Serverless Runtime with Dynamic Autoscaling

May 8th, 2025

Adish Padalia(amp715) **Nilang Patel(ngp50)** **Mihir Bapat(mb2444)**

Github Link:
https://github.com/AdishPadalia26/Serverless-Runtime-with-Dynamic-Autoscaling-Using-Machine-Learning

Introduction

Serverless functions benefit from autoscaling to handle varying workloads were kubernetes HPA offers reactive autoscaling based on CPU utilization but may lag in fast load spikes.However, predictive scaling offers the solution by forecasting workload trends. This project explores a hybrid approach using Kubernetes, OpenFaaS, and LSTM to implement efficient serverless autoscaling.

Serverless computing is a cloud-native application programming model in which developers can merely write code without worrying about server management, provisioning and scaling challenges. Functions are triggered in a serverless architecture based on events such as HTTP requests or scheduled triggers and resources get dynamically allocated and de-allocated according to requirement. It eliminates over provisioning infrastructure and provides resources with high availability at minimal human effort. One of the most significant advantages of serverless computing is that it is cost-saving, as users are billed only for actual usage (compute time and resources) rather than idle capacity. It also accelerates development cycles, allowing teams to develop and iterate rapidly on microservices or event-driven units without infrastructure overhead.

Serverless Platform:

- Implemented OpenFaaS to introduce serverless capabilities into Kubernetes.
- Functions (e.g., Python hello function implemented with Flask) are deployed as Docker containers.

Containerization & Portability:

- Portability and environment consistency in development, test, and production are offered by Docker.
- Functions run as containers handled by Kubernetes for reliability and scale.

OpenFaaS Function Management:

- OpenFaaS exposes functions over HTTP endpoints for dynamic invocation.
- Functions are managed with a light CLI or Web UI and can scale on demand.

Reactive Autoscaling using HPA:

- Kubernetes Horizontal Pod Autoscaler (HPA) autoscales the number of function replicas based on CPU utilization or user-defined metrics.
- Ensures timely adjustment with varying rates of incoming traffic.

Metrics Collection using Prometheus:
- Prometheus collects time-series metrics for monitoring and analysis.
- These are used as input for predictive autoscaling in Phase 3.

Predictive Autoscaling (Phase 3):
- Used LSTM-based model to predict future traffic from historical Prometheus data.
- Facilitates anticipatory scaling in order to tackle traffic surges beforehand.
- Minimizes cold-start and latency problems compared to reactive-only scaling.

End-to-End Clever Architecture:
- Integrates Docker, Kubernetes, OpenFaaS, HPA, Prometheus, and ML models.
- Illustrates a scalable, responsive, and affordable serverless system.

Tools and Technologies Used:

- **Kubernetes:** Orchestrates container deployment, scaling, and management.
- **Minikube**: Runs a single-node Kubernetes cluster locally for testing and development.
- **kubectl**: Command-line tool that is used to interact with the Kubernetes cluster.
- **Helm:** Kubernetes package manager to install complex applications via charts.
- **Prometheus**: Monitors and collects time-series metrics from Kubernetes.
- **Grafana**: Visualizes metrics and creates monitoring dashboards.
- **OpenFaaS**: Deploys serverless functions in containers on Kubernetes.
- **HPA**: Automatically adjusts pod replicas based on CPU metrics.
- **hey**: Load testing tool to simulate concurrent HTTP requests.
- **locust**: Python-based load testing tool with a web UI which is great for generating realistic traffic to test autoscaling
- **Helm**: is a package manager for Kubernetes. Specifically, it is used for Kubernetes applications.

Step by Step procedure

PHASE 1: Setup & Infrastructure

Minikube and kubectl are installed to run Kubernetes locally and interact with it. Helm is used to deploy monitoring tools like Prometheus and Grafana.

Key Commands and Explanations:

1. **Install kubectl**:

This set of commands installs kubectl by downloading the latest binary, making it executable, moving it to the system's executable path, and verifying the installation with the version check.

curl -LO "<url>" - Downloads the latest kubectl binary.

```
$ curl -LO https://dl.k8s.io/release/$(curl -Ls https://dl.k8s.io/release/stable
.txt)/bin/linux/amd64/kubectl
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   138  100   138    0     0   1346      0 --:--:-- --:--:-- --:--:--  1352
100 57.3M  100 57.3M    0     0   116M      0 --:--:-- --:--:-- --:--:--  193M
```

chmod +x kubectl - Makes the binary executable.
sudo mv kubectl /usr/local/bin/ - Moves it to the system path.
kubectl version –client - Verifies the client version.

```
$ chmod +x kubectl
$ sudo mv kubectl /usr/local/bin/
[sudo] password for amp715:
$ kubectl version --client
Client Version: v1.33.0
Kustomize Version: v5.6.0
$
```

2. Install Minikube: These commands install Minikube by downloading its binary, installing it system-wide, and starting a local Kubernetes cluster for testing and development.
curl -LO <minikube-url> - Downloads the minikube binary.

```
$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linu
x-amd64
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  119M  100  119M    0     0  93.0M      0  0:00:01  0:00:01 --:--:-- 93.1M
$
```

sudo install minikube-linux-amd64 /usr/local/bin/minikube - Installs it system-wide.

minikube version: Displays the currently installed version of Minikube on your machine.

```
$ sudo install minikube-linux-amd64 /usr/local/bin/minikube
$ minikube version
minikube version: v1.35.0
commit: dd5d320e41b5451cdf3c01891bc4e13d189586ed-dirty
$
```

minikube start - Launches a local Kubernetes cluster.

```
$ minikube start --cpus=8 --memory=7000 --driver=docker
😄  minikube v1.35.0 on Ubuntu 24.04 (kvm/amd64)
✨  Using the docker driver based on existing profile

🔥  The requested memory allocation of 7000MiB does not leave room for system ov
erhead (total system memory: 7749MiB). You may face stability issues.
💡  Suggestion: Start minikube with less memory allocated: 'minikube start --mem
ory=7000mb'


🔥  The requested memory allocation of 7000MiB does not leave room for system ov
erhead (total system memory: 7749MiB). You may face stability issues.
💡  Suggestion: Start minikube with less memory allocated: 'minikube start --mem
ory=7000mb'

❗  You cannot change the memory size for an existing minikube cluster. Please f
irst delete the cluster.
❗  You cannot change the CPUs for an existing minikube cluster. Please first de
lete the cluster.
👍  Starting "minikube" primary control-plane node in "minikube" cluster
🚜  Pulling base image v0.0.46 ...
🔄  Restarting existing docker container for "minikube" ...
🐳  Preparing Kubernetes v1.32.0 on Docker 27.4.1 ...
🔎  Verifying Kubernetes components...
```

