2022 Spring CSIT6000O Course Project: Object Detection on AKS (Canvas Group 10)

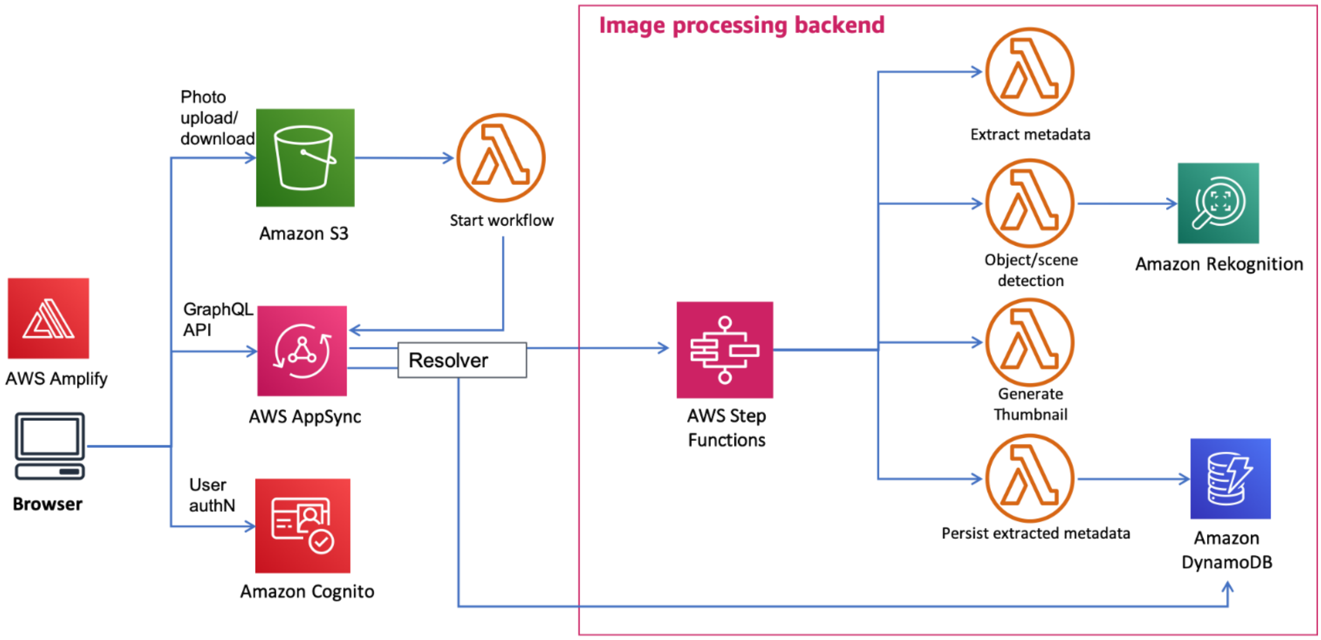
Introduction

In this project, our group reimplements the Image Recognition serverless application from AWS Samples[1]. AWS has provided a reference architecture that received some updates in the past. To summarize, the usage pattern of the application is described as follows:

1) A user arrives at the landing page and sign-in to continue
2) In the next page, the user can create photo albums. Selecting an album will bring the user to the album page.
3) In the album page, user can upload images to AWS S3 object storage for processing.
4) Each image upload will trigger a processing workflow controlled by AWS Step Function in the back. The image file and metadata are checked for supported image format (JPG and PNG) with AWS Lambda functions, else the image is not processed any further. The metadata is cleaned to a pre-defined schema and passed along. The accepted image then undergoes the following, also with AWS Lambda:
    a. The image is resized to a thumbnail for display on the album page. The image is stored in AWS S3 and the key to the thumbnail is passed along.
    b. Objects in the image are detected by invoking AWS Rekognition machine learning image analysis. The detected labels are passed along

    The various pieces of data (metadata, thumbnail key, labels) are stored in AWS Dynamo DB persistent storage.
5) The album page will check if corresponding rows for the user's images have appeared in AWS Dynamo DB. If so, the page will retrieve the information, fetch the thumbnails from AWS S3 and display the thumbnails along with some metadata (e.g. creation time) and the detected labels.
6) The user can choose to log out.
   Options to remove information is omitted.

