Homwork 3

Shiraz University

Cloud Computing – Spring 2025

Instructor: Dr.Khunjosh
Students: Sina Hakimzadeh
Student IDs: 40131857
Assignment: Exploring Modern Cloud Paradigms: Kubernetes (K8S), Serverless and Ray
Due Date: *June 15, 2025*

Introduction

This report explores three major cloud computing technologies: Kubernetes, serverless computing with OpenFaaS, and distributed computing with Ray. All tasks were completed using a local Kubernetes cluster created with kind (Kubernetes IN Docker). The assignment is divided into three parts. First, I set up the Kubernetes cluster, deployed a basic Nginx web server, exposed it for browser access, scaled it, and added monitoring with Prometheus and Grafana. Second, I installed OpenFaaS to test serverless computing by writing and deploying a simple Python function. Third, I deployed a Ray cluster on Kubernetes and ran a distributed Python application, using the Ray Dashboard to track the workload. Each part includes commands, screenshots, and explanations. The report ends with a comparison between Ray and OpenFaaS to highlight their differences in how they work and when each is best used. This assignment gave me practical experience with cloud-native tools and showed how they can be used together in real applications.

1 Prerequisites & Installation

This project requires a set of tools commonly used in Kubernetes-based cloud development. The following components must be installed on a macOS system before proceeding:

- **kubectl**: The Kubernetes command-line tool used to interact with clusters.
- **kind** (Kubernetes IN Docker): A tool for running local Kubernetes clusters using Docker.
- **Lens**: A powerful IDE for managing and monitoring Kubernetes clusters.
- **faas-cli**: The command-line interface for OpenFaaS, used for managing serverless functions (optional, depending on project requirements).
- **Helm**: A package manager for Kubernetes that simplifies deployment of applications.

Installation Steps

1. Install the required tools using Homebrew:

```
brew install kubectl
brew install kind
brew install faas-cli
brew install helm
brew install --cask lens
```

2. Verify each installation:

```
kubectl version --client
kind version
faas-cli version
helm version
```

Lens can be launched from the Applications folder or via Spotlight search.

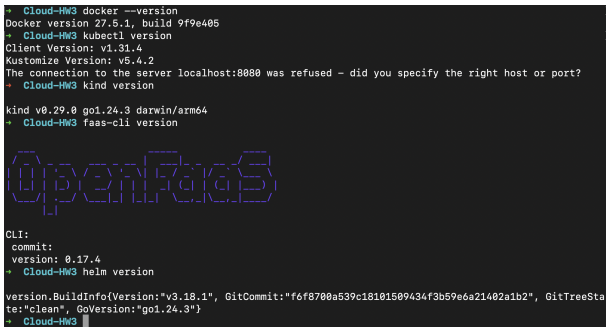


Figure 1: Installation Verification

2 Kubernetes Fundamentals with kind

In this part of homework, I had a practical experience with Kubernetes which is a container orchestration platform that automates the deployment, management, and scaling of containerized applications. We are using a single system to simulate a cloud environment, So we need to use a container to run Kubernetes in it.

Task 2.1: Create a Kind Cluster

To begin working with Kubernetes locally, we use **Kind (Kubernetes IN Docker)** to create a lightweight, single-node Kubernetes cluster.

Pull Kubernetes Image

To create a Kubernetes cluster, we pull its image from docker hub using the following command:

```
docker pull kindest/node:v1.33.1
```

Cluster Creation Steps

The following command was used to create the cluster:

```
kind create cluster --name cloud-homework
```

This command initiates the setup of a local cluster named `cloud-homework`. Kind uses Docker to spin up the Kubernetes control-plane inside a container.

Cluster Verification

To verify the cluster was created successfully and is operational, we used:

```
kubectl cluster-info --context kind-cloud-homework
kubectl get nodes
```

The output confirms:

- The Kubernetes control plane is running.
- CoreDNS is operational.
- The single node is in the `Ready` state with Kubernetes version v1.33.1.

What is CoreDNS? CoreDNS is a built-in tool in Kubernetes wchich helps services find each other by name. It works like a phonebook for the cluster. Instead of using IP addresses, apps can use names like `nginx-service` to talk to each other. CoreDNS turns those names into the correct IP addresses automatically. This makes it easier for different parts of your app to connect inside the cluster.

Command Output

Figure 2 shows the terminal output confirming the successful creation and readiness of the Kind cluster.

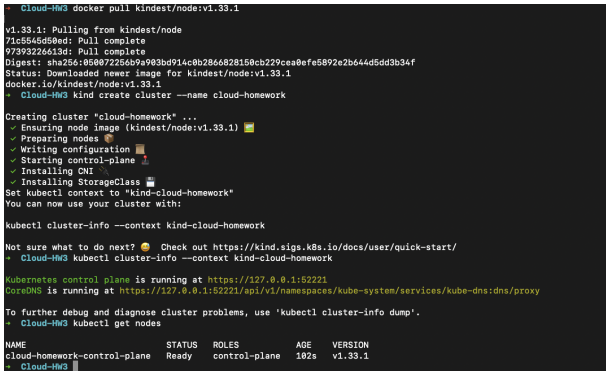


Figure 2: Successful creation of the Kind cluster and node readiness

Task 2.2: Deploy a Simple Application

In this task, I deployed a basic Nginx web server inside our Kubernetes cluster. This required writing two manifest files: one to create the Deployment, and another to expose the application through a Service.

What is Nginx? Nginx is a software that works as a web server and helps deliver websites to users. It can serve files like images and web pages quickly and can also send requests to other programs that create dynamic content. Nginx is very fast and can handle many users visiting a website at the same time. It is also used to make websites more secure and reliable by managing traffic and encrypting data. Additionally, it can act as a load balancer in the system.

Explanation of YAML Fields

Below is a breakdown of each field used in the Deployment and Service YAML files. Each field defines a specific configuration that Kubernetes uses to run and expose the application.