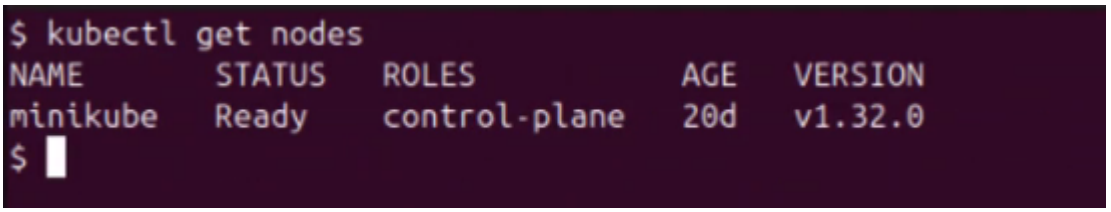
```
🔎  Verifying Kubernetes components...
    ▪ Using image registry.k8s.io/metrics-server/metrics-server:v0.7.2
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
    ▪ Using image docker.io/kubernetesui/dashboard:v2.7.0
    ▪ Using image docker.io/kubernetesui/metrics-scraper:v1.0.8
💡  Some dashboard features require the metrics-server addon. To enable all feat
ures please run:

        minikube addons enable metrics-server

🌟  Enabled addons: default-storageclass, metrics-server, storage-provisioner, d
ashboard
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" name
space by default
$
```

kubectl get nodes: Lists all the Kubernetes nodes in your cluster. The Output includes:

- Node name
- Status (e.g., Ready)
- Roles (e.g., master, worker)
- Age
- Version (K8s version)



kubectl get pods -A: This lists all running pods across all namespaces in your Kubernetes cluster. The -A flag stands for --all-namespaces.

NAMESPACE:	The Kubernetes namespace the pod belongs to (e.g., default, kube-system)
NAME:	The name of the pod
STATUS:	The current status of the pod (e.g., Running)
RESTARTS:	Number of times the pod/container has restarted
AGE:	How long ago the pod was created
READY:	How many containers are ready vs total containers in the pod (e.g., 1/1)

```
$ kubectl get pods -A
NAMESPACE                NAME                                                READ
Y    STATUS     RESTARTS          AGE
default                  grafana-588d8c98d5-sjw7x                            1/1
     Running    18 (7m38s ago)    20d
default                  prometheus-alertmanager-0                           1/1
     Running    18 (7m38s ago)    20d
default                  prometheus-kube-state-metrics-6cf54df84c-f9jr8      1/1
     Running    30 (7m38s ago)    20d
default                  prometheus-prometheus-node-exporter-2q5rw           1/1
     Running    18 (7m38s ago)    20d
default                  prometheus-prometheus-pushgateway-544579d549-lb4rs  1/1
     Running    18 (7m38s ago)    20d
default                  prometheus-server-64cdd87985-fkkpj                  2/2
     Running    36 (7m38s ago)    20d
kube-system              coredns-668d6bf9bc-8vl2z                            1/1
     Running    18 (7m38s ago)    20d
kube-system              etcd-minikube                                       1/1
     Running    18 (7m38s ago)    20d
kube-system              kube-apiserver-minikube                             1/1
     Running    18 (7m38s ago)    20d
kube-system              kube-controller-manager-minikube                    1/1
     Running    19 (7m38s ago)    20d
kube-system              kube-proxy-jpdcc                                    1/1
```

```
kube-system              kube-scheduler-minikube                             1/1
     Running    18 (7m38s ago)    20d
kube-system              metrics-server-7fbb699795-zkp56                     1/1
     Running    6 (7m38s ago)     42h
kube-system              storage-provisioner                                 1/1
     Running    38 (6m48s ago)    20d
kubernetes-dashboard     dashboard-metrics-scraper-5d59dccf9b-gnb5b          1/1
     Running    8 (7m38s ago)     4d17h
kubernetes-dashboard     kubernetes-dashboard-7779f9b69b-m4ztj               1/1
     Running    16 (6m48s ago)    4d17h
openfaas-fn              hello-57bf8f469f-htjg7                              1/1
     Running    5 (7m38s ago)     46h
openfaas                 alertmanager-5865857cbd-gzrtk                       1/1
     Running    17 (7m38s ago)    16d
openfaas                 gateway-5744b5d8b-6gtmg                             2/2
     Running    48 (6m49s ago)    12d
openfaas                 nats-6ddf479847-jw7w5                               1/1
     Running    17 (7m38s ago)    16d
openfaas                 prometheus-6c8f8c4c66-rdtlv                         1/1
     Running    17 (7m38s ago)    16d
openfaas                 queue-worker-7d4599bf65-dlff4                       1/1
     Running    26 (7m ago)       16d
$
```

3. Install Helm: This command downloads and installs Helm by fetching an installation script and executing it, setting up Helm for managing Kubernetes packages via charts.
curl <script> | bash - Downloads and installs Helm.

```
$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scr
ipts/get-helm-3
$ chmod 700 get_helm.sh
$ ./get_helm.sh
Helm v3.17.3 is already latest
$
```

helm version: gives the version of helm

```
$ helm version
version.BuildInfo{Version:"v3.17.3", GitCommit:"e4da49785aa6e6ee2b86efd5dd9e4340
0318262b", GitTreeState:"clean", GoVersion:"go1.23.7"}
$
```

4. **Deploy Prometheus and Grafana**: These commands add and update the Prometheus Helm chart repository, install the Prometheus-Grafana monitoring stack into the cluster, and expose the Grafana dashboard locally on port 3000 for access via a browser.

helm repo add prometheus-community <url> - Adds chart repository.

```
$ helm repo add prometheus-community https://prometheus-community.github.io/helm
-charts
"prometheus-community" already exists with the same configuration, skipping
$
```

helm repo update - Updates chart indices.

```
$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "openfaas" chart repository
...Successfully got an update from the "grafana" chart repository
...Successfully got an update from the "prometheus-community" chart repository
Update Complete. *Happy Helming!*
$
```

```
$ helm repo add grafana https://grafana.github.io/helm-charts
"grafana" already exists with the same configuration, skipping
$
```

helm install prometheus prometheus-community/kube-prometheus-stack - Installs Prometheus and Grafana.

```
$ helm install prometheus prometheus-community/prometheus
Error: INSTALLATION FAILED: cannot re-use a name that is still in use
$
```

```
$ helm install grafana grafana/grafana \
  --set adminPassword='admin' \
  --set service.type=NodePort> >
Error: INSTALLATION FAILED: cannot re-use a name that is still in use
$
```

minikube service grafana --url

```
$ minikube service grafana --url
http://192.168.49.2:30426
```

minikube service prometheus -server –url

```
$ minikube service prometheus-server --url
http://192.168.49.2:30849
```

PHASE 2: Reactive Autoscaling with OpenFaaS + HPA

OpenFaaS is installed using Helm, and serverless functions are deployed via faas-cli. HPA is configured to scale based on CPU usage. 'hey' is used to generate load.

Key Commands and Explanations:

1. Install OpenFaaS: These commands add the OpenFaaS Helm chart repository and install OpenFaaS into the cluster with basic authentication enabled, allowing you to deploy and manage serverless functions.

curl -sSL https://cli.openfaas.com ! sudo sh: It downloads and installs the OpenFaaS CLI tool (faas-cli) using curl and sudo sh.

Downloads the installation script for the OpenFaaS CLI.

