Bulding the image from homework 05

Go to the `ML-zoomcamp/05Deploy/Homework` repo and build the docker image from the Dockerfile of the homework 05.

`Terminal`

```
marcos@marcos:~$ docker build -t zoomcamp-model:hw10 .
```

---

the content of the Dockerfile is:

`Dockerfile`

```
# Start with the existing image as a base
FROM svizor/zoomcamp-model:3.10.12-slim

# Set environment variables
ENV PYTHONUNBUFFERED=TRUE

# Install pipenv
RUN pip --no-cache-dir install pipenv

# Set the working directory inside the container
WORKDIR /app

# Copy the Flask script, Pipenv files into the container
COPY ["question6.py", "Pipfile", "Pipfile.lock", "./"]

# Install Python dependencies and clean cache
RUN pipenv install --deploy --system && \
rm -rf /root/.cache

# Port the app runs on
EXPOSE 9696

# Run Gunicorn
ENTRYPOINT ["gunicorn", "--bind", "0.0.0.0:9696", "question6:app"]
```

Question 1

Run it to test that it's working locally:

`docker run -it --rm -p 9696:9696 zoomcamp-model:hw10`
And in another terminal, execute `q6_test.py` file:

`python q6_test.py`
You should see this:

`{'get_credit': True, 'get_credit_probability': <value>}`
Here `<value>` is the probability of getting a credit card. You need to choose the right one.

- 0.3269
- 0.5269
- **0.7269**
- 0.9269

In [ ]:
```
import requests

url = "http://localhost:9696/predict"
client = {"job": "retired", "duration": 445, "poutcome": "success"}
response = requests.post(url, json=client).json()

print(response)
```

```
{'Credit probability': 0.726936946355423}
```

Installing `kubectl` and `kind`

You need to install:

- `kubectl` - https://kubernetes.io/docs/tasks/tools/ (you might already have it - check before installing)
- `kind` - https://kind.sigs.k8s.io/docs/user/quick-start/

Question 2

What's the version of `kind` that you have?

Use `kind --version` to find out.

`Terminal`

```
marcos@marcos:~$ kind --version
kind version 0.20.0
```

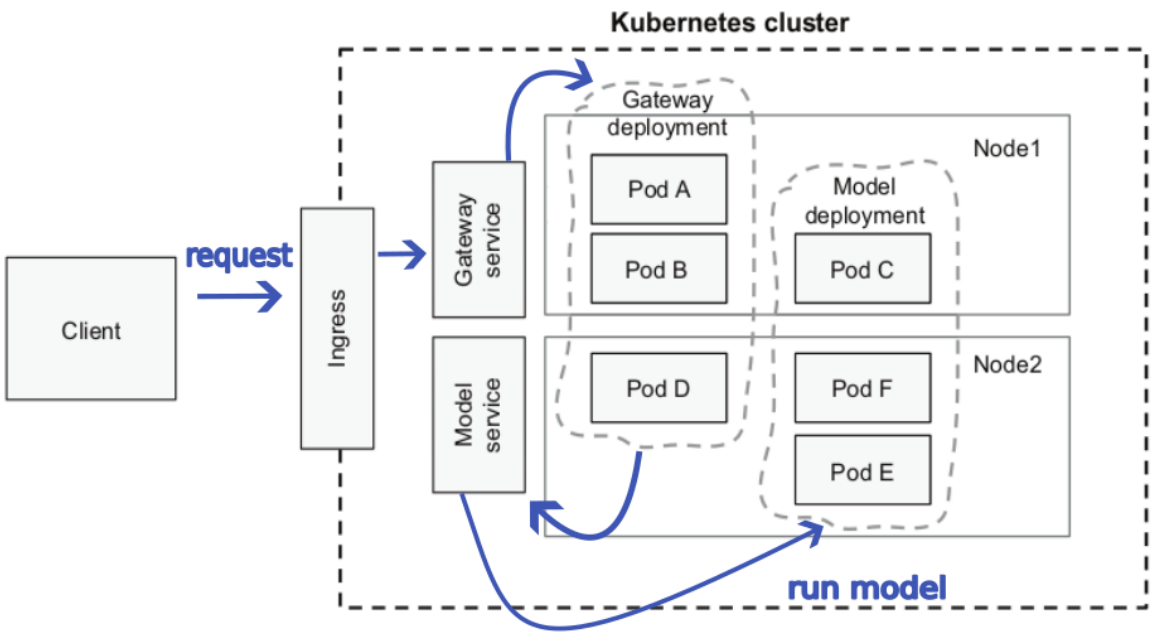
Kubernetes

Docker is used for running containers, which are packages of code and dependencies ensuring applications run efficiently across different environments (computers, cloud, virtual machines). However, for managing these containers, especially at large scale, is necessary the use of Kubernetes tool.

