploying with Kubernetes

First, start minikube cluster with `minikube start`. Then enable tunneling with `minikube tunnel`, this command requires sudo (or administrator in windows) privileges, because it will open a reserved port with number lower than 1024. In particular, it will open ports 80 and 443 to enable HTTP and HTTPS requests.

With this configuration, we can start creating deployments using the command `kubectl apply -f deployments.yml`.

The file *deployments.yml* contains all the deployment configurations. The six deployments have the same general configurations: exposing port 8501 (the port on which TensorFlow serving serves the model), policy for pulling image only if not present in the local database, number of replicas equal to 2 to counter back issues with a pod. In this way, when listing the deployments, each of them will have a counter of 2 pods in a ready and available state.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: "540-deployment"
  labels:
    app: "540"
spec:
  replicas: 2
  selector:
    matchLabels:
      app: "540"
  template:
    metadata:
      labels:
        app: "540"
    spec:
      containers:
      - name: "540"
        image: nanigo/540:latest
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8501
```

Figure 8: Example of deployment for patient 540

```
(base) PS C:\Users\Luigina> kubectl get deployments
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
540-deployment    2/2     2            2           30h
544-deployment    2/2     2            2           30h
552-deployment    2/2     2            2           30h
559-deployment    2/2     2            2           30h
563-deployment    2/2     2            2           30h
567-deployment    2/2     2            2           30h
```

Figure 9: Active deployments in Kubernetes cluster

After generating the pods, it is possible to associate the services with the deployments to make them accessible through the network. In this architecture, one service is associated with a single deployment, but since a deployment manages two pods, a Kubernetes service exposes a service provided by the pods. Because TensorFlow serving API expects all REST requests on port 8501, each service is connected to that port to link the pods, but the services themselves are listening on external ports, specifically from 8080 to 8085.

Using the service, the pod that runs the serving of the model of the patient 540, will receive all requests on port 8501 forwarded by the service associated with 540, which expects requests on port 8080 (therefore, this is the port to connect to).

```yaml
apiVersion: v1
kind: Service
metadata:
  name: "svc-540-service"
  labels:
    app: "540"
spec:
  selector:
    app: "540"
  type: NodePort
  ports:
    - name: http
      port: 8080
      targetPort: 8501
```

Figure 9: Example of service for patient 540

```
(base) PS C:\Users\Luigina> kubectl get svc
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes        ClusterIP   10.96.0.1       <none>        443/TCP          19d
svc-540-service   NodePort    10.108.67.0     <none>        8080:32759/TCP   2d4h
svc-544-service   NodePort    10.99.174.30    <none>        8083:32626/TCP   2d4h
svc-552-service   NodePort    10.100.124.55   <none>        8084:31811/TCP   2d4h
svc-559-service   NodePort    10.97.71.235    <none>        8081:31839/TCP   2d4h
svc-563-service   NodePort    10.104.172.69   <none>        8085:32135/TCP   2d4h
svc-567-service   NodePort    10.98.171.87    <none>        8082:30559/TCP   2d4h
```

Figure 10: Active services in Kubernetes cluster

Since all REST requests must be sent to services, to make them accessible from the outside there are two ways: using an ingress (exposed by minikube tunneling) or port-forwarding the services.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/rewrite-target: /v1/models/$1:predict
spec:
  rules:
    - http:
        paths:
        - path: /540/v1/models/(.*):predict
          pathType: Prefix
          backend:
            service:
              name: "svc-540-service"
              port:
                number: 8080
        - path: /559/v1/models/(.*):predict
          pathType: Prefix
          backend:
            service:
              name: "svc-559-service"
              port:
                number: 8081
```

Figure 11: Ingress for the services

The chosen solution is using an ingress because it has various benefits apart from creating an access point. For completeness purposes, to perform port-forwarding on Kubernetes service that exposes patient 540's model it is sufficient to run `kubectl port-forward svc/svc-540-service 5400:8081`. Port forwarding can be done using the same port exposed by a service, but for simplicity reasons, it has been considered that using *patient_number*+0 (i.e. patient 540 will be exposed on port 5400) will be more effective. This solution does not require minikube tunneling but has the drawback that the port forward must be done for each service.