Deployment File (`nginx-deployment.yaml`)

- `apiVersion: apps/v1`
  Specifies the version of the Kubernetes API. For Deployments, we use `apps/v1`.
- `kind: Deployment`
  Indicates the type of resource being created, in this case, a Deployment.
- `metadata: name: nginx-deployment`
  Provides a name for the Deployment so it can be referenced later.
- `spec: replicas: 1`
  Sets the number of copies (replicas) of the pod that should be running. Here, we run one instance.
- `selector: matchLabels: app: nginx`
  Matches pods that have the label `app: nginx`. This tells the Deployment which pods it should manage.
- `template: metadata: labels: app: nginx`
  Assigns a label to the pod so it can be identified by the Deployment and Service.
- `spec: containers:`
  Defines the container to run inside each pod.
  - `name: nginx` – A name for the container.
  - `image: nginx:latest` – Specifies the Docker image to use, in this case, the latest Nginx image.
  - `containerPort: 80` – The port on which the container listens for HTTP traffic.

Applying the Configuration

To deploy the application, I used the following commands:

```
kubectl apply -f nginx-deployment.yaml
```

I verified the deployment using:

```
kubectl get pods
```

Task 2.3: Expose the application

In this task, I am aimed to make my deployed Nginx web page accessible from our browser(Expose the deployed application). I used the **NodePort** method, which is simple and works well with Kind. It exposes the application on a specific port on the local machine.

Service File (`nginx-service.yaml`)

- `apiVersion: v1`
  Indicates the version of the API used for creating the Service.
- `kind: Service`
  Specifies that this resource is a Service.
- `metadata: name: nginx-service`
  Assigns a name to the Service.
- `spec: type: NodePort`
  Defines the Service type. `NodePort` allows access from outside the cluster using a fixed port.
- `selector: app: nginx`
  Routes traffic to pods that have the label `app: nginx`.
- `ports:`
  Describes how the Service handles networking.
  - `port: 80` – The port the Service exposes within the cluster.
  - `targetPort: 80` – The port on the container the traffic is sent to.
  - `nodePort: 30080` – The external port on the host machine to access the application.

What is NodePort? NodePort is one of the Kubernetes service types used to expose a pod to external traffic. It opens a specific port on each node in the cluster and forwards traffic to the associated service, allowing access from outside the cluster. For example, if the NodePort is set to 30080, the service can be accessed at `http://<node-ip>:30080`. While convenient for local development and testing, NodePort has limitations such as a fixed port range and less flexibility for production environments. Compared to other service types, `ClusterIP` is used for internal communication only, and `LoadBalancer` provides a public IP for direct access, typically in cloud-based deployments.

Validation These commands confirmed that the Nginx pod was running and that the service was successfully exposed at port 30080.

```
→  Cloud-HW3 ls
nginx-deployment.yaml nginx-service.yaml
→  Cloud-HW3 kubectl apply -f nginx-deployment.yaml
deployment.apps/nginx-deployment created
→  Cloud-HW3 kubectl apply -f nginx-service.yaml

service/nginx-service created
→  Cloud-HW3 kubectl get pods
NAME                               READY   STATUS         RESTARTS   AGE
nginx-deployment-96b9d695-gbb5q    0/1     ErrImagePull   0          13s
→  Cloud-HW3 kubectl get svc

NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP        98m
nginx-service   NodePort    10.96.191.136   <none>        80:30080/TCP   12s
→  Cloud-HW3
```

Figure 3: Deployment & Service Validation

Why the deployment is not ready? This is an issue I faced during deployment with `kubectl apply -f nginx-deployment.yaml`. When I used this command the Nginx image did not pulled from Docker Hub, So, I pulled the image manually. This happened because I was connected to a VPN.

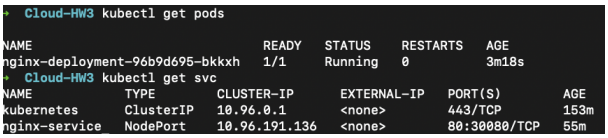


Figure 4: Deployment & Service Validation After Pulling Image from Docker Hub

Task 2.4: Scale the application

In this task, I scaled the application by increasing the number Nginx pods running in my Kubernetes cluster to improve **Availability**.

I used the following command to scale the application:

```
kubectl scale deployment nginx-deployment --replicas=3
```

Figure 5: Scaling Nginx which took about 6-7 seconds

Why does scaling take a few seconds? When I scaled the Nginx Deployment from one replica to multiple replicas, I observed a short delay of approximately 6–7 seconds before the new pods became ready. This delay is expected and results from several background processes Kubernetes performs. First, each new replica must be scheduled to a node and initialized. If the image is not already present or if the `imagePullPolicy` is set to `Always`, Kubernetes attempts to pull the Docker image again, which takes time even if the image is available locally. Additionally, starting a container involves a small overhead due to loading the container, binding to network ports, and launching the application process (in this case, Nginx). Even without custom health checks or readiness probes, the system waits for the pod to be fully initialized before marking it as `Running`. Therefore, a startup time of several seconds is normal for even lightweight applications.

Task 2.5: Access the Application

After deploying and exposing the Nginx application using a `NodePort` service on port 30080, I attempted to access it via `http://localhost:30080`. However, no response was received. This is because, in Kind, NodePort services are not directly accessible from the host's `localhost` by default.

Access via Port Forwarding To work around this, I used Kubernetes port forwarding, which redirects traffic from a port on the host machine to a port inside the cluster:

```
kubectl port-forward svc/nginx-service 8080:80
```

This allowed me to access the application in the browser at:

`http://localhost:8080`

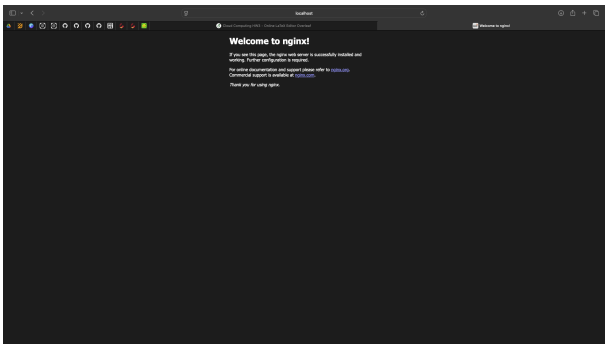


Figure 6: Enginx Welcome Web Page

Task 2.6: Install Prometheus and Grafana using Helm

What is Helm? **Helm** is a package manager for Kubernetes that simplifies the deployment and management of applications within a cluster. It uses pre-configured application templates called *charts*, which package together Kubernetes manifests like Deployments, Services, and ConfigMaps. Helm enables developers and operators to define, install, upgrade, and rollback applications consistently and efficiently. It abstracts away much of the complexity of writing raw YAML by allowing parameterized templates and reusable configurations, making it an essential tool for managing complex cloud-native applications.