Kubernetes is an open-source platform that manage container applications and services. It enables declarative configuration and automation, making it easier to handle applications in various environments like physical servers, virtual machines, or cloud systems.

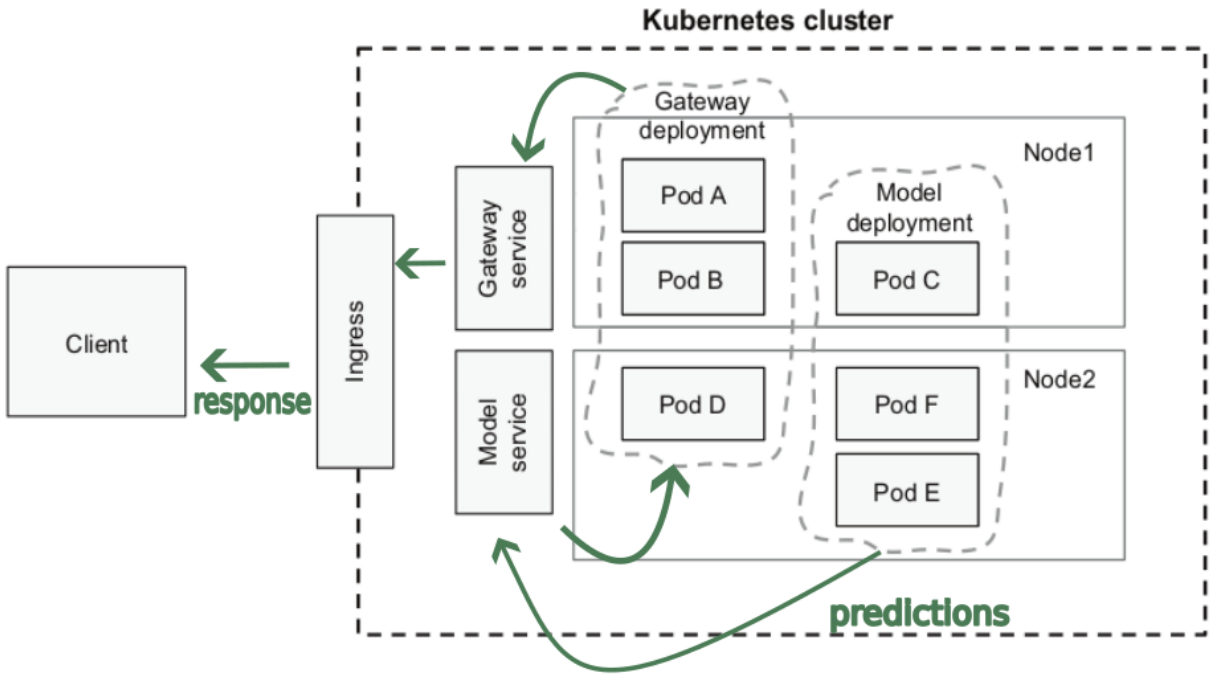
While Docker focuses on packaging and running individual containers, Kubernetes provides a comprehensive framework for orchestrating those containers in a clustered environment.

The main unit of abstraction in Kubernetes is a **pod**. A pod can contain one or more containers, typically Docker containers. The pods live within a **node**, which can be either a virtual machine or a physical computer that runs the Kubernetes processes. The nodes are grouped into **Kubernetes clusters**.



The pods within a deployment (a group of pods) usually contain instances of the same container image. A client can make a request, where the entry point is the **Ingress**, which communicates with the external gateway service. The **gateway service** (often referred to as an ingress controller) is responsible for routing the requests to the appropriate **gateway deployment**. The **gateway deployment** then routes the requests to the **model service**, which is responsible for routing the requests to the **model deployment**.