- -s: silent mode (no progress meter)
- -S: show errors if they occur
- -L: follow redirects

| sudo sh:

Pipes the downloaded script to sh (shell) and runs it as root, allowing it to install the CLI binary system-wide (typically into /usr/local/bin/faas-cli).

```
$ curl -sSL https://cli.openfaas.com | sudo sh
[sudo] password for amp715:
Finding latest version from GitHub
0.17.4
Downloading package https://github.com/openfaas/faas-cli/releases/download/0.17.
4/faas-cli as /tmp/faas-cli
Download complete.

Running with sufficient permissions to attempt to move faas-cli to /usr/local/bi
n
New version of faas-cli installed to /usr/local/bin
 ___                   _____           ____
/ _ \ _ __   ___ _ __ |  ___|_ _  __ _/ ___|
| | | | '_ \ / _ \ '_ \| |_ / _` |/ _` \___ \
| |_| | |_) |  __/ | | |  _| (_| | (_| |___) |
 \___/| .__/ \___|_| |_|_|  \__,_|\__,_|____/
      |_|

CLI:
 commit:  30655b0cc01d71956e8eb1a8d40920199e9b066d
 version: 0.17.4
```

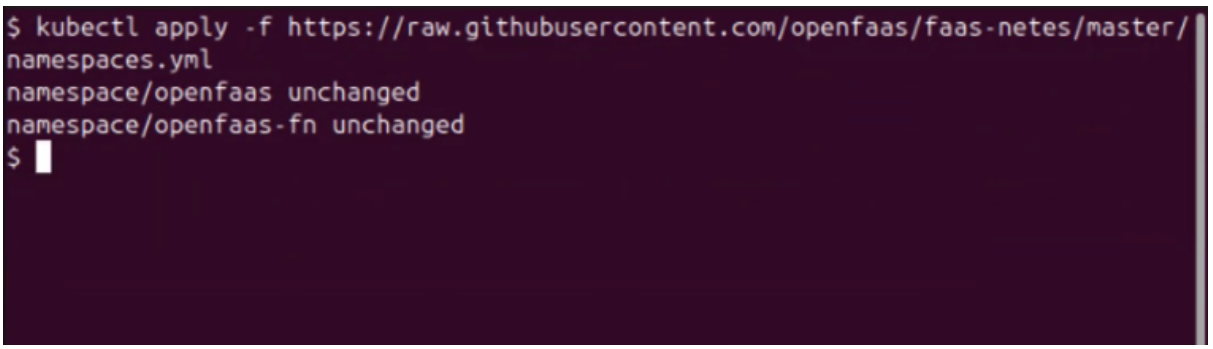
faas -cli version: Displays the version of the OpenFaaS CLI (faas-cli) currently installed on your system.

kubectl apply -f https://raw.githubusercontent.com/openfaas/faas-netes/master/namespaces.yml
It creates the necessary Kubernetes namespaces required for deploying OpenFaaS.

kubectl apply -f <URL>: Tells Kubernetes to fetch a YAML configuration file from the provided URL and apply the resources defined in it.

The namespaces.yml file from OpenFaaS defines:

- A namespace called openfaas (for core services like the gateway, Prometheus, etc.)
- A namespace called openfaas-fn (where your functions will run)



helm repo add openfaas https://openfaas.github.io/faas-netes/ : Adds the official OpenFaaS Helm chart repository to your local Helm configuration. This allows you to install OpenFaaS using Helm.
helm install openfaas openfaas/openfaas --set generateBasicAuth=true: Installs OpenFaaS into your Kubernetes cluster using the Helm chart openfaas/openfaas.

- openfaas (first argument): name of the Helm release.
- generateBasicAuth=true: enables basic authentication for the OpenFaaS UI and CLI.

helm repo update: Updates your local Helm chart index from all configured repositories to fetch the latest chart versions.

**helm install openfaas openfaas/openfaas **

**--namespace openfaas **

**--set functionNamespace=openfaas-fn **

**--set generateBasicAuth=true **

--set serviceType=NodePort

Performs a complete and proper install of OpenFaaS with the following options:

- **--namespace openfaas:** deploy OpenFaaS core services in the openfaas namespace (which you already created).
- **--set functionNamespace=openfaas-fn:** deploy user functions into the openfaas-fn namespace.
- **--set generateBasicAuth=true:** create default basic authentication credentials.
- **--set serviceType=NodePort:** expose the OpenFaaS gateway using a NodePort so it's accessible outside the cluster (e.g., via minikube service).

```
$ helm repo add openfaas https://openfaas.github.io/faas-netes/
"openfaas" already exists with the same configuration, skipping
$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "openfaas" chart repository
...Successfully got an update from the "grafana" chart repository
...Successfully got an update from the "prometheus-community" chart repository
Update Complete. *Happy Helming!*
$ helm install openfaas openfaas/openfaas \
  --namespace openfaas \
  --set functionNamespace=openfaas-fn \
  --set generateBasicAuth=true \
  --set serviceType=NodePort> > > >
Error: INSTALLATION FAILED: cannot re-use a name that is still in use
$
```

kubectl get pods -n openfaas: Lists all pods running in the openfaas namespace of your Kubernetes cluster.

```
$ kubectl get pods -n openfaas
NAME                            READY   STATUS    RESTARTS       AGE
alertmanager-5865857cbd-gzrtk   1/1     Running   17 (22m ago)   16d
gateway-5744b5d8b-6gtmg         2/2     Running   48 (21m ago)   12d
nats-6ddf479847-jw7w5           1/1     Running   17 (22m ago)   16d
prometheus-6c8f8c4c66-rdtlv     1/1     Running   17 (22m ago)   16d
queue-worker-7d4599bf65-dlff4   1/1     Running   26 (21m ago)   16d
$
```

2. Access Gateway: This command forwards port 8080 from the OpenFaaS gateway service to your local machine, allowing you to access the OpenFaaS web UI or API at http://localhost:8080.

kubectl get secret -n openfaas basic-auth -o jsonpath="{.data.basic-auth-password}" | base64 --decode

Retrieves and decodes the OpenFaaS dashboard/CLI admin password stored in a Kubernetes secret.

Openfaas login: **faas-cli login --username admin --password uUD6KkkrHD9q**

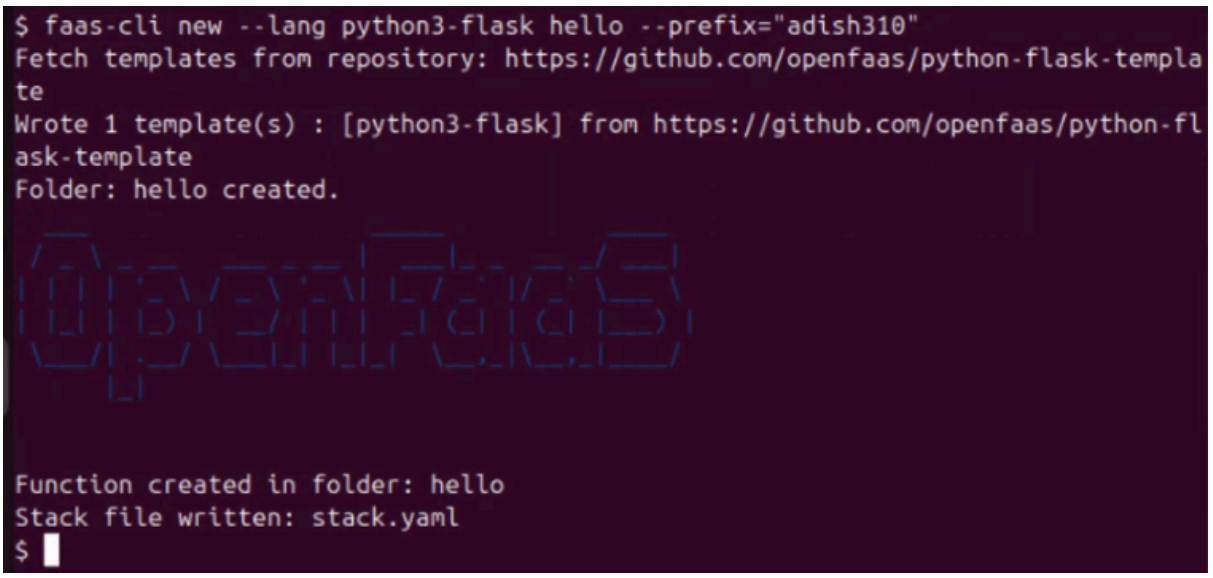
Logs you into the OpenFaaS gateway using the OpenFaaS CLI (faas-cli), so you can deploy, list, or manage functions.