The following diagram showcases AWS's reference architecture, which is slightly outdated. For example, in the new serverless image processing example[2] AWS EventBridge has taken over the cloud event and messaging role between AWS S3 and AWS Step Function.



[1] AWS Serverless Reference Architecture: Image Recognition and Processing Backend https://github.com/aws-samples/lambda-refarch-imagerecognition
[2] AWS Serverless Image Processing Workshop https://www.image-processing.serverlessworkshops.io/

Our project will largely follow this reference architecture to provide a similar usage pattern. For simplicity, we have removed the following:

- User authentication
  This is added in later versions of the AWS Sample.
- Album functionality
  There is only one album per user, and automated checking of the persistent database for image process result is replaced by a manual refresh button.

The application built in this project has the other functionalities provided in the AWS Sample.

Deployment environment

Before further discussing our own architecture, we will first explain the deployment environment we have chosen for this project. Our overall goal is always to reimplement the application – and the cluster environment will decide how close can we get. On one end, it is possible for each group member to run kind or minikube locally and share container images within the group. This approach is very convenient but remove some aspects of working with a more production-oriented cluster.

- Cluster behaviours are different. For instances, most of the time there is no need to consider the cluster network with most things made available on localhost. There is also much less deployment delay, connection resets, etc. which has far-reaching effects when designing component pods.
- While there is problem related to working a remote cluster, there is also no easy way to collaboratively examine the same cluster. Notably, there are all sorts of real-life constraints, and subtle difference in local environment can frequently be a source of bugs and conflicts.

On the other end, it is possible to use kubeadm to set up a cluster from clean VMs. We must admit that it will be a good learning experience, however the amount of maintenance work will distract from the serverless application itself.

- There is no easy balance between the cost keeping a cluster running, and the effort in maintaining the cluster gracefully through all operations. Particularly, cluster management is a thorny problem no matter if one person is fully responsible (for the extra workload) or if (irrelevant) cluster admin is done by everyone.
- The cascade effect on the project when technical difficulties arise.

Unfortunately, AWS Academy does not include AWS EKS (Elastic Kubernetes Service) and the inconvenient timer for lab environment brings more trouble to having a cluster on AWS EC2. At the end, we have elected to use Microsoft Azure AKS (Azure Kubernetes Services) with free credits. The version of Kubernetes is 1.21.9.

Our cluster is based on 2 standard B2ms VMs[3]

- 2 vCPUs, 8 GB RAM and 16 GB temporary SSD storage
- Burstable and can run at lower performance when idle

IAM and role management is done through AKS, which makes it easy to give each member access to all the necessary resources. The cluster is shut down about half of the time so we can keep the cluster longer; it only takes a few minutes for the cluster owner to start up the cluster and everyone can deploy and test on the cluster in real time.

