Serverless Web Application: To-Do List

1 Introduction

In this project, we have reimplemented an AWS serverless web application[1], "To-Do List". The key features of this application include user authentication, task management (addition and deletion), as well as image uploading and recognition.

We replaced AWS components with robust open-source alternatives to provide a fully-functional replica of the original application. The replacements included using MySQL in place of DynamoDB, employing Flask framework as a serverless function with OpenFaaS as a substitute for AWS Lambda, switching from S3 to MinIO, employing OpenFaaS Gateway as an alternative to the API Gateway, and utilizing TensorFlow serving instead of Rekognition.

The end result is a versatile application, deployable directly onto a Kubernetes cluster as a set of Docker images. We provide detailed documentation for configuring the environment and deploying applications on an AWS EC2 instance. Our development scope was all-inclusive, spanning the front-end, back-end, database, storage systems, and machine learning systems. We not only maintained the application's core functionalities but also harnessed the adaptability and control inherent to open-source alternatives.

The structure of the report is as follows: In Section 2, we will discuss the overall structure and functionality of our program. In Section 3, we'll introduce the backend services developed using the Flask framework and managed by OpenFaaS. The Section 4 will cover the integration of different parts of the system such as image storage, image classification, and data storage. Section 5 will guide you through the process of deploying this application onto a Kubernetes cluster. Finally, the last chapter will wrap up the key points discussed throughout the article.

2 Software Design

The project is deployed to a single AWS EC2 instance running the minikube Kubernetes cluster. Figure 1 illustrates the fundamental functionalities of this program. Users can log in with a specific account. Once logged in, they can browse, add, and delete to-do items. For each to-do item, users can upload a picture and obtain its labels.

Figure 2 displays the architecture of our project. Users interact with the frontend to access services, which is developed using the Vue.js framework. This frontend invokes the public APIs of the OpenFaaS backend to deliver the services. The backend is developed using the Flask framework and is managed by OpenFaaS in the form of

[1] https://github.com/aws-samples/serverless-tasks-webapp

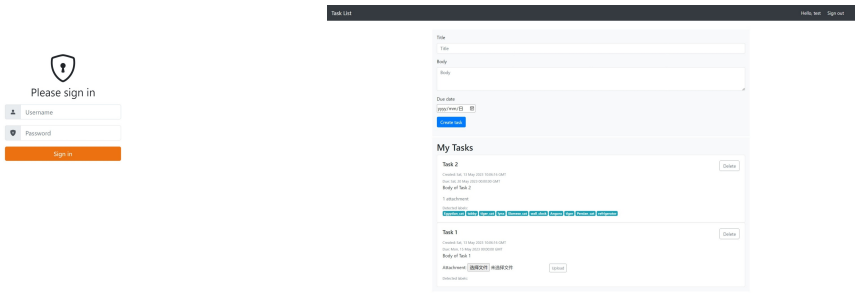

Figure 1: The Frontend UI

Docker containers. Images uploaded by users are stored in the MinIO bucket, and the backend retrieves image labels by calling the image classification model deployed via TensorFlow Serving. Both user and to-do list data are stored in a MySQL database.