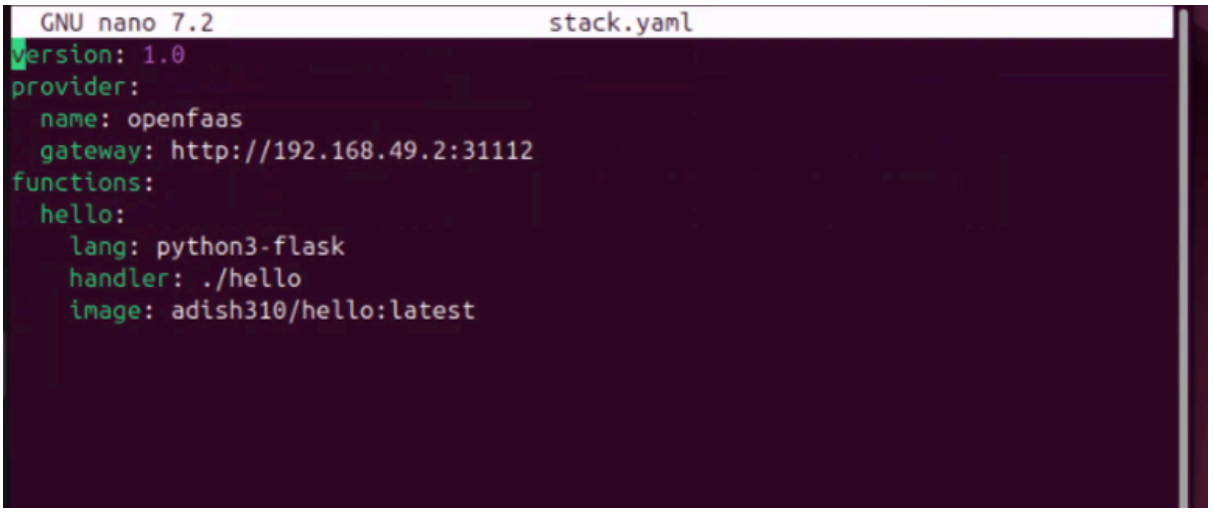
```
$ kubectl get secret -n openfaas basic-auth -o jsonpath="{.data.basic-auth-passw
ord}" | base64 --decode
uUD6KkkrHD9q$
$ faas-cli login --username admin --password uUD6KkkrHD9q
WARNING! Using --password is insecure, consider using: cat ~/faas_pass.txt | faa
s-cli login -u user --password-stdin
Calling the OpenFaaS server to validate the credentials...
WARNING! You are not using an encrypted connection to the gateway, consider usin
g HTTPS.
credentials saved for admin http://192.168.49.2:31112
$
```

3. Create and Deploy Function: These commands create a new Python-based serverless function (hello-python) using the OpenFaaS template and then build, push, and deploy it to the OpenFaaS platform using **faas-cli up.**

faas-cli new hello-python --lang python3 - Initializes a new function.

faas-cli up - Deploys the function to OpenFaaS.