In this task, I installed the **kube-prometheus-stack** Helm chart, which bundles both **Prometheus** and **Grafana** for monitoring Kubernetes clusters.

- **Prometheus** is an open-source monitoring system that collects metrics from configured targets at given intervals.
- **Grafana** is an analytics and visualization tool that displays data from Prometheus using customizable dashboards.

2.6.1 Add Required Helm Repository

In this step, I added the official Prometheus Helm chart repository to my local Helm client. When I run Helm commands, it looks for charts in its configured list of repositories. By default, Helm only includes a limited set of sources, so I had to explicitly add the `prometheus-community` repository, which contains the `kube-prometheus-stack` chart.

```
helm repo add prometheus-community
    ↪ https://prometheus-community.github.io/helm-charts
helm repo update
```

The first command registers the Prometheus community Helm chart source to my local system, and the second command updates the local cache to fetch the latest list of charts and versions available in that repository.

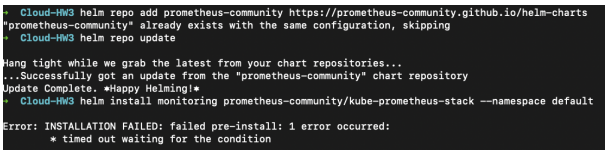


Figure 7: Adding Helm repo and update

2.6.2 Install the Kube-Prometheus-Stack

Then, I installed the kube-prometheus-stack using:

```
helm install monitoring prometheus-community/kube-prometheus-stack
```

Image Pull Issues: Due to **VPN restrictions and regional Docker registry blocking**, several images required by the Helm chart failed to pull automatically. To resolve this, I manually downloaded and loaded those images into my Kind cluster using the following approach:

1. Pull the alternative image from a working registry:

```
docker pull bitnami/kube-state-metrics:2.10.1
```

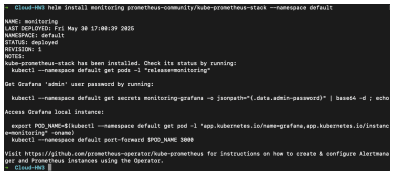
2. Tag the image to match what Kubernetes expects:

```
docker tag bitnami/kube-state-metrics:2.10.1
    ↪ registry.k8s.io/kube-state-metrics/kube-state-metrics:v2.15.0
```

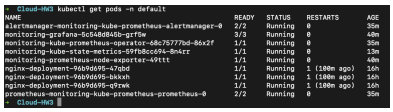
3. Load the image into the Kind cluster:

```
kind load docker-image
    ↪ registry.k8s.io/kube-state-metrics/kube-state-metrics:v2.15.0 --name
    ↪ cloud-homework
```

This approach resolved the `ImagePullBackOff` errors and allowed the stack to deploy successfully.



(a) Successful Helm installation of kube-prometheus-stack



(b) All monitoring pods running successfully

(c) Pulling image manually due to regional/VPN restrictions

Figure 8: Installing kube-prometheus-stak

2.6.3 Access Prometheus and Grafana Dashboards

To access the dashboards from my browser, I used port forwarding, since services in Kind are not exposed externally by default.

Access Grafana :

```
kubectl port-forward svc/monitoring-grafana 3000:80
# Access via: http://localhost:3000
# Default credentials: admin / prom-operator
```

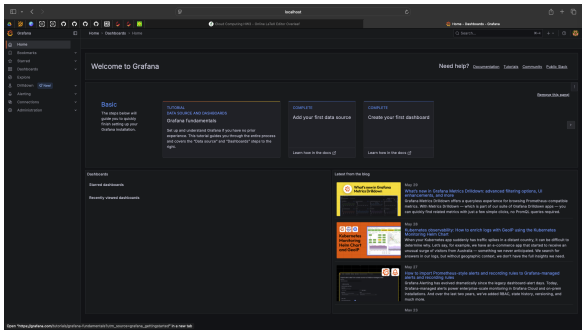
If the password had changed, I retrieved it using:

```
kubectl get secret --namespace default monitoring-grafana -o
    ↪ jsonpath="{.data.admin-password}" | base64 --decode
```

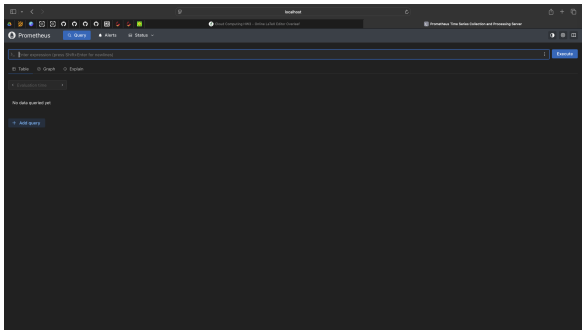
Access Prometheus :

```
kubectl port-forward svc/monitoring-kube-prometheus-prometheus 9090:9090
# Access via: http://localhost:9090
```

These steps allowed me to successfully access and use the monitoring stack locally.



(a) Grafana Dashboard running locally



(b) Prometheus UI for querying metrics

Figure 9: Grafana and Prometheus dashboards after deployment

Task 2.7: Access the K8S Cluster via Lens

Lens is a Kubernetes IDE that provides a GUI for managing and visualizing clusters. It simplifies working with Kubernetes by showing real-time logs, resource metrics, and cluster configuration.

To access the Kubernetes cluster via Lens, I opened the Lens application. By default, Lens automatically read the cluster configurations from the `~/.kube/config` file. If the cluster was created using `kind`, and no custom kubeconfig path was used, it should appear under the **Local kubeconfig** section in the Lens interface.

No manual import is necessary unless the configuration file is located elsewhere. Simply select the cluster from the list to connect and manage workloads, view resources, and monitor the cluster visually through the Lens dashboard.