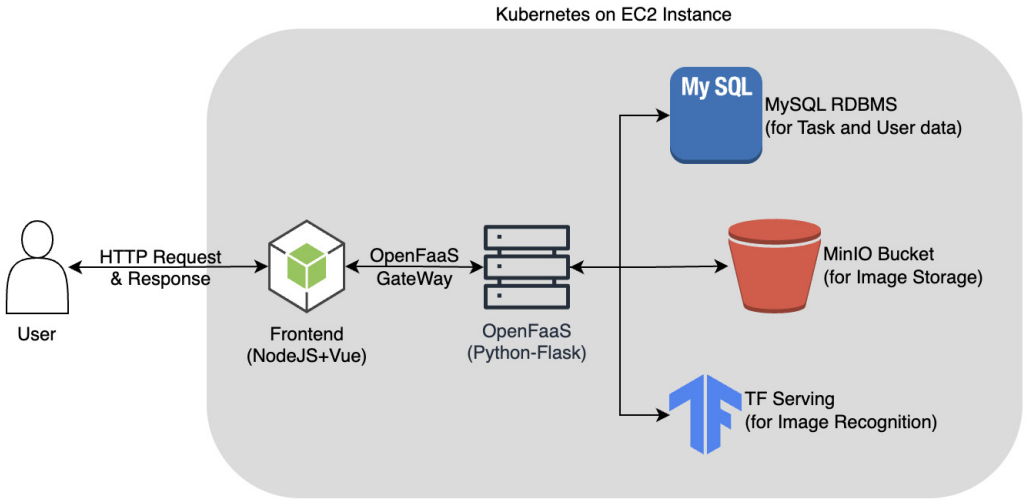

Figure 2: The application architecture

3 Backend

3.1 Preliminary: FaaS and OpenFaas

Function-as-a-Service (FaaS) is a serverless model that allows developers to build, execute, and manage application functionalities without the complexities of infrastructure typically associated with microservices applications. OpenFaaS is an open-source framework that enables the implementation of serverless architecture on Kubernetes, utilizing Docker containers for function storage and execution. It simplifies the deployment of event-driven functions and microservices on Kubernetes, eliminating the need for repetitive coding. By packaging code or binaries into Docker images, developers can achieve highly scalable endpoints with auto-scaling and metrics.

3.2 Reimplementing the Backend with OpenFaaS and Flask

In this project, we replaced AWS Lambda with OpenFaaS to implement a serverless architecture for the backend. First, we re-implemented all backend functionalities

using the Flask framework, including user authentication, to-do list management, and image recognition. Flask is a Python micro web framework, allowing us to build these services with greater flexibility and control. Then, we packaged the backend service into a Docker container and deployed it through OpenFaaS via configuration. Considering that the backend relies on the services provided by MinIO, MySQL, and TensorFlow Serving, we needed to add corresponding environment variables for access paths in the configuration file.

By following this process, we were able to successfully transition from AWS Lambda to OpenFaaS, while maintaining a flexible backend thanks to Python and Flask.

We also utilized Arkade, an open-source CLI to deploy OpenFaaS. Arkade simplifies the management of applications by leveraging the capabilities of Kubernetes. It offers a convenient interface with strongly-typed flags, enabling straightforward configuration of popular options available in helm charts.

3.3 Backend API

login

This function is designed to handle a POST request. The function utilizes the Flask framework's **@app.route** decorator to specify the route for this particular endpoint. It executes a SQL query to extract the username and password from the request, and checks if the provided credentials match those stored in the database. Then it generates an authentication token using the JWT library, containing the username, current time, and expiration time. Finally, the fhttps://www.overleaf.com/project/645f63eacb7962e836dfd344unction updates the token for the user in the database, commits the changes to the database and returns the generated token as the response.

create_task

This function handles a POST request made to the "/tasks" endpoint in an application. It is decorated with the **@auth** decorator, authentication is required to access this endpoint. When a POST request is made to this endpoint, it executes a SQL query using the **CURSOR** object to insert a new task into a table named "Task". The query is constructed using string interpolation to include the title, body, and due date values retrieved from the request. The changes made to the database are committed using **CONN.commit()** to ensure persistence. If successful, the function returns a message indicating that the data was inserted successfully.

delete_task

The delete_task() function handles a DELETE request in an application. Authentication is required to access this endpoint. When a delete request is received, it retrieves the upload value from the database by executing a SELECT query based on the provided task_id. If the upload value exists, it uses the MINIO_CLIENT library to remove corresponding objects from a bucket in a cloud storage service, and a success message is returned to the client.

get_tasks

The function handles a GET request made to the "/tasks" endpoint in an application. It retrieves all tasks from the database and returns them as a list of dictionaries. The

"labels" value for each task is split into a list. The function returns the list of tasks as the response to the client.

predict_images

The function handles a PUT request. After reading and processing the image data from the request stream, data is uploaded to a cloud storage service using the MINIO_CLIENT library, with the filename and data length. The function calls get_labels() to generate labels for the uploaded image. It executes an UPDATE query on the "Task" table to update the task's labels. The **if __name__ == "__main__"** block starts the Flask application, allowing it to run on the specified host address.

4 Other Components

This chapter primarily discusses the functionalities of the various components apart from the backend, with their deployment to be covered in the following chapter.

4.1 Frontend

Developed with the Vue.js framework and JavaScript, the frontend of our application largely draws its codebase from the original AWS project. It initiates the user experience by acquiring an identity token upon login. This token is subsequently presented with each backend interaction to authenticate user identity.

In response to the evolution of backend service interfaces, we've implemented necessary adjustments. These modifications include changing the base path of the backend services. In the initial project, images were directly stored in S3. However, due to TensorFlow Serving's inability to directly access images from the storage bucket as AWS Rekognition can from S3, uploaded images have to be forwarded via the backend. This shift has led to an essential revision in the logic of the associated functions.

4.2 Database

We have chosen MySQL, a relational database, to replace AWS DynamoDB. Although the latter is a NoSQL database, considering that the data schema we need to store is very simple, using an RDBMS does not pose significant differences. In this project, the database consists of two tables: User and Task. Their schemes are shown in table 1 and 2.