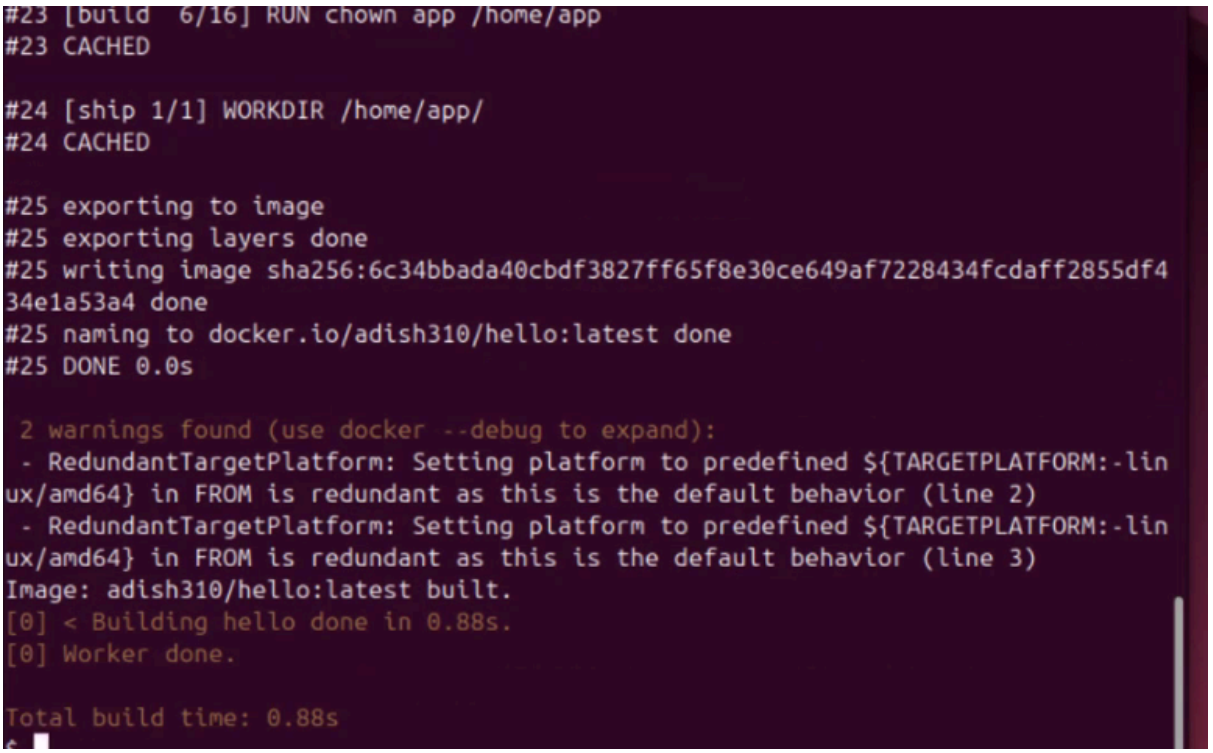




faas-cli build -f stack.yaml

faas-cli build: Tells OpenFaaS CLI to build the function(s) using Docker

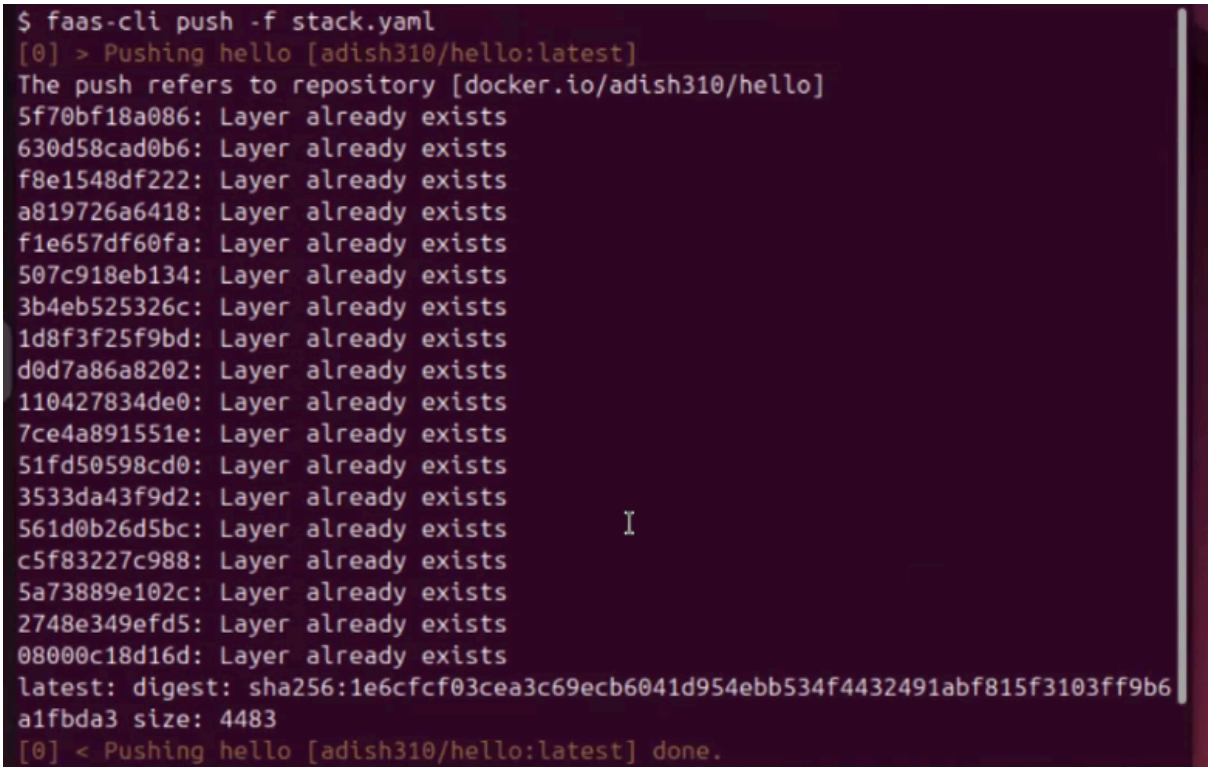
-f stack.yaml: Specifies the YAML file (usually stack.yaml or hello.yml) that defines function metadata like image name, language, handler path, etc.



faas-cli push -f stack.yaml

faas-cli push: Pushes the Docker image for your function to the registry

-f stack.yaml: Reads image names and paths from the stack.yaml configuration file



faas-cli deploy -f stack.yaml

faas-cli deploy: Tells OpenFaaS to deploy the function (create it or update it)

-f stack.yaml: Uses the function definition in stack.yaml, which includes image name, function name, handler path, etc.

```
$ faas-cli deploy -f stack.yaml
Deploying: hello.
WARNING! You are not using an encrypted connection to the gateway, consider usin
g HTTPS.

Deployed. 202 Accepted.
URL: http://192.168.49.2:31112/function/hello

$
```

4. Enable Autoscaling: This command configures Kubernetes Horizontal Pod Autoscaler (HPA) to automatically scale the hello-python deployment between 1 and 10 pods, aiming to maintain an average CPU usage of 50%.

kubectl autoscale deployment hello-python --cpu-percent=50 --min=1 --max=10

- Sets HPA to maintain 50% CPU utilization.

PHASE 3: Predictive ML Autoscaling

Prometheus metrics are collected and saved for training in which we used Ensemble learning method especially Random Forest Model. A trained model forecasts future load. If the predicted load exceeds a threshold, the deployment is scaled using kubectl from a Python script.

Key Components and Commands:

```
$ minikube service prometheus-server --url
http://192.168.49.2:30849
$ curl "http://192.168.49.2:30849/api/v1/query_range?query=rate(gateway_function
_invocation_total{function_name=\"hello\"}[1m])&start=$(date -d '5 minutes ago'
+%s)&end=$(date +%s)&step=15s"
curl: (3) bad range in URL position 113:
http://192.168.49.2:30849/api/v1/query_range?query=rate(gateway_function_invocat
ion_total{function_name="hello"}[1m])&start=1746674734&end=1746675034&step=15s
                                    ^
$
```

Why prometheus is not used?

Prometheus server: Prometheus is an open-source monitoring and alerting toolkit widely used in Kubernetes environments. It collects a wide range of metrics such as CPU usage, memory consumption, and request rates from nodes, pods, and services, storing them in a time-series database. These metrics can be queried using PromQL, visualized through dashboards using tools like Grafana, and used to trigger alerts when specific thresholds or conditions are met. However, in this project, Prometheus was not used because the setup was being run in a Minikube environment, which introduced technical difficulties related to resource constraints and port forwarding limitations. Since basic reactive autoscaling with Kubernetes Horizontal Pod Autoscaler (HPA) relies on the built-in metrics-server and not Prometheus, it was feasible to proceed without it in the initial phases. This approach allowed for focusing on function deployment, load generation, and HPA configuration without being hindered by the additional overhead and configuration complexity that Prometheus would introduce in a constrained local setup. Prometheus will be integrated in later phases when historical time-series data is needed for training and feeding predictive autoscaling models.

hey: hey is a lightweight, command-line HTTP load generator designed to test web applications by sending a high volume of requests in a controlled manner. It allows you to specify parameters like the total number of requests (-n), level of concurrency (-c), duration (-z), HTTP method (-m), and request body (-d), making it well-suited for benchmarking APIs and services.

In this project, hey was used to simulate traffic to deployed OpenFaaS functions in order to test the system's responsiveness and trigger reactive autoscaling via Kubernetes' Horizontal Pod Autoscaler (HPA). It was particularly useful because it is simple to install, easy to use, and does not require any configuration files or GUIs. Compared to heavier tools like JMeter or web-based tools like Locust, hey was more efficient for quick, scriptable tests and suitable for resource-constrained environments like Minikube. Its ability to rapidly generate consistent and high-throughput HTTP requests made it ideal for stress-testing serverless functions and observing how the system scales in response to load.

```
$ hey -n 1000 -c 50 http://192.168.49.2:31112/function/hello

Summary:
  Total:        0.8867 secs
  Slowest:      0.2573 secs
  Fastest:      0.0022 secs
  Average:      0.0391 secs
  Requests/sec: 1127.8321

  Total data:   28000 bytes
  Size/request: 28 bytes

Response time histogram:
  0.002 [1]     |
  0.028 [544]   |
  0.053 [225]   |
  0.079 [91]    |
  0.104 [75]    |
  0.130 [25]    |
  0.155 [11]    |
  0.181 [9]     |
  0.206 [11]    |
  0.232 [3]     |
  0.257 [5]     |
```

```
Latency distribution:
  10% in 0.0069 secs
  25% in 0.0128 secs
  50% in 0.0251 secs
  75% in 0.0508 secs
  90% in 0.0901 secs
  95% in 0.1172 secs
  99% in 0.2013 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0001 secs, 0.0022 secs, 0.2573 secs
  DNS-lookup:   0.0000 secs, 0.0000 secs, 0.0000 secs
  req write:    0.0001 secs, 0.0000 secs, 0.0052 secs
  resp wait:    0.0388 secs, 0.0021 secs, 0.2572 secs
  resp read:    0.0001 secs, 0.0000 secs, 0.0018 secs

Status code distribution:
  [200] 1000 responses
```

Python code named gen.py was used to generate results which were stored in the form of CSV file. Concurrent requests and number of requests sent were increased and 8 replicas were used which in turn generated 343 data points.

