Project Report on

“Implementing C and Python grading functionality in serverless Architecture ”

Submitted by :
Anshika Saxena
(23210018)

Under the guidance of
Prof. Sameer G Kulkarni

Parallel And Distributed Systems
Department of Computer Science and Engineering
IIT Gandhinagar.

23/April/2024

Abstract

Traditional compilation methods for C and Python code often rely on stateful servers, resulting in challenges such as limited scalability, inefficient resource utilization, and complex operational management. To address these issues, this project aims to evaluate the efficacy of serverless computing for code compilation tasks in both C and Python languages. By implementing two distinct serverless functions—one for compiling C code and another for compiling Python code—we seek to demonstrate the advantages of serverless architectures over traditional stateful server methods. Through rigorous performance evaluation, including analysis of throughput, latency, scalability, and resource utilization, our objective is to showcase the superior efficiency and operational simplicity offered by serverless computing for compilation tasks, thereby highlighting its superiority over stateful server approaches.

Table of Contents

"Why choose serverless?"

Serverless–Serverless architecture is a way to build and run applications and services without managing infrastructure.

The decision to adopt a serverless approach for this project stems from several key advantages offered by serverless computing:

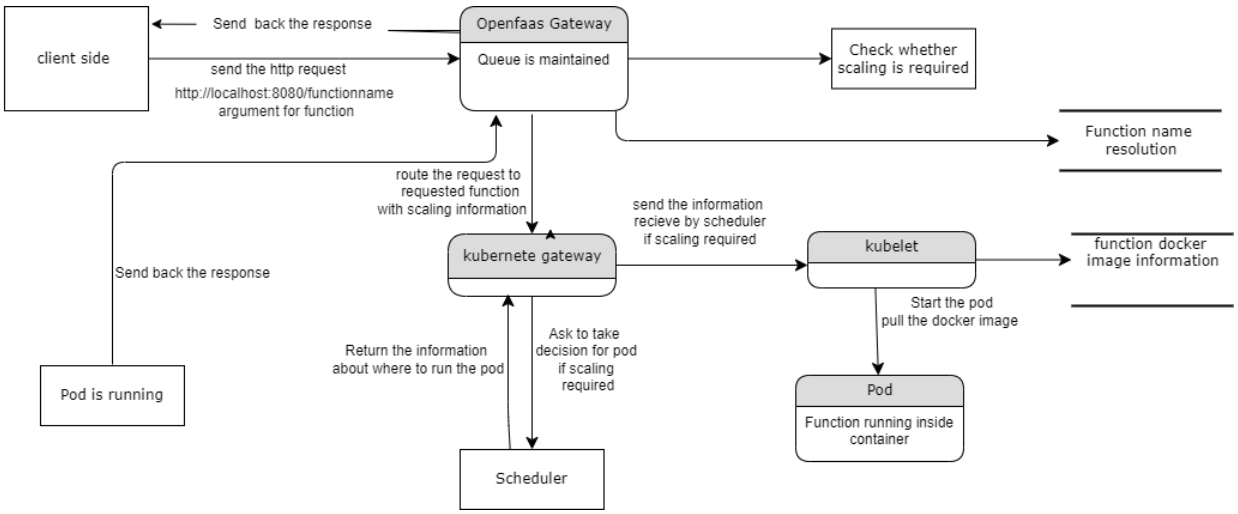
1. Scalability: Serverless functions scale automatically in response to incoming requests or events. This eliminates the need for manual provisioning or managing of server resources, ensuring that the system can handle varying workloads efficiently.

2. Cost-Efficiency: With serverless computing, users only pay for the compute resources consumed during function execution, typically measured in milliseconds. This pay-as-you-go pricing model eliminates the overhead costs of maintaining idle server instances, making it a cost-effective solution for sporadic workloads. Online compilers are not used frequently, so why reserve the resources for this and pay for the idle state also, instead of that, we can ask for the resources when we needed.