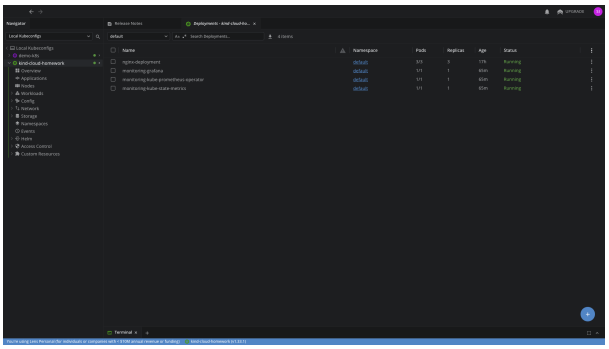


Figure 10: Lens Application Environment

System Hierarchy Overview

```
Local macOS Host
|
+-- Docker
    |
    +-- kind Cluster ("cloud-homework")
        |
        +-- Core Kubernetes System
        |   +-- kube-apiserver
        |   +-- kube-scheduler
        |   +-- etcd
        |   +-- CoreDNS
        |
        +-- User Deployed Workloads
        |   +-- Nginx Deployment
        |   |   +-- Namespace: default
        |   |   +-- Exposed via NodePort on port 30080
        |   |
        |   +-- Monitoring Stack (Prometheus + Grafana)
        |   |   +-- Namespace: default
        |   |   +-- Prometheus
        |   |   +-- Grafana
        |   |   +-- Node Exporter
        |   |   +-- kube-state-metrics
```

As you can see , this is the state of our system which contains its hierarchy. I used a kind docker image to create a Kubernetes node on my local computer. This cluster contains a *Core Kubernetes System* which is the fundamental infrastructure of any Kubernetes cluster. Each component is described as follow:

- **kube-apiserver**: The front-end of the Kubernetes control plane; all internal and external communication with the cluster goes through this REST API server.
- **etcd**: A distributed key-value store that holds all cluster configuration data and state. It is the single source of truth for the cluster.
- **kube-scheduler**: Assigns pods to nodes based on resource availability and scheduling policies.

- **CoreDNS**: Provides internal DNS services, allowing pods and services to discover each other using domain names.
- **kubelet**: An agent that runs on each node; ensures the containers in a pod are running as expected.

Also, there is two deployed application including Nginx and Monitoring Stack and all components of each one.

3 Serverless Deployment on K8S

In this section, I deployed a serverless application using OpenFaaS framework on K8S.

Task 3.1: Install OpenFaaS

I used the recommended option to deploy serverless functions in Kubernetes, **OpenFaaS**. It is a lightweight framework that simplifies the development and deployment of serverless applications.

Installation Procedure

I used the official OpenFaaS Helm chart documentation to install OpenFaaS while getting some help from ChatGPT for its instructions, I executed the following steps:

1. **Add OpenFaaS Helm repository**:

```
helm repo add openfaas https://openfaas.github.io/faas-netes/
helm repo update
```

2. **Create required namespaces**:

```
kubectl create namespace openfaas
kubectl create namespace openfaas-fn
```

3. **Generate and store basic authentication credentials**:

```
PASSWORD=$(head -c 12 /dev/urandom | shasum | cut -d' ' -f1)
kubectl -n openfaas create secret generic basic-auth \
  --from-literal=basic-auth-user=admin \
  --from-literal=basic-auth-password="$PASSWORD"

echo "OpenFaaS admin password: $PASSWORD"
```

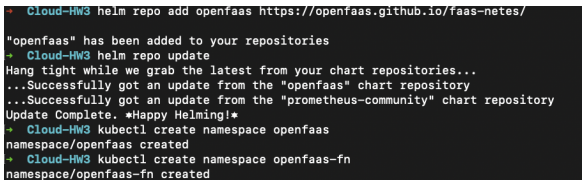
4. **Install OpenFaaS with Helm**:

```
helm upgrade openfaas openfaas/openfaas \
  --namespace openfaas \
  --set basic_auth=true \
  --set gateway.upstreamTimeout=120s \
  --set gateway.readTimeout=120s \
  --set gateway.writeTimeout=120s \
  --set faasnetes.imagePullPolicy=IfNotPresent \
  --install
```

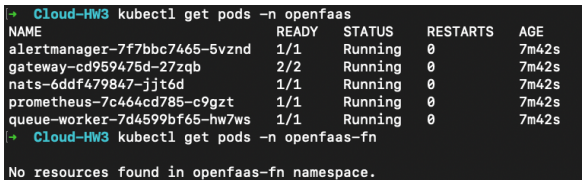
Verification of Deployment

I verified the successful deployment of OpenFaaS by checking the pods in the `openfaas` namespace:

```
kubectl get pods -n openfaas
```



(a) Creating Namespaces



(b) OpenFaaS pods Status

Figure 11: OpenFaaS pods successfully running in the openfaas namespace

All core components were in `Running` state, confirming that the control plane of OpenFaaS was active and healthy.

No Deployed Functions Yet

To confirm the absence of functions (as none were deployed yet), I checked the `openfaas-fn` namespace:

```
kubectl get pods -n openfaas-fn
# Output: No resources found in openfaas-fn namespace.
```

This is expected behavior at this stage since function deployment is addressed in the next task.

Task 3.2: Develop a simple function

In this task, I pulled OpenFaaS template and use them to write a function for serverless environment. I used OpenFaaS official documentation to do the task.

Pulling Official Templates

Before creating a new function, it's recommended to download the official templates:

```
faas-cli template pull
```

This command pulls standard OpenFaaS language templates into the `./template` directory of the current working directory.

Classic vs. of-watchdog Templates

There are two types of templates in OpenFaaS:

- **Classic Templates** – Use the `watchdog` binary and communicate via standard input/output (STDIO). These are recommended for beginners and for following tutorials.
- **of-watchdog Templates** – Communicate using HTTP and offer greater flexibility and performance. These are better suited for production environments or when custom HTTP handling is required.

Exploring the Template Store

OpenFaaS provides a built-in template store where users can browse and pull available templates. To list templates:

```
faas-cli template store list
```

Pulling a Specific Template

To pull Python 3 HTTP template:

```
faas-cli template store pull python3-http
```

This makes the template available locally under the `./template` directory.