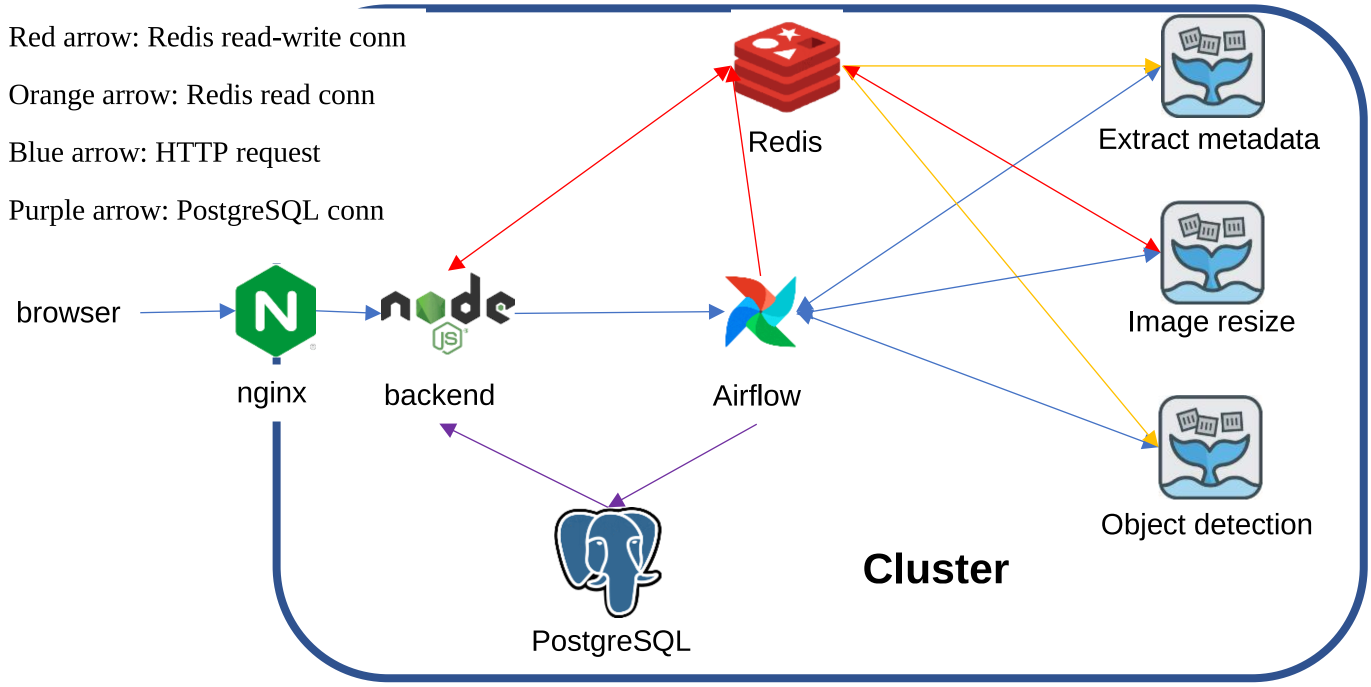
As a final note, there are some aspects of our deliverables that are more or less 'hard-coded'. The main offender being authentication information baked into the source code, instead of using Secrets properly. Some components (Redis and Open FaaS) create their authentication secret when deployed, and we admit hard coding these passwords is not ideal in more than one way. We know that it is possible to create the secrets needed by the pods in their respective namespace and giving code running inside the container access to the secrets. The quickest way is to modify the YAML file and pass reference to the secrets as environment variables[4]; we did not standardize this requirement early on. Meanwhile, the container images responsible for most of the image processing backend is hosted on a private Docker Hub repository, and proper `kubernetes.io/dockerconfigjson` Secret is used. There are other minor issues like namespace naming (as all deployment uses Kubernetes DNS for Services) which are defined using kubectl before installation with Helm/Arkade. Otherwise, most changes are cleanly listed in the YAML files modifying Helm charts and thus portable.

[3] B-series burstable virtual machine sizes https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable
[4] Using Secrets as environment variables https://kubernetes.io/docs/concepts/configuration/secret/#using-secrets-as-environment-variables

Tools (Other than Docker)

- Kubectl[5]
  Specific to this project: Azure CLI (after log-in) set up seamless access to the cluster in one step by setting the context `~/.kube/config.json`. For direct access to many components, either `kubectl port-forward` or `LoadBalancer` Service type is used.
- Helm/ Arkade[6, 7]
  For larger installations Helm charts are usually used to orchestrate the proper installation initialization sequence and configuration of different resource. The source of the Helm Charts used will be listed together with the YAML files. Arkade from the author of Open FaaS is another interface built on top of Helm for quick close-to-default installation.
- minikube[5]
  Local development might also involve other CLIs (Redis and PostgreSQL).

