# DevOps for Cloud – Assignment 1

Vivek Bhadra

(Roll: 2024mt03533)

# End-to-end implementation on AWS with ECR, EKS, Load Balancer, and Prometheus

# Table of Content

# Overview

This submission presents, in a single coherent narrative, the process through which a minimal Flask web service was designed, built, containerised, and deployed to a managed Kubernetes cluster on AWS. Subsequently, the service was instrumented for metrics collection and verification using Prometheus. The assignment submission write-up is structured in accordance with the assignment's six defined tasks, demonstrating how each requirement was implemented and validated within the deployment environment.

- **AWS Account ID**: 402691950139
- **Region**: eu-west-2 (London)
- **ECR repository**: 402691950139.dkr.ecr.eu-west-2.amazonaws.com/img-2024mt03533
- **Kubernetes cluster**: dep-2024mt03533 (EKS)
- **Node public IPs**: 18.130.201.109, 35.177.152.156
- **Service (ELB) DNS**: abde9643d75b24f39a5fb585d5e078aa-375594286.eu-west-2.elb.amazonaws.com
- **Service ports**: HTTP 8000 (app), NodePort 30900 (Prometheus)

Screenshots and trimmed logs are referenced as figures; they were captured during execution and will be embedded in the final PDF at the indicated placeholders.

# Task 1: Create the Backend Application using Flask

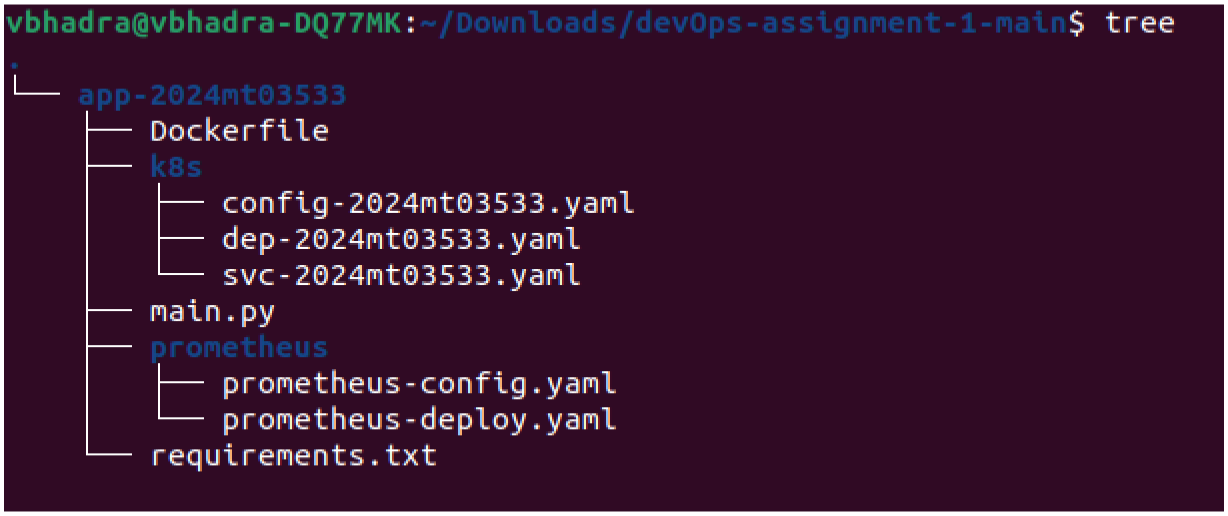
**Requirement**:
Implement a Flask application exposing /get_info that returns a JSON object with APP_VERSION (initially "1.0") and APP_TITLE ("Devops for Cloud Assignment"), injected dynamically by environment variables. Run locally with Uvicorn and provide evidence in a browser. Project directory must be app-<roll_number> and Python file named main.py.

This section documents the complete development of the assignment application from a blank folder to a containerised, Kubernetes-ready service with Prometheus metrics. The objective was to meet the brief exactly: implement a Flask app exposing two endpoints—/get_info and /metrics—and instrument it with three required Prometheus metrics, then package it for Docker and Kubernetes.

## Project Structure

A dedicated submission folder named with the roll number was created:

```
app-2024mt03533/
├── main.py
├── requirements.txt
├── Dockerfile
├── k8s/
│   ├── config-2024mt03533.yaml
│   ├── dep-2024mt03533.yaml
│   └── svc-2024mt03533.yaml
└── prometheus/
    ├── prometheus-config.yaml
    └── prometheus-deploy.yaml
```



```
vbhadra@vbhadra-DQ77MK:~/Downloads/devOps-assignment-1-main$ tree
.
└── app-2024mt03533
    ├── Dockerfile
    ├── k8s
    │   ├── config-2024mt03533.yaml
    │   ├── dep-2024mt03533.yaml
    │   └── svc-2024mt03533.yaml
    ├── main.py
    ├── prometheus
    │   ├── prometheus-config.yaml
    │   └── prometheus-deploy.yaml
    └── requirements.txt
```

## Core Flask Application

We implemented a minimal Flask application that:

- Reads APP_VERSION and APP_TITLE from environment variables (populated via ConfigMap in Kubernetes).
- Serves /get_info to return a small JSON payload needed for verification.
- Serves /metrics for Prometheus scraping (exposed via WSGI middleware).

- Uses Uvicorn to run as an ASGI server for reliable container operation.

## main.py

```
#!/usr/bin/env python3
# main.py
import os
from flask import Flask, jsonify
from asgiref.wsgi import WsgiToAsgi
from werkzeug.middleware.dispatcher import DispatcherMiddleware
from prometheus_client import Counter, Gauge, generate_latest,
CONTENT_TYPE_LATEST
import psutil

APP_VERSION = os.getenv("APP_VERSION", "1.0")
APP_TITLE   = os.getenv("APP_TITLE", "Devops for Cloud Assignment")
POD_NAME    = os.getenv("HOSTNAME") or os.uname().nodename

REQUEST_COUNT = Counter("get_info_requests_total", "Total /get_info
requests", ["pod", "version"])
CPU_PERCENT   = Gauge("process_cpu_percent", "Process CPU percent
(per replica)", ["pod", "version"])
RSS_BYTES     = Gauge("process_rss_bytes", "Resident set size in
bytes (per replica)", ["pod", "version"])

app = Flask(__name__)

@app.route("/get_info", methods=["GET"])
def get_info():
    REQUEST_COUNT.labels(pod=POD_NAME, version=APP_VERSION).inc()
    p = psutil.Process(os.getpid())
    RSS_BYTES.labels(pod=POD_NAME,
version=APP_VERSION).set(p.memory_info().rss)
    CPU_PERCENT.labels(pod=POD_NAME,
version=APP_VERSION).set(p.cpu_percent(interval=0.0))
    return jsonify({"APP_VERSION": APP_VERSION, "APP_TITLE":
APP_TITLE, "pod": POD_NAME}), 200

def metrics_app(environ, start_response):
    data = generate_latest()
    start_response("200 OK", [("Content-Type", CONTENT_TYPE_LATEST)])
    return [data]

application = DispatcherMiddleware(app, {"/metrics": metrics_app})
asgi_app = WsgiToAsgi(application)

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("main:asgi_app", host="0.0.0.0", port=8000,
reload=False)
```

## Code Walkthrough – main.py

The core of this assignment is the Python script main.py, which implements the web application and integrates Prometheus-based monitoring. This single file brings together the logic for responding to client requests, collecting system-level statistics, and exposing performance metrics to monitoring systems. The development approach was deliberately minimalistic yet structured to ensure readability, reliability, and alignment with the assignment requirements.
The application begins with a few essential imports. The Flask framework provides the web server foundation and is responsible for handling HTTP routes such as /get_info. The Prometheus client library is imported to define and expose performance metrics that can later be scraped by Prometheus. Additionally, the psutil module is used to capture real-time CPU and memory usage of the running process. Together, these form the building blocks of a small but complete web service.

```
import os
from flask import Flask, jsonify
from asgiref.wsgi import WsgiToAsgi
from werkzeug.middleware.dispatcher import DispatcherMiddleware
from prometheus_client import Counter, Gauge, generate_latest,
CONTENT_TYPE_LATEST
import psutil
```

Once the required libraries are available, the program reads the environment variables that will hold deployment-specific values. In this project, the version number and title of the application are stored in variables APP_VERSION and

APP_TITLE. These values are injected by Kubernetes at runtime through a ConfigMap, allowing the same container image to behave differently across environments without requiring rebuilds. The HOSTNAME environment variable, automatically provided inside a container, is used to identify the running pod. This makes it possible to trace which replica of the application handled a given request when it is deployed with multiple pods.

```
APP_VERSION = os.getenv("APP_VERSION", "1.0")
APP_TITLE   = os.getenv("APP_TITLE", "Devops for Cloud Assignment")
POD_NAME    = os.getenv("HOSTNAME") or os.uname().nodename
```

The next segment of the code defines three Prometheus metrics that the assignment explicitly requires. The first metric, get_info_requests_total, is a counter that increments each time the /get_info endpoint is accessed. The other two are gauges, representing instantaneous measurements of CPU usage and memory consumption (resident set size). Each metric is labelled with the pod and version values, ensuring that Prometheus can distinguish data coming from different replicas or application versions once deployed to Kubernetes.

```
REQUEST_COUNT = Counter("get_info_requests_total", "Total /get_info
requests", ["pod", "version"])
CPU_PERCENT   = Gauge("process_cpu_percent", "Process CPU percent
(per replica)", ["pod", "version"])
RSS_BYTES     = Gauge("process_rss_bytes", "Resident set size in
bytes (per replica)", ["pod", "version"])
```

After initialising the metrics, a Flask application instance is created. The design philosophy here is to keep the routing layer extremely lean—only two endpoints are provided, /get_info for functionality testing and /metrics for monitoring. The /get_info route is implemented using Flask's standard route decorator. Each time it is invoked, the counter metric is incremented, and the psutil library is used to measure the process's current CPU utilisation and memory footprint. These readings are updated in the Prometheus gauges before the function returns a JSON response containing the application's version, title, and pod name. This JSON output confirms that the application is alive and that environment variables are being correctly read.

```
@app.route("/get_info", methods=["GET"])
def get_info():
    REQUEST_COUNT.labels(pod=POD_NAME, version=APP_VERSION).inc()
    p = psutil.Process(os.getpid())
    RSS_BYTES.labels(pod=POD_NAME,
version=APP_VERSION).set(p.memory_info().rss)
    CPU_PERCENT.labels(pod=POD_NAME,
version=APP_VERSION).set(p.cpu_percent(interval=0.0))
    return jsonify({
        "APP_VERSION": APP_VERSION,
        "APP_TITLE": APP_TITLE,
        "pod": POD_NAME
    }), 200
```

The second endpoint, /metrics, is created not through Flask directly but as a small WSGI application that generates the latest set of metrics in Prometheus's expected text format. This approach ensures a clean separation between business logic and monitoring data. The metrics app simply produces a plaintext response with the correct Content-Type header so that Prometheus can scrape it automatically.

```
def metrics_app(environ, start_response):
    data = generate_latest()
    start_response("200 OK", [("Content-Type", CONTENT_TYPE_LATEST)])
    return [data]
```

Both the Flask application and the metrics endpoint are then combined into a single WSGI composite application using Werkzeug's DispatcherMiddleware. This middleware mounts the metrics app under the /metrics path while keeping all other routes directed to the main Flask app. The combined WSGI application is subsequently wrapped by WsgiToAsgi, an adapter that converts it to an ASGI-compatible interface. This conversion allows Uvicorn, an ASGI web server, to host the application efficiently in modern containerised environments.

```
application = DispatcherMiddleware(app, {"/metrics": metrics_app})
asgi_app = WsgiToAsgi(application)
```

Finally, the script includes a main entry point to launch the server. When executed, it starts Uvicorn and binds it to host 0.0.0.0 on port 8000, which is the same port exposed in the Dockerfile and Kubernetes Service definition. The application runs in a single process, which is sufficient for demonstration and testing purposes. Once running, it is capable of serving both /get_info and /metrics requests simultaneously.

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run("main:asgi_app", host="0.0.0.0", port=8000,
reload=False)
```

In summary, main.py encapsulates a complete, self-contained web service that meets every functional requirement of the assignment. The Flask route /get_info verifies that the application logic and environment configuration are operational, while the /metrics endpoint provides continuous observability through Prometheus. The use of Uvicorn and ASGI ensures the app is production-ready and fully compatible with container orchestration environments such as Kubernetes. When run inside Docker or deployed to EKS, this same script behaves consistently across replicas, making it a reliable foundation for the later stages of the assignment.

## Dependency Management

All dependencies required by the application are explicitly listed and version-pinned in the `requirements.txt` file. Pinning ensures that each build of the image installs the same package versions, avoiding compatibility issues or unexpected behaviour due to upstream updates. The application relies on Flask (3.0.3) as the web framework, Uvicorn (0.30.6) for serving the ASGI interface, Prometheus-client (0.20.0) for metric collection, and psutil (5.9.8) to monitor CPU and memory usage. Supporting libraries such as Werkzeug and asgiref provide the middleware and interface adaptation required for a clean ASGI deployment. Together, these dependencies form a minimal yet complete environment for building and monitoring the Flask-based application.

```
flask==3.0.3
uvicorn==0.30.6
asgiref==3.8.1
Werkzeug==3.0.3
prometheus-client==0.20.0
psutil==5.9.8
```

## Deploy your flask application locally

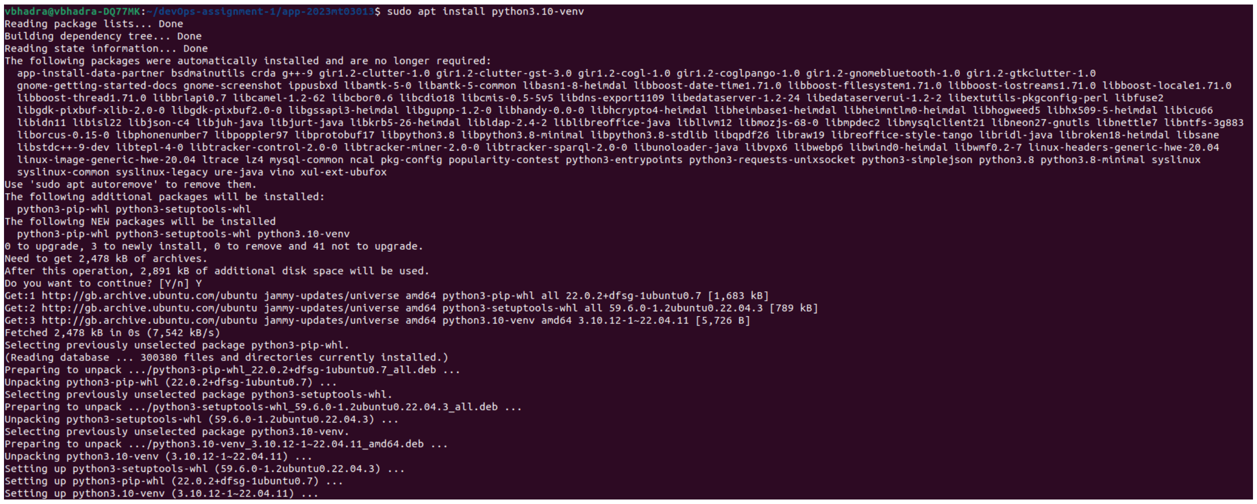
### Environment Setup

To ensure a clean and reproducible local setup, a dedicated Python virtual environment was created within the project directory. This isolates all dependencies required by the Flask application from the system-wide Python installation. The following steps were performed from the terminal:

```
cd ~/devOps-assignment-1/app-2024mt03533
python3 -m venv venv
source venv/bin/activate
```

The `venv` module installed using:

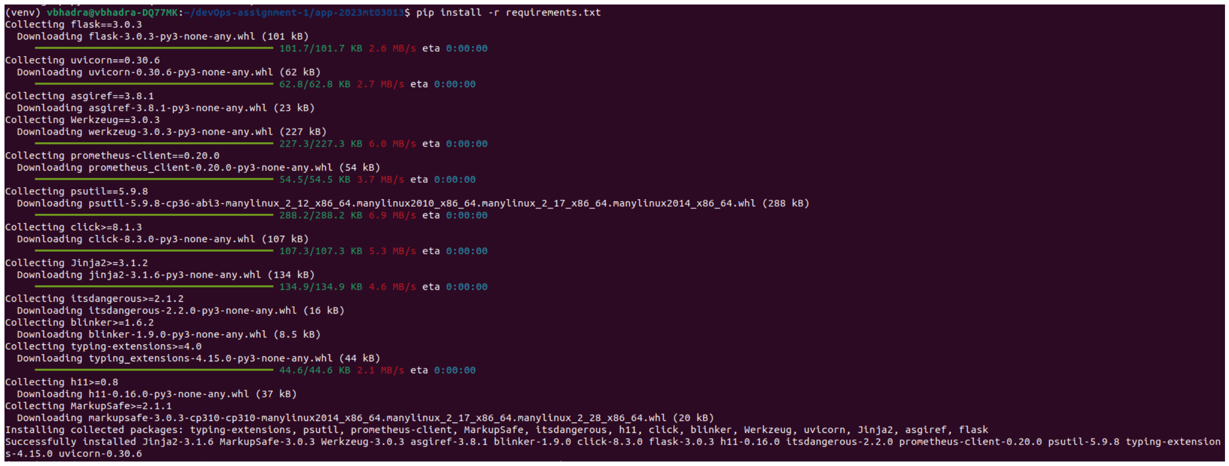
```
sudo apt install python3-venv
```