3. Operational Simplicity: Serverless architectures simplify operational tasks such as deployment, monitoring, and scaling. Developers can focus on writing code without worrying about managing server infrastructure, operating systems, or runtime environments. The main task is compiling the code and sending the response accordingly. Why does the developer care about maintaining the state?

4 . Flexibility: Since application logic is encapsulated within individual functions, the addition of new functionalities, such as integrating a Java compiler or incorporating the current logic into another application becomes straightforward. It's merely a matter of creating a new function and the gateway will seamlessly manage its integration.

Architecture Diagram



Tools used in the project

1. Docker Desktop – Where my Kubernetes cluster is going to run.

2. Minikube - Minikube provides a local Kubernetes cluster for development

3. Kubectl- It is the command-line tool for interacting with Kubernetes clusters.
4. Helm – Helm is a package manager for Kubernetes that simplifies the process of deploying, managing, and upgrading complex applications on Kubernetes clusters

5. Openfaas Cli – It is used to interact with the Openfaas platform.
6. Apache Benchmark - To send multiple requests at the same time to the server. For Testing Purpose.

"Steps involved in setting up Kubernetes with OpenFaaS on my local machine:"

1. Installed Docker Desktop and logged in. Then, accessed settings and enabled Kubernetes. Once Kubernetes is enabled, Docker Desktop will automatically set up and manage a single-node Kubernetes cluster .

2. Install the minikube and open the terminal write

   minikube start

The above command starts a local Kubernetes cluster on the docker desktop

```
PS C:\kubeless> minikube start
W0423 08:30:28.994708    5192 main.go:291] Unable to resolve the current Docker CLI context "default": context "default": context not found: open C:\Users\a
nshi\.docker\contexts\meta\37a8eec1ce19687d132fe29051dca629d164e2c4958ba141d5f4133a33f0688f\meta.json: The system cannot find the path specified.
* minikube v1.32.0 on Microsoft Windows 11 Home Single Language 10.0.22631.3296 Build 22631.3296
* minikube 1.33.0 is available! Download it: https://github.com/kubernetes/minikube/releases/tag/v1.33.0
* To disable this notice, run: 'minikube config set WantUpdateNotification false'

* Using the docker driver based on existing profile
* Starting control plane node minikube in cluster minikube
* Pulling base image ...
* Updating the running docker "minikube" container ...
* Preparing Kubernetes v1.28.3 on Docker 24.0.7 ...
* Configuring bridge CNI (Container Networking Interface) ...
* Verifying Kubernetes components...
  - Using image registry.k8s.io/metrics-server/metrics-server:v0.6.4
  - Using image docker.io/kubernetesui/dashboard:v2.7.0
  - Using image docker.io/kubernetesui/metrics-scraper:v1.0.8
  - Using image gcr.io/k8s-minikube/storage-provisioner:v5
* Some dashboard features require the metrics-server addon. To enable all features please run:

        minikube addons enable metrics-server


* Enabled addons: storage-provisioner, default-storageclass, metrics-server, dashboard
* Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
PS C:\kubeless>
```

3. Install the kubectl

4. Install the open-faas CLI by downloading the binary

   https://github.com/openfaas/faas-cli/releases

5. Download helm

   After downloading the helm , Add the OpenFaaS chart repository and deploy OpenFaaS by running the following commands:

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

   helm repo update

6. Create namespace openfaas-fn using kubectl

   kubectl create ns openfaas-fn

7. Install openfaas using helm

   helm upgrade --install openfaas openfaas/openfaas--namespace openfaas --set basic_auth=true

8. Now the deployment is complete i can access the openfaas gateway using port forwarding

9. kubectl port-forward svc/gateway -n openfaas 8080:8080

   This command forwards traffic from your local machine's port 8080 to the OpenFaaS gateway service running in the Kubernetes cluster.