A service in Kubernetes is an abstraction that defines a logical set of pods, essentially serving as an entry point to a group of pods (deployment). A service ensures that the requests are routed to the appropriate pods within a deployment.



The response route after a request is made follows the reverse path. The **model deployment** processes the request and sends the response back to the **model service**. The **model service** routes the response to the **gateway deployment**. The **gateway deployment** passes the response to the **gateway service**, which then sends it to the **Ingress**. Finally, the **Ingress** routes the response back to the client.

Creating a cluster

Now let's create a cluster with `kind`:

```
kind create cluster
```
`Terminal`

```
marcos@marcos:~$ sudo kind create cluster

Creating cluster "kind" ...
 ✓ Ensuring node image (kindest/node:v1.27.3) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Have a question, bug, or feature request? Let us know! https://kind.sigs.k8s.io/#community 🙂
```

And check with `kubectl` that it was successfully created:

```
kubectl cluster-info
```
`Terminal`

```
marcos@marcos:~$ sudo kubectl cluster-info
Kubernetes control plane is running at https://127.0.0.1:45357
CoreDNS is running at https://127.0.0.1:45357/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Question 3

Now let's test if everything works. Use `kubectl` to get the list of running services.

What's `CLUSTER-IP` of the service that is already running there?

- **10.96.0.1**

`Terminal`

```
marcos@marcos:~$ sudo kubectl get services
[sudo] password for marcos:
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   39m
```

Question 4

To be able to use the docker image we previously created (`zoomcamp-model:hw10`), we need to register it with `kind`.

What's the command we need to run for that?

- `kind create cluster`
- `kind build node-image`
- **`kind load docker-image`**
- `kubectl apply`

Let's check the docker images to ensure that the Docker image `zoomcamp-model:hw10` is available.

`Terminal`

```
marcos@marcos:~$ sudo docker images
[sudo] password for marcos:
REPOSITORY              TAG           IMAGE ID       CREATED        SIZE
zoomcamp-model          hw10          4af085cd30a2   25 hours ago   432MB
svizor/zoomcamp-model   3.10.12-slim  08266c8f0c4b   8 weeks ago    147MB
```

With kind we created a local cluster by running each Kubernetes component as a separate Docker container. A local Docker network is then used to interconnect these containers, simulating the nodes of a Kubernetes cluster. With this approach, we can use the `kind load docker-image` command to load the Docker image into the local cluster.

`Terminal`

```
marcos@marcos:~$ sudo kind load docker-image zoomcamp-model:hw10
Image: "zoomcamp-model:hw10" with ID
"sha256:4af085cd30a2b4a3e7b3ede2a57bce241b8fd8508b117d671e6c59ead838e57f" not yet present on node "kind-
control-plane", loading...
```

Once the Docker image is loaded into kind, it becomes available in the local Kubernetes cluster. To actually run the image, a Kubernetes deployment (group of pods) or pod needs to be created that references this image. This setup allows us to emulate a Kubernetes cluster locally, testing our application as if it were running in a real cluster environment.

deployment and service

Question 5

Now let's create a deployment config (e.g. `deployment.yaml`):

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: credit
spec:
  selector:
    matchLabels:
      app: credit
  replicas: 1
  template:
    metadata:
      labels:
        app: credit
    spec:
      containers:
      - name: credit
        image: <Image>
        resources:
          requests:
            memory: "64Mi"
            cpu: "100m"
          limits:
            memory: <Memory>
            cpu: <CPU>
        ports:
        - containerPort: <Port>
```

Replace `<Image>`, `<Memory>`, `<CPU>`, `<Port>` with the correct values.

What is the value for `<Port>`?

- **9696**

Apply this deployment using the appropriate command and get a list of running Pods. You can see one running Pod.

A YAML file, standing for "YAML Ain't Markup Language". Is a human-readable data serialization format commonly used for configuration files and data exchange. Its cross-language compatibility and readability have made it a popular choice.