For the Task table, the "upload" attribute indicates whether it has an image attachment. For the User table, the "token" attribute is generated by the backend during login and is written to the database. During authentication, the backend retrieves the corresponding user's token from the database.

4.3 Image Recognition Service

We deploy the EfficientNet V2 model, obtained from TensorFlow Hub[2], using TensorFlow Serving as a replacement for AWS's image recognition service. This powerful model is capable of predicting the probability of an input image belonging to any of the 1000 classes within the ImageNet dataset.

[2]https://tfhub.dev/google/imagenet/efficientnet_v2_imagenet21k_ft1k_s/classification/2

Table 1: Task Table

Column	Data Type	Constraints
id	INT	PRIMARY KEY AUTO_INCREMENT
title	VARCHAR(255)	NOT NULL
body	TEXT	NOT NULL
dueDate	DATE	NOT NULL
createdAt	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP
upload	TINYINT(1)	DEFAULT 0
labels	VARCHAR(255)	DEFAULT ""

Table 2: User Table

Column	Data Type	Constraints
username	VARCHAR(255)	PRIMARY KEY
password	VARCHAR(255)	NOT NULL
token	VARCHAR(255)	

During the image recognition process, the backend forwards the image to TensorFlow Serving through its RESTful API. The model then calculates the probabilities for each class. Finally, the top 10 classes with the highest probabilities are selected as the image's labels.

4.4 Object Storage

We have opted to utilize MinIO, an open-source object storage system, as a substitute for S3 in our project. MinIO offers complete compatibility with all S3 APIs, presenting a seamless integration advantage. Upload images are stored within a designated MinIO bucket. When their corresponding tasks are removed, associated images are also deleted.

MinIO also provides a user-friendly web interface for managing files. This interface is accessible through the public IP of the EC2 instance and port 9090 as shown in fig 3, allowing us to easily navigate and inspect the images uploaded by users.

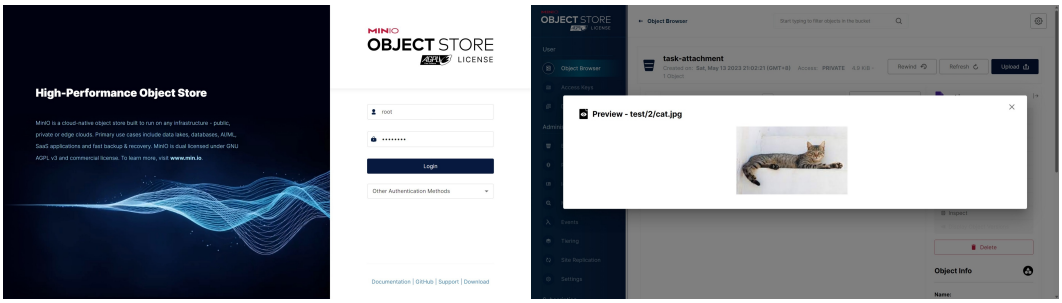

Figure 3: MinIO Management Interface: Log in and View Files

5 Kubernetes Deployment

The functionalities discussed above are encapsulated in to several Docker images, which can be deployed into a Kubernetes cluster. In this section, we discuss more

technical details in addition to the previous sections about the manipulation over the internal and external interactions of the system, primarily in networking context. Simultaneously, we exhibit the key configurations for building and deploying the containers for individual functionalities.

5.1 Web Frontend

Container Construction. The NodeJS-based web service is built sequentially based on two public Docker images, node:14 and node:14-alpine. In the first stage, we state as the base image the node:14, which is based on a Debian-based Linux distribution and includes a more complete set of libraries and tools. By virtue of the rich environment and tooling provided, we can easily install dependencies and run build scripts. After this *build* stage, we adopt node:14-alpine as the base image for the final production stage. This base image is a lightweight and minimal Linux distribution, whose smaller footprint leads to reduced storage requirements and potentially faster container startup times. We then inform Docker through the Dockerfile that the container will start the http server by **CMD ["http-server", "dist"]**, and that it will listen for incoming network connections on port 8080 by **EXPOSE 8080**. In addition, as the frontend is designed to access the OpenFaaS gateway through the public IP address and port number (by default 31112), before building Docker image, we replace in advance the reserved field for a base url (i.e., http://localhost:5000 in /webapp/src/gloopts.js) with the public IP of the host machine and the path of the OpenFaaS service.

Service Configuration. The major issue for the deployment of web frontend module is the networking configuration, which should allow access of the web resources outside the Kubernetes cluster and the host machine through a public IP address. While Kubernetes intrinsically supports the exposure of an application to the public internet by creating an Ingress resource, with this measure we encountered flexibility and compatibility challenges. Therefore, an alternative solution is adopted, which comprises the internal traffic policy and the port-forwarding techniques. Specifically, we apply **ClusterIP** as the type of the service with TCP port 8080, indicating that the service is accessible only from within the cluster. We then use the **port-forward** function from *kubectl*, which sets up a port-forwarding mechanism that maps a local port (80) to the service's port (8080) in the cluster.