The Kubernetes ingress solution requires the enabling of the ingress controller (that uses reverse proxy and load balancer) and the minikube tunneling because, in this application, minikube has been used to simulate a cloud architecture. As deployments and services, the instruction is `kubectl apply`

`-f ingress.yml`. The ingress class is nginx, it automatically protects from simple attacks like DDoS or files that are too big (error 413).

When defining the ingress, it has been specified that the access protocol is HTTP, services' names and their ports to which the ingress will be connected, and a path for each service. The specified path takes into account the fact that TensorFlow Serving expects a specific path to infer models, next called *inference path*. To make the ingress work, there must be a different path for each service. This is done by placing before the *inference path* the patient's number (i.e. to infer the model of patient 540 the final path will be '/540/v1/models/540:predict'). In the ingress definition, the rule *rewrite-target* removes this new part, inferring the models with the correct path. This rule, moreover, does not forward to the server requests written in any other format, also protecting from SQL injection.

```
(base) PS C:\Users\Luigina> kubectl get ingress
NAME      CLASS    HOSTS   ADDRESS        PORTS   AGE
ingress   <none>   *       192.168.49.2   80      29h
```

Figure 12: Active ingress in Kubernetes cluster

The ingress consequently exposes to port 80 (because it is HTTP only) the cluster with address 192.168.49.2. This is the default address of the minikube cluster in its docker network (next described), and it is not accessible as-is from the local network; thanks to tunneling this IP is mapped onto localhost and can be accessed invoking 127.0.0.1 instead of minikube IP in a local browser to infer the models. Without minikube, in a real cluster architecture, the exposed IP will be the cloud one (excluding other routings from the cluster to the entry point of the cloud). Apart from using the IP address, it is possible to query the cluster with a specific hostname (e.g. glucose-predictor.info), if it is registered or simply associated with the IP address by adding a row in /etc/hosts. It will not refer to a DNS server, but it will use its local mapping.

Web application

For the implementation of the web application, Streamlit has been used. With this app, the patient can select its identifier and upload a CSV file where in each row the last 15 measures of carbohydrates, insulin, and glucose are registered. The web app does an HTTP POST request to the correct model for the patient and will show a plot with the estimated glucose values for each row.

This is the interface of the web application:

Figure 13: Interface of the web app

This is the plot with the estimated glucose values:

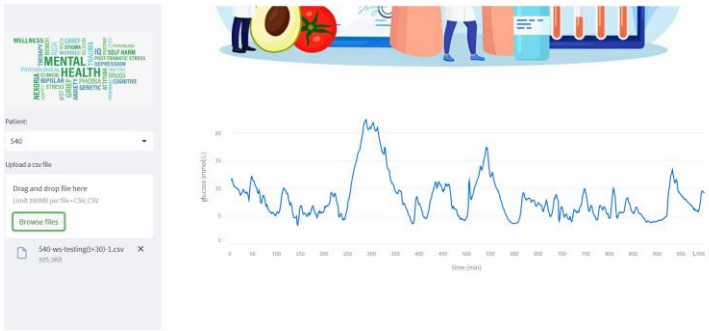

Figure 14: Interface of the web app after the inference of data registered by patient 540

Workflow for deploying with docker

For the deployment of the web app, Docker has been adopted. Docker builds images by reading the instructions from a Dockerfile which is a text document that contains all the commands a user could call on the command line to assemble an image. Thus, the first step is to build the docker image *glucose-prediction* `docker build --tag glucose-prediction`. Then, it's necessary to run the image with `docker run --publish 8501:8501 --name=glucose-prediction-web-app --net=minikube glucose-prediction` which first creates a container over the specified image, and then starts it. Streamlit default port is 8501, on which the container listens.

```
FROM python:3.9-slim-buster

WORKDIR /app

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

# Install dependencies
RUN pip3 install -r requirements.txt

# Expose port
EXPOSE 8501

COPY . /app

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

Figure 15: Dockerfile

The Dockerfile starts with a FROM instruction. It sets the base image for the container: in our case, it is a lightweight image that comes with the latest version of Python 3.9. The WORKDIR instruction sets the working directory for any instruction that follows the file. The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime. The file installs all the required libraries and then creates the image associated with the application.

Connect one deployment to another