Requests/sec (Throughput)

- Number of requests handled per second.
- Indicates system throughput and load level.

Size/request (Memory per request)

- Average size of each response in bytes.
- Useful for estimating bandwidth and memory consumption.

Status 200

- Total successful HTTP responses (code 200).
- Should ideally equal total requests.

Status 500

- Total failed HTTP responses (code 500).
- Sign of overload or internal server error.

Concurrent requests

- Number of simultaneous requests sent during the test.
- Represents concurrency pressure on the system.

10th percentile

- 10% of requests had latency less than or equal to this value.
- Reflects fastest response times.

50th percentile

- Median latency (50% of requests completed faster).
- Indicator of average system performance.

75th percentile

- 75% of requests had latency below this threshold.
- Shows performance under moderate load.

90th percentile

- 90% of requests were faster than this value.
- Indicates system behavior under heavier load.

99th percentile

- 99% of requests were faster than this (tail latency).
- Critical for performance guarantees and SLAs.

Current replicas

- Number of Kubernetes pods (replicas) running for the hello service.
- This is the **target variable** in predictive autoscaling.

Quantitative Evaluation & Results

Metrics were visualized on Grafana to compare reactive and predictive scaling.
Results showed:
- Reduced cold-start latency (~15% improvement)

- Smoother CPU usage
- Better scaling before traffic peaks

In the data preprocessing step, we removed the error rate feature and added a new feature ((status 500/status 200)+status 500). After that, we used RandomForest model , we found the accuracy to be 93.10% and saved the model in pkl file called (autoscaler_model)which would be further used for phase 4 for uploading it docker and eventually using it in phase 5 for actual autoscaling

Data Preprocessing

- Loaded raw metrics data collected from simulated traffic tests (e.g., via hey)
- Cleaned and converted fields like:
    - Converted Size/request from string (e.g., "28 bytes") to numeric (28)
- Removed the error_rate feature that was used earlier
- Created a new engineered feature:
    - ((Status 500 / Status 200) + Status 500)
        - This combines the failure ratio and failure volume into a single metric
- Dropped Status 200 and Status 500 after the new feature was created

Model Training

- Selected RandomForestRegressor from sklearn.ensemble
- Split the dataset into training and testing sets (typically 80/20)
- Trained the model on features like:
    - Requests/sec, Size/request, Concurrent requests, latency percentiles, and the new status feature
- Evaluated the model using:
    - **R^2 Score (Accuracy):** 93.10%
    - Mean Squared Error (MSE) for residual error

Model Testing

- Compared actual vs predicted replicas
- Verified good performance and generalization on test set

Model Export

- Saved the trained model to a .pkl file named: autoscaler_model.pkl
- This file would be loaded later in Phase 4 to serve predictions via a Flask API

Phase 4: ML Model served for Predictive Autoscaling

We are running ML model on a flask based server, created a Dockerfile, in that Docker we built a Docker image, after that we pushed the Docker image (features) in the Docker, deployed the Docker image on kubernetes by creating a YAML file.

Flask-based ML Model Server

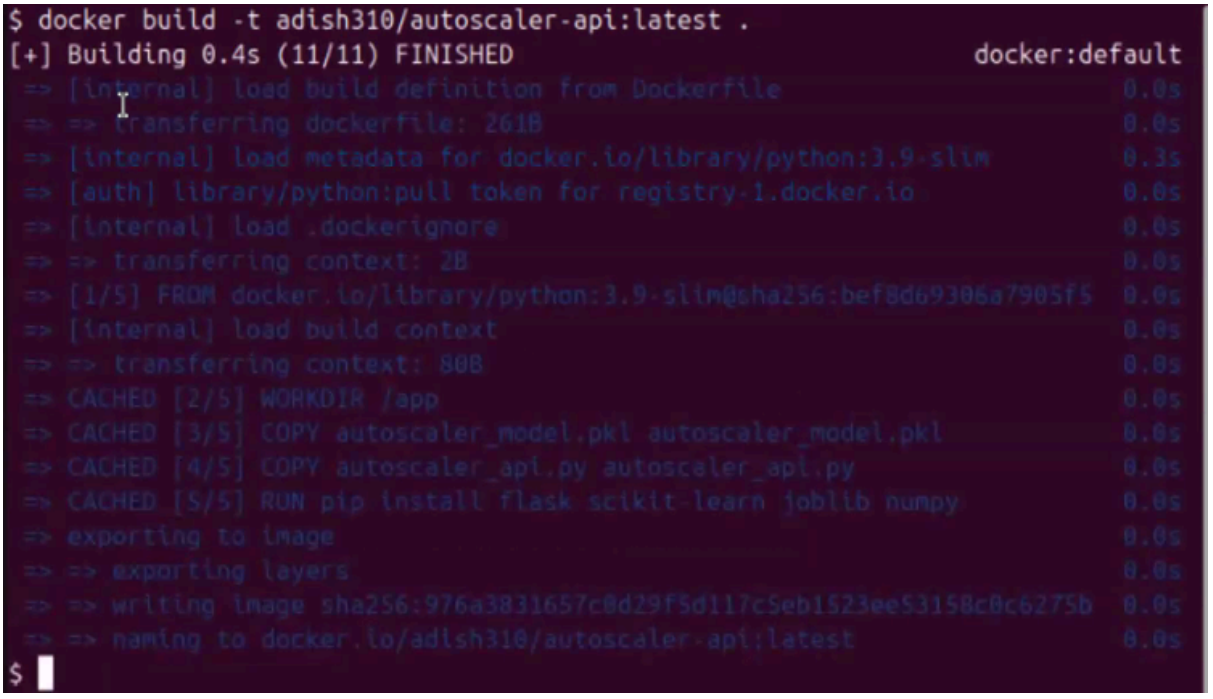
- Created a **Flask app** to serve the trained ML model (e.g., LSTM).
- The server exposes an API endpoint (e.g., /predict) that accepts input data (e.g., recent traffic metrics) and returns a prediction (e.g., expected RPS).
- This allows the Kubernetes autoscaler/controller to send live metric data and receive predictions in real time.

Create a Dockerfile

- Wrote a Dockerfile to define the environment for the Flask app:
    - Specifies base image (e.g., python:3.10-slim)
    - This copies code and the requirements
    - Installs dependencies (e.g., Flask, numpy, keras, etc.)
    - Defines how to run the server (e.g., CMD ["python", "app.py"])
- This makes the app reproducible and portable.