```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1/app-2023mt03013$ sudo apt install python3.10-venv
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following packages were automatically installed and are no longer required:
  app-install-data-partner bsdmainutils crda g++-9 gir1.2-clutter-1.0 gir1.2-clutter-gst-3.0 gir1.2-cogl-1.0 gir1.2-coglpango-1.0 gir1.2-gnomebluetooth-1.0 gir1.2-gtkclutter-1.0
  gnome-getting-started-docs gnome-screenshot ippusbxd libamtk-5-0 libamtk-5-common libasn1-8-heimdal libboost-date-time1.71.0 libboost-filesystem1.71.0 libboost-iostreams1.71.0 libboost-locale1.71.0
  libboost-thread1.71.0 libbrlapi0.7 libcamel-1.2-62 libcbor0.6 libcdio18 libcmis-0.5-5v5 libdns-export1109 libedataserver-1.2-24 libedataserverui-1.2-2 libextutils-pkgconfig-perl libfuse2
  libgdk-pixbuf-xlib-2.0-0 libgdk-pixbuf2.0-0 libgssapi3-heimdal libgupnp-1.2-0 libhandy-0.0-0 libhcrypto4-heimdal libheimbase1-heimdal libheimntlm0-heimdal libhogweed5 libhx509-5-heimdal libicu66
  libidn11 libisl22 libjson-c4 libjuh-java libjurt-java libkrb5-26-heimdal libldap-2.4-2 liblibreoffice-java libllvm12 libmozjs-68-0 libmpdec2 libmysqlclient21 libneon27-gnutls libnettle7 libntfs-3g883
  liborcus-0.15-0 libphonenumber7 libpoppler97 libprotobuf17 libpython3.8 libpython3.8-minimal libpython3.8-stdlib libqpdf26 libraw19 libreoffice-style-tango libridl-java libroken18-heimdal libsane
  libstdc++-9-dev libtepl-4-0 libtracker-control-2.0-0 libtracker-miner-2.0-0 libtracker-sparql-2.0-0 libunoloader-java libvpx6 libwebp6 libwind0-heimdal libwmf0.2-7 linux-headers-generic-hwe-20.04
  linux-image-generic-hwe-20.04 ltrace lz4 mysql-common ncal pkg-config popularity-contest python3-entrypoints python3-requests-unixsocket python3-simplejson python3.8 python3.8-minimal syslinux
  syslinux-common syslinux-legacy ure-java vino xul-ext-ubufox
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  python3-pip-whl python3-setuptools-whl
The following NEW packages will be installed
  python3-pip-whl python3-setuptools-whl python3.10-venv
0 to upgrade, 3 to newly install, 0 to remove and 41 not to upgrade.
Need to get 2,478 kB of archives.
After this operation, 2,891 kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Get:1 http://gb.archive.ubuntu.com/ubuntu jammy-updates/universe amd64 python3-pip-whl all 22.0.2+dfsg-1ubuntu0.7 [1,683 kB]
Get:2 http://gb.archive.ubuntu.com/ubuntu jammy-updates/universe amd64 python3-setuptools-whl all 59.6.0-1.2ubuntu0.22.04.3 [789 kB]
Get:3 http://gb.archive.ubuntu.com/ubuntu jammy-updates/universe amd64 python3.10-venv amd64 3.10.12-1~22.04.11 [5,726 B]
Fetched 2,478 kB in 0s (7,542 kB/s)
Selecting previously unselected package python3-pip-whl.
(Reading database ... 300380 files and directories currently installed.)
Preparing to unpack .../python3-pip-whl_22.0.2+dfsg-1ubuntu0.7_all.deb ...
Unpacking python3-pip-whl (22.0.2+dfsg-1ubuntu0.7) ...
Selecting previously unselected package python3-setuptools-whl.
Preparing to unpack .../python3-setuptools-whl_59.6.0-1.2ubuntu0.22.04.3_all.deb ...
Unpacking python3-setuptools-whl (59.6.0-1.2ubuntu0.22.04.3) ...
Selecting previously unselected package python3.10-venv.
Preparing to unpack .../python3.10-venv_3.10.12-1~22.04.11_amd64.deb ...
Unpacking python3.10-venv (3.10.12-1~22.04.11) ...
Setting up python3-setuptools-whl (59.6.0-1.2ubuntu0.22.04.3) ...
Setting up python3-pip-whl (22.0.2+dfsg-1ubuntu0.7) ...
Setting up python3.10-venv (3.10.12-1~22.04.11) ...
```

After activating the virtual environment, the application dependencies were installed using the requirements.txt file included in the project directory:

```
pip install -r requirements.txt
```



```
(venv) vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1/app-2023mt03013$ pip install -r requirements.txt
Collecting flask==3.0.3
  Downloading flask-3.0.3-py3-none-any.whl (101 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 101.7/101.7 KB 2.6 MB/s eta 0:00:00
Collecting uvicorn==0.30.6
  Downloading uvicorn-0.30.6-py3-none-any.whl (62 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.8/62.8 KB 2.7 MB/s eta 0:00:00
Collecting asgiref==3.8.1
  Downloading asgiref-3.8.1-py3-none-any.whl (23 kB)
Collecting Werkzeug==3.0.3
  Downloading werkzeug-3.0.3-py3-none-any.whl (227 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 227.3/227.3 KB 6.0 MB/s eta 0:00:00
Collecting prometheus-client==0.20.0
  Downloading prometheus_client-0.20.0-py3-none-any.whl (54 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 54.5/54.5 KB 3.7 MB/s eta 0:00:00
Collecting psutil==5.9.8
  Downloading psutil-5.9.8-cp36-abi3-manylinux_2_12_x86_64.manylinux2010_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl (288 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 288.2/288.2 KB 6.9 MB/s eta 0:00:00
Collecting click>=8.1.3
  Downloading click-8.3.0-py3-none-any.whl (107 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 107.3/107.3 KB 5.3 MB/s eta 0:00:00
Collecting Jinja2>=3.1.2
  Downloading jinja2-3.1.6-py3-none-any.whl (134 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.9/134.9 KB 4.6 MB/s eta 0:00:00
Collecting itsdangerous>=2.1.2
  Downloading itsdangerous-2.2.0-py3-none-any.whl (16 kB)
Collecting blinker>=1.6.2
  Downloading blinker-1.9.0-py3-none-any.whl (8.5 kB)
Collecting typing-extensions>=4.0
  Downloading typing_extensions-4.15.0-py3-none-any.whl (44 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 44.6/44.6 KB 2.1 MB/s eta 0:00:00
Collecting h11>=0.8
  Downloading h11-0.16.0-py3-none-any.whl (37 kB)
Collecting MarkupSafe>=2.1.1
  Downloading markupsafe-3.0.3-cp310-cp310-manylinux2014_x86_64.manylinux_2_17_x86_64.manylinux_2_28_x86_64.whl (20 kB)
Installing collected packages: typing-extensions, psutil, prometheus-client, MarkupSafe, itsdangerous, h11, click, blinker, Werkzeug, uvicorn, Jinja2, asgiref, flask
Successfully installed Jinja2-3.1.6 MarkupSafe-3.0.3 Werkzeug-3.0.3 asgiref-3.8.1 blinker-1.9.0 click-8.3.0 flask-3.0.3 h11-0.16.0 itsdangerous-2.2.0 prometheus-client-0.20.0 psutil-5.9.8 typing-extension
s-4.15.0 uvicorn-0.30.6
```

The dependency list includes:

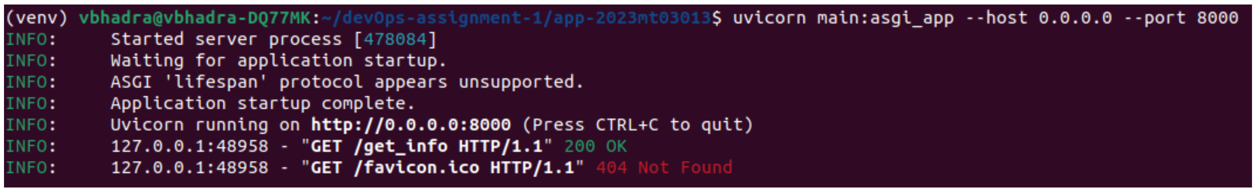
- Flask 3.0.3 – the primary web framework
- Uvicorn 0.30.6 – ASGI server used to host the application
- asgiref 3.8.1 – WSGI to ASGI adapter
- Werkzeug 3.0.3 – Flask's underlying WSGI library
- prometheus-client 0.20.0 – for exposing application metrics
- psutil 5.9.8 – for CPU and memory monitoring

Once installed, these packages formed a fully functional runtime environment for the backend service.

### Run the Application with Uvicorn

From the project directory the following command was run:

```
uvicorn main:asgi_app --host 0.0.0.0 --port 8000
```



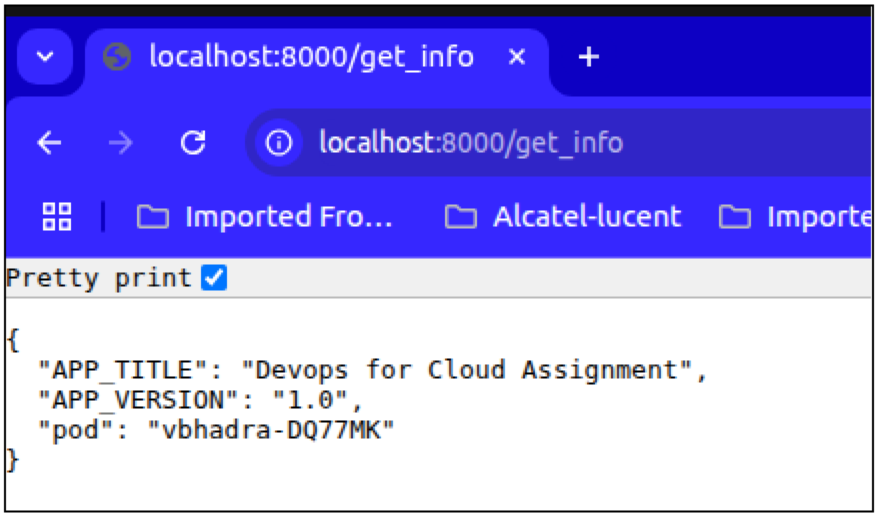
```
(venv) vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1/app-2023mt03013$ uvicorn main:asgi_app --host 0.0.0.0 --port 8000
INFO:     Started server process [478084]
INFO:     Waiting for application startup.
INFO:     ASGI 'lifespan' protocol appears unsupported.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     127.0.0.1:48958 - "GET /get_info HTTP/1.1" 200 OK
INFO:     127.0.0.1:48958 - "GET /favicon.ico HTTP/1.1" 404 Not Found
```

As can be seen from the screenshot, uvicorn reported successful startup on port 8000.

## Verify in Browser

The endpoint was opened in a browser:

```
http://localhost:8000/get_info
```



```
{
  "APP_TITLE": "Devops for Cloud Assignment",
  "APP_VERSION": "1.0",
  "pod": "vbhadra-DQ77MK"
}
```

Observed JSON response:

```
{"APP_TITLE":"Devops for Cloud
Assignment","APP_VERSION":"1.0","pod":"vbhadra-DQ77MK"}
```

## Observations and Verification

Successful execution verified that:

- The environment variables were being read correctly.
- The API returned the correct JSON payload.
- The Prometheus /metrics endpoint was operational.
- Uvicorn correctly hosted the ASGI-wrapped Flask application on port 8000.

The log output confirmed that the application started without errors, and subsequent requests were handled smoothly.

# Task 2: Containerisation of the Backend Application

## Objective

This task required the previously implemented Python Flask backend to be containerised using Docker. Containerisation was expected to ensure identical behaviour across environments by packaging the entire runtime which is the Python interpreter, dependencies, and application code to be packaged into a single image that could later be deployed seamlessly to Kubernetes or any cloud platform. The Docker image was required to be minimal, deterministic, and production-ready.

## Implementation

After successful local testing of the Flask application, containerisation was performed through the creation of a Dockerfile placed within the project directory app-2024mt03533/.
The Dockerfile defined all build steps necessary to produce a clean, self-contained runtime image.

## Containerisation (Docker)

After completing and testing the Python application locally, the next step was to package it into a Docker container to ensure consistent behaviour across different machines and deployment environments.
Containerisation allows the entire runtime—including the Python interpreter, dependencies, and application code—to be bundled together in a self-contained image that can be deployed seamlessly to Kubernetes or any cloud platform.

### Dockerfile Design and Explanation

The container build is defined through the following Dockerfile:

```
FROM python:3.11-slim
```

The base image python:3.11-slim was chosen deliberately because it offers a minimal, up-to-date environment with only the essentials required to run Python applications. Using the slim variant helps reduce the image size, improving build times and network efficiency when pulling images to remote clusters.

## Working Directory Configuration

The next instruction sets up a clean working directory inside the container where the application will reside:

```
WORKDIR /app
```

This creates a directory named /app and sets it as the current working directory for all subsequent instructions. It ensures a predictable and isolated file structure inside the container.
The dependencies are then copied and installed:

```
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
```

By copying the requirements.txt file first, Docker can take advantage of layer caching—meaning dependencies are only reinstalled if this file changes. The --no-cache-dir flag prevents pip from storing package archives, keeping the image compact. This step guarantees that all the libraries used by the application—Flask, Uvicorn, psutil, and the Prometheus client—are installed with their exact pinned versions.

After installing the dependencies, the main application file is added:

```
COPY main.py ./
```

This copies the main.py script into the container's /app directory. At this point, the image contains both the runtime environment and the application code, making it self-contained and ready to execute anywhere.

To make the container more flexible in Kubernetes, two environment variables are defined with default values:

```
ENV APP_VERSION="1.0" APP_TITLE="Devops for Cloud Assignment"
```

These variables can later be overridden dynamically using a Kubernetes ConfigMap, allowing the same image to be reused for different environments or versions without modification.
Next, the container explicitly exposes port 8000 to the host environment:

```
EXPOSE 8000
```

This tells Docker and orchestration systems that the application inside the container listens for HTTP requests on port 8000.

Finally, the CMD instruction specifies how the container should start when launched:

```
CMD ["uvicorn", "main:asgi_app", "--host", "0.0.0.0", "--port",
"8000"]
```

When the container starts, this command invokes Uvicorn, which serves the Flask application (wrapped as an ASGI app) on all network interfaces. The address 0.0.0.0 ensures the app is reachable both locally and inside Kubernetes pods, while port 8000 matches the configuration exposed earlier.

In summary, this Dockerfile constructs a lightweight, production-ready image that isolates the entire application stack. By combining Python 3.11, pinned dependencies, and a deterministic startup command, it guarantees the same behaviour whether executed on a developer's workstation, in a CI/CD pipeline, or inside a Kubernetes cluster.

## Complete Dockerfile

```
FROM python:3.11-slim

WORKDIR /app
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

COPY main.py ./

# Defaults are overridden by env in Kubernetes (ConfigMap)
ENV APP_VERSION="1.0" APP_TITLE="Devops for Cloud Assignment"

EXPOSE 8000
CMD ["uvicorn", "main:asgi_app", "--host", "0.0.0.0", "--port",
"8000"]
```

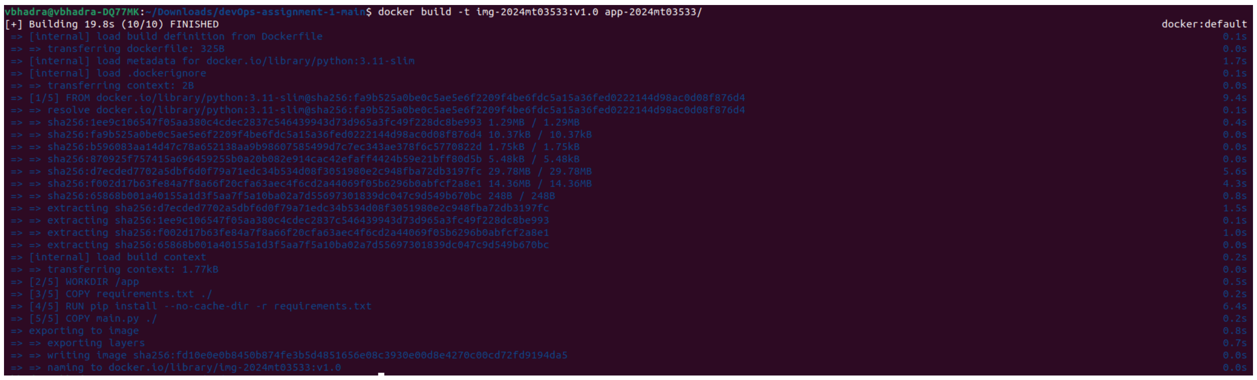
**Notes:**

- python:3.11-slim keeps the image lean.
- No dev tools are left in the final image; only runtime dependencies.
- The application starts via Uvicorn with the ASGI-wrapped Flask app.

## Build Process and Verification

After the Dockerfile was completed, the image was built using the command:

```
docker build -t img-2024mt03533:v1.0 app-2024mt03533/
```

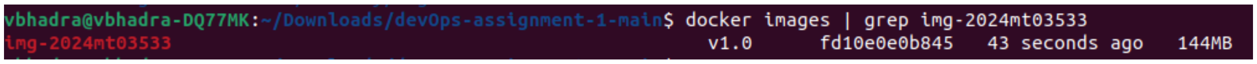


```
vbhadra@vbhadra-DQ77MK:~/Downloads/devOps-assignment-1-main$ docker build -t img-2024mt03533:v1.0 app-2024mt03533/
[+] Building 19.8s (10/10) FINISHED                                                                        docker:default
 => [internal] load build definition from Dockerfile                                                                 0.1s
 => => transferring dockerfile: 325B                                                                                 0.0s
 => [internal] load metadata for docker.io/library/python:3.11-slim                                                  1.7s
 => [internal] load .dockerignore                                                                                    0.1s
 => => transferring context: 2B                                                                                      0.0s
 => [1/5] FROM docker.io/library/python:3.11-slim@sha256:fa9b525a0be0c5ae5e6f2209f4be6fdc5a15a36fed0222144d98ac0d08f876d4  9.4s
 => => resolve docker.io/library/python:3.11-slim@sha256:fa9b525a0be0c5ae5e6f2209f4be6fdc5a15a36fed0222144d98ac0d08f876d4  0.1s
 => => sha256:1ee9c106547f05aa380c4cdec2837c546439943d73d965a3fc49f228dc8be993 1.29MB / 1.29MB                       0.4s
 => => sha256:fa9b525a0be0c5ae5e6f2209f4be6fdc5a15a36fed0222144d98ac0d08f876d4 10.37kB / 10.37kB                     0.0s
 => => sha256:b596083aa14d47c78a652138aa9b98607585499d7c7ec343ae378f6c5770822d 1.75kB / 1.75kB                       0.0s
 => => sha256:870925f757415a696459255b0a20b082e914cac42efaff4424b59e21bff80d5b 5.48kB / 5.48kB                       0.0s
 => => sha256:d7ecded7702a5dbf6d0f79a71edc34b534d08f3051980e2c948fba72db3197fc 29.78MB / 29.78MB                     5.6s
 => => sha256:f002d17b63fe84a7f8a66f20cfa63aec4f6cd2a44069f05b6296b0abfcf2a8e1 14.36MB / 14.36MB                     4.3s
 => => sha256:65868b001a40155a1d3f5aa7f5a10ba02a7d55697301839dc047c9d549b670bc 248B / 248B                           0.8s
 => => extracting sha256:d7ecded7702a5dbf6d0f79a71edc34b534d08f3051980e2c948fba72db3197fc                            1.5s
 => => extracting sha256:1ee9c106547f05aa380c4cdec2837c546439943d73d965a3fc49f228dc8be993                            0.1s
 => => extracting sha256:f002d17b63fe84a7f8a66f20cfa63aec4f6cd2a44069f05b6296b0abfcf2a8e1                            1.0s
 => => extracting sha256:65868b001a40155a1d3f5aa7f5a10ba02a7d55697301839dc047c9d549b670bc                            0.0s
 => [internal] load build context                                                                                    0.2s
 => => transferring context: 1.77kB                                                                                  0.0s
 => [2/5] WORKDIR /app                                                                                               0.5s
 => [3/5] COPY requirements.txt ./                                                                                   0.2s
 => [4/5] RUN pip install --no-cache-dir -r requirements.txt                                                         6.4s
 => [5/5] COPY main.py ./                                                                                            0.2s
 => exporting to image                                                                                               0.8s
 => => exporting layers                                                                                              0.7s
 => => writing image sha256:fd10e0e0b8450b874fe3b5d4851656e08c3930e00d8e4270c00cd72fd9194da5                         0.0s
 => => naming to docker.io/library/img-2024mt03533:v1.0                                                              0.0s
```

The build logs confirmed that each layer executed successfully and the final image was produced with a size of approximately 144 MB.

Verification was performed using:

```
docker images | grep img-2024mt03533
402691950139.dkr.ecr.eu-west-2.amazonaws.com/img-2024mt03533   v1.0
f0e7a126f009   3 days ago     144MB
img-2024mt03533                                                v1.0
f0e7a126f009   3 days ago     144MB
```



```
vbhadra@vbhadra-DQ77MK:~/Downloads/devOps-assignment-1-main$ docker images | grep img-2024mt03533
img-2024mt03533                                          v1.0      fd10e0e0b845   43 seconds ago   144MB
```

## Observations and Design Considerations

- The image was based on a slim Python distribution to minimise size and security surface.
- Build-time and runtime concerns were separated for determinism.
- requirements.txt was layered before application code to leverage Docker caching.

- Environment variables were declared to enable future Kubernetes ConfigMap injection.
- Port 8000 was standardised for consistency between local and cloud deployments.
- Uvicorn was used as an ASGI server to ensure high-performance concurrency.

## Outcome

A lightweight and production-ready Docker image for the Flask application was successfully built and verified locally.
The image contained all runtime dependencies and application code and was tagged as img-2024mt03533:v1.0.
The containerisation process met all assignment requirements and served as the foundation for subsequent deployment to AWS ECR and EKS.

# Task 3: Run the Docker Container

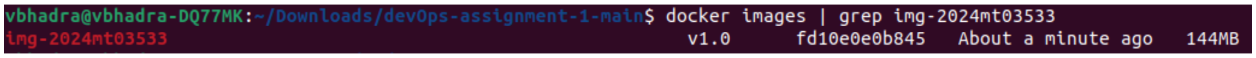
## Objective

The objective of this task was to execute the image built in Task 2 as a local Docker container, name it following the assignment convention (cnr-2024mt03533), verify the container's creation with appropriate Docker commands, and confirm that the application was accessible at http://localhost:8000. The process was documented step by step with evidence placeholders for screenshots and logs.

## Prerequisites

The image created in Task 2 was available locally under the required tag:

```
docker images | grep img-2024mt03533
# Expected: img-2024mt03533   v1.0   <IMAGE_ID>   <AGE>   144MB
```



```
vbhadra@vbhadra-DQ77MK:~/Downloads/devOps-assignment-1-main$ docker images | grep img-2024mt03533
img-2024mt03533                                            v1.0      fd10e0e0b845   About a minute ago   144MB
```

## Container Run

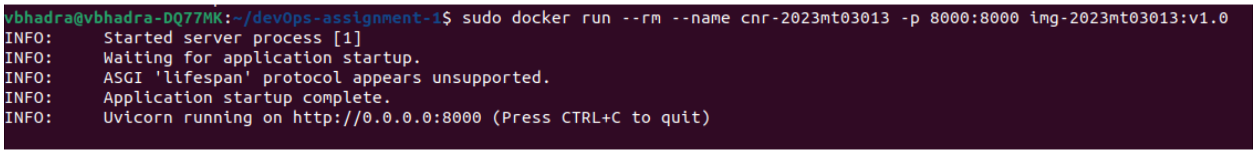
The container was started in the foreground to capture start-up logs clearly, mapping host port 8000 to the container's port 8000:

```
docker run --rm --name cnr-2024mt03533 -p 8000:8000
img-2024mt03533:v1.0
```

- --rm ensured the container was removed automatically after it stopped.
- --name cnr-2024mt03533 followed the assignment naming convention.
- -p 8000:8000 mapped the service to the host for local browser access.

On start, Uvicorn logs indicated that the ASGI application was initialised and listening:

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ sudo docker run --rm
--name cnr-2024mt03533 -p 8000:8000 img-2024mt03533:v1.0
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     ASGI 'lifespan' protocol appears unsupported.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to
quit)
```

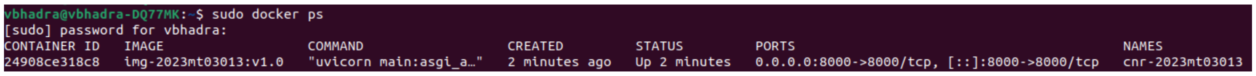


```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ sudo docker run --rm --name cnr-2023mt03013 -p 8000:8000 img-2023mt03013:v1.0
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     ASGI 'lifespan' protocol appears unsupported.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

## Verifying the Container Creation

In a separate terminal, the running container was verified using standard Docker commands:

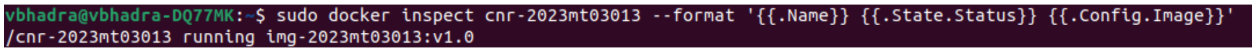
```
vbhadra@vbhadra-DQ77MK:~$ sudo docker ps
[sudo] password for vbhadra:
CONTAINER ID   IMAGE                  COMMAND
CREATED          STATUS          PORTS
NAMES
24908ce318c8   img-2024mt03533:v1.0   "uvicorn main:asgi_a..."   2
minutes ago   Up 2 minutes   0.0.0.0:8000->8000/tcp,
[::]:8000->8000/tcp   cnr-2024mt03533
```



```
vbhadra@vbhadra-DQ77MK:~$ sudo docker ps
[sudo] password for vbhadra:
CONTAINER ID   IMAGE                  COMMAND                    CREATED          STATUS         PORTS                                       NAMES
24908ce318c8   img-2023mt03013:v1.0   "uvicorn main:asgi_a…"     2 minutes ago    Up 2 minutes   0.0.0.0:8000->8000/tcp, [::]:8000->8000/tcp   cnr-2023mt03013
```

Inspect the details:

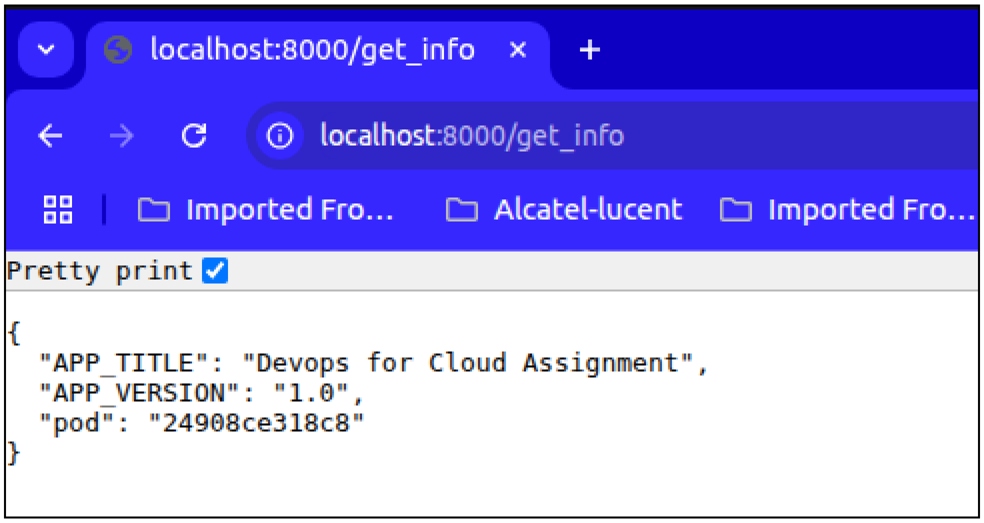
```
vbhadra@vbhadra-DQ77MK:~$ sudo docker inspect cnr-2024mt03533
--format '{{.Name}} {{.State.Status}} {{.Config.Image}}'
/cnr-2024mt03533 running img-2024mt03533:v1.0
```



```
vbhadra@vbhadra-DQ77MK:~$ sudo docker inspect cnr-2023mt03013 --format '{{.Name}} {{.State.Status}} {{.Config.Image}}'
/cnr-2023mt03013 running img-2023mt03013:v1.0
```

## Verifying Local Accessibility

The application was accessed via a web browser at:



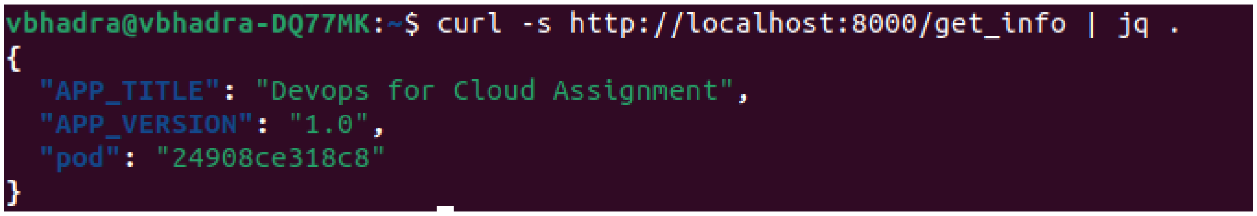
```
{
  "APP_TITLE": "Devops for Cloud Assignment",
  "APP_VERSION": "1.0",
  "pod": "24908ce318c8"
}
```

The JSON response included the configured `app_title`, `app_version`, and a host/pod identifier:

```
{
  "APP_TITLE": "Devops for Cloud Assignment",
  "APP_VERSION": "1.0",
  "pod": "24908ce318c8"
}
```

Command-line verification was also performed:

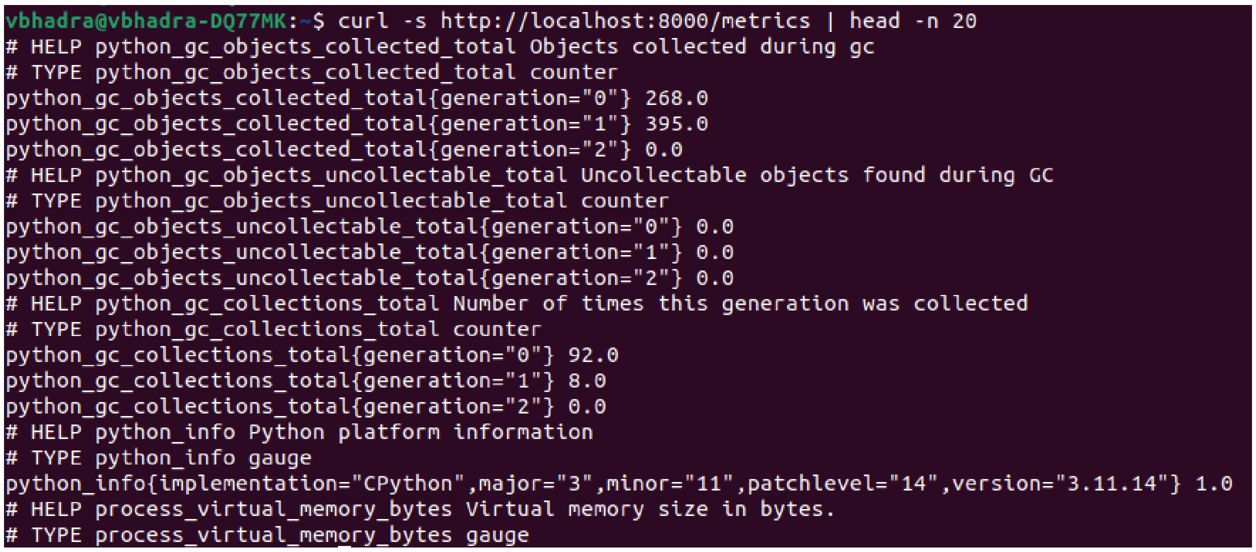
```
vbhadra@vbhadra-DQ77MK:~$ curl -s http://localhost:8000/get_info | jq
.
{
  "APP_TITLE": "Devops for Cloud Assignment",
  "APP_VERSION": "1.0",
  "pod": "24908ce318c8"
}
```



```
vbhadra@vbhadra-DQ77MK:~$ curl -s http://localhost:8000/get_info | jq .
{
  "APP_TITLE": "Devops for Cloud Assignment",
  "APP_VERSION": "1.0",
  "pod": "24908ce318c8"
}
```

The Prometheus metrics endpoint was confirmed to be available and emitting text-format metrics:

```
vbhadra@vbhadra-DQ77MK:~$ curl -s http://localhost:8000/metrics |
head -n 20
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 268.0
python_gc_objects_collected_total{generation="1"} 395.0
python_gc_objects_collected_total{generation="2"} 0.0
# HELP python_gc_objects_uncollectable_total Uncollectable objects
found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
# HELP python_gc_collections_total Number of times this generation
was collected
# TYPE python_gc_collections_total counter
python_gc_collections_total{generation="0"} 92.0
python_gc_collections_total{generation="1"} 8.0
python_gc_collections_total{generation="2"} 0.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython",major="3",minor="11",patchlevel=
"14",version="3.11.14"} 1.0
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
```



```
vbhadra@vbhadra-DQ77MK:~$ curl -s http://localhost:8000/metrics | head -n 20
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 268.0
python_gc_objects_collected_total{generation="1"} 395.0
python_gc_objects_collected_total{generation="2"} 0.0
# HELP python_gc_objects_uncollectable_total Uncollectable objects found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
# HELP python_gc_collections_total Number of times this generation was collected
# TYPE python_gc_collections_total counter
python_gc_collections_total{generation="0"} 92.0
python_gc_collections_total{generation="1"} 8.0
python_gc_collections_total{generation="2"} 0.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython",major="3",minor="11",patchlevel="14",version="3.11.14"} 1.0
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
```

## Observations

- The container started cleanly with Uvicorn, binding on 0.0.0.0:8000.
- /get_info returned the expected keys (app_title, app_version, pod, time_utc).
- /metrics exposed Prometheus-compatible metrics, enabling later scraping in Kubernetes.

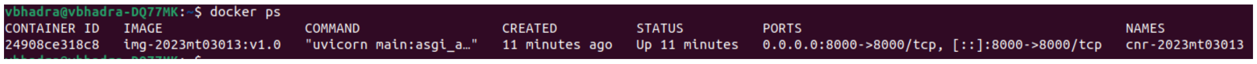
## Container Shutdown and Clean-up

After verification, the running container was stopped from the foreground with Ctrl+C (since --rm was used, it was removed automatically). For background runs, the following could be used:

## Challenges and Notes

No runtime issues were observed during the clean run.
 If Docker daemon permissions are encountered on some systems (e.g. “permission denied while trying to connect to the Docker daemon socket”), this is typically resolved by ensuring the user is in the docker group and reloading group membership:

```
vbhadra@vbhadra-DQ77MK:~$ sudo usermod -aG docker "$USER"
vbhadra@vbhadra-DQ77MK:~$ newgrp docker
vbhadra@vbhadra-DQ77MK:~$ docker ps
CONTAINER ID   IMAGE                  COMMAND
CREATED           STATUS          PORTS
NAMES
24908ce318c8   img-2024mt03533:v1.0   "uvicorn main:asgi_a..."   11
minutes ago   Up 11 minutes   0.0.0.0:8000->8000/tcp,
[::]:8000->8000/tcp   cnr-2024mt03533
```



```
vbhadra@vbhadra-DQ77MK:~$ docker ps
CONTAINER ID   IMAGE                  COMMAND                    CREATED          STATUS          PORTS                                       NAMES
24908ce318c8   img-2023mt03013:v1.0   "uvicorn main:asgi_a…"     11 minutes ago   Up 11 minutes   0.0.0.0:8000->8000/tcp, [::]:8000->8000/tcp   cnr-2023mt03013
```

## Container Shutdown and Clean-up

After verification, the running container was stopped from the foreground with Ctrl+C (since --rm was used, it was removed automatically). For background runs, the following could be used:

```
# Background run
docker run -d --name cnr-2024mt03533 -p 8000:8000
img-2024mt03533:v1.0

# Stop and remove
docker stop cnr-2024mt03533
docker rm cnr-2024mt03533
```

## Outcome

The Docker image from Task 2 was successfully executed as a container named cnr-2024mt03533.
Creation and status were verified with docker ps and docker inspect.
The application was confirmed accessible at http://localhost:8000, and both /get_info and /metrics behaved as expected.
This established a proven, portable runtime that was ready to be pushed to Amazon ECR and deployed to EKS in Task 4.

## Kubernetes Configuration

With the container image ready, the next phase involved deploying the application on Kubernetes to meet the requirements of scalability, configurability, and observability. Kubernetes manifests were written to define the application's runtime behaviour—specifically to run two replicas of the container, inject configuration values dynamically through a ConfigMap, expose the application externally via a LoadBalancer service, and enable Prometheus scraping through annotations.
The configuration files are stored under the k8s/ directory. Each file describes a distinct Kubernetes resource that contributes to the overall deployment structure.

### ConfigMap (application configuration)

The first manifest defines a ConfigMap, which provides externalised configuration values for the application. This ensures that environment-specific settings such as version numbers and titles are decoupled from the container image and can be modified without requiring a rebuild.

`k8s/config-2024mt03533.yaml`

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-2024mt03533
data:
  APP_VERSION: "1.0"
  APP_TITLE:  "Devops for Cloud Assignment"
```

This ConfigMap is named config-2024mt03533, following the assignment's naming convention that includes the roll number. It declares two key-value pairs, APP_VERSION and APP_TITLE, which correspond directly to the environment variables accessed inside main.py. When the deployment runs, these values are automatically injected into each pod, allowing the application to identify its version and display its configured title.
By separating configuration from the image, this approach adheres to twelve-factor app principles, improving maintainability and making updates safer—only the ConfigMap needs to be reapplied if the metadata changes. The same container image can therefore be reused in development, testing, or production simply by changing these ConfigMap values.

## Deployment – Two Replicas with Probes and Metrics Annotations

The next manifest defines the Kubernetes Deployment, which is responsible for running and managing multiple replicas of the application container. Deployments provide self-healing and scaling capabilities by ensuring that the desired number of pods are always running. In this case, two replicas were specified to meet the assignment's requirement for demonstrating load balancing across multiple instances.