Our application architecture



Red arrow: Redis read-write conn

Orange arrow: Redis read conn

Blue arrow: HTTP request

Purple arrow: PostgreSQL conn

In essence, the following is the direct correspondence between AWS products and their open-source alternatives.

AWS S3	AWS Step Function	AWS Lambda	AWS DynamoDB	AWS Rekognition
Redis[8]	Apache Airflow[9]	Open FaaS[10]	PostgreSQL[11]	Tensorflow[12]

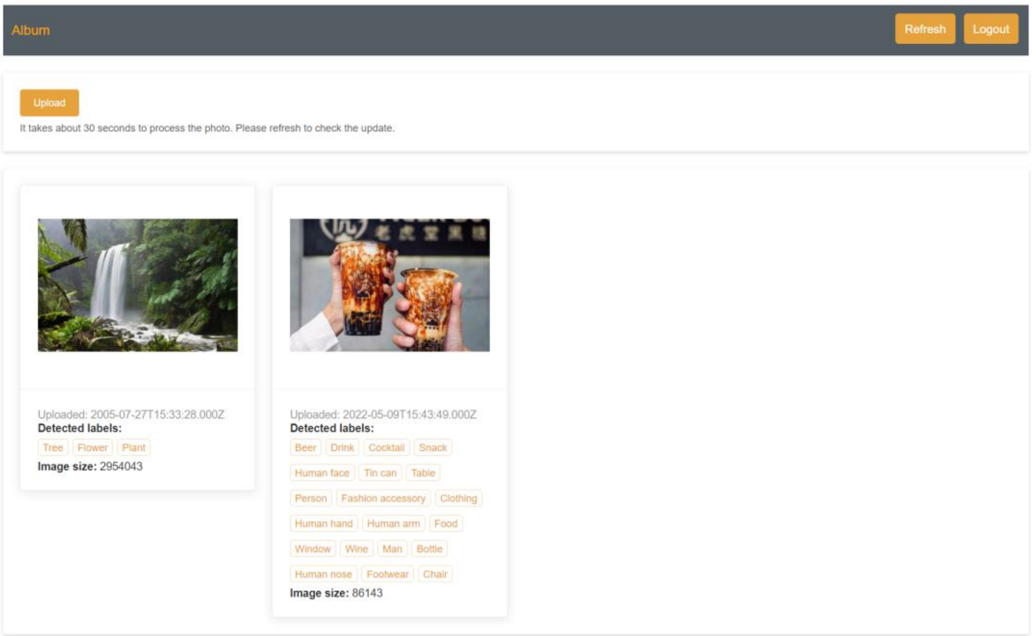
Apache Airflow and Open FaaS provide REST API, which means the majority of communication is done by HTTP traffic inside the cluster between services. PostgreSQL as well as the free version of Redis depends on language specific clients for connections using TCP. For TensorFlow, see the section on Object Detection under Open FaaS for the use of TF Hub model for obtaining image labels.

In addition, Vue.js[13] is used for constructing the user interfaces and nginx[14] is used for reverse proxy server. The backend is a Node.js[15] runtime.

[5] kubectl and minikube https://kubernetes.io/docs/tasks/tools/
[6] Helm https://helm.sh/
[7] Arkade https://github.com/alexellis/arkade
[8] Redis https://redis.io
[9] Apache Airflow https://airflow.apache.org/
[10] Open Faas https://www.openfaas.com/
[11] PostgreSQL https://www.postgresql.org/
[12] Tensorflow https://www.tensorflow.org/
[13] Vue.js https://vuejs.org/
[14] nginx https://www.nginx.com/
[15] Node.js https://nodejs.org/en/

Vue

Vue is a JavaScript framework for building user interfaces. It builds on top of standard HTML, CSS and JavaScript, and provides a responsive model that helps users efficiently develop front end web application. When the development version is completed, run `npm run build` to build the release version based on webpack.