Listing Available Templates Locally

Once pulled, available templates can be listed using:

```
faas-cli new --list
```

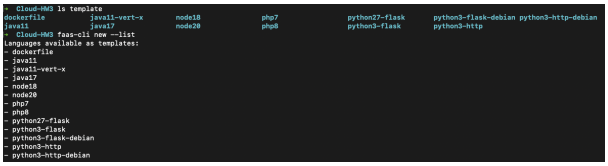


Figure 12: Installed Templates

Creating a Function

To create a new function named `to-uppercase`, I used the following command:

```
faas-cli new to-uppercase --lang python3
```

This command creates a `stack.yaml` file(if does not exist), pull templates(if did not pull) and a directory named `to-uppercase` which contains some files:

- `handler.py` – Main function code executed by OpenFaaS.
- `handler_test.py` – Unit tests for the function logic.
- `requirements.txt` – Python dependencies to be installed.
- `tox.ini` – Config for running tests with `tox`.

stack.yaml This file is a configuration file that defines how your functions should be built and deployed. The command I used to create these files, writes the `stack.yaml` file by itself based on the function name i specified.

```
version: 1.0
provider:
  name: openfaas
  gateway: http://127.0.0.1:8080
functions:
  to-uppercase:
    lang: python3-http
    handler: ./to-uppercase
    image: to-uppercase:latest
```

As you can see, it specified the provider names which is OpenFaaS and the functions. The only defined function is `to-uppercase`. It also specified the location of the function file and its language using `lang` keyword.

The function logic as mentioned above is written in `handler.py` file. The function is passed two arguments, *event* and *context*. The documentation describes them as follow:

- **Event** contains data about the request, including: - body - headers - method - query - path
- **Context** contains basic information about the function, including: - hostname

The function itself is written as follow:

```
def handle(event, context):

    body = event.get("body", "")
    return {
        "statusCode": 200,
        "body": str(body).upper()
    }
```

This function gets the input data from *event* and return its upper case.

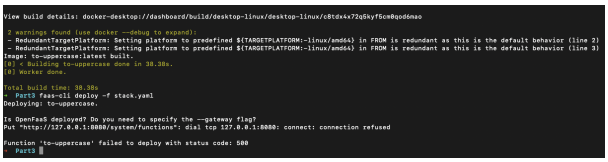


Figure 13: Result of Built & deploy

Task 3.3: Deploy the Function

To build the function, I used the following command. This command pulled an image named `python:3.12-alpine` and took about 38 seconds to complete.

```
faas-cli build -f stack.yaml
```

Then I used the following command to deploy the function:

```
faas-cli deploy -f stack.yaml
```

But as shown in Figure 13, the deployment is failed.

By default, faas-cli deploy tries to connect to http://127.0.0.1:8080, assuming that the OpenFaaS gateway is running locally and accessible on port 8080. However OpenFaaS is running inside the Kubernetes cluster, not on my host directly. I need to either port-forward the gateway to my local machine or specify the correct gateway address.

Task 3.3: Deploy the Function

After building the function using the OpenFaaS template engine, the next step was to deploy it on the Kubernetes cluster via the OpenFaaS gateway. The deployment process involved the following:

1. First, the function image was built using:

```
faas-cli build -f stack.yaml
```

2. Due to limitations in the free Community Edition of OpenFaaS, local images were not supported. Therefore, I had to tag and push the image to Docker Hub:

```
docker tag to-uppercase sinahkz/to-uppercase:latest
docker push sinahkz/to-uppercase:latest
```

3. Then, I logged into the OpenFaaS gateway using the previously generated password:

```
faas-cli login --username admin --password-stdin --gateway
    ↪ http://127.0.0.1:8080
```

4. Finally, the function was deployed using:

```
faas-cli deploy -f stack.yaml --gateway http://127.0.0.1:8080
```

Challenges Faced and Solutions:

- **Image Push Requirement:** The Community Edition of OpenFaaS only allows deploying public images. This led to a 400 error until the image was pushed to Docker Hub.
  Solution: I tagged the image and pushed it using Docker commands, then updated the `stack.yaml` to reference the public image.
- **Handler Signature Error:** An incorrect function signature in `handler.py` caused repeated 500 errors.
  Solution: I updated the handler function to accept the correct number and type of arguments as required by the OpenFaaS Python HTTP template.
- **Byte Response Format:** The output appeared as `B'HELLO WORLD'`, indicating a byte-string response.
  Solution: I decoded the byte stream using Python's `decode()` method or cast it to string to ensure readable output.

Task 3.4: Invoke the Function

Once deployed, the function was invoked both via the OpenFaaS UI and CLI to test its behavior. The function was designed to receive a string and return it in uppercase.

```
echo -n "hello world" | faas-cli invoke to-uppercase --gateway
    ↪ http://127.0.0.1:8080
```

Output:

```
B'HELLO WORLD'
```

The `B` prefix in the output indicates that the function returned a byte string. This is expected behavior for OpenFaaS Python functions. If a plain string output is required, the handler can be adjusted to decode the byte stream before returning.

Task 3.7: Function Resource Monitoring and Analysis via Lens

After successfully deploying and invoking the `to-uppercase` serverless function, I conducted a resource usage analysis to observe its behavior under load. For this purpose, I utilized the Lens Kubernetes IDE, which provides integrated metrics dashboards for each pod, and designed a custom Python script to continuously invoke the function over time.

Performance Testing Approach

To generate consistent load on the function and observe its runtime behavior, I wrote a Python script that sends random string inputs to the function's endpoint for one minute. This not only simulated real-world usage but also allowed me to monitor CPU and memory usage through the Prometheus metrics exposed in Lens.

Resource Consumption Insights

Figure 14 and Figure 15 summarize the observed metrics:

- **CPU Usage:** The function pod peaked around `0.38` cores during active invocation, which indicates lightweight processing under HTTP-based serverless execution.
- **Memory Usage:** The memory consumption remained stable, averaging close to `100MB`, with minimal fluctuation, demonstrating efficient memory management without leaks.
- **Pod Status:** Throughout the testing, the function's pod remained in a `Running` state, managed by its ReplicaSet, indicating robustness under repeated access.

This testing method used scripts and monitoring tools in Lens to show that the serverless function runs well and reliably in the local kind cluster.