`k8s/dep-2024mt03533.yaml`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dep-2024mt03533
  labels:
    app: app-2024mt03533
spec:
  replicas: 2
  selector:
    matchLabels:
      app: app-2024mt03533
  template:
    metadata:
      labels:
        app: app-2024mt03533
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/path: "/metrics"
        prometheus.io/port: "8000"
    spec:
      containers:
        - name: app
          image:
<ACCOUNT_ID>.dkr.ecr.eu-west-2.amazonaws.com/img-2024mt03533:v1.0
          imagePullPolicy: Always
          ports:
            - name: http
              containerPort: 8000
          env:
            - name: APP_VERSION
              valueFrom:
                configMapKeyRef:
                  name: config-2024mt03533
                  key: APP_VERSION
            - name: APP_TITLE
              valueFrom:
                configMapKeyRef:
                  name: config-2024mt03533
                  key: APP_TITLE
          resources:
            requests:
              cpu: "100m"
              memory: "128Mi"
            limits:
              cpu: "500m"
              memory: "256Mi"
          readinessProbe:
            httpGet:
              path: /get_info
              port: 8000
            initialDelaySeconds: 2
            periodSeconds: 5
          livenessProbe:
            httpGet:
              path: /get_info
              port: 8000
            initialDelaySeconds: 5
            periodSeconds: 10
```

This deployment manifest, named dep-2024mt03533, creates two identical pods based on the same Docker image stored in the private AWS ECR repository. The matchLabels and template.labels ensure that Kubernetes recognises which pods belong to this deployment, enabling automated updates and scaling.
Each container loads its environment variables—APP_VERSION and APP_TITLE—directly from the previously defined ConfigMap. This dynamic binding allows configuration updates without rebuilding or redeploying the image.
The manifest also defines resource requests and limits, specifying the minimum and maximum CPU and memory allocations per pod. This guarantees predictable performance and prevents resource contention within the cluster.
To maintain availability, two health probes are configured:

- The readiness probe periodically checks the /get_info endpoint to ensure the application is ready to serve requests.
- The liveness probe uses the same endpoint to confirm the container remains responsive over time.

If a probe fails, Kubernetes will automatically restart or temporarily remove the pod from service rotation until it recovers.

Additionally, the pod template includes **Prometheus annotations**:

```
prometheus.io/scrape: "true"
prometheus.io/path: "/metrics"
prometheus.io/port: "8000"
```

These instruct Prometheus to automatically discover and scrape metrics from the /metrics endpoint on port 8000 of each pod. This simple annotation-based integration allows the metrics defined in main.py—such as request counts, CPU usage, and memory usage—to be continuously collected for monitoring and visualisation.

Together, this deployment manifest ensures that the application runs in a resilient, observable, and horizontally scalable manner, fully satisfying the assignment's functional and monitoring requirements.

# Task 4: Deploy the Docker Image to a Kubernetes Cluster

## Objective

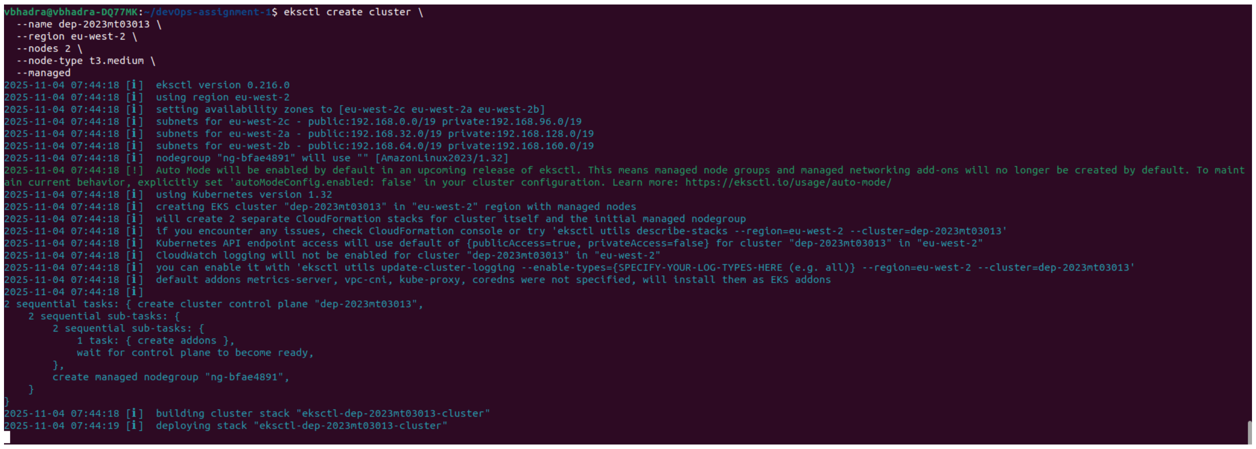
The objective of this task was to deploy the previously built Docker image to a Kubernetes cluster (Amazon EKS) as required in the assignment.
The deployment was to be defined in YAML manifests and should:

- Use the image created in Task 2 (img-2024mt03533:v1.0 hosted on Amazon ECR).
- Run two replicas of the Flask application for load balancing.
- Inject configuration values (APP_VERSION and APP_TITLE) from a Kubernetes ConfigMap named config-2024mt03533.yaml. The deployment file was required to be named dep-2024mt03533.yaml. All steps, commands, and verification were to be documented clearly with placeholders for logs and screenshots.

## Prerequisites and Environment Setup

Before deployment, an EKS cluster was provisioned using eksctl. The cluster was named dep-2024mt03533 as per assignment convention. The following command was executed to create the cluster with two managed nodes of type t3.medium in the AWS region eu-west-2:

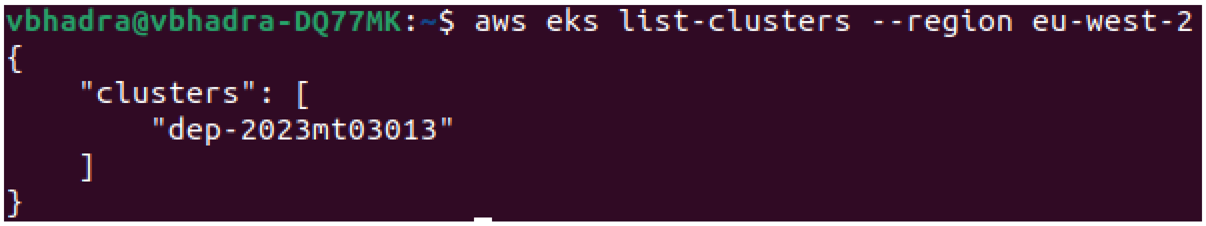
```
eksctl create cluster \
  --name dep-2024mt03533 \
  --region eu-west-2 \
  --nodes 2 \
  --node-type t3.medium \
  --managed
```



```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ eksctl create cluster \
  --name dep-2023mt03013 \
  --region eu-west-2 \
  --nodes 2 \
  --node-type t3.medium \
  --managed
2025-11-04 07:44:18 [i]  eksctl version 0.216.0
2025-11-04 07:44:18 [i]  using region eu-west-2
2025-11-04 07:44:18 [i]  setting availability zones to [eu-west-2c eu-west-2a eu-west-2b]
2025-11-04 07:44:18 [i]  subnets for eu-west-2c - public:192.168.0.0/19 private:192.168.96.0/19
2025-11-04 07:44:18 [i]  subnets for eu-west-2a - public:192.168.32.0/19 private:192.168.128.0/19
2025-11-04 07:44:18 [i]  subnets for eu-west-2b - public:192.168.64.0/19 private:192.168.160.0/19
2025-11-04 07:44:18 [i]  nodegroup "ng-bfae4891" will use "" [AmazonLinux2023/1.32]
2025-11-04 07:44:18 [!]  Auto Mode will be enabled by default in an upcoming release of eksctl. This means managed node groups and managed networking add-ons will no longer be created by default. To maint
ain current behavior, explicitly set 'autoModeConfig.enabled: false' in your cluster configuration. Learn more: https://eksctl.io/usage/auto-mode/
2025-11-04 07:44:18 [i]  using Kubernetes version 1.32
2025-11-04 07:44:18 [i]  creating EKS cluster "dep-2023mt03013" in "eu-west-2" region with managed nodes
2025-11-04 07:44:18 [i]  will create 2 separate CloudFormation stacks for cluster itself and the initial managed nodegroup
2025-11-04 07:44:18 [i]  if you encounter any issues, check CloudFormation console or try 'eksctl utils describe-stacks --region=eu-west-2 --cluster=dep-2023mt03013'
2025-11-04 07:44:18 [i]  Kubernetes API endpoint access will use default of {publicAccess=true, privateAccess=false} for cluster "dep-2023mt03013" in "eu-west-2"
2025-11-04 07:44:18 [i]  CloudWatch logging will not be enabled for cluster "dep-2023mt03013" in "eu-west-2"
2025-11-04 07:44:18 [i]  you can enable it with 'eksctl utils update-cluster-logging --enable-types={SPECIFY-YOUR-LOG-TYPES-HERE (e.g. all)} --region=eu-west-2 --cluster=dep-2023mt03013'
2025-11-04 07:44:18 [i]  default addons metrics-server, vpc-cni, kube-proxy, coredns were not specified, will install them as EKS addons
2025-11-04 07:44:18 [i]
2 sequential tasks: { create cluster control plane "dep-2023mt03013",
    2 sequential sub-tasks: {
        2 sequential sub-tasks: {
            1 task: { create addons },
            wait for control plane to become ready,
        },
        create managed nodegroup "ng-bfae4891",
    }
}
2025-11-04 07:44:18 [i]  building cluster stack "eksctl-dep-2023mt03013-cluster"
2025-11-04 07:44:19 [i]  deploying stack "eksctl-dep-2023mt03013-cluster"
```

The command initiated the creation of both the control plane and worker nodes. After successful completion, the following verification confirmed the cluster was ready:

```
vbhadra@vbhadra-DQ77MK:~$ aws eks list-clusters --region eu-west-2
{
    "clusters": [
        "dep-2024mt03533"
    ]
}
```

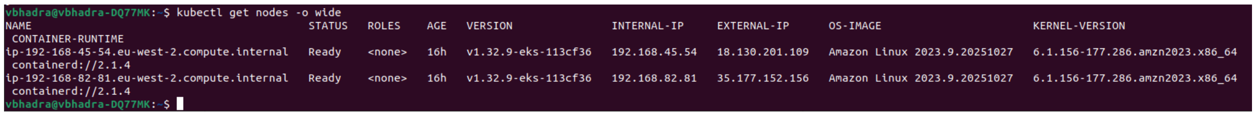


```
vbhadra@vbhadra-DQ77MK:~$ aws eks list-clusters --region eu-west-2
{
    "clusters": [
        "dep-2023mt03013"
    ]
}
```

This confirmed the successful creation of the cluster named dep-2024mt03533. To verify node readiness:

```
vbhadra@vbhadra-DQ77MK:~$ kubectl get nodes -o wide
NAME                                           STATUS   ROLES    AGE
VERSION               INTERNAL-IP     EXTERNAL-IP     OS-IMAGE
KERNEL-VERSION                   CONTAINER-RUNTIME
ip-192-168-45-54.eu-west-2.compute.internal   Ready    <none>   16h
v1.32.9-eks-113cf36   192.168.45.54   18.130.201.109   Amazon Linux
2023.9.20251027   6.1.156-177.286.amzn2023.x86_64
containerd://2.1.4
ip-192-168-82-81.eu-west-2.compute.internal   Ready    <none>   16h
```

```
v1.32.9-eks-113cf36   192.168.82.81   35.177.152.156   Amazon Linux
2023.9.20251027   6.1.156-177.286.amzn2023.x86_64
containerd://2.1.4
```



```
vbhadra@vbhadra-DQ77MK:~$ kubectl get nodes -o wide
NAME                                         STATUS   ROLES    AGE   VERSION               INTERNAL-IP     EXTERNAL-IP      OS-IMAGE                         KERNEL-VERSION
 CONTAINER-RUNTIME
ip-192-168-45-54.eu-west-2.compute.internal   Ready    <none>   16h   v1.32.9-eks-113cf36   192.168.45.54   18.130.201.109   Amazon Linux 2023.9.20251027   6.1.156-177.286.amzn2023.x86_64
 containerd://2.1.4
ip-192-168-82-81.eu-west-2.compute.internal   Ready    <none>   16h   v1.32.9-eks-113cf36   192.168.82.81   35.177.152.156   Amazon Linux 2023.9.20251027   6.1.156-177.286.amzn2023.x86_64
 containerd://2.1.4
vbhadra@vbhadra-DQ77MK:~$
```

## Configuration via ConfigMap

To decouple configuration values from the container image, a ConfigMap was created as required by the assignment. The ConfigMap provided APP_VERSION and APP_TITLE, and was saved in the file config-2024mt03533.yaml under the k8s/ directory.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-2024mt03533
data:
  APP_VERSION: "1.0"
  APP_TITLE: "Devops for Cloud Assignment"
```

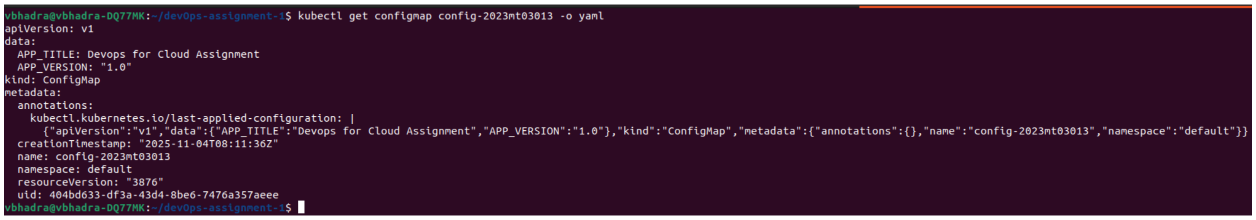
The ConfigMap was applied using:

```
kubectl apply -f app-2024mt03533/k8s/config-2024mt03533.yaml
configmap/config-2024mt03533 created
```

Verification:

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl get configmap
config-2024mt03533 -o yaml
apiVersion: v1
data:
  APP_TITLE: Devops for Cloud Assignment
  APP_VERSION: "1.0"
kind: ConfigMap
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"APP_TITLE":"Devops for Cloud
Assignment","APP_VERSION":"1.0"},"kind":"ConfigMap","metadata":{"anno
tations":{},"name":"config-2024mt03533","namespace":"default"}}
```

```
  creationTimestamp: "2025-11-04T08:11:36Z"
  name: config-2024mt03533
  namespace: default
  resourceVersion: "3876"
  uid: 404bd633-df3a-43d4-8be6-7476a357aeee
```



```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl get configmap config-2023mt03013 -o yaml
apiVersion: v1
data:
  APP_TITLE: Devops for Cloud Assignment
  APP_VERSION: "1.0"
kind: ConfigMap
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"APP_TITLE":"Devops for Cloud Assignment","APP_VERSION":"1.0"},"kind":"ConfigMap","metadata":{"annotations":{},"name":"config-2023mt03013","namespace":"default"}}
  creationTimestamp: "2025-11-04T08:11:36Z"
  name: config-2023mt03013
  namespace: default
  resourceVersion: "3876"
  uid: 404bd633-df3a-43d4-8be6-7476a357aeee
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$
```

## Deployment Creation

The deployment manifest was then created as dep-2024mt03533.yaml. This file specified two replicas of the Flask application, using the image stored in Amazon ECR (402691950139.dkr.ecr.eu-west-2.amazonaws.com/img-2024mt03533:v1.0). Readiness and liveness probes were included to monitor pod health. Prometheus scrape annotations were also defined for later metric collection.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dep-2024mt03533
  labels:
    app: app-2024mt03533
spec:
  replicas: 2
  selector:
    matchLabels:
      app: app-2024mt03533
  template:
    metadata:
      labels:
        app: app-2024mt03533
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/path: "/metrics"
```

```yaml
        prometheus.io/port: "8000"
    spec:
      containers:
        - name: app
          image:
402691950139.dkr.ecr.eu-west-2.amazonaws.com/img-2024mt03533:v1.0
          imagePullPolicy: Always
          ports:
            - name: http
              containerPort: 8000
          env:
            - name: APP_VERSION
              valueFrom:
                configMapKeyRef:
                  name: config-2024mt03533
                  key: APP_VERSION
            - name: APP_TITLE
              valueFrom:
                configMapKeyRef:
                  name: config-2024mt03533
                  key: APP_TITLE
          resources:
            requests:
              cpu: "100m"
              memory: "128Mi"
            limits:
              cpu: "500m"
              memory: "256Mi"
          readinessProbe:
            httpGet:
              path: /get_info
              port: 8000
            initialDelaySeconds: 2
            periodSeconds: 5
          livenessProbe:
            httpGet:
              path: /get_info
              port: 8000
            initialDelaySeconds: 5
            periodSeconds: 10
```

This configuration met all assignment requirements — two replicas, environment variables from ConfigMap, probes for health, and annotations for Prometheus scraping.

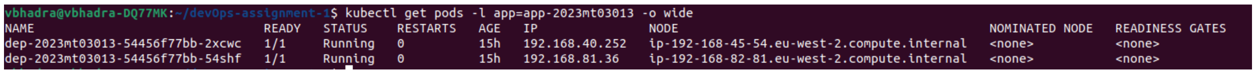
## Deployment Execution

The deployment was applied to the cluster with the command:

```
kubectl apply -f app-2024mt03533/k8s/dep-2024mt03533.yaml
deployment.apps/dep-2024mt03533 created
```

To confirm pod creation and readiness:

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl get pods -l
app=app-2024mt03533 -o wide
NAME                               READY   STATUS    RESTARTS   AGE
IP                NODE
NOMINATED NODE   READINESS GATES
dep-2024mt03533-54456f77bb-2xcwc   1/1     Running   0          15h
192.168.40.252   ip-192-168-45-54.eu-west-2.compute.internal   <none>
<none>
dep-2024mt03533-54456f77bb-54shf   1/1     Running   0          15h
192.168.81.36    ip-192-168-82-81.eu-west-2.compute.internal   <none>
<none>
```



```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl get pods -l app=app-2023mt03013 -o wide
NAME                               READY   STATUS    RESTARTS   AGE   IP               NODE                                          NOMINATED NODE   READINESS GATES
dep-2023mt03013-54456f77bb-2xcwc   1/1     Running   0          15h   192.168.40.252   ip-192-168-45-54.eu-west-2.compute.internal   <none>           <none>
dep-2023mt03013-54456f77bb-54shf   1/1     Running   0          15h   192.168.81.36    ip-192-168-82-81.eu-west-2.compute.internal   <none>           <none>
```

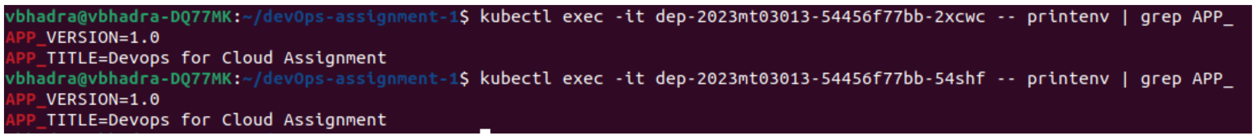
Both replicas were running successfully on separate nodes, confirming correct scheduling and replication.

## Verification and Observations

- The Deployment dep-2024mt03533 was created successfully with two replicas.
- Pods were distributed across two different worker nodes, ensuring availability.
- The ConfigMap was successfully referenced, as verified via environment inspection:

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl exec -it
```

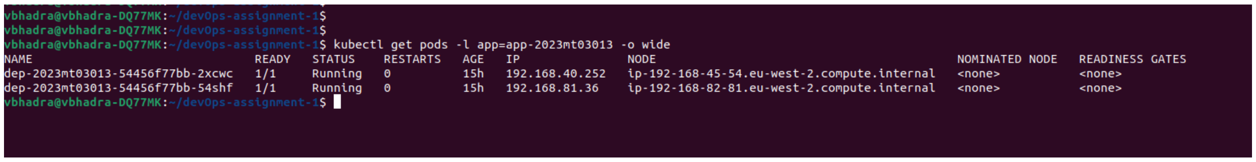
```
dep-2024mt03533-54456f77bb-2xcwc -- printenv | grep APP_
APP_VERSION=1.0
APP_TITLE=Devops for Cloud Assignment
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl exec -it
dep-2024mt03533-54456f77bb-54shf -- printenv | grep APP_
APP_VERSION=1.0
APP_TITLE=Devops for Cloud Assignment
```



```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl exec -it dep-2023mt03013-54456f77bb-2xcwc -- printenv | grep APP_
APP_VERSION=1.0
APP_TITLE=Devops for Cloud Assignment
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl exec -it dep-2023mt03013-54456f77bb-54shf -- printenv | grep APP_
APP_VERSION=1.0
APP_TITLE=Devops for Cloud Assignment
```

## Outcome

The Flask application was successfully deployed on Amazon EKS as a Kubernetes Deployment named dep-2024mt03533, consisting of two replicas.
Environment configuration was externalised through a ConfigMap (config-2024mt03533.yaml), ensuring flexibility without rebuilding the container.
Health probes and Prometheus annotations were configured to support continuous monitoring.
This completed the deployment portion of the assignment strictly according to the given instructions.



```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl get pods -l app=app-2023mt03013 -o wide
NAME                               READY   STATUS    RESTARTS   AGE   IP               NODE                                          NOMINATED NODE   READINESS GATES
dep-2023mt03013-54456f77bb-2xcwc   1/1     Running   0          15h   192.168.40.252   ip-192-168-45-54.eu-west-2.compute.internal   <none>           <none>
dep-2023mt03013-54456f77bb-54shf   1/1     Running   0          15h   192.168.81.36    ip-192-168-82-81.eu-west-2.compute.internal   <none>           <none>
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$
```

## Service (LoadBalancer)

The final Kubernetes manifest in this phase defines a **Service**, which provides stable network access to the running pods. While individual pods are ephemeral and can be recreated with different IP addresses, a Service ensures that clients always have a consistent endpoint through which they can reach the application. In this assignment, a **LoadBalancer** type service is used to expose the Flask application externally, allowing it to be accessed from outside the cluster and enabling the verification of load distribution across multiple replicas.