It takes about 30 seconds to process the photo. Please refresh to check the update.

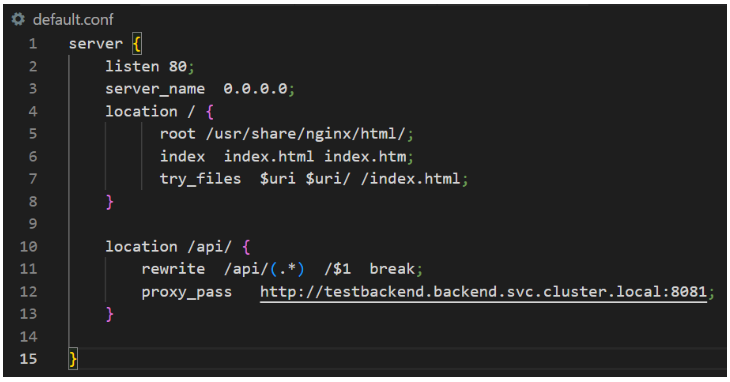
Uploaded: 2005-07-27T15:33:28.000Z
Detected labels:
Tree Flower Plant
Image size: 2954043

Uploaded: 2022-05-09T15:43:49.000Z
Detected labels:
Beer Drink Cocktail Snack
Human face Tin can Table
Person Fashion accessory Clothing
Human hand Human arm Food
Window Wine Man Bottle
Human nose Footwear Chair
Image size: 86143

Page displaying thumbnail and labels of uploaded images

Nginx

nginx is an HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server. It allows the web pages to be accessed from internet, and forward requests from frontend to backend to avoid Cross Domain Issue. I create a customized nginx docker image by copy the website codes and configuration file into the official nginx image. Then deploy this image on Kubernetes in the `frontend` namespace and create the load balancer.



default.conf

```
server {
    listen 80;
    server_name  0.0.0.0;
    location / {
        root /usr/share/nginx/html/;
        index  index.html index.htm;
        try_files  $uri $uri/ /index.html;
    }

    location /api/ {
        rewrite  /api/(.*)  /$1  break;
        proxy_pass   http://testbackend.backend.svc.cluster.local:8081;
    }

}
```

Node.js

Node.js is an asynchronous event-driven JavaScript runtime and designed to build scalable network applications. It can easily create a backend server for the website. The program contains several APIs to communicate with frontend and database:

- `/up`
  It accepts a POST request and receive a photo from website. It will store the photo into Redis `redis-master` and send a POST request to the Airflow REST API[16] at `airflow-cluster-web` to kickstart the image processing workflow.
- `/all/:user`
  This GET request will query the PostgreSQL DB at `pgsql-postgresql`, retrieve the data about photos' that this user uploaded from metadata table.
- `/pic/:user/:file`
  This GET request will return the photo file from Redis at `redis-master`, by using user and filename is the key (see section on Redis).

The backend program is packeted into an image with node.js runtime environment. It is deployed in `backend` namespace.

[16] Airflow API (stable) https://airflow.apache.org/docs/apache-airflow/stable/stable-rest-api-ref.html