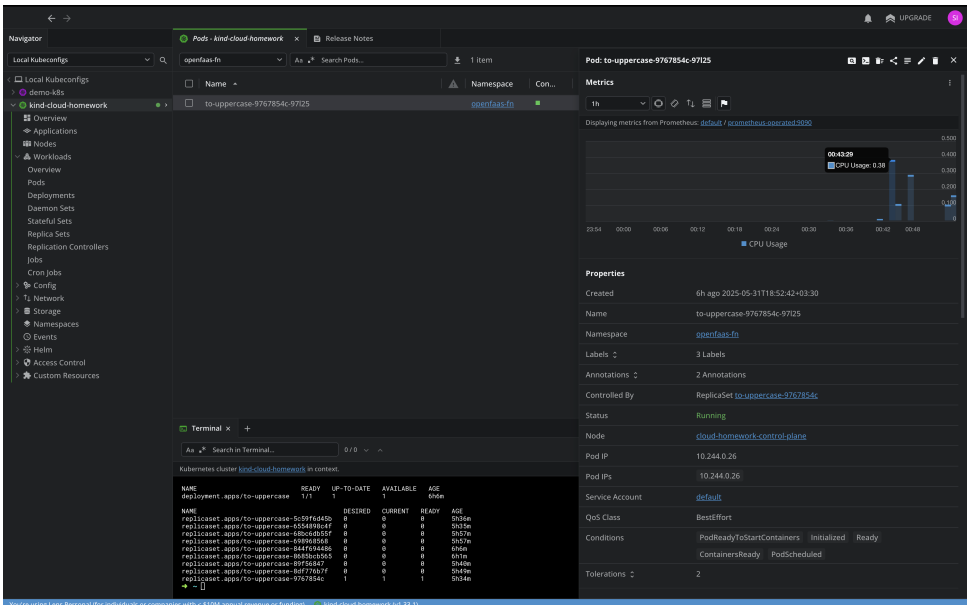


Figure 14: CPU usage of `to-uppercase` function under scripted load

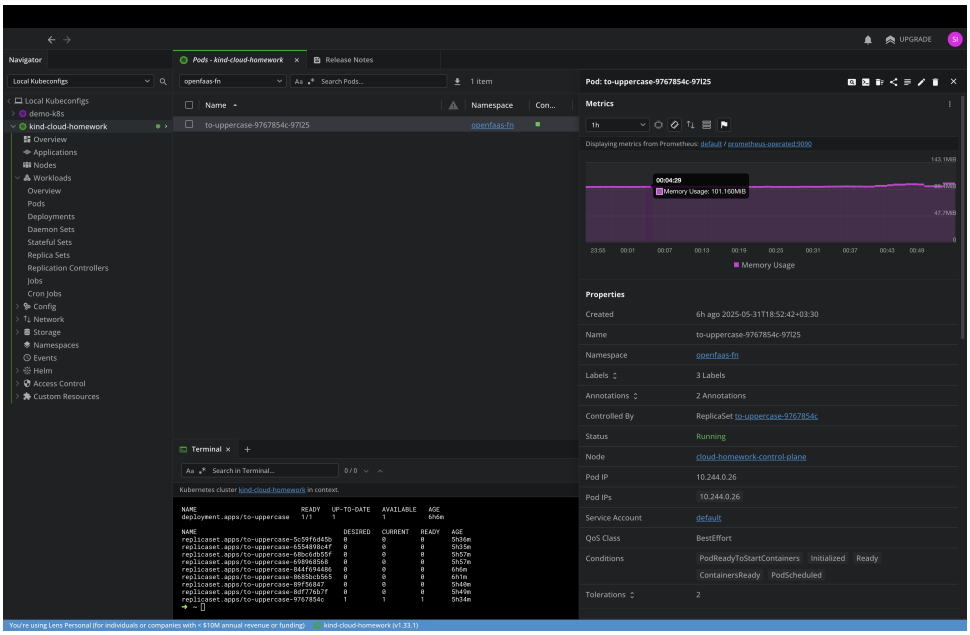


Figure 15: Memory usage of `to-uppercase` function during test period

4 Ray on K8S

Task 4.1: Deploy a Ray Cluster on Kubernetes

To deploy a Ray cluster on my local `kind` Kubernetes setup, I used the official `KubeRay` Helm charts as documented in the Ray project's official guide. I was working on an Apple Silicon (ARM64) machine, it was important to select a compatible Ray image tag.

Step 1: Install the KubeRay Operator The KubeRay operator is required to manage Ray clusters declaratively. I installed it using Helm with a custom image tag compatible with Apple Silicon:

```
helm repo add kuberay https://ray-project.github.io/kuberay-helm/
helm repo update
# Install both CRDs and KubeRay operator v1.3.0.
helm install kuberay-operator kuberay/kuberay-operator --version 1.3.0
# Install CRD and KubeRay operator.
kubectl create -k
    ↪ "github.com/ray-project/kuberay/ray-operator/config/default?ref=v1.3.0"
```

After resolving various Helm ownership conflicts (due to previously applied resources), the operator pod started successfully:

```
→  Part4 kubectl get pods

NAME                                                READY   STATUS    RESTARTS        AGE
alertmanager-monitoring-kube-prometheus-alertmanager-0   2/2     Running   14 (160m ago)   4d9h
kuberay-operator-6ddb59999c-2t1f8                   1/1     Running   0               22m
monitoring-grafana-5c548d845b-grf5w                 3/3     Running   21 (160m ago)   4d9h
monitoring-kube-prometheus-operator-68c75777bd-86x2f   1/1     Running   14 (159m ago)   4d9h
monitoring-kube-state-metrics-59fb8cc694-8n4rr      1/1     Running   14 (159m ago)   4d8h
monitoring-prometheus-node-exporter-49ttt           1/1     Running   7 (160m ago)    4d9h
nginx-deployment-96b9d695-47qbd                     1/1     Running   8 (160m ago)    5d1h
nginx-deployment-96b9d695-bkkxh                     1/1     Running   8 (160m ago)    5d1h
nginx-deployment-96b9d695-q9rwk                     1/1     Running   8 (160m ago)    5d1h
prometheus-monitoring-kube-prometheus-prometheus-0  2/2     Running   14 (160m ago)   4d9h
raycluster-kuberay-head-ppqjm                       1/1     Running   0               19m
raycluster-kuberay-workergroup-worker-9ptrz         1/1     Running   0               19m
```

Step 2: Deploy the RayCluster To launch the actual Ray cluster, I used the Helm chart `ray-cluster` and specified the ARM64-compatible Ray image:

```
helm install raycluster kuberay/ray-cluster --version 1.3.0 --set
    ↪ 'image.tag=2.41.0-aarch64
```

The chart automatically created a head node and one worker node, as specified in the default configuration.