```
(base) idamaruotto@MacBook-Pro-3 docker-deployed-application % docker build --tag glucose-predic
tion .
[+] Building 2.2s (2/3)
      [+] Building 2.6s (4/9)

 => [internal] load build definition from Dockerfile
 0.1ss
 => => transferring dockerfile: 407B
 0.0ss
[+] Building 2.7s (6/9)
      => [internal] load build definition from Dockerfile
      0.1s
 => => transferring dockerfile: 407B
0.0ss => [internal] load .dockerignore
      0.1s
 => => transferring context: 2B
0.0ss => [internal] load metadata for docker.io/library/python:3.9-slim-buster
[+] Building 85.0s (10/10) FINISHED
 => [internal] load build definition from Dockerfile                                0.1s
 => => transferring dockerfile: 407B                                                0.0s
 => [internal] load .dockerignore                                                   0.1s
 => => transferring context: 2B                                                     0.0s
 => [internal] load metadata for docker.io/library/python:3.9-slim-buster           2.0s
 => [1/5] FROM docker.io/library/python:3.9-slim-buster@sha256:ef30644d6bf6f30a2efceac16e  0.0s
 => [internal] load build context                                                   0.1s
 => => transferring context: 2.34kB                                                 0.0s
 => CACHED [2/5] WORKDIR /app                                                       0.0s
 => [3/5] COPY  requirements.txt ./requirements.txt                                0.1s
 => [4/5] RUN pip3 install -r requirements.txt                                     74.3s
 => [5/5] COPY . /app                                                               0.2s
 => exporting to image                                                              7.6s
 => => exporting layers                                                             7.6s
 => => writing image sha256:daffaa56c58931d5c032bf6d9026dba8f77540c56df3f9fbb7cc3c19ab4c5  0.0s
 => => naming to docker.io/library/glucose-prediction                               0.0s
(base) idamaruotto@MacBook-Pro-3 docker-deployed-application % docker run --publish 8501:8501 --
name=glucose-prediction-web-app --net=minikube glucose-prediction

Collecting usage statistics. To deactivate, set browser.gatherUsageStats to False.


  You can now view your Streamlit app in your browser.