Redis

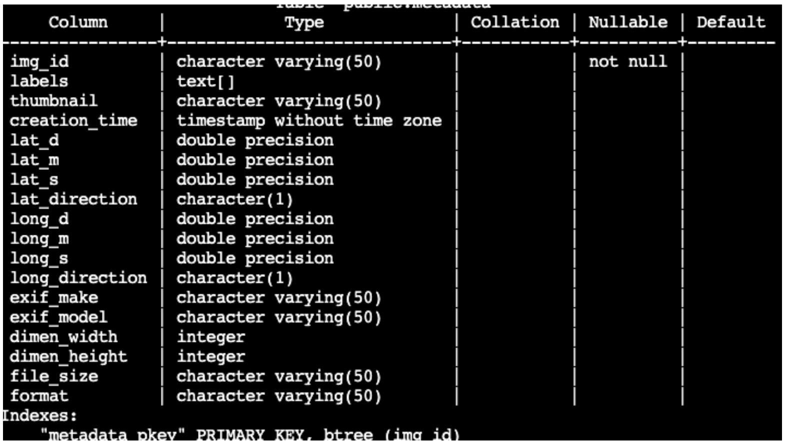
Redis at its simplest is a key-value storage, where both key and value can take binary data. It substitutes AWS S3 object storage. The ability to store binary string safely[17] as string is taken advantage to store user uploaded image as well as processed thumbnail (for JPG and PNG which are binary files). The key to each image file is then stored in the persistent DB. As such, each key is either `user/image.ext` or `user/image_thumbnail.ext`. To store the value, the website backend will buffer the content of the multipart HTTP request and directly save the buffer content to Redis through Redis client. The reverse is true when an image is loaded with the help of image processing modules.

Redis is deployed with Arkade in the namespace `redis`. Two main services are created by the underlying Helm chart: `redis-master` and `redis-replicas`. `redis-master` supports read-write access but is of course easily congested since there is only 1 replica. The second service is for read-only access with multiple replicas updating itself regularly.

PostgreSQL

PostgreSQL is a robust open-source database management system that supports both SQL relational query and JSON non-relational query. PostgreSQL is deployed by using the Helm chart provided by bitnami[18] which is a library for software packages. Because PostgreSQL is a stateful application in Kubernetes, it should collaborate with the Persistent Volume (PV)[19], a piece of storage in the cluster that has independent lifecycle, to create and manage the permanent storage of pods.

The database will store the extracted metadata of images, e.g. `latitude` and `fileSize` etc., labels detected from the model and the Redis key to the generated thumbnails. Accordingly, the table `metadata` is constructed to store above information where the Redis key to the original image `img_id` is set as the primary key. The scheme of the table is shown below. Since most images do not have complete metadata, all attribute columns are set to be nullable, which is also the default value.



```
                      Table "public.metadata"
     Column      |            Type             | Collation | Nullable | Default
-----------------+-----------------------------+-----------+----------+---------
 img_id          | character varying(50)       |           | not null |
 labels          | text[]                      |           |          |
 thumbnail       | character varying(50)       |           |          |
 creation_time   | timestamp without time zone |           |          |
 lat_d           | double precision            |           |          |
 lat_m           | double precision            |           |          |
 lat_s           | double precision            |           |          |
 lat_direction   | character(1)                |           |          |
 long_d          | double precision            |           |          |
 long_m          | double precision            |           |          |
 long_s          | double precision            |           |          |
 long_direction  | character(1)                |           |          |
 exif_make       | character varying(50)       |           |          |
 exif_model      | character varying(50)       |           |          |
 dimen_width     | integer                     |           |          |
 dimen_height    | integer                     |           |          |
 file_size       | character varying(50)       |           |          |
 format          | character varying(50)       |           |          |
Indexes:
    "metadata_pkey" PRIMARY KEY, btree (img_id)
```

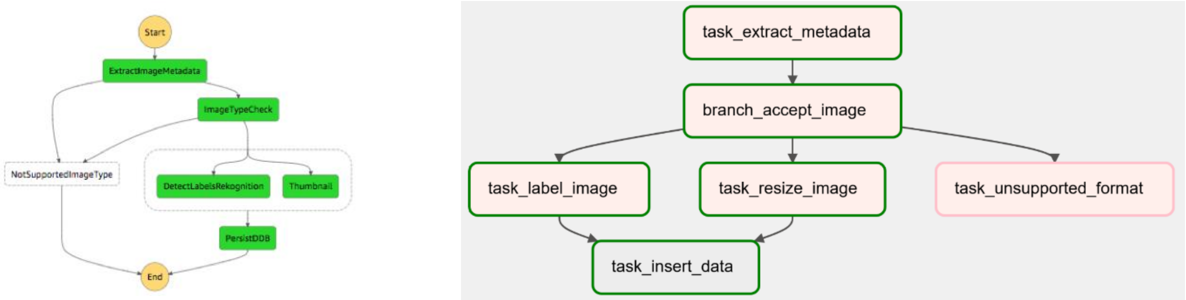
PostgreSQL schema of the table `metadata`