`deployment.yaml`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: credit
spec:
  selector:
    matchLabels:
```

```
    app: credit
replicas: 1
template:
  metadata:
    labels:
      app: credit
  spec:
    containers:
    - name: credit
      image: zoomcamp-model:hw10
      resources:
        requests:
          memory: "64Mi"
          cpu: "100m"
        limits:
          memory: "256Mi"
          cpu: "200m"
      ports:
      - containerPort: 9696
```

The `requests` field specifies the minimum CPU and memory required by the Pod, while the `limits` field specifies the maximum CPU and memory allowed by the Pod. The `containerPort` field specifies the port on which the container listens for requests.

Now lets apply the deployment:

`Termninal`

```
marcos@marcos:~/GitHub/ML-zoomcamp/10kubernetes$ sudo kubectl apply -f deployment.yaml

deployment.apps/credit created
```

---

and check the deployment and pods:

`Termninal`

```
marcos@marcos:~$ sudo kubectl get deployment
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
credit   1/1     1            1           32s

marcos@marcos:~$ sudo kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
credit-59d5ff45f-zjxg8    1/1     Running   0          46s
```

Question 6

Let's create a service for this deployment (`service.yaml`):

```
apiVersion: v1
kind: Service
metadata:
  name: <Service name>
spec:
  type: LoadBalancer
  selector:
    app: <???>
  ports:
  - port: 80
    targetPort: <PORT>
```

Fill it in. What do we need to write instead of `<???>` ?

Apply this config file.

- **credit**

From the `deployment.yaml` file, the label assigned to our pods is `app: credit` . In `service.yaml` file, we need to use this label in the `selector` to ensure the service routes traffic to these pods. The `name` of the service, `credit-service` , is used to identify the service within the Kubernetes cluster. The `targetPort` field specifies the port on which the service will send requests to the pods.

`service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: credit
```

```
spec:
  type: LoadBalancer
  selector:
    app: credit
  ports:
  - port: 80
    targetPort: 9696
```

Let's apply the service configuration and check:

`Termninal`

```
marcos@marcos:~/GitHub/ML-zoomcamp/10kubernetes$ sudo kubectl apply -f service.yaml
service/credit-service created

marcos@marcos:~$ sudo kubectl get services
NAME             TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
credit-service   LoadBalancer   10.96.206.120   <pending>     80:31813/TCP   74s
kubernetes       ClusterIP      10.96.0.1       <none>        443/TCP        3h16m
```

Testing the service

We can test our service locally by forwarding the port 9696 on our computer to the port 80 on the service:

`kubectl port-forward service/<Service name> 9696:80`
Run `q6_test.py` (from the homework 5) once again to verify that everything is working. You should get the same result as in Question 1.

To make a request to the service running in the cluster, we need to forward the port 9696 on our computer to the port 80 on the service. This can be done using the `kubectl port-forward` command. The `service.yaml` file specifies that the service listens on port 80, while the `deployment.yaml` file specifies that the pods listen on port 9696. Therefore, we need to forward port 9696 on our computer to port 80 on the service:

```
marcos@marcos:~$ sudo kubectl port-forward service/credit 9696:80
Forwarding from 127.0.0.1:9696 -> 9696

Handling connection for 9696
```
Now, with the port forwarding in place, we can test the service's response by making requests to `http://localhost:9696/predict`, which will be routed to the credit-service:

In [ ]:
```
# Testing the cluster service response
url = "http://localhost:9696/predict"
client = {"job": "retired", "duration": 445, "poutcome": "success"}
response = requests.post(url, json=client).json()

print(response)
```

```
{'Credit probability': 0.726936946355423}
```

Autoscaling

Now we're going to use a HorizontalPodAutoscaler (HPA for short) that automatically updates a workload resource (such as our deployment), with the aim of automatically scaling the workload to match demand.