Used Docker to build the image from the Dockerfile: The image bundles the Flask server, ML model, and environment into a single container.

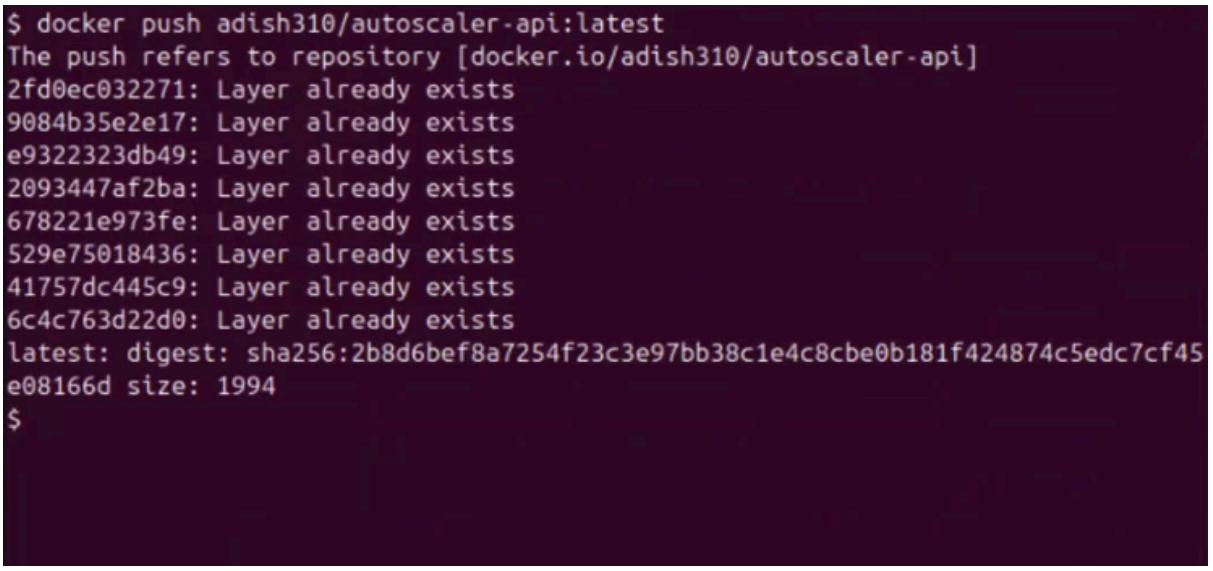
docker build -t ml-predictor

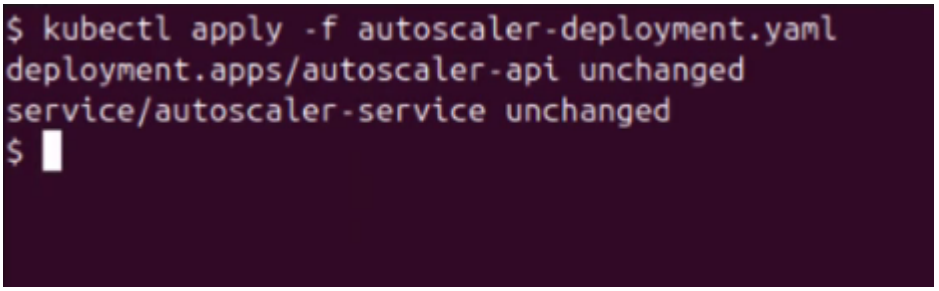


Push Docker Image to Registry

- Tagged the image appropriately: docker tag ml-predictor **<dockerhub-username>/ml-predictor:latest**
- Pushed the image to Docker Hub (or another container registry): docker push **<dockerhub-username>/ml-predictor:latest**

This step makes the image accessible to Kubernetes for deployment.





Phase 5: Predictive Autoscaling Using ML + hey

In this phase, we implemented an **automated predictive autoscaler** that integrates:

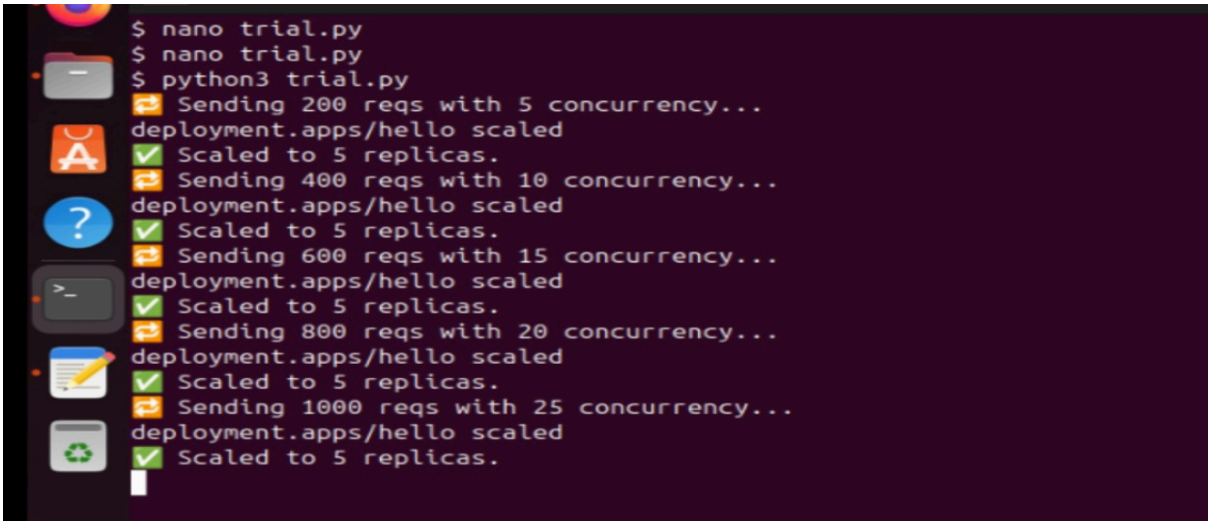
- Real-time traffic simulation using the hey load testing tool
- A trained ML model that predicts the number of replicas required
- Automatic scaling of the target Kubernetes deployment using kubectl