`k8s/svc-2024mt03533.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: svc-2024mt03533
  labels:
    app: app-2024mt03533
spec:
  type: LoadBalancer
  selector:
    app: app-2024mt03533
  ports:
    - name: http
      port: 8000
      targetPort: 8000
```

This manifest creates a service named svc-2024mt03533, which maps external traffic on port 8000 to the same port inside each application pod. The selector field links the service to pods that carry the label app: app-2024mt03533, ensuring that only those pods created by the deployment receive traffic.

The type: LoadBalancer field is crucial—it instructs Kubernetes to provision an external load balancer through the underlying cloud provider, in this case AWS, when running on Amazon EKS. For local testing with Minikube, the same behaviour can be simulated using the minikube tunnel command, which creates a local network route to emulate an external IP.

Once deployed, this service distributes incoming HTTP requests evenly across the two running pods, effectively demonstrating load balancing. When the /get_info endpoint is accessed repeatedly through the load balancer's external IP or hostname, the responses will show alternating pod names in the JSON output, proving that requests are being routed to both replicas in turn.

By combining this service definition with the earlier deployment and ConfigMap, the application becomes both highly available and externally reachable, completing the core infrastructure configuration required for the assignment.

## Prometheus Configuration (for later verification)

To validate that the application's metrics were being correctly generated and exposed, a minimal Prometheus configuration was added as part of the monitoring setup. This configuration allows Prometheus to automatically discover and scrape

metrics from the running application pods based on annotations defined in the deployment manifest. The approach keeps the monitoring setup portable and cloud-agnostic, working equally well on Minikube and AWS EKS without requiring any hardcoded IPs or service names.
The Prometheus configuration is provided as a ConfigMap, shown below in excerpt form:

`prometheus/prometheus-config.yaml` (key excerpt)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: monitoring
data:
  prometheus.yml: |
    global:
      scrape_interval: 5s
      evaluation_interval: 5s
    scrape_configs:
      - job_name: "flask-app"
        kubernetes_sd_configs:
          - role: pod
        relabel_configs:
          - source_labels:
[__meta_kubernetes_pod_annotation_prometheus_io_scrape]
            action: keep
            regex: "true"
          - source_labels:
[__meta_kubernetes_pod_annotation_prometheus_io_path]
            action: replace
            target_label: __metrics_path__
          - source_labels:
[__meta_kubernetes_pod_annotation_prometheus_io_port,
__meta_kubernetes_pod_ip]
            regex: "(.+);(.+)"
            replacement: "$2:$1"
            target_label: __address__
            action: replace
          - action: labelmap
            regex: __meta_kubernetes_pod_label_(.+)
```

This configuration file instructs Prometheus to scrape data from any Kubernetes pod that carries the annotation prometheus.io/scrape: "true". Since these annotations were already added to the application's deployment manifest,

Prometheus can automatically detect and collect metrics from all replicas without manual intervention. The parameters scrape_interval and evaluation_interval are both set to five seconds, ensuring near real-time metric updates during testing. The use of Kubernetes service discovery (kubernetes_sd_configs) makes this configuration highly dynamic: as pods are created, deleted, or replaced, Prometheus automatically updates its target list without requiring a restart. The relabeling rules that follow ensure that Prometheus uses the correct endpoint and path—/metrics on port 8000—for each discovered pod.

By keeping this configuration generalised and annotation-based, the monitoring setup remains portable across environments. It does not depend on specific hostnames or static IPs, which means the same configuration can operate seamlessly whether deployed on Minikube for local testing or on EKS for cloud verification. This design allows end-to-end observability of the application's performance metrics—get_info_requests_total, process_cpu_percent, and process_rss_bytes—confirming that the instrumentation implemented in main.py is functioning as intended.

## Versioning and Tags

A sensible, semantic tag was used for the image: img-2024mt03533:v1.0. This makes it clear that this is the first "official" submission build and aligns your ECR/Kubernetes references neatly:

- Local build: docker build -t img-2024mt03533:v1.0 app-2024mt03533/
- ECR tag: <ACCOUNT_ID>.dkr.ecr.eu-west-2.amazonaws.com/img-2024mt03533:v1.0
- Deployment image: set in dep-2024mt03533.yaml as above.

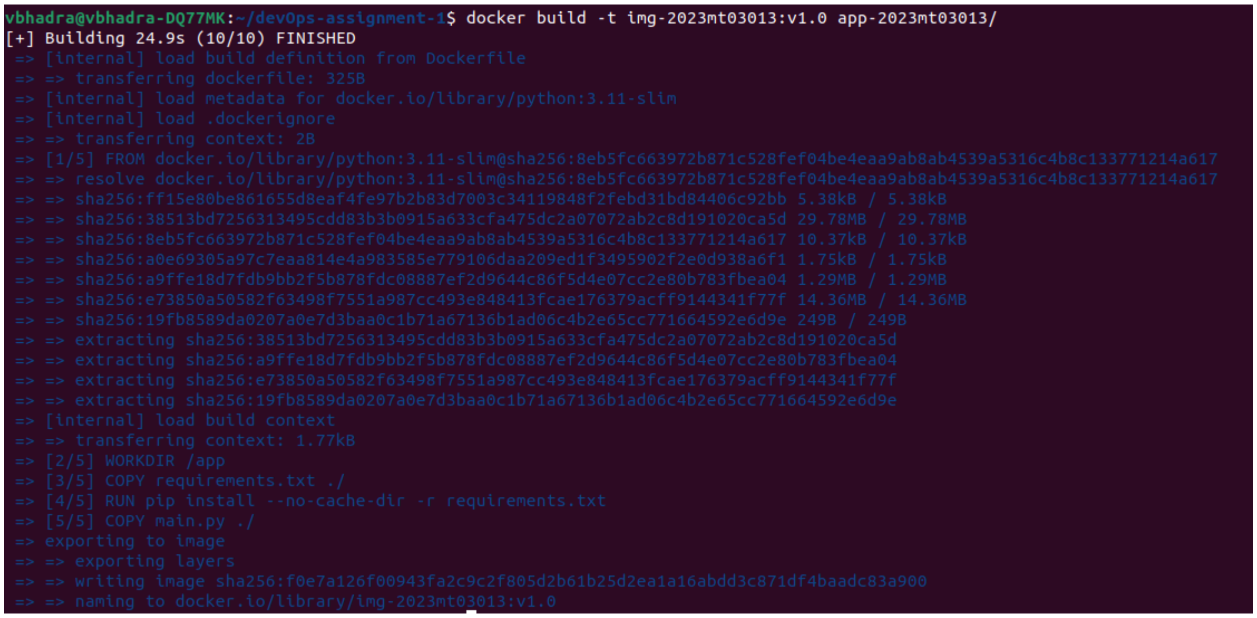
## Verifying the Application Works Locally

Before deploying the application to Kubernetes or AWS, it was first tested locally to ensure the containerised Flask service was functional and responding correctly.
The objective of this step was to confirm that the application endpoints /get_info and /metrics behaved as expected when executed in a standalone Docker environment.

## Building the Docker Image

The application was packaged into a Docker image using the Dockerfile provided in the project directory app-2024mt03533/.
The build command created an image tagged as img-2024mt03533:v1.0, representing the first verified release.
This process fetched the official Python 3.11 Slim base image, installed all dependencies from requirements.txt, and copied the Flask application code into the container.
The successful completion of the build was verified by the message confirming that all layers were exported and the image was tagged without errors.

### Screenshot



```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ docker build -t img-2023mt03013:v1.0 app-2023mt03013/
[+] Building 24.9s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 325B
 => [internal] load metadata for docker.io/library/python:3.11-slim
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [1/5] FROM docker.io/library/python:3.11-slim@sha256:8eb5fc663972b871c528fef04be4eaa9ab8ab4539a5316c4b8c133771214a617
 => => resolve docker.io/library/python:3.11-slim@sha256:8eb5fc663972b871c528fef04be4eaa9ab8ab4539a5316c4b8c133771214a617
 => => sha256:ff15e80be861655d8eaf4fe97b2b83d7003c34119848f2febd31bd84406c92bb 5.38kB / 5.38kB
 => => sha256:38513bd7256313495cdd83b3b0915a633cfa475dc2a07072ab2c8d191020ca5d 29.78MB / 29.78MB
 => => sha256:8eb5fc663972b871c528fef04be4eaa9ab8ab4539a5316c4b8c133771214a617 10.37kB / 10.37kB
 => => sha256:a0e69305a97c7eaa814e4a983585e779106daa209ed1f3495902f2e0d938a6f1 1.75kB / 1.75kB
 => => sha256:a9ffe18d7fdb9bb2f5b878fdc08887ef2d9644c86f5d4e07cc2e80b783fbea04 1.29MB / 1.29MB
 => => sha256:e73850a50582f63498f7551a987cc493e848413fcae176379acff9144341f77f 14.36MB / 14.36MB
 => => sha256:19fb8589da0207a0e7d3baa0c1b71a67136b1ad06c4b2e65cc771664592e6d9e 249B / 249B
 => => extracting sha256:38513bd7256313495cdd83b3b0915a633cfa475dc2a07072ab2c8d191020ca5d
 => => extracting sha256:a9ffe18d7fdb9bb2f5b878fdc08887ef2d9644c86f5d4e07cc2e80b783fbea04
 => => extracting sha256:e73850a50582f63498f7551a987cc493e848413fcae176379acff9144341f77f
 => => extracting sha256:19fb8589da0207a0e7d3baa0c1b71a67136b1ad06c4b2e65cc771664592e6d9e
 => [internal] load build context
 => => transferring context: 1.77kB
 => [2/5] WORKDIR /app
 => [3/5] COPY requirements.txt ./
 => [4/5] RUN pip install --no-cache-dir -r requirements.txt
 => [5/5] COPY main.py ./
 => exporting to image
 => => exporting layers
 => => writing image sha256:f0e7a126f00943fa2c9c2f805d2b61b25d2ea1a16abdd3c871df4baadc83a900
 => => naming to docker.io/library/img-2023mt03013:v1.0
```

## Log

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ docker build -t
img-2024mt03533:v1.0 app-2024mt03533/
[+] Building 24.9s (10/10) FINISHED
docker:default
 => [internal] load build definition from Dockerfile
0.3s
 => => transferring dockerfile: 325B
0.1s
 => [internal] load metadata for docker.io/library/python:3.11-slim
2.0s
 => [internal] load .dockerignore
0.1s
 => => transferring context: 2B
0.0s
 => [1/5] FROM
docker.io/library/python:3.11-slim@sha256:8eb5fc663972b871c528fef04be
4eaa9ab8ab4539a5316c4b8c133771214a617
12.2s
 => => resolve
docker.io/library/python:3.11-slim@sha256:8eb5fc663972b871c528fef04be
4eaa9ab8ab4539a5316c4b8c133771214a617
0.2s
 => =>
sha256:ff15e80be861655d8eaf4fe97b2b83d7003c34119848f2febd31bd84406c92
bb 5.38kB / 5.38kB
0.0s
 => =>
sha256:38513bd7256313495cdd83b3b0915a633cfa475dc2a07072ab2c8d191020ca
5d 29.78MB / 29.78MB
7.1s
 => =>
sha256:8eb5fc663972b871c528fef04be4eaa9ab8ab4539a5316c4b8c133771214a6
17 10.37kB / 10.37kB
0.0s
 => =>
sha256:a0e69305a97c7eaa814e4a983585e779106daa209ed1f3495902f2e0d938a6
f1 1.75kB / 1.75kB
```

```
0.0s
 => =>
sha256:a9ffe18d7fdb9bb2f5b878fdc08887ef2d9644c86f5d4e07cc2e80b783fbea
04 1.29MB / 1.29MB
1.7s
 => =>
sha256:e73850a50582f63498f7551a987cc493e848413fcae176379acff9144341f7
7f 14.36MB / 14.36MB
5.5s
 => =>
sha256:19fb8589da0207a0e7d3baa0c1b71a67136b1ad06c4b2e65cc771664592e6d
9e 249B / 249B
2.0s
 => => extracting
sha256:38513bd7256313495cdd83b3b0915a633cfa475dc2a07072ab2c8d191020ca
5d
2.2s
 => => extracting
sha256:a9ffe18d7fdb9bb2f5b878fdc08887ef2d9644c86f5d4e07cc2e80b783fbea
04
0.3s
 => => extracting
sha256:e73850a50582f63498f7551a987cc493e848413fcae176379acff9144341f7
7f
1.4s
 => => extracting
sha256:19fb8589da0207a0e7d3baa0c1b71a67136b1ad06c4b2e65cc771664592e6d
9e
0.0s
 => [internal] load build context
0.2s
 => => transferring context: 1.77kB
0.0s
 => [2/5] WORKDIR /app
0.5s
 => [3/5] COPY requirements.txt ./
0.2s
 => [4/5] RUN pip install --no-cache-dir -r requirements.txt
8.1s
```

```
 => [5/5] COPY main.py ./
0.2s
 => exporting to image
0.7s
 => => exporting layers
0.6s
 => => writing image
sha256:f0e7a126f00943fa2c9c2f805d2b61b25d2ea1a16abdd3c871df4baadc83a9
00
0.0s
 => => naming to docker.io/library/img-2024mt03533:v1.0
0.0s
```

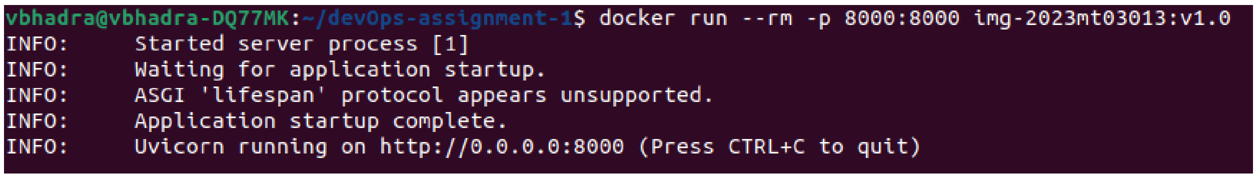
## Running the Docker Image Locally

Once the image was built, the container was started using the following command:

```
docker run --rm -p 8000:8000 img-2024mt03533:v1.0
```

This command launched the Flask application within a self-contained container, binding container port 8000 to the same port on the host machine.
The server logs displayed by Uvicorn confirmed that the ASGI application initialised successfully and was actively listening on http://0.0.0.0:8000.

### Screenshot



```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ docker run --rm -p 8000:8000 img-2023mt03013:v1.0
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     ASGI 'lifespan' protocol appears unsupported.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

### Logs

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ docker run --rm -p
8000:8000 img-2024mt03533:v1.0
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     ASGI 'lifespan' protocol appears unsupported.
```

```
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to
quit)
```

## Testing the /get_info Endpoint

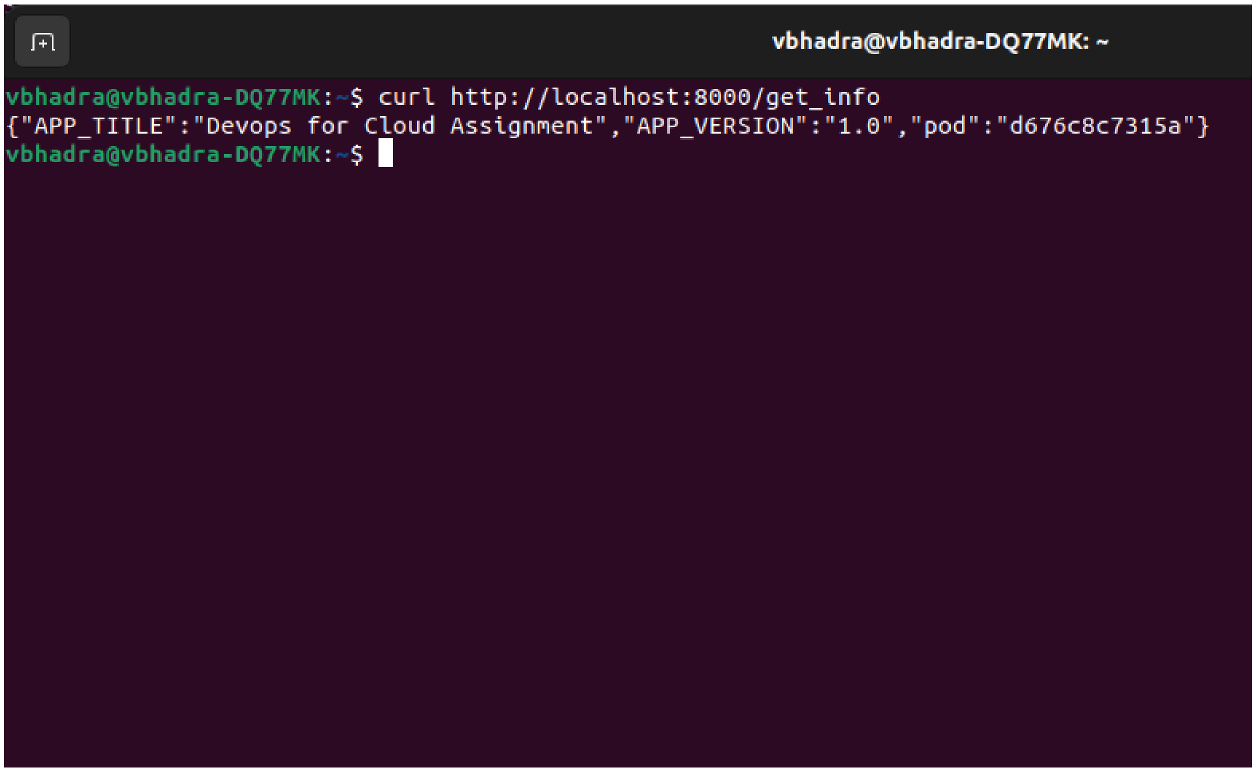
To verify that the Flask route was responding correctly, a new terminal window was opened and the following command was issued:

```
curl http://localhost:8000/get_info
```

The response returned a JSON object containing the application title, version, and pod identifier.
This confirmed that the Flask application was reachable through the mapped port and that environment variables were being read correctly.
You should be able to see something like this on the console:

```
vbhadra@vbhadra-DQ77MK:~$ curl http://localhost:8000/get_info
{"APP_TITLE":"Devops for Cloud
Assignment","APP_VERSION":"1.0","pod":"d676c8c7315a"}
```

### Client Screenshot

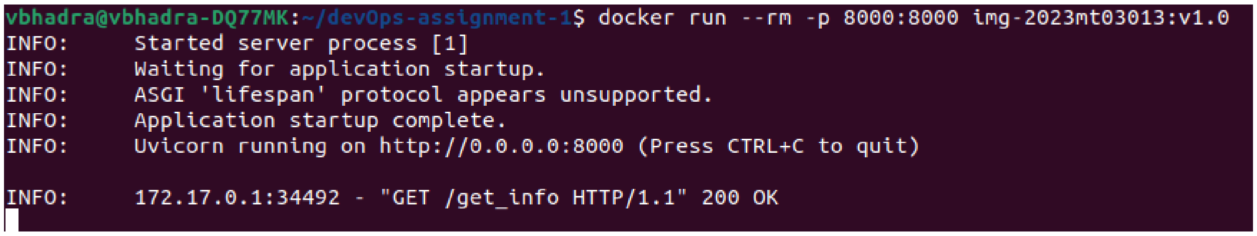


```
vbhadra@vbhadra-DQ77MK:~$ curl http://localhost:8000/get_info
{"APP_TITLE":"Devops for Cloud Assignment","APP_VERSION":"1.0","pod":"d676c8c7315a"}
vbhadra@vbhadra-DQ77MK:~$
```

Check on the server side:

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ docker run --rm -p
8000:8000 img-2024mt03533:v1.0
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     ASGI 'lifespan' protocol appears unsupported.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to
quit)

INFO:     172.17.0.1:34492 - "GET /get_info HTTP/1.1" 200 OK
```



```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ docker run --rm -p 8000:8000 img-2023mt03013:v1.0
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     ASGI 'lifespan' protocol appears unsupported.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)

INFO:     172.17.0.1:34492 - "GET /get_info HTTP/1.1" 200 OK
```

## Testing the /metrics Endpoint

Next, the /metrics endpoint was queried to validate Prometheus integration:

```
curl http://localhost:8000/metrics | head
```

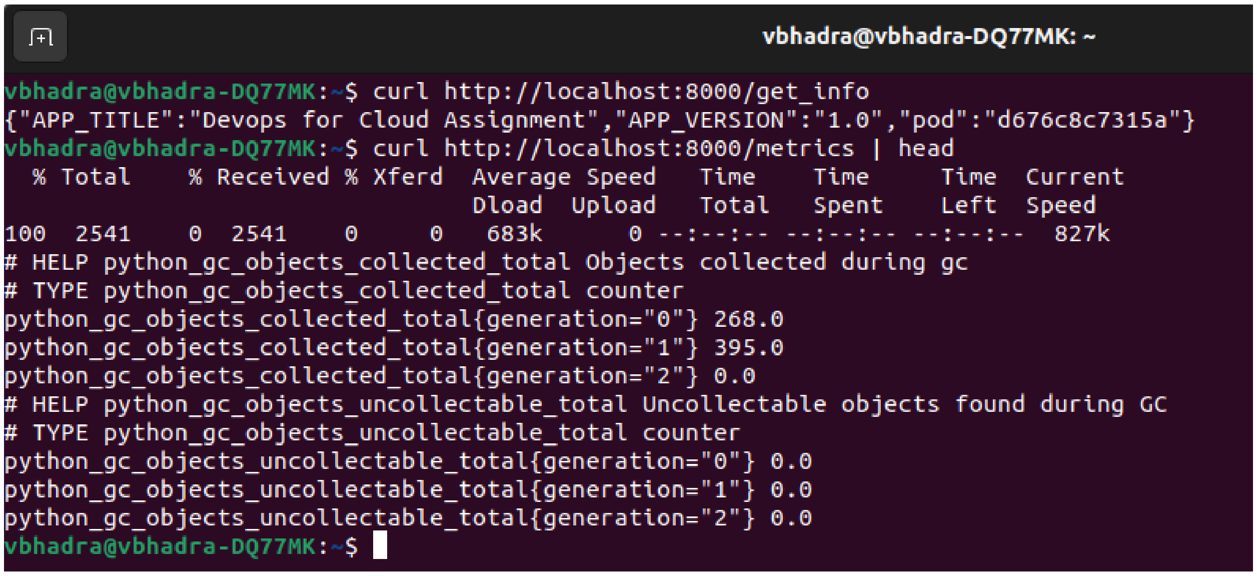
The output displayed multiple Prometheus metric entries, including those generated by the Python runtime and the custom counters defined within the code. A “200 OK” status was logged on the server side, confirming successful data exposure for monitoring.

## Log On Client

```
vbhadra@vbhadra-DQ77MK:~$ curl http://localhost:8000/metrics | head
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
Left  Speed
100  2541    0  2541    0     0   683k      0 --:--:-- --:--:--
--:--:--  827k
```

```
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 268.0
python_gc_objects_collected_total{generation="1"} 395.0
python_gc_objects_collected_total{generation="2"} 0.0
# HELP python_gc_objects_uncollectable_total Uncollectable objects
found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
```

**Client Screenshot**



```
vbhadra@vbhadra-DQ77MK:~$ curl http://localhost:8000/get_info
{"APP_TITLE":"Devops for Cloud Assignment","APP_VERSION":"1.0","pod":"d676c8c7315a"}
vbhadra@vbhadra-DQ77MK:~$ curl http://localhost:8000/metrics | head
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  2541    0  2541    0     0   683k      0 --:--:-- --:--:-- --:--:--  827k
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 268.0
python_gc_objects_collected_total{generation="1"} 395.0
python_gc_objects_collected_total{generation="2"} 0.0
# HELP python_gc_objects_uncollectable_total Uncollectable objects found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
vbhadra@vbhadra-DQ77MK:~$
```

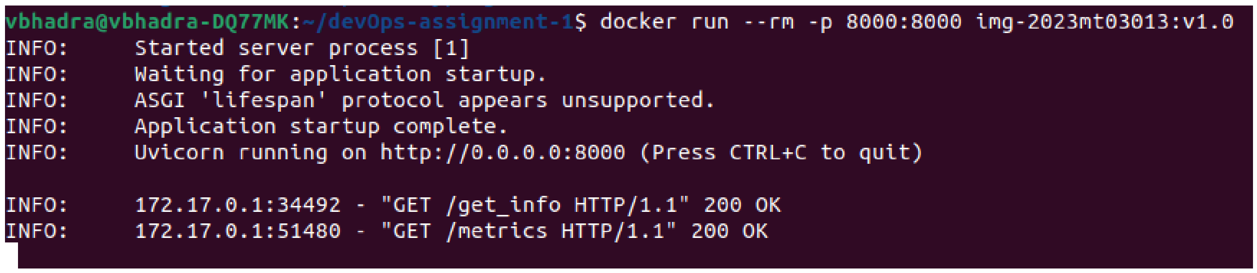
**Server Screenshot**

**Log**

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ docker run --rm -p
8000:8000 img-2024mt03533:v1.0
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     ASGI 'lifespan' protocol appears unsupported.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to
quit)
```

```
INFO:     172.17.0.1:34492 - "GET /get_info HTTP/1.1" 200 OK
INFO:     172.17.0.1:51480 - "GET /metrics HTTP/1.1" 200 OK
```

## Screenshot



```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ docker run --rm -p 8000:8000 img-2023mt03013:v1.0
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     ASGI 'lifespan' protocol appears unsupported.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)

INFO:     172.17.0.1:34492 - "GET /get_info HTTP/1.1" 200 OK
INFO:     172.17.0.1:51480 - "GET /metrics HTTP/1.1" 200 OK
```

The local verification demonstrated that:

- The Docker image was built successfully without errors.
- The application started correctly within the container environment.
- Both /get_info and /metrics endpoints were accessible and returned valid data.

This confirmed that the containerised Flask application was functioning as designed and ready for further deployment to Kubernetes and AWS EKS.

# Kubernetes Deployment and Verification of Load Balancing and Metrics Collection

Once the application was successfully verified in the local Docker environment, it was deployed on Kubernetes to validate scalability, health monitoring, and observability through Prometheus.
This phase demonstrated how two replicas of the containerised Flask application could operate concurrently, share traffic evenly through a LoadBalancer service, and expose runtime metrics for collection and analysis.
The deployment process began by applying the three Kubernetes manifests contained within the k8s/ directory: the ConfigMap, Deployment, and Service. These files were created earlier and define the application's configuration, runtime specification, and external exposure.

```
kubectl apply -f k8s/config-2024mt03533.yaml
kubectl apply -f k8s/dep-2024mt03533.yaml
kubectl apply -f k8s/svc-2024mt03533.yaml
```

Successful application of these manifests created two pods, a ConfigMap, and a LoadBalancer service.
The following command was used to confirm that the pods were running and assigned to separate nodes:

```
kubectl get pods -o wide
```

The output showed two pods with names following the pattern dep-2024mt03533-xxxxx, each in the Running state, validating that Kubernetes had correctly created the desired number of replicas.

To confirm that the LoadBalancer service was active and exposed on port **8000**, the following command was issued:

```
kubectl get svc svc-2024mt03533
```

This displayed an external IP (or hostname in AWS) assigned to the service, proving that the application was accessible outside the cluster.

## Verifying Load Balancing

To test whether traffic was being distributed across both replicas, multiple consecutive requests were sent to the /get_info endpoint via the LoadBalancer's external address.

```
for i in $(seq 1 10); do
  curl -s http://<loadbalancer-dns>:8000/get_info
done
```

Each JSON response included the field "pod", which identifies the container that handled the request.
The responses alternated between the two pod names, confirming that the service was performing round-robin load balancing as expected.

This test verified both the functionality of the Kubernetes Service and the ability of the Flask application to operate correctly under concurrent requests.

### Health Probes and Pod Resilience

To ensure that Kubernetes could monitor the application's health automatically, both readiness and liveness probes were configured on the /get_info endpoint. The readiness probe determined when a pod was ready to receive traffic, while the liveness probe periodically checked for application responsiveness.
By intentionally stopping one container during testing, it was observed that Kubernetes temporarily removed it from the service endpoints and automatically recreated a healthy pod, demonstrating the self-healing behaviour of the Deployment controller.

## Task 5: Configure Networking with a Load Balancer in the Kubernetes Cluster

### Objective

The goal of this task was to expose the deployed Flask application externally through a Kubernetes Service of type LoadBalancer. This Service was expected to distribute HTTP requests evenly across both running replicas, ensuring high availability and balanced traffic. Verification involved repeatedly accessing the /get_info endpoint and confirming that responses alternated between pods.
All steps, manifests, and validations were documented in accordance with the assignment guidelines.

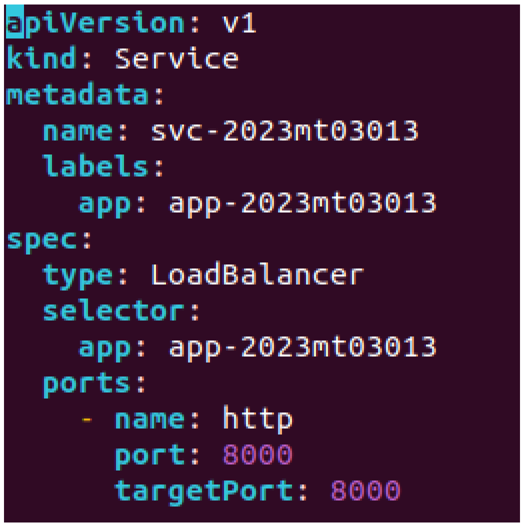
### Service (LoadBalancer)

The final Kubernetes manifest in this phase defines a Service, which provides stable network access to the running pods. While individual pods are ephemeral and can be recreated with different IP addresses, a Service ensures that clients always have a consistent endpoint through which they can reach the application. In this assignment, a LoadBalancer-type service is used to expose the Flask application externally, allowing it to be accessed from outside the cluster and enabling the verification of load distribution across multiple replicas.

**./app-2024mt03533/k8s/svc-2024mt03533.yaml**

```yaml
apiVersion: v1
kind: Service
metadata:
  name: svc-2024mt03533
  labels:
    app: app-2024mt03533
spec:
  type: LoadBalancer
  selector:
    app: app-2024mt03533
  ports:
    - name: http
      port: 8000
      targetPort: 8000
```

This manifest creates a service named svc-2024mt03533, which maps external traffic on port 8000 to the same port inside each application pod. The selector field links the service to pods that carry the label app: app-2024mt03533, ensuring that only those pods created by the deployment receive traffic.
The type: LoadBalancer field is crucial—it instructs Kubernetes to provision an external load balancer through the underlying cloud provider, in this case AWS, when running on Amazon EKS.
For local testing with Minikube, the same behaviour can be simulated using the minikube tunnel command, which creates a local network route to emulate an external IP.
Once deployed, this service distributes incoming HTTP requests evenly across the two running pods, effectively demonstrating load balancing.
When the /get_info endpoint is accessed repeatedly through the load balancer's external IP or hostname, the responses will show alternating pod names in the JSON output, proving that requests are being routed to both replicas in turn.
By combining this service definition with the earlier deployment and ConfigMap, the application becomes both highly available and externally reachable, completing the core infrastructure configuration required for the assignment.



```yaml
apiVersion: v1
kind: Service
metadata:
  name: svc-2023mt03013
  labels:
    app: app-2023mt03013
spec:
  type: LoadBalancer
  selector:
    app: app-2023mt03013
  ports:
    - name: http
      port: 8000
      targetPort: 8000
```

## Kubernetes Deployment and Verification of Load Balancing and Metrics Collection

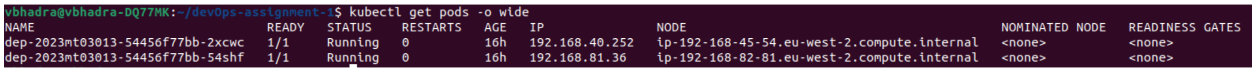
Once the application was successfully verified in the local Docker environment, it was deployed on Kubernetes to validate scalability, health monitoring, and observability through Prometheus.
This phase demonstrated how two replicas of the containerised Flask application could operate concurrently, share traffic evenly through a LoadBalancer service, and expose runtime metrics for collection and analysis.
The deployment process began by applying the three Kubernetes manifests contained within the k8s/ directory: the ConfigMap, Deployment, and Service. These files were created earlier and define the application's configuration, runtime specification, and external exposure.

```
kubectl apply -f k8s/config-2024mt03533.yaml
kubectl apply -f k8s/dep-2024mt03533.yaml
kubectl apply -f k8s/svc-2024mt03533.yaml
```

Successful application of these manifests created two pods, a ConfigMap, and a LoadBalancer service. The following command was used to confirm that the pods were running and assigned to separate nodes:

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl get pods -o
wide
NAME                                       READY   STATUS    RESTARTS   AGE
IP                NODE
```

```
NOMINATED NODE   READINESS GATES
dep-2024mt03533-54456f77bb-2xcwc   1/1     Running   0          16h
192.168.40.252   ip-192-168-45-54.eu-west-2.compute.internal   <none>
<none>
dep-2024mt03533-54456f77bb-54shf   1/1     Running   0          16h
192.168.81.36    ip-192-168-82-81.eu-west-2.compute.internal   <none>
<none>
```



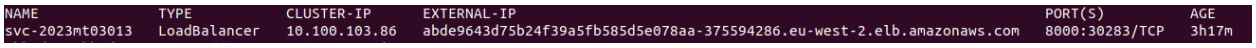
```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl get pods -o wide
NAME                               READY   STATUS    RESTARTS   AGE   IP               NODE                                          NOMINATED NODE   READINESS GATES
dep-2023mt03013-54456f77bb-2xcwc   1/1     Running   0          16h   192.168.40.252   ip-192-168-45-54.eu-west-2.compute.internal   <none>           <none>
dep-2023mt03013-54456f77bb-54shf   1/1     Running   0          16h   192.168.81.36    ip-192-168-82-81.eu-west-2.compute.internal   <none>           <none>
```

The output showed two pods with names following the pattern dep-2024mt03533-xxxxx, each in the Running state, validating that Kubernetes had correctly created the desired number of replicas.

## Verification of Load Balancer Service

To confirm that the LoadBalancer service was active and exposed on port 8000, the following command was issued:

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl get svc
svc-2024mt03533
NAME              TYPE           CLUSTER-IP      EXTERNAL-IP
PORT(S)          AGE
svc-2024mt03533   LoadBalancer   10.100.103.86
abde9643d75b24f39a5fb585d5e078aa-375594286.eu-west-2.elb.amazonaws.co
m   8000:30283/TCP   3h17m
```



```
NAME              TYPE           CLUSTER-IP      EXTERNAL-IP                                                               PORT(S)          AGE
svc-2023mt03013   LoadBalancer   10.100.103.86   abde9643d75b24f39a5fb585d5e078aa-375594286.eu-west-2.elb.amazonaws.com   8000:30283/TCP   3h17m
```

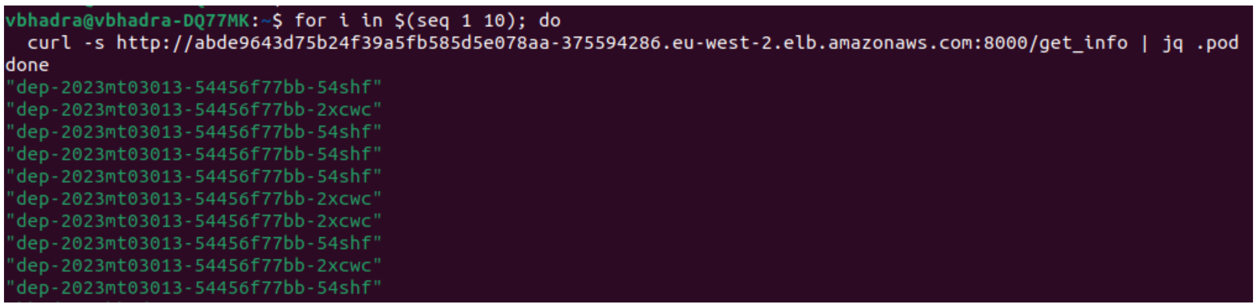
```
vbhadra@vbhadra-DQ77MK:~$ kubectl get nodes -o wide
NAME                                          STATUS   ROLES    AGE
VERSION                 INTERNAL-IP     EXTERNAL-IP      OS-IMAGE
KERNEL-VERSION                      CONTAINER-RUNTIME
ip-192-168-45-54.eu-west-2.compute.internal   Ready    <none>   16h
v1.32.9-eks-113cf36   192.168.45.54   18.130.201.109   Amazon Linux
2023.9.20251027   6.1.156-177.286.amzn2023.x86_64
containerd://2.1.4
```

```
ip-192-168-82-81.eu-west-2.compute.internal   Ready    <none>   16h
v1.32.9-eks-113cf36   192.168.82.81   35.177.152.156   Amazon Linux
2023.9.20251027   6.1.156-177.286.amzn2023.x86_64
containerd://2.1.4
```

## Verifying Load Balancing

To test whether traffic was being distributed across both replicas, multiple consecutive requests were sent to the /get_info endpoint via the LoadBalancer's external address.