Apache Airflow

Apache Airflow is a workflow management platform, particularly for data pipeline. It is possible to define workflows programmatically, which can then be scheduled for regular execution or triggered. In our project, it substitutes AWS Step Function, while removing the need for AWS EventBridge since the backend has direct access to Airflow's API.

The workflow object in Airflow is aptly named as DAG and is written in Python with Airflow 2's TaskFlow API. The workflow itself is defined as a function, and tasks within are defined as child function inside. The logic can pretty much be expressed as if one is writing normal functions (with decorators), while Airflow monitor each of the tasks and handle all the intermediate communications. In our case, each step in the image processing workflow is triggered as API calls to various providers (Redis, PostgreSQL and Open FaaS), which manage their own containers. One can then easily program intricate control (retries, timeout, task dependencies, etc.) conceptually without worrying about every particularity like synchronization, error handling, etc. The added flexibility means our workflow is further simplified.

[17] Redis String https://redis.io/docs/manual/data-types/#strings
[18] bitnami https://bitnami.com/
[19] Kubernetes persistent volume https://kubernetes.io/docs/concepts/storage/persistent-volumes/



Graphical representation of AWS Sample workflow and our workflow

The workflow itself is triggered (starting a DAG run) by an external POST request. The single piece of information for the DAG run is `conf: {“image_key”: “user_id/image.ext”}`. This information is available to all task instances. At the start of the DAG run, the metadata extraction function is invoked, which will check both file extension and metadata for file format. Depending on the output, the DAG run either continue to the next image processing tasks or terminate (`task_unsupported_format`). For termination, we choose to directly update Redis key that supposed to hold the thumbnail with a failure message; this information is not used right now.

Airflow itself is deployed user the community managed Helm chart[20] – easier to work with than the official one where everything is in full Kubernetes model. Some cluster specific configurations (including connection to other services on cluster) are provided through our own YAML file. Workflow definitions and other files (SQL query) need to be baked into the Airflow image.

Open FaaS

Open FaaS is a mature, simple-to-use serverless platform. Through its CLI `faas-cli`, Open FaaS provide function templates in various languages[21], complete with Dockerfile and boilerplates for handling requests, performing health checks, etc. Much of the configuration of the function (container) is specified through a YML file, which `faas-cli` will handle when deploying the function onto the cluster. For example, the object detection function require slightly more time to run (between 30 seconds and a minute) which is accommodated with extended timeout. During testing, only one replica is maintained for each function. Instructions are provided in the `open-faas` source code file.

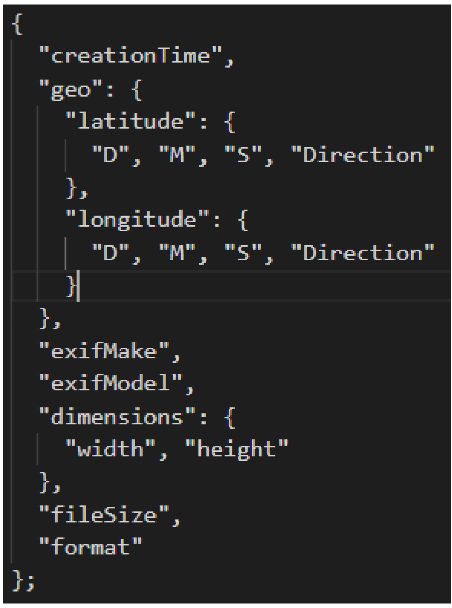
For the actual function image, we only need to concern ourselves with the actual image processing implementation. Each function is fully defined by the source code file, the YML file and language-specific files for dependencies. It should be noted that for some languages there are fewer options in how requests can be handled. In our case, the Python template has access to request body and not query. Open FaaS is deployed with Arkade.