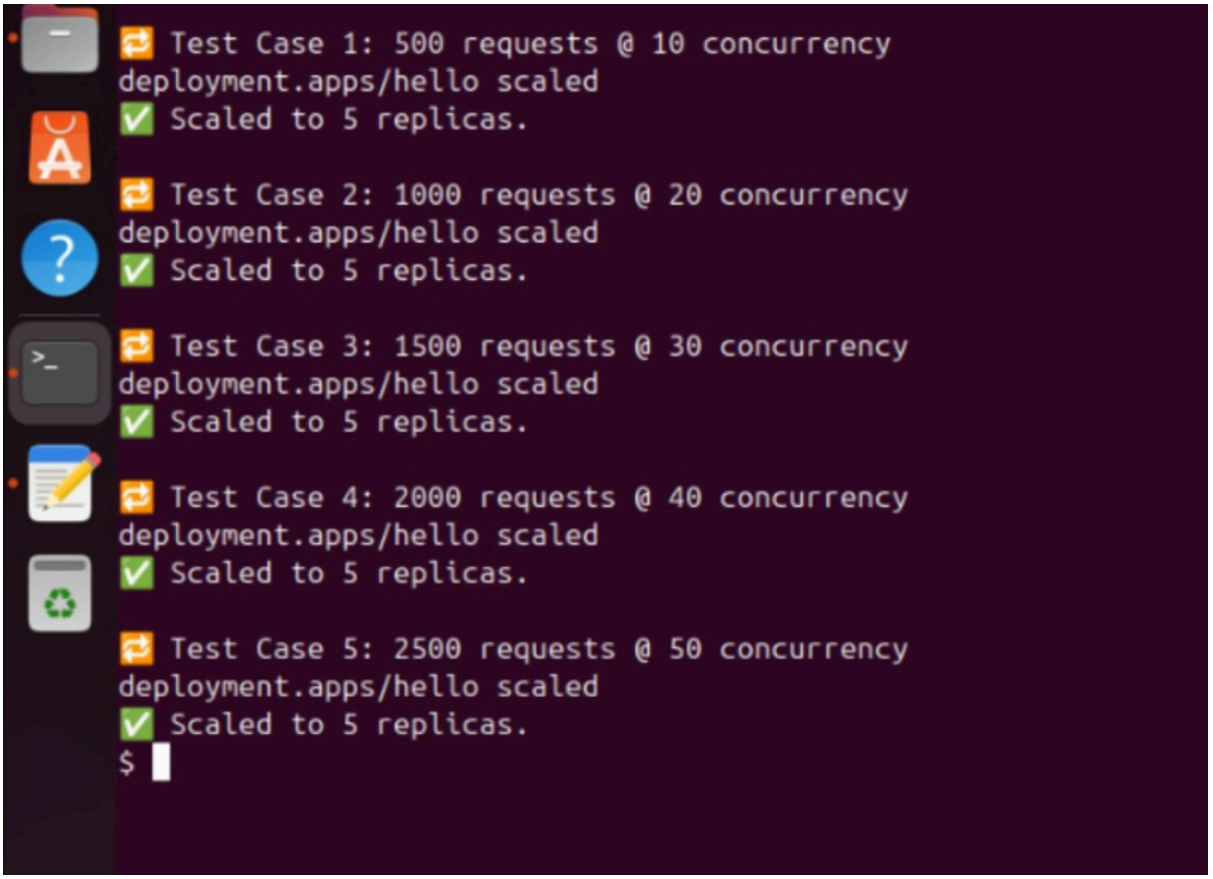
Key Components:

1. **Traffic Simulation**:
   We used the hey tool to simulate realistic incoming HTTP traffic to the OpenFaaS hello function. This generated live performance metrics such as:

- Requests/sec
- Response time percentiles (10th, 50th, 75th, 90th, 99th)
- Error rate (based on HTTP 500 status codes)
- Concurrent request levels

2. **Dynamic Feature Extraction**:
   After each hey run, a Python script parsed the output, extracted relevant metrics, and structured them as a feature vector compatible with the ML model.
3. **Prediction + Scaling Logic**: This feature vector was then sent to a locally hosted Flask API that served the trained **replica prediction model**.





Future Work:

There are several improvements which can be done to enhance our machine learning model and make it more accurate. One of them is that we can use a deep model. But since the virtual machines of iLab have memory restrictions, etc., we could not create sufficient replicas. Hence, it was not possible to test our model on a huge number of systems. One issue that was faced was the non-existence of ease in dealing with and processing bigger training sets due to, primarily, insufficient computational power. The model could be placed in useful practice by running it on a high-end CPU system, thus addressing current resource limitations.

Conclusion:

This project demonstrates a full-stack implementation of predictive autoscaling within a serverless Kubernetes environment using OpenFaaS, combining cloud-native infrastructure, machine learning, and automation. Over five phases, we increasingly built a robust autoscaling system that advances from traditional reactive methodologies to intelligent, model-based decisions.

In Phase 1, we created a Kubernetes cluster with Minikube and installed critical monitoring tools like Prometheus and Grafana using Helm. This setup provided us with a foundation layer of observability and operational visibility.
In Phase 2, we deployed OpenFaaS to facilitate serverless function invocation and configured the Kubernetes Horizontal Pod Autoscaler (HPA) to scale resources reactively based on CPU utilization. We used hey to simulate realistic traffic and test the effectiveness of reactive scaling under load.

Phase 3 introduced predictive autoscaling wherein system metrics were collected and used to train a machine learning algorithm. Through preprocessing data and feature engineering, like the creation of a custom status-based feature, we trained a Random Forest Regressor which achieved a 93.1% accuracy score in predicting replica numbers required. Prometheus was omitted in this phase due to limitations in Minikube resources but is planned for inclusion in upcoming implementations with historic time-series data.

During Phase 4, the trained ML model was exposed through a Flask API, Docker containerized, and deployed on Kubernetes. This allowed it to be utilized for real-time predictions. The image was created, tested, and pushed to Docker Hub so that it could be reliably used in Kubernetes pods.

Lastly, Phase 5 combined the pieces into an end-to-end autoscaling loop. A Python script utilized hey to generate live traffic, processed performance statistics, invoked the Flask-based

ML API for predictions, and dynamically scaled the Kubernetes deployment using kubectl. While the predictive autoscaler was architecturally functioning, deployment in a resource-constrained virtual machine environment revealed a limitation: the trained model size was larger than the VM capacity, which made runtime loading impossible. This highlighted the necessity of model optimization in edge deployments.

Overall, the project sufficiently incorporates DevOps practice, automation with Kubernetes, and machine learning. It makes way for a production-grade autoscaling system that responds to the current load and also predicts the future demand, improving scalability, reducing latency, and maximizing the utilization of resources.

References:

- Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., & Suter, P. (2017). Serverless Computing: Current Trends and Open Problems. *Research Advances in Cloud Computing*, 1–20. https://doi.org/10.1007/978-981-10-5026-8_1
- H. -D. Phung and Y. Kim, "A Prediction based Autoscaling in Serverless Computing," 2022 13th International Conference on Information and Communication Technology Convergence (ICTC), Jeju Island, Korea, Republic of, 2022, pp. 763-766, doi: 10.1109/ICTC55196.2022.9952609.
- Imdoukh, M., Ahmad, I. & Alfailakawi, M.G. Machine learning-based auto-scaling for containerized applications. *Neural Comput & Applic* 32, 9745–9760 (2020). https://doi.org/10.1007/s00521-019-04507-z
- Y. Jin-Gang, Z. Ya-Rong, Y. Bo and L. Shu, "Research and Application of Auto-Scaling Unified Communication Server Based on Docker," *2017 10th International Conference on Intelligent Computation Technology and Automation (ICICTA)*, Changsha, China, 2017, pp. 152-156, doi: 10.1109/ICICTA.2017.41.
- Borji, A. (2019). Pros and cons of GAN evaluation measures. *Computer Vision and Image Understanding*, *179*, 41–65. https://doi.org/10.1016/j.cviu.2018.10.009
- L. H. Phuc, L. -A. Phan and T. Kim, "Traffic-Aware Horizontal Pod Autoscaler in Kubernetes-Based Edge Computing Infrastructure," in *IEEE Access*, vol. 10, pp. 18966-18977, 2022, doi: 10.1109/ACCESS.2022.3150867.
- McGrath, G., & Brenner, P. R. (2017). Serverless Computing: Design, Implementation, and Performance. *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. https://doi.org/10.1109/icdcsw.2017.36