Step 3: Confirm Deployment I verified the creation and readiness of the RayCluster resource:

```
→  Part4 kubectl get rayclusters
NAME                 DESIRED WORKERS   AVAILABLE WORKERS   CPUS   MEMORY   GPUS   STATUS   AGE
raycluster-kuberay   1                 1                   2      3G       0      ready    11m
```

Step 4: Verify Pod Status I confirmed that both the head and worker pods were running and ready:

```
→  Part4 # View the pods in the RayCluster named "raycluster-kuberay"
kubectl get pods --selector=ray.io/cluster=raycluster-kuberay
NAME                                          READY   STATUS    RESTARTS   AGE
raycluster-kuberay-head-ppqjm                 1/1     Running   0          17m
raycluster-kuberay-workergroup-worker-9ptrz   1/1     Running   0          17m
→  Part4
```

At this point, the Ray cluster is operational and ready to serve applications.

Task 4.2: Run a Simple Ray Application

To write a python script for my ray cluster, ray library must be installed through ray. I tried to install the ray library and run it through my local MacBook, but there is an mismatching issue between my ray cluster and the library I installed. Additionally, there is no other ray library version which supported on my system, so i had to run my code inside the head pod to bypass this problem.

Step 1: Create a Simple Python File To test if everything working right, I wrote a simple ray python file to ensure it.

```
import ray
ray.init()
```

Step 2: Copy Python File Into Head Pod I copied the python file into my ray head pod `/tmp` address by using

```
kubectl cp rayTask.py raycluster-kuberay-head-ppqjm:/tmp/
```

Step 3: Run the Script To run the python script I used the following command:

```
kubectl exec -it raycluster-kuberay-head-ppqjm -- python3 /tmp/rayTask.py
```

Also there is an alternative way to run the script through the head pod command line by using the following command:

```
kubectl exec -it raycluster-kuberay-head-ppqjm -- bash
```

By running the file, I got this message which confirms that everything is working:

```
(base) ray@raycluster-kuberay-head-ppqjm:~$ python3 rayTask.py
2025-06-04 16:05:16,787 INFO worker.py:1614 -- Using address 127.0.0.1:6379 set in the environment variable RAY_ADDRESS
2025-06-04 16:05:16,787 INFO worker.py:1654 -- Connecting to existing Ray cluster at address: 10.244.0.2:6379...
2025-06-04 16:05:16,794 INFO worker.py:1832 -- Connected to Ray cluster. View the dashboard at 10.244.0.2:8265
```

Figure 16: Python File Validation

Test MapReduce Task: As mentioned in the Homework documentation. I must deploy an intermediate example on ray. I chose MapReduce task and follow the the instructions from the ray official documentation.

The attached script is using *Ray* which is a framework for distributed computing on a Ray cluster, to implement a simplified MapReduce-like word count over the Zen of Python. Here's a brief explanation of the Ray-specific components:

- `ray.init(address="auto")`: Connects the script to a running Ray cluster. In this case, it assumes that it is running inside a Ray head node or pod.
- `@ray.remote`: This decorator is used to define functions that can be executed in parallel as Ray tasks. In my script, both the `apply_map` and `apply_reduce` functions are defined as remote tasks.

- `apply_map.remote(...)`: Launches the map phase in parallel across different partitions of the input corpus. Each map task creates key-value pairs based on the first letter of each word to prepare for partitioning across reducers.
- `ray.get(...)`: Used to collect the results from the asynchronous Ray tasks once they are complete. Here, it gathers the mapped outputs and, later, the reduced outputs.
- `apply_reduce.remote(...)`: Executes the reduce phase, combining counts of words that were assigned to the same reducer. Each reduce task aggregates the key-value pairs (word counts) emitted by the map phase.

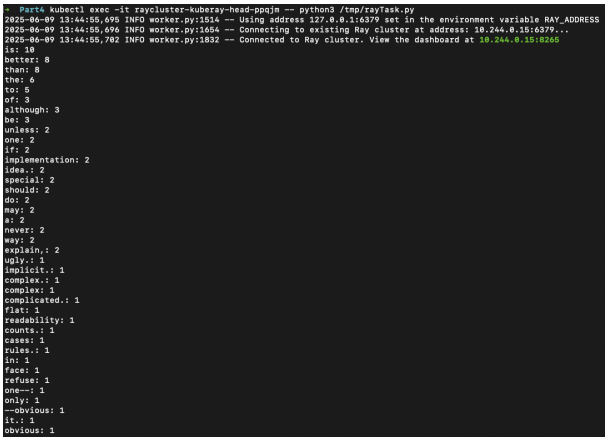


Figure 17: MapReduce Output Result

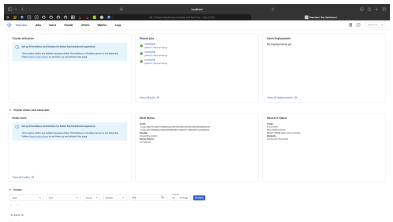
The complete result is included in a text file named `MapReduce_out.txt`.

Task 4.3: Access the Ray Dashboard

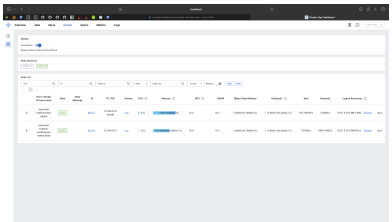
To monitor and validate Ray cluster activity, I accessed the Ray Dashboard by port-forwarding the dashboard port exposed by the Ray head node:

```
kubectl port-forward raycluster-kuberay-head-ppqjm 8265:8265
```

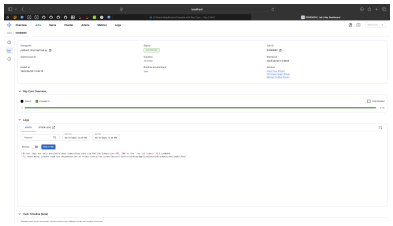
The dashboard provides a real-time view of job status, node resource usage, active worker processes, and historical logs.