5.2 OpenFaaS Gateway

OpenFaaS Installation. The installation of OpenFaaS involves a complex of issues, including:

- checking for any necessary dependencies,
- pulling the required Docker images for OpenFaaS components (e.g., the gateway, queue-worker) core services,
- createing Kubernetes resources to deploy OpenFaaS components (e.g., creating deployments, services, secrets, config map)

Arkade, as a tool designed for installing and managing Kubernetes applications, can automate these installation steps.

Container Construction. Consecutively, we deploy a container that encapsulates the services for handling request from the web frontend. There are two base images,

the python:3.10-slim-buster for the python-flask API implementation, and the of-watchdog:0.9.11 from ghcr.io/openfaas. After settling the environment variable **fprocess** with the main script (i.e., app.py) for the python-flask API, the **fwatchdog** command is configured to run at the the commence of the container, whose process intercepts the request from the web frontend and passes it to the function handler specified in app.py.

Service Configuration. Using Arkade installation, we obtain a base deployment object **deploy/gateway** and a corresponding service object **svc/gateway**. After confirming the ready state of the OpenFaaS through the deployment object, we configure a port forwarding from the local machine port 8080 to the node port 8080 using the service object. As currently we have not materialized an OpeenFaaS service using the customized Docker image, the faas-service.yml indicates the related configurations, where the **gateway** is settled as http://127.0.0.1:8080, referring to the OpenFaaS gateway endpoint on the local machine as we previously discussed. Additionally, the yml specifies the environmental variables indicating the service names of the MySQL, MinIO, and TenserFlow Serving. We use **faas-cli**, which is an encapsulation of the complexity of Kubernetes manifests, to build the image at runtime. The service is finally deployed using **faas-cli deploy** with the configurations in faas-service.yml.

5.3 Backend Services

MySQL. The MySQL service is deployed using the official MySQL Docker image mysql:latest, which is based on a Debian-based Linux distribution and includes all the necessary components and dependencies to run a MySQL database server. In this case, we leverage the convenience and reliability of the official image to ensure a stable and up-to-date MySQL deployment. The MySQL root password is set using the environment variable MYSQL_ROOT_PASSWORD with the value 123456, ensuring the access from the flask service. To persist the data and make it available across container restarts, a volume is mounted to the container using "volumeMounts", named "mysql-data" and its host path is specified as path/mysql, where path is to be replaced by $(pwd). Consequently, by a conventional mounting of the volume mysql-data to the container at the path /Docker-entrypoint-initdb.d, a customized sql file is loaded into that path for database initialization. To enable communication with the MySQL service, a Kubernetes Service object named mysql-service is created and selects pods with the label "app: mysql" to direct traffic to the MySQL deployment. The Service is also of type "ClusterIP" like web frontend, which means it is only accessible within the Kubernetes cluster. It exposes the MySQL service on port 3306, the default port used by MySQL, and targets the container port 3306.

MinIO. In order to provide persistent storage for the Minio deployment, a PersistentVolumeClaim (PVC) is defined, which specifies the access mode as "ReadWriteOnce", meaning it can be mounted by a single node for read and write operations. The PVC requests 1Gi of storage capacity. The container in the deployment uses the official Minio Docker image minio/minio:latest, which is provided with the args as the command to start the Minio server with data directory set to /storage (storing a volume that references the PVC created earlier) and the console port set to 9090. The container then exposes two ports: 9000 and 9090, where 9000 is used for HTTP access to the Minio server and 9090 is used for the Minio console. As the service

is only required from the OpenFaaS service, we set the service type as ClusterIP, making it accessible only within the Kubernetes cluster.

Tensorflow Serving. The deployment template of Tensorflow Serving uses the image tensorflow/serving to start a container, which is the official TensorFlow Serving Docker image. We also define a volume using the path of the preloaded efficientnet model on the host machine, which is mounted into the container to allow the container to access the TensorFlow model. To allow access of the TensorFlow Serving service from outside the container or from other components within the cluster, we expose the port 8501 of the container, making it accessible within the cluster.

6 Conclusion

Our reimplementation of the "To-Do List" application using robust open-source alternatives has proven to be a success, effectively demonstrating the flexibility and control afforded by these technologies. Our approach has allowed us to maintain the core functionalities of the original AWS application while offering the advantage of adaptability inherent to open-source systems. We believe this project serves as a valuable blueprint for those seeking to explore and adopt serverless architectures.