Use the following command to create the HPA:

`kubectl autoscale deployment credit --name credit-hpa --cpu-percent=20 --min=1 --max=3`
You can check the current status of the new HPA by running:

`kubectl get hpa`
The output should be similar to the next:

```
NAME              REFERENCE                TARGETS    MINPODS    MAXPODS    REPLICAS    AGE
credit-hpa    Deployment/credit    1%/20%     1          3          1           27s
```
`TARGET` column shows the average CPU consumption across all the Pods controlled by the corresponding deployment. Current CPU consumption is about 0% as there are no clients sending requests to the server.

first download the metrics-server and apply it:

`Terminal`

```
marcos@marcos:~/GitHub/ML-zoomcamp/10kubernetes$ wget
https://raw.githubusercontent.com/pythianarora/total-practice/master/sample-kubernetes-code/metrics-
```

```
server.yaml

marcos@marcos:~/GitHub/ML-zoomcamp/10kubernetes$ sudo kubectl apply -f metrics-server.yaml


serviceaccount/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
service/metrics-server created
deployment.apps/metrics-server created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created
```

Autoscaling refers to the dynamic allocation of resources to a service based on current demand. It's used to maintain application performance and reduce costs. During times of high demand, the HPA will increase the number of Pods to prevent overloading and maintain performance. During times of low demand, the HPA will reduce the number of Pods to reduce costs. We can do this as follows:

`Terminal`

```
marcos@marcos:~$ sudo kubectl autoscale deployment credit --name credit-hpa --cpu-percent=20 --min=1 --
max=3

horizontalpodautoscaler.autoscaling/credit-hpa autoscaled
```

```
marcos@marcos:~$ sudo kubectl get hpa

NAME         REFERENCE           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
credit-hpa   Deployment/credit   1%/20%    1         3         1          3m5s
```

```
marcos@marcos:~$ sudo kubectl describe hpa credit-hpa

Name:                                                  credit-hpa
Namespace:                                             default
Labels:                                                <none>
Annotations:                                           <none>
CreationTimestamp:                                     Mon, 04 Dec 2023 16:09:40 -0300
Reference:                                             Deployment/credit
Metrics:                                               ( current / target )
  resource cpu on pods  (as a percentage of request):  1% (1m) / 20%
Min replicas:                                          1
Max replicas:                                          3
Deployment pods:                                       1 current / 1 desired
Conditions:

  Type            Status  Reason              Message
  ----            ------  ------              -------
  AbleToScale     True    ReadyForNewScale    recommended size matches current size
  ScalingActive   True    ValidMetricFound    the HPA was able to successfully calculate a replica ...
  ScalingLimited  False   DesiredWithinRange  the desired count is within the acceptable range

Events:           <none>
```

The parameters `--cpu-percent=20`, `--min=1`, and `--max=3` indicate that the HPA should maintain CPU usage of the Pods at 20% of the requested CPU. If usage goes above that, the HPA will start creating new Pods (up to a maximum of 3). If the usage is low, it will reduce the number of Pods (but never below 1).

Increase the load

Let's see how the autoscaler reacts to increasing the load. To do this, we can slightly modify the existing script by putting the operator that sends the request to the credit service into a loop.

```
while True:
    sleep(0.1)
    response = requests.post(url, json=client).json()
    print(response)
```

Now you can run this script.

To test the HPA, we can artificially increase the load on our service to continuously send requests our service. This simulates high demand, prompting the HPA to scale out the number of Pods to maintain performance. Let's make the following coding running indefinitely:

In [ ]:
```python
from time import sleep

url = "http://localhost:9696/predict"
client = {"job": "retired", "duration": 445, "poutcome": "success"}
while True:
    sleep(0.1)
    response = requests.post(url, json=client).json()
```

Question 7

Run `kubectl get hpa credit-hpa --watch` command to monitor how the autoscaler performs. Within a minute or so, you should see the higher CPU load; and then - more replicas. What was the maximum amount of the replicas during this test?

- 1
- **2**
- 3
- 4

Terminal

```
marcos@marcos:~$ kubectl port-forward service/credit 9696:80
marcos@marcos:~$ sudo kubectl get hpa credit-hpa --watch
NAME         REFERENCE           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
credit-hpa   Deployment/credit   1%/20%    1         3         1          15m
credit-hpa   Deployment/credit   15%/20%   1         3         1          16m
....

credit-hpa   Deployment/credit   32%/20%   1         3         2          16m

....
```

When the CPU load (TARGETS) exceeds the threshold (20%), the HPA will increase the number of replicas (REPLICAS) to distribute the load across more pods. The maximum number of replicas it can scale to is determined by MAXPODS.