10. We can access the openfaas gateway ui by login. Open a web browser and navigate to http://localhost:8080 it will ask for the username and password

Username - admin and to retrieve the password, we will run the below command on the terminal and get the password

kubectl -n openfaas get secret basic-auth -o jsonpath="{.data.basic-auth-password}" | ForEach-Object { [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($_)) }

Openfaas have two namespaces

- `openfaas` - core components like gateway, alertmanager
- `openfaas-fn` - developer deployed functions

Now Openfaas set up is ready, and now we can deploy the serverless functions.

Application Logic of Functions

Openfaas provide some inbuilt template , so for this project i used the Python3 templates

To use the template download the templates on your current directory using

faas-cli template store list

After running above command it will download all the templates provided by openfaas

Run this command to create the function using the template

faas-cli new --lang <language> <function-name>

In language write the template name and function name is anything.

Function 1:--

I created the function Pythontimeout

This function receive the file from client and compile using python compiler and send error if any as response else the output.

In this it will compile for max upto 1 minute if still it is not getting it will send the Time LImit Exceeded Error .

The given below is yml file in that i changed the gateway to localhost and port 8080 and in image i add my docker registry name which is hkranshi.

```
version: 1.0
provider:
  name: openfaas
  gateway: http://localhost:8080
functions:
  pythontimeout:
    lang: python3
    handler: ./pythontimeout
    image: hkranshi/pythontimeout:latest
```

And handler.py code contain main business logic

```
import sys
import signal
import subprocess
import os

def handle(req):
    try:
        # Write the received text to a temporary Python file
        with open("/tmp/temp_script.py", "w") as f:
            f.write(req)

        def timeout_handler(signum, frame):
```

```
            raise TimeoutError("Execution time exceeded 20 seconds")

        # Set a signal alarm to trigger after 20 seconds
        signal.signal(signal.SIGALRM, timeout_handler)
        signal.alarm(60)  # 20 seconds timeout

        # Execute the temporary Python file using python3 command and capture output
        process = subprocess.Popen(['python3', '/tmp/temp_script.py'],
stdout=subprocess.PIPE, stderr=subprocess.PIPE)
        output, error = process.communicate()

        # Disable the alarm after execution
        signal.alarm(0)

        if process.returncode == 0:
            # Encode output with 'latin1' and handle errors gracefully
            encoded_output = output.decode('latin1', errors='replace').encode('utf-8',
errors='replace')
            return encoded_output.decode('utf-8')  # Compilation successful, return output
        else:
            # Encode error with 'latin1' and handle errors gracefully
            encoded_error = error.decode('latin1', errors='replace').encode('utf-8',
errors='replace')
            return f"Error during compilation: {encoded_error.decode('utf-8')}"
    except TimeoutError as te:
        return f"Error: {te}"
```

```
    except Exception as e:
        return f"Error during compilation: {e}"
```

After edit both the yml and handler.py file run the below command

`faas-cli up -f <yaml-file>`

This command contain the three things

1. It will build the docker image based on the configuration provided

   Building a serverless function involves preparing the code and its dependencies to be packaged into a Docker image.

   The output of the build process is a Docker image containing the function code and dependencies, ready to be deployed
2. Pulling a Docker image involves retrieving the pre-built container image from a container registry.
3. Deploy:
   a. Deploying serverless functions involves instructing the OpenFaaS platform to create instances of the functions using the Docker images.
   b. During the deployment process, the OpenFaaS platform pulls the necessary Docker images (if not already available locally), creates Kubernetes resources (such as deployments and services) to manage the function instances, and configures routing for incoming requests to the deployed functions.
   c. Deploying functions may also involve specifying configuration parameters such as function names, HTTP triggers, environment variables, and resource limits.

d. Once deployed, the serverless functions are ready to receive and process requests from clients or other services according to their defined triggers and endpoints.

After this function is ready to serve the requests.