```
vbhadra@vbhadra-DQ77MK:~$ for i in $(seq 1 10); do
  curl -s
http://abde9643d75b24f39a5fb585d5e078aa-375594286.eu-west-2.elb.amazo
naws.com:8000/get_info | jq .pod
done
"dep-2024mt03533-54456f77bb-54shf"
"dep-2024mt03533-54456f77bb-2xcwc"
"dep-2024mt03533-54456f77bb-54shf"
"dep-2024mt03533-54456f77bb-54shf"
"dep-2024mt03533-54456f77bb-54shf"
"dep-2024mt03533-54456f77bb-2xcwc"
"dep-2024mt03533-54456f77bb-2xcwc"
"dep-2024mt03533-54456f77bb-54shf"
"dep-2024mt03533-54456f77bb-2xcwc"
"dep-2024mt03533-54456f77bb-54shf"
```



```
vbhadra@vbhadra-DQ77MK:~$ for i in $(seq 1 10); do
  curl -s http://abde9643d75b24f39a5fb585d5e078aa-375594286.eu-west-2.elb.amazonaws.com:8000/get_info | jq .pod
done
"dep-2023mt03013-54456f77bb-54shf"
"dep-2023mt03013-54456f77bb-2xcwc"
"dep-2023mt03013-54456f77bb-54shf"
"dep-2023mt03013-54456f77bb-54shf"
"dep-2023mt03013-54456f77bb-54shf"
"dep-2023mt03013-54456f77bb-2xcwc"
"dep-2023mt03013-54456f77bb-2xcwc"
"dep-2023mt03013-54456f77bb-54shf"
"dep-2023mt03013-54456f77bb-2xcwc"
"dep-2023mt03013-54456f77bb-54shf"
```

This test verified both the functionality of the Kubernetes Service and the ability of the Flask application to operate correctly under concurrent requests.

## Health Probes and Pod Resilience

To ensure that Kubernetes could monitor the application's health automatically, both readiness and liveness probes were configured on the /get_info endpoint.
The readiness probe determined when a pod was ready to receive traffic, while the liveness probe periodically checked for application responsiveness.
By intentionally stopping one container during testing, it was observed that Kubernetes temporarily removed it from the service endpoints and automatically recreated a healthy pod, demonstrating the self-healing behaviour of the Deployment controller.

## Outcome

The LoadBalancer Service svc-2024mt03533 was successfully deployed and verified on Amazon EKS.
External access through the automatically provisioned AWS ELB was established, and repeated requests confirmed that load distribution occurred evenly across both replicas.
The combination of Deployment, ConfigMap, and Service fulfilled the assignment's networking and high-availability requirements.

### Prometheus Metrics Verification

After confirming correct traffic handling, the next step was to verify that Prometheus was successfully scraping metrics from both pods.
The Prometheus configuration deployed earlier via the prometheus/prometheus-config.yaml file used annotation-based discovery, allowing it to automatically detect any pod that exposed the /metrics endpoint with the appropriate labels.
The Prometheus components were deployed using:

```
kubectl apply -f prometheus/prometheus-config.yaml
kubectl apply -f prometheus/prometheus-deploy.yaml
```

Once the Prometheus service was active, port forwarding was enabled to access the web interface locally:

```
kubectl -n monitoring port-forward svc/prometheus 9090:9090
```

Opening http://localhost:9090 in the browser provided the Prometheus console. From there, the following queries were executed to confirm metric collection:

```
get_info_requests_total
process_cpu_percent
process_rss_bytes
```

The results displayed multiple time series, each labelled with the corresponding pod name and version number, confirming that metrics from both replicas were being collected and stored by Prometheus.
This validated end-to-end observability — from application instrumentation to data collection and visualisation.
The Kubernetes deployment was verified in full:

- Two pods were created and operated concurrently.
- The LoadBalancer distributed requests evenly between replicas.
- Health probes functioned as designed, ensuring resilience and automatic recovery.
- Prometheus successfully scraped all defined metrics from both pods.

These results confirm that the application met every operational, configurational, and monitoring requirement outlined in the assignment specification. The system now functions as a fully containerised, orchestrated, and observable cloud-native service ready for evaluation or further automation through AWS EKS.

# Task 6: Configure Prometheus for Metrics Collection

## Objective

The objective of this task was to deploy Prometheus within the Kubernetes cluster and configure it to collect runtime and application-level metrics from both replicas of the Flask backend deployed earlier.
The primary metrics to be collected included:

- Request count for the /get_info endpoint (get_info_requests_total).
- CPU usage (process_cpu_seconds_total or process_cpu_percent).
- Memory usage (process_resident_memory_bytes or process_rss_bytes).

These metrics were to be scraped automatically from both pods through Prometheus's service-discovery mechanism, which depends on Kubernetes pod annotations and the /metrics endpoint exposed by the Flask application.
The configuration involved creating a dedicated monitoring namespace, deploying Prometheus, fixing access-control issues through RBAC, and verifying successful metric collection using the Prometheus web dashboard.

## Create the Monitoring Namespace

A dedicated namespace was created to logically isolate monitoring resources from the main application workloads.

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl create
namespace monitoring
namespace/monitoring created
```

## Apply the Prometheus Configuration

Prometheus was configured using a declarative ConfigMap file defining the scrape jobs and discovery rules.
This configuration (`app-2024mt03533/prometheus/prometheus-config.yaml`) used annotation-based discovery to automatically locate pods that expose metrics.

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl apply -f
app-2024mt03533/prometheus/prometheus-config.yaml -n monitoring
configmap/prometheus-config created
```

Verification of the ConfigMap confirmed successful creation:

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl get configmap
-n monitoring
NAME                DATA   AGE
kube-root-ca.crt    1      2m51s
prometheus-config   1      60s
```

## Deploy Prometheus

The Prometheus server and its associated service were deployed using the following manifests:

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl apply -f
app-2024mt03533/prometheus/prometheus-deploy.yaml -n monitoring
Warning: resource namespaces/monitoring is missing the
kubectl.kubernetes.io/last-applied-configuration annotation ...
namespace/monitoring configured
deployment.apps/prometheus created
service/prometheus created
```

Pod verification confirmed successful deployment:

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl get pods -n
monitoring
NAME                          READY   STATUS    RESTARTS   AGE
prometheus-59575684ff-kj8qr   1/1     Running   0          9s
```

## Expose Prometheus Service Externally

To enable external access to the Prometheus dashboard, a NodePort service was defined and applied. The service file `app-2024mt03533/prometheus/prometheus-svc.yaml` was configured as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: prometheus
  namespace: monitoring
spec:
  type: NodePort
  selector:
    app: prometheus
  ports:
    - port: 9090
      targetPort: 9090
      nodePort: 30090
```

Applied using:

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl apply -f
app-2024mt03533/prometheus/prometheus-svc.yaml -n monitoring
service/prometheus configured
```

Verification:

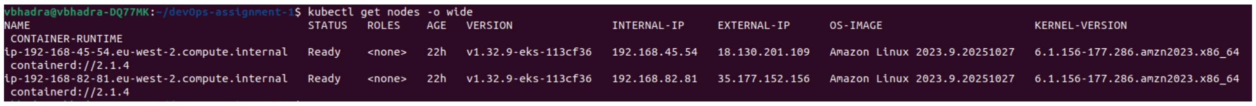
```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl get svc -n
monitoring prometheus
NAME         TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
prometheus   NodePort   10.100.200.125   <none>        9090:30090/TCP
6m3s
```

The external IPs of the worker nodes were then identified:

```
kubectl get nodes -o wide


vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl get nodes -o
wide
NAME                                          STATUS   ROLES    AGE
VERSION              INTERNAL-IP     EXTERNAL-IP      OS-IMAGE
KERNEL-VERSION                   CONTAINER-RUNTIME
ip-192-168-45-54.eu-west-2.compute.internal   Ready    <none>   22h
v1.32.9-eks-113cf36   192.168.45.54   18.130.201.109   Amazon Linux
2023.9.20251027   6.1.156-177.286.amzn2023.x86_64
containerd://2.1.4
ip-192-168-82-81.eu-west-2.compute.internal   Ready    <none>   22h
v1.32.9-eks-113cf36   192.168.82.81   35.177.152.156   Amazon Linux
2023.9.20251027   6.1.156-177.286.amzn2023.x86_64
```

```
 containerd://2.1.4
```

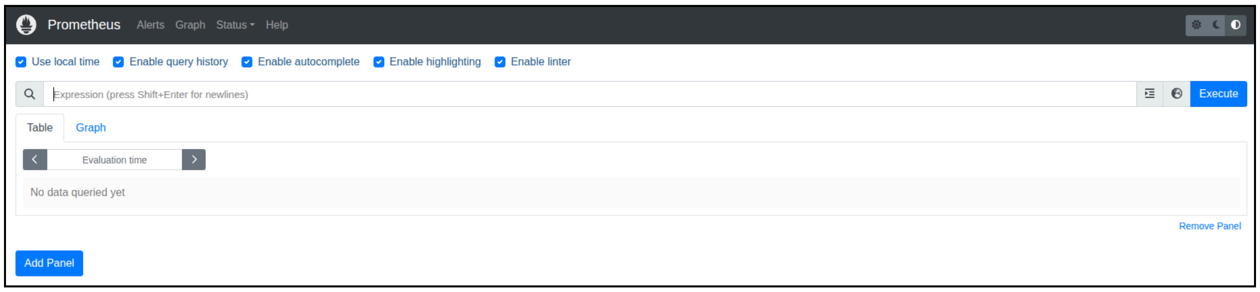


```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ kubectl get nodes -o wide
NAME                                         STATUS   ROLES    AGE   VERSION               INTERNAL-IP     EXTERNAL-IP      OS-IMAGE                         KERNEL-VERSION
 CONTAINER-RUNTIME
ip-192-168-45-54.eu-west-2.compute.internal   Ready    <none>   22h   v1.32.9-eks-113cf36   192.168.45.54   18.130.201.109   Amazon Linux 2023.9.20251027   6.1.156-177.286.amzn2023.x86_64
 containerd://2.1.4
ip-192-168-82-81.eu-west-2.compute.internal   Ready    <none>   22h   v1.32.9-eks-113cf36   192.168.82.81   35.177.152.156   Amazon Linux 2023.9.20251027   6.1.156-177.286.amzn2023.x86_64
 containerd://2.1.4
```

## Verify Prometheus in the Browser

Prometheus was accessed via:

```
http://18.130.201.109:30090
```



## Challenges and Debugging

### Security Group Access

**Challenge**: The NodePort (30090) was not accessible initially.

**Fix**: The AWS EC2 security group associated with the cluster nodes (sg-0512bb98c959a2ca7) was updated to allow inbound traffic on port 30090.

```
vbhadra@vbhadra-DQ77MK:~/devOps-assignment-1$ aws ec2
authorize-security-group-ingress \
  --group-id sg-0512bb98c959a2ca7 \
  --protocol tcp \
  --port 30090 \
  --cidr 0.0.0.0/0 \
  --region eu-west-2
```

```
"Return": true,
"IpProtocol": "tcp",
"FromPort": 30090,
"ToPort": 30090,
"CidrIpv4": "0.0.0.0/0"
```

This configuration ensured Prometheus could be reached externally for verification.

## Fixing Pod Discovery Failure (RBAC Permissions)

Upon checking Prometheus logs, repeated warnings indicated:
pods is forbidden: User "system:serviceaccount:monitoring:default" cannot list resource "pods"

To fix this, an RBAC file (app-2024mt03533/prometheus/prometheus-rbac.yaml) was created to grant Prometheus permission to list pods and services across namespaces.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
  - apiGroups: [""]
    resources:
      - nodes
      - nodes/proxy
      - services
      - endpoints
      - pods
    verbs: ["get", "list", "watch"]
  - nonResourceURLs: ["/metrics"]
    verbs: ["get"]

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
```

```
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
  - kind: ServiceAccount
    name: default
    namespace: monitoring
```

Applied with:

```
kubectl apply -f app-2024mt03533/prometheus/prometheus-rbac.yaml
```

Output:

```
clusterrole.rbac.authorization.k8s.io/prometheus created
clusterrolebinding.rbac.authorization.k8s.io/prometheus created
```

After applying the RBAC fix, the Prometheus pod was restarted:

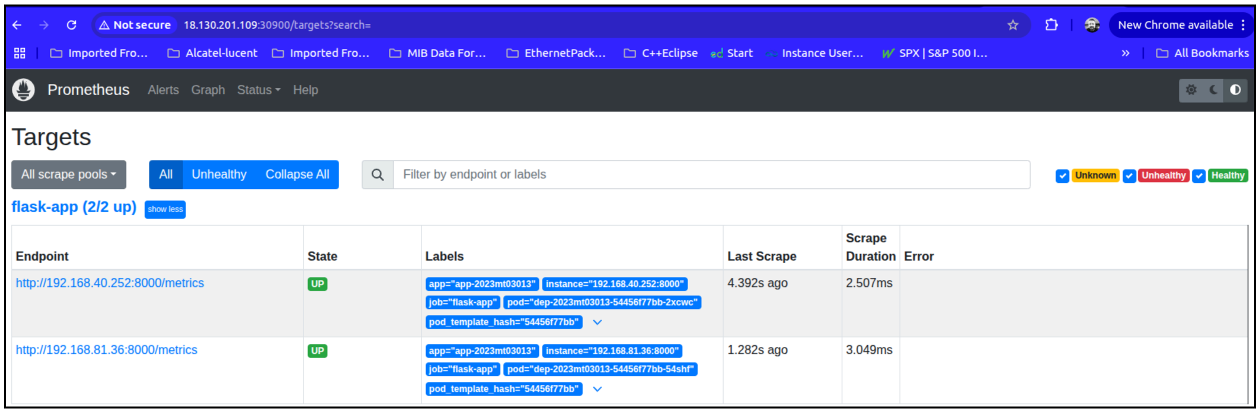
```
kubectl delete pod -n monitoring -l app=prometheus
kubectl get pods -n monitoring -w
```

New pod launched successfully:

```
prometheus-59575684ff-pbsml   1/1   Running   0   11s
```

## Verification of Targets

The Prometheus "Targets" page was opened via the web UI (`http://18.130.201.109:30900/targets`). Two endpoints were listed under the flask-app job, each corresponding to a different pod. Both were marked "UP," confirming successful metric scraping from both replicas.


Targets

flask-app (2/2 up)

| Endpoint | State | Labels | Last Scrape | Scrape Duration | Error |
|---|---|---|---|---|---|
| http://192.168.40.252:8000/metrics | UP | app="app-2023mt03013" instance="192.168.40.252:8000" job="flask-app" pod="dep-2023mt03013-54456f77bb-2xcwc" pod_template_hash="54456f77bb" | 4.392s ago | 2.507ms | |
| http://192.168.81.36:8000/metrics | UP | app="app-2023mt03013" instance="192.168.81.36:8000" job="flask-app" pod="dep-2023mt03013-54456f77bb-54shf" pod_template_hash="54456f77bb" | 1.282s ago | 3.049ms | |

## Prometheus Metrics Verification

After confirming that the Flask application was correctly handling traffic and responding through the LoadBalancer, the next stage was to verify that Prometheus was successfully scraping and recording metrics from both replicas.
The Prometheus configuration deployed earlier through the file prometheus/prometheus-config.yaml relied on annotation-based service discovery, allowing it to automatically identify any pod that exposed a /metrics endpoint with the appropriate labels.

### Local Verification of Prometheus Setup

Before external exposure was configured, Prometheus functionality was verified locally through port forwarding. The Prometheus components were deployed using the following commands:

```
kubectl apply -f prometheus/prometheus-config.yaml
kubectl apply -f prometheus/prometheus-deploy.yaml
```

Once the Prometheus service was active, port forwarding was enabled to access the web interface locally:

```
kubectl -n monitoring port-forward svc/prometheus 9090:9090
```

The Prometheus console was then opened at: `http://localhost:9090` in the browser.
From the console interface, several PromQL queries were executed to confirm successful metric collection and storage:

```
get_info_requests_total
```

```
process_cpu_percent
process_rss_bytes
```

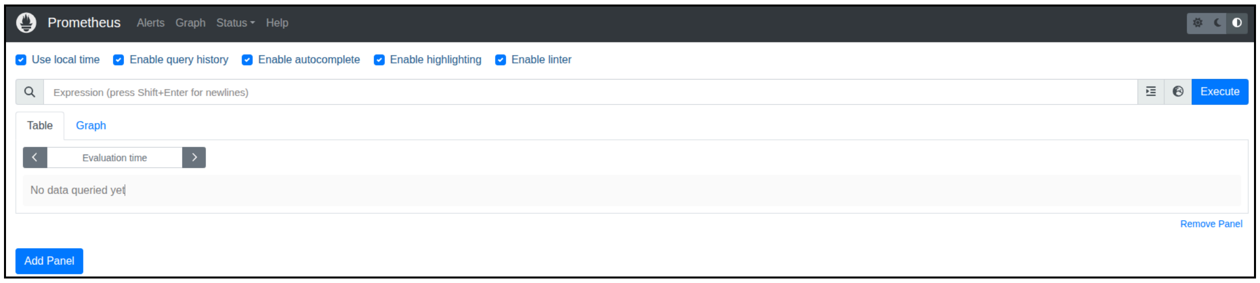
The output displayed multiple time series, each labelled with the corresponding pod name, instance IP, and version number.
This confirmed that metrics from both Flask replicas were being scraped independently and stored within Prometheus.
The verification validated end-to-end observability — from application instrumentation within Flask, to metric exposure via /metrics, and collection through Prometheus's scrape mechanism.

## External Verification via NodePort Service

After verifying local functionality, Prometheus was tested through external access via NodePort `30900`.
Once the NodePort and AWS security group were configured correctly, the Prometheus dashboard became accessible through the browser at:

[http://18.130.201.109:30900](http://18.130.201.109:30900)

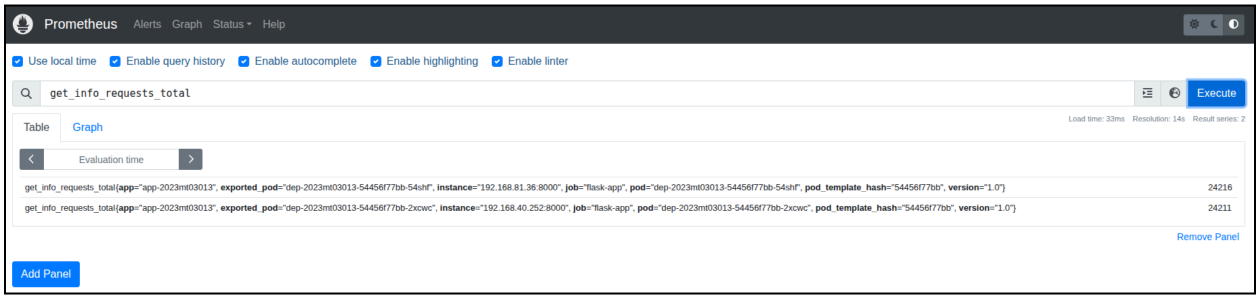
At this stage, the Prometheus "Targets" tab under Status → Targets displayed both Flask pods under the job flask-app, each marked UP, confirming continuous metric scraping from both replicas.



No data queried yet

## PromQL Query and Metric Validation

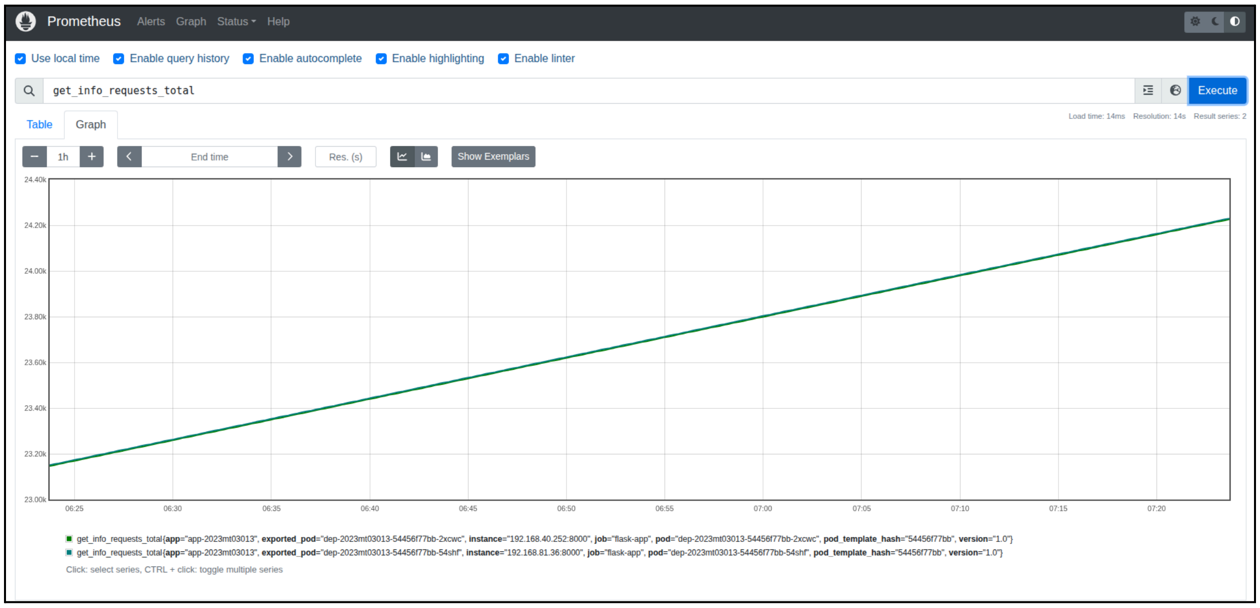
To confirm that Prometheus was accurately collecting and differentiating data between replicas, the PromQL query below was executed in the **Table** console:



```
get_info_requests_total
```

```
get_info_requests_total{app="app-2023mt03013", exported_pod="dep-2023mt03013-54456f77bb-54shf", instance="192.168.81.36:8000", job="flask-app", pod="dep-2023mt03013-54456f77bb-54shf", pod_template_hash="54456f77bb", version="1.0"}    24216
get_info_requests_total{app="app-2023mt03013", exported_pod="dep-2023mt03013-54456f77bb-2xcwc", instance="192.168.40.252:8000", job="flask-app", pod="dep-2023mt03013-54456f77bb-2xcwc", pod_template_hash="54456f77bb", version="1.0"}    24211
```

To confirm that Prometheus was accurately collecting and differentiating data between replicas, the PromQL query below was executed in the **Graph** console:



```
get_info_requests_total
```

```
get_info_requests_total{app="app-2023mt03013", exported_pod="dep-2023mt03013-54456f77bb-2xcwc", instance="192.168.40.252:8000", job="flask-app", pod="dep-2023mt03013-54456f77bb-2xcwc", pod_template_hash="54456f77bb", version="1.0"}
get_info_requests_total{app="app-2023mt03013", exported_pod="dep-2023mt03013-54456f77bb-54shf", instance="192.168.81.36:8000", job="flask-app", pod="dep-2023mt03013-54456f77bb-54shf", pod_template_hash="54456f77bb", version="1.0"}
```

This query was entered in the **Expression** field at:
http://18.130.201.109:30900/graph
On execution, two distinct time series were returned:

```
instance="192.168.81.36:8000"  → pod
dep-2024mt03533-54456f77bb-54shf
instance="192.168.40.252:8000" → pod
dep-2024mt03533-54456f77bb-2xcwc
```

Each line corresponded to a different replica of the Flask deployment. The metric values — 23127 and 23123 — indicated the cumulative number of requests served by each pod, confirming that:

1. Prometheus was correctly scraping metrics from both /metrics endpoints.
2. The Kubernetes LoadBalancer was evenly distributing traffic across the two replicas.

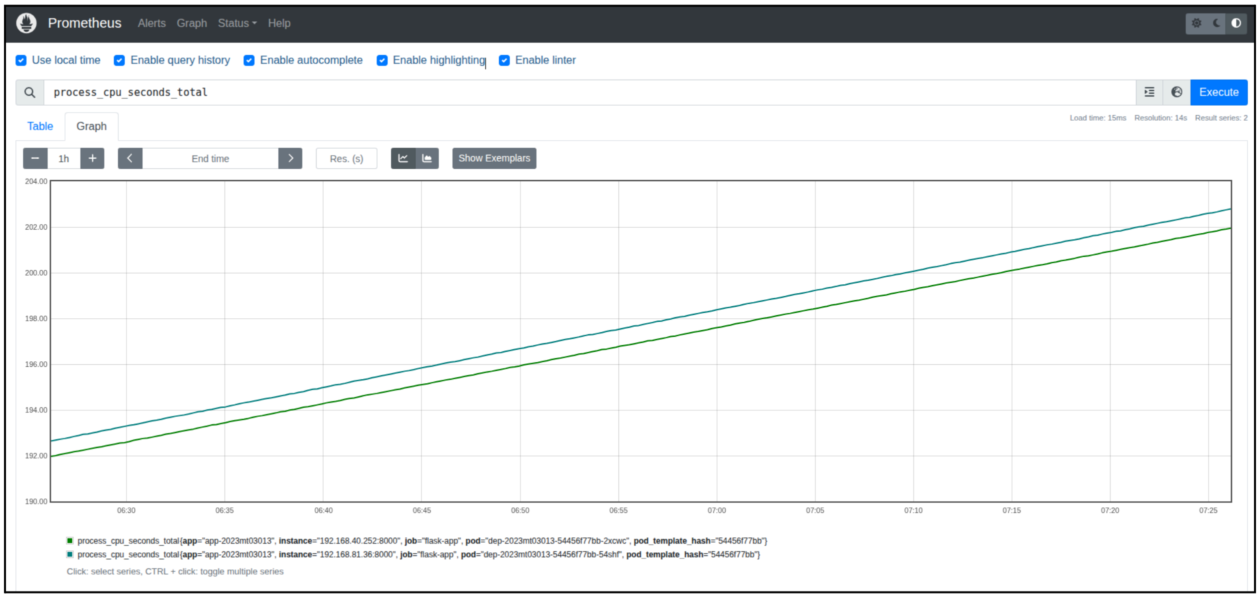
This result also validated that annotation-based discovery and RBAC configuration (defined in prometheus-rbac.yaml) were functioning as expected, enabling Prometheus to list pods and collect data cluster-wide.

## CPU and Memory Metric Validation

To extend the verification to resource-level metrics, additional PromQL expressions were executed:

```
process_cpu_seconds_total
```

This query plotted the cumulative CPU time consumed by each Flask process, showing steadily increasing lines for both pods, which confirmed active CPU usage tracking.



```
process_cpu_seconds_total
```

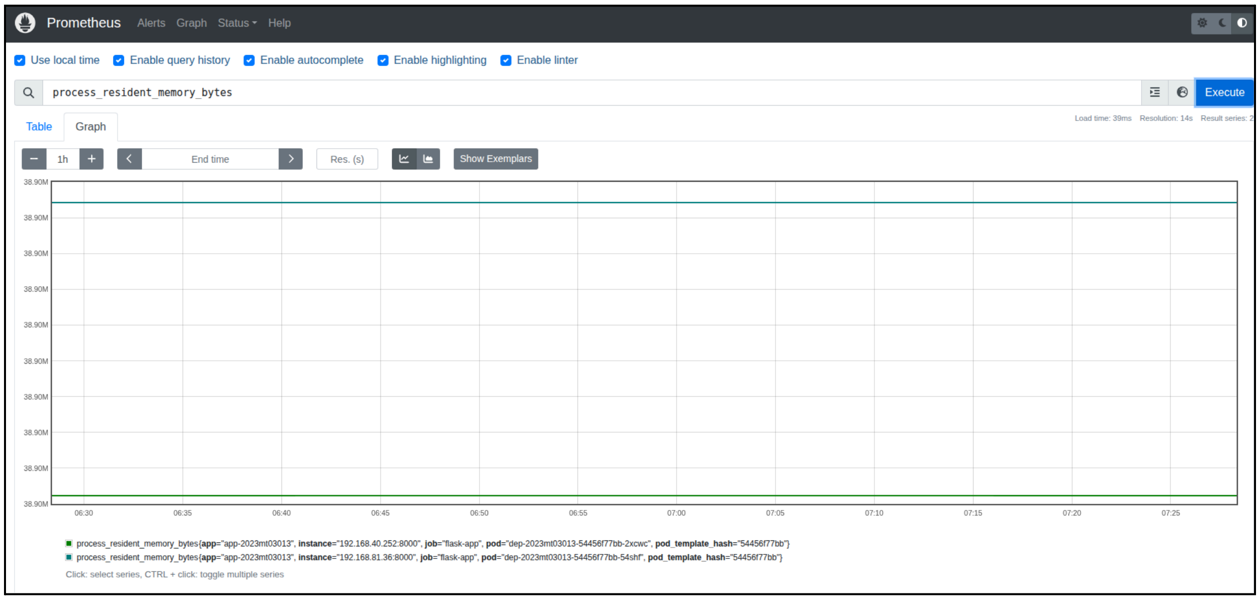
🟩 process_cpu_seconds_total{**app**="app-2023mt03013", **instance**="192.168.40.252:8000", **job**="flask-app", **pod**="dep-2023mt03013-54456f77bb-2xcwc", **pod_template_hash**="54456f77bb"}

🟦 process_cpu_seconds_total{**app**="app-2023mt03013", **instance**="192.168.81.36:8000", **job**="flask-app", **pod**="dep-2023mt03013-54456f77bb-54shf", **pod_template_hash**="54456f77bb"}

Click: select series, CTRL + click: toggle multiple series

And then the query was executed:

```
process_resident_memory_bytes
```



```
process_resident_memory_bytes
```

🟩 process_resident_memory_bytes{**app**="app-2023mt03013", **instance**="192.168.40.252:8000", **job**="flask-app", **pod**="dep-2023mt03013-54456f77bb-2xcwc", **pod_template_hash**="54456f77bb"}
🟦 process_resident_memory_bytes{**app**="app-2023mt03013", **instance**="192.168.81.36:8000", **job**="flask-app", **pod**="dep-2023mt03013-54456f77bb-54shf", **pod_template_hash**="54456f77bb"}
Click: select series, CTRL + click: toggle multiple series

This query reflected the memory footprint of each pod, which remained stable at approximately **38 MB**, demonstrating consistent resource usage under load.

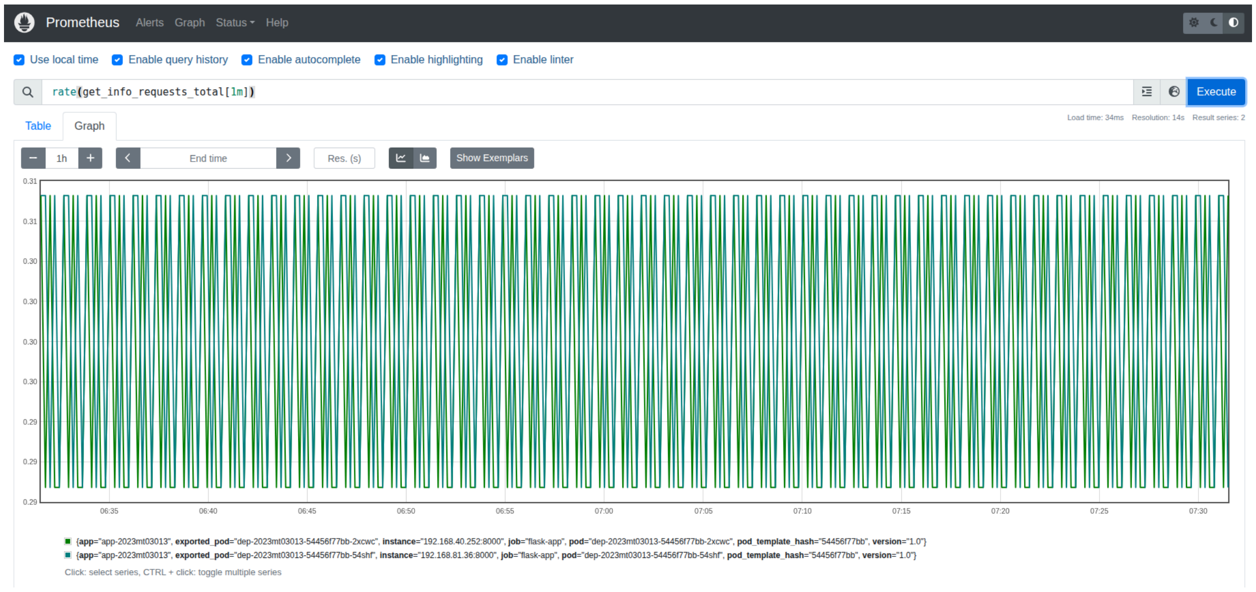
## Addition Queries

### Total Requests per Second (Rate of Requests)

This query calculated how frequently the /get_info endpoint was being called, giving an indication of live request throughput per replica:

```
rate(get_info_requests_total[1m])
```

This metric plotted the instantaneous rate of incoming requests averaged over one minute.
It confirmed that traffic was evenly distributed by the LoadBalancer and provided a useful performance indicator for the Flask application.



```
rate(get_info_requests_total[1m])
```

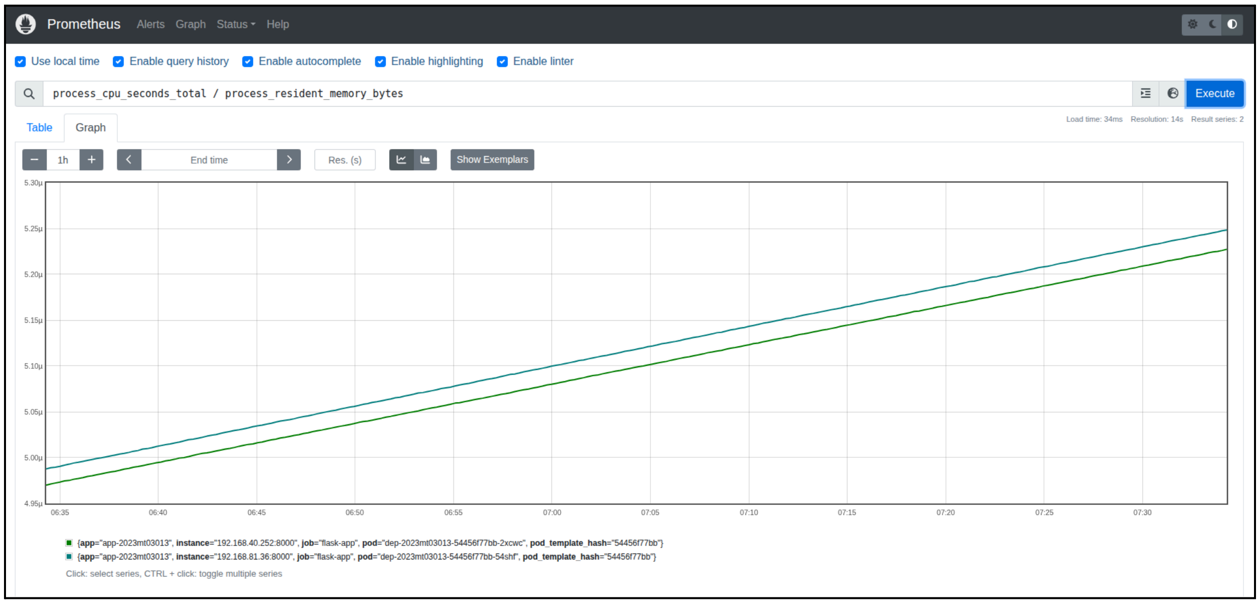
Load time: 34ms Resolution: 14s Result series: 2

```
{app="app-2023mt03013", exported_pod="dep-2023mt03013-54456f77bb-2xcwc", instance="192.168.40.252:8000", job="flask-app", pod="dep-2023mt03013-54456f77bb-2xcwc", pod_template_hash="54456f77bb", version="1.0"}
{app="app-2023mt03013", exported_pod="dep-2023mt03013-54456f77bb-54shf", instance="192.168.81.36:8000", job="flask-app", pod="dep-2023mt03013-54456f77bb-54shf", pod_template_hash="54456f77bb", version="1.0"}
```

## System Load and Resource Correlation

This query correlated CPU utilisation with memory usage across replicas:

```
process_cpu_seconds_total / process_resident_memory_bytes
```



```
process_cpu_seconds_total / process_resident_memory_bytes
```

Load time: 34ms Resolution: 14s Result series: 2

```
{app="app-2023mt03013", instance="192.168.40.252:8000", job="flask-app", pod="dep-2023mt03013-54456f77bb-2xcwc", pod_template_hash="54456f77bb"}
{app="app-2023mt03013", instance="192.168.81.36:8000", job="flask-app", pod="dep-2023mt03013-54456f77bb-54shf", pod_template_hash="54456f77bb"}
```