(a) Ray Dashboard Overview



(b) Cluster Nodes and Resources



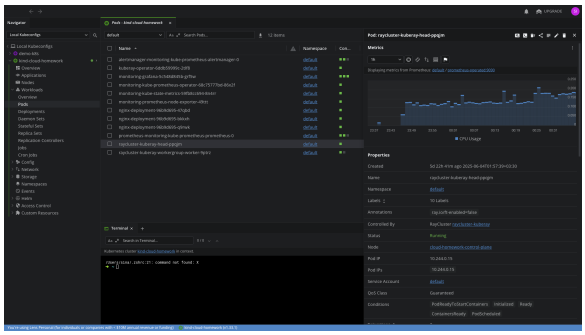
(c) Ray Job Execution Details

Figure 18: Ray Dashboard: cluster status, job logs, and resource usage

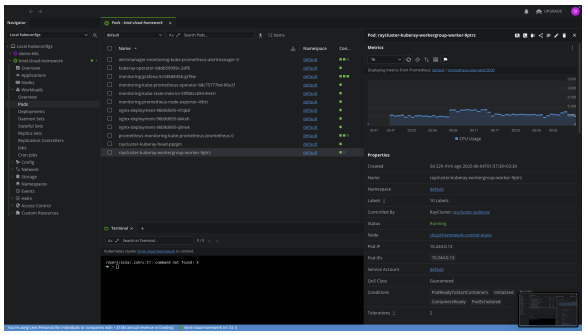
Task 4.4: Check Cluster Status in Lens

Having previously set up Lens to manage my `kind` cluster, I used it again here to monitor the status and performance of the deployed Ray cluster. This inspection focused specifically on validating that Ray components were correctly running and actively participating in distributed task execution.

- Both the Ray head pod and worker pod were confirmed to be in `Running` state.
- CPU usage graphs in the right-hand pane indicated that both pods were actively executing tasks, with measurable and sustained load.



(a) Lens: Head pod status and CPU usage metrics



(b) Lens: Worker pod status and metrics

Figure 19: Ray pods running inside the `kind-cloud-homework` cluster as viewed in Lens

Comparison of Ray and Serverless

Ray and OpenFass are designed to simplify distributed computing, but they are different on their structure and architecture. OpenFass framework is designed for request-based tasks where each task is stateless; additionally, in these systems each task is handled by a function which we call function as a service. On the other hand, Ray is designed for stateful and parallel tasks such as training models.

1. Architecture

Ray Ray uses a head node and worker nodes like the previous technologies we had in previous assignments. the head node acts as the control plane for the Ray cluster and also the global scheduler and object store manager is placed in the head node. Workers' tasks are assigned by the head node and they process the task.

OpenFaaS On the other hand, OpenFaaS is a serverless platform which works only using stateless functions. Users deploy a set of functions and based on the requests to the cloud servers, relevant function will be run on its own container which isolates the function from others.

2. Execution model

Ray Ray's execution model is designed for fine-grained, distributed parallel computing and supports both stateless and stateful computation. Ray's task is a stateless function which receives input, computes a result, and returns it. Actors are Python classes which maintain internal state across method calls. Actors are scheduled similarly to tasks but are bound to specific worker processes, preserving local memory/state.

OpenFaaS OpenFaaS follows a serverless, request-response execution model, aligning with the principles of Function-as-a-Service. Each function is independent, stateless, short-lived, and containerized that exposes an HTTP endpoint. When a request is sent to the OpenFaaS gateway, it forwards it to the appropriate function pod or scales it up if none exist.

3. Use Cases

Ray Ray are widely used in machine learning training, data processing pipelines and real-time inference, where statful, long running, and parallel computing are necessary.

OpenFaaS On the other hand, OpenFaaS is used for microservices, API backends, and event-driven. But researchers are focused on the potentials of function as a service to utilize them for AI task such as inference. One of the biggest bottlenecks in serverless environments is utilizing GPUs for AI inference which today is one of the research topics. StreamBox has a creative idea to utilize them by using streams as a lightweight sandbox.

4. Choosing Between Them

Use Ray when your application requires fine-grained control over distributed computation, needs to maintain state across tasks, or benefits from automatic scaling of parallel workloads. Use OpenFaaS when building stateless services that respond to HTTP requests or events, particularly when simplicity, fast deployment, and scalability are key concerns.

Final Cluster Structure

```
Local macOS Host
|
+-- Docker
    |
    +-- kind Cluster ("cloud-homework")
        |
        +-- Core Kubernetes System
        |   +-- kube-apiserver
        |   +-- kube-scheduler
        |   +-- etcd
        |   +-- CoreDNS
        |
        +-- User Deployed Workloads
            +-- Nginx Deployment
            |   +-- Namespace: default
            |   +-- Exposed via NodePort on port 30080
            |
            +-- Monitoring Stack (Prometheus + Grafana)
            |   +-- Namespace: default
            |   +-- Prometheus
            |   +-- Grafana
            |   +-- Node Exporter
            |   +-- kube-state-metrics
            |
            +-- OpenFaaS Platform
            |   +-- Namespace: openfaas
            |   +-- Gateway
            |   +-- faas-netes
            |   +-- Queue Worker
            |   +-- Prometheus (OpenFaaS metrics)
            |   +-- Alertmanager
            |
            +-- OpenFaaS Functions
            |   +-- Namespace: openfaas-fn
            |   +-- to-uppercase (Python function)
            |
            +-- Ray Cluster
                +-- Namespace: default
                +-- Ray Head Pod
                |   +-- Ray Dashboard (port 8265)
                +-- Ray Worker Pod(s)
```

Conclusion

In conclusion, this assignment provided hands-on experience with three important areas of modern cloud computing: Kubernetes, serverless platforms, and distributed computing frameworks. I began by setting up and working with a local Kubernetes cluster using kind, deploying and managing containerized applications like Nginx. I then explored serverless computing by installing OpenFaaS, building a Python function, and learning how stateless functions are deployed and scaled. Finally, I deployed a Ray cluster on Kubernetes and ran a distributed Python task to understand how Ray handles parallel and stateful workloads. Each step deepened my understanding of how these technologies function individually and together in a cloud-native environment. Overall, this project helped me connect theory to practice and better understand the real-world use cases for Kubernetes, serverless platforms, and distributed systems like Ray.