Command to invoke the function is –**curl -X POST -H "Content-Type: text/plain" --data-binary @input.py** http://localhost:8080/function/pythontimeout**.**

Function 2–

This function will compile the c code so compiling the c code we need to setup the gcc compiler in the container image .

To setup the gcc container what i did i edit the python3 template file and add the below command

```
# Install C compiler and development libraries
RUN apk --no-cache add gcc libc-dev
```

It will install the gcc compiler in the docker image

After this i created the new function cc .

Business logic for cc function –

```
import subprocess
import os

def handle(req):
    # Extract the C code from the request
    c_code = req

    # Write the C code to a temporary file
    with open('/tmp/code.c', 'w') as f:
        f.write(c_code)
```

```
    # Compile the C code
    try:
        # Compile and generate the binary file within 30 seconds
        subprocess.run(['timeout', '30s', 'gcc', '/tmp/code.c',
'-o', '/tmp/compiled'], check=True)

        # Execute the binary file within 30 seconds
        result = subprocess.run(['timeout', '30s',
'/tmp/compiled'], capture_output=True, text=True, check=True)

        # Get the output of the executed binary file
        output = result.stdout
    except subprocess.CalledProcessError as e:
        output = f"Compilation failed: {e.stderr}\n"

    # Return the compilation result
    return output
```

After implementing this, I deploy this function using **faas-cli up if cc.yml** command

Now i client can access both the function through http request.

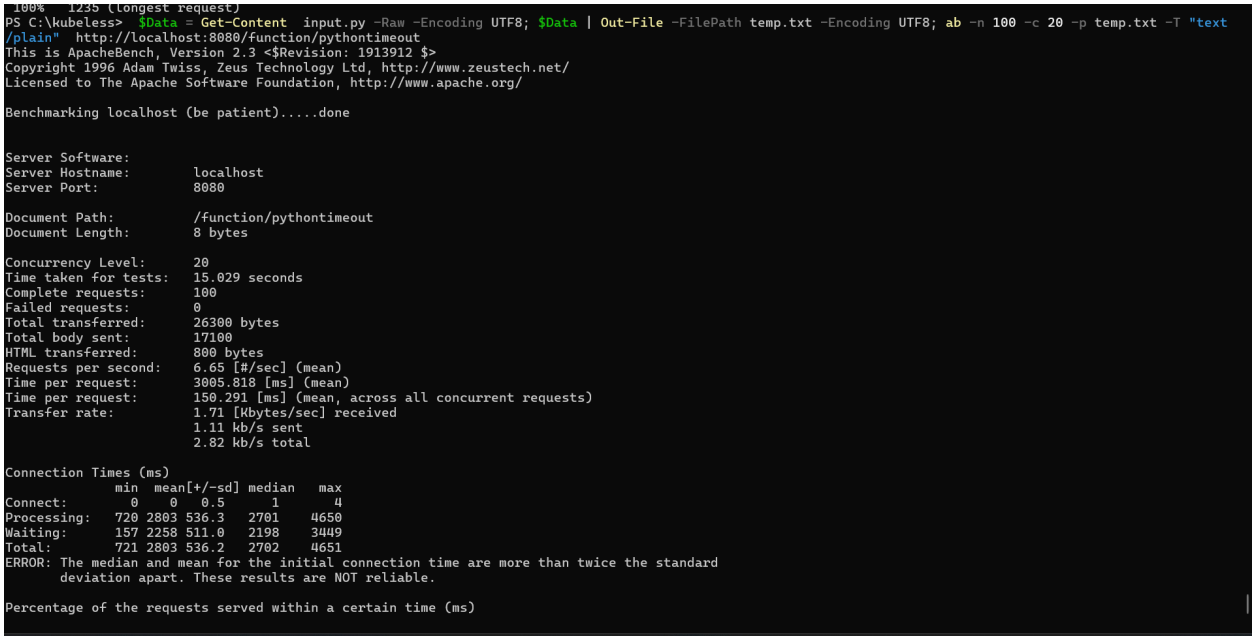
Performance Metrics of the function

To check the performance like how the server will behave if multiple request will come at the same time