  Network URL: http://192.168.49.3:8501
  External URL: http://93.34.32.9:8501
```

Figure 16: Deployment of the web app with Docker

Because docker containers can access an internet connection, in a real case the application would be finished. Since this architecture is built in a single computer using minikube to simulate the cluster, it is fundamental to connect the two docker containers (web application and minikube) to the same docker network to make them communicate. Docker network can be designed custom by running `docker network create new_network`, then connect all desired containers to the network by running `docker network connect new_network container_name`, and by inspecting containers properties, their IP address can be obtained. Minikube, when installed, creates its dedicated docker network named minikube, therefore, it has been decided that the best solution, for the small scale of this project, is connecting the web app container to the minikube docker network, by running `docker network connect minikube glucose-prediction-web-app`. Finally, to make the whole orchestration work, the IP to which the app must connect to perform inference is `minikube ip`.

Security webinar

One of the main problems of web or software applications is their vulnerabilities. Specifically, for cloud applications, one of the most used attacks is the DDoS (Distributed Denial of Service) attack.

The most common way to keep track of any possible attack is to:

1. Detect and Inspect the traffic flowing through the network;
2. Assess if the traffic could be malignant (this can be done with the help of software);
3. Either block the traffic, alert it or allow the traffic.
   Occasionally, good traffic will be blocked.

The best way to detect DDoS is to first build a baseline, i.e. how the application usually behaves, in terms of the number of requests, speed of requests, and content, and then compare the behavior of the application at the moment with the baseline, to detect the change in the speed of response, type of traffic, number of errors. When the traffic has been detected, it is necessary to act.

Nginx offers solutions to protect applications from DDoS attacks, and it can be used with Kubernetes. Examples of attacks Nginx protects from:

Figure 17: attacks Nginx protects from

Nginx checks for different parameters to assess whether the application is under attack or not. Apart from checking the traffic incoming, it also checks the responses of the application and then defined a violation level to identify the messages incoming as either threats or approved data. The engine is shown below.

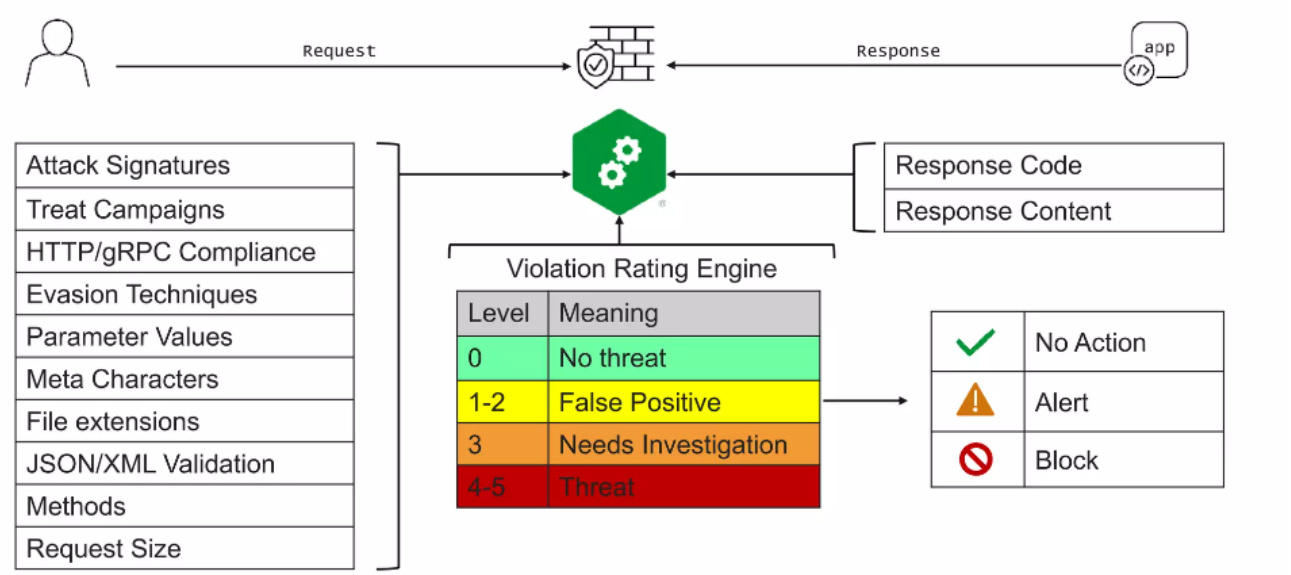

Figure 18: How the Nginx engine works

Depending on the level of the threat, different actions are taken incrementally, if the previous actions did not solve the anomaly issue:
First: block bad IPs, to make sure that the good traffic is still coming to the network;
Second: block bad requests, some of the good requests will be blocked;
Third: global rate limit.

Particularly, protecting apps in Kubernetes can be performed using an Ingress, much like we did with our application. It exposes the application no matter how big it is, exposing Kubernetes pods but checking and re-directing the traffic to each of them only if the request has been made on the right path, hiding their IP and Port.

Benefits of an Nginx ingress controller include a reduction of the complexity of the environment, custom resources, deployment performed on any cloud, path-based routing, TLS encryption, SSO, and it is less complex compared to the standard Kubernetes Ingress API.