Metadata extraction (`extract-metadata-redis`)

The function is implemented with Node.js 12 using the `sharp` image processing module[22] for its speed. The function input is the key (`user_id/image.ext`) to the image stored on Redis (loading from `redis-replicas`). The image metadata is extracted from the fetched image file and cleaned. The `exif-reader` module[23] is used to parse the EXIF raw buffer in the metadata. The image’s extension and metadata are both checked in this function. The function returns the formatted metadata JSON to Airflow, or nothing if the image type is not supported.

Thumbnail generation (`image-resize-redis`)

The function is largely the same as the previous function. Instead of reading metadata, the function resizes the image loaded from `redis-master` at fixed aspect ratio down to a default maximum height/ width (adjustable). The thumbnail is stored on `redis-master` under a new key (`user_id/image_thumbnail.ext`). The function then returns this new key to Airflow be stored in the persistent DB.



```
{
  "creationTime",
  "geo": {
    "latitude": {
      "D", "M", "S", "Direction"
    },
    "longitude": {
      "D", "M", "S", "Direction"
    }
  },
  "exifMake",
  "exifModel",
  "dimensions": {
    "width", "height"
  },
  "fileSize",
  "format"
};
```

[20] Airflow Helm Chart (User Community) https://github.com/airflow-helm/charts
[21] OpenFaaS Classic templates https://github.com/openfaas/templates
[22] sharp https://sharp.pixelplumbing.com/
[23] exif-reader https://www.npmjs.com/package/exif-reader

Object detection (`tfhub-od-redis`)

The function runs on a TensorFlow container that serves a TensorFlow Hub model[24]. The object detection model of choice is the SSD-based model (trained on OpenImage v4) with MobileNet V2 (trained on ImageNet) as feature extractor. This particular TF Hub model is chosen for its speed (the model stored as a static graph and the speed of MobileNetV2) and large set of labels recognised (600). Unfortunately, TensorFlow Serving[25] is not applicable. It is not possible to cleanly pull the graph out of the TF Hub model and build a pipeline, since the human readable labels are handled internally in the TF Hub model. This means the main, larger, TensorFlow container is used just to load the image into Tensor.

The function takes the input key and loads the image from `redis-replicas`. The image (tensor) is passed to the model, which output labels corresponding to each detection in the image. For simplicity, the set of labels, with no filtering for probability, is returned in the response.

Since it is wasteful to download the model every time the function is invoked, there is a separate long running process (`model_loader.py`) in the container that pre-loads the model and constantly wait for socket connection from the actual function script `handler.py` being run each time Object detection is invoked. The request is passed along to the long running process and the labels are passed back. The model itself is small enough to be baked into the container image, which means Open FaaS can start up extra pods with slightly less cool start.

Conclusion

In this project, we have re-implemented the Image Recognition serverless application from AWS Sample on a Kubernetes cluster. Nearly all functionalities are available in the same form, with modifications that reflect the higher flexibility we have on a standalone cluster. The infrastructure deployed on the cluster provides in fact the same kind of extensibility as the AWS sample while giving us full information and control. There are some deficits regarding security that would be addressed in a production environment – Kubernetes has adequate tools to address them. Overall, it is a successful experience as we have no experience developing on a distributed cluster prior to the project.

[24] TensorFlow Hub (Model) https://tfhub.dev/google/openimages_v4/ssd/mobilenet_v2/1
[25] TensorFlow Serving https://www.tensorflow.org/tfx/guide/serving