To measure the performance i used apache benchmark which will give the everything

Command – $Data = Get-Content input.py -Raw -Encoding UTF8; $Data | Out-File -FilePath temp.txt -Encoding UTF8; ab -n 100 -c 20 -p temp.txt -T "text/plain" http://localhost:8080/function/pythontimeout

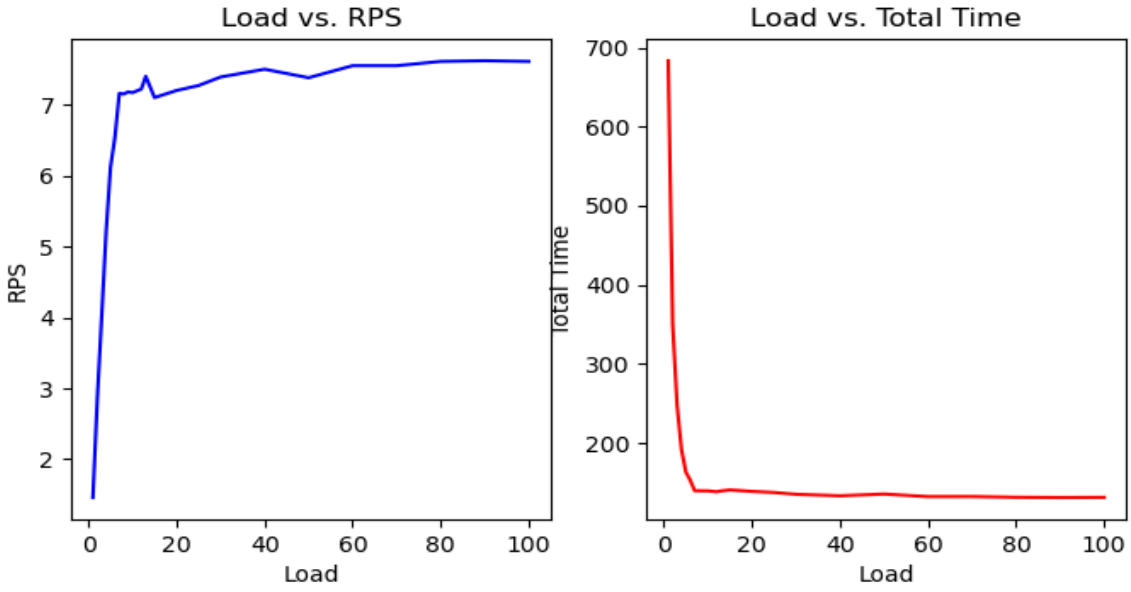
Where -n represent the total number of requests and -c represent the load means at a time how many request simultaneously go to server



To check whether the server responds correctly or not we can use ‘ -v 2 ‘ flag

Command – $Data = Get-Content input.py -Raw -Encoding UTF8; $Data | Out-File -FilePath temp.txt -Encoding UTF8; ab -n 1000 -c 200 -p temp.txt -T "text/plain" -v 2 http://localhost:8080/function/pythontimeout

It will tell the individual time taken by each request, and using this, I have plotted the throughput and latency graph



To get the cpu utilization and other resource utilization by pod

We can run — kubectl top pods -n openfaas-fn

The future outcome of this project

Online Compilers : Extend the project to support additional programming languages beyond C and Python. By adding support for languages such as Java, Go, or Rust, the platform can cater to a wider range of developers and use cases.

Online Coding Environment: Offer an online coding environment where users can write, compile, and run code in various programming languages directly within a web browser. Users can write code in the provided editor, submit it to the serverless functions for compilation, and view the output or errors in real-time.

Interactive Learning: Create interactive coding tutorials and exercises where users can practice coding concepts and algorithms. Users can write code to solve problems, submit it for compilation, and receive immediate feedback on their solutions.