FRANKFURT UNIVERSITY OF APPLIED SCIENCES

CI/CD of Cloud Functions including the Service by using Infrastructure as Code

Cloud Computing SS2022

Submitted by:

Tobias Maas
Tobias Schiffhauer
Raoul Neumann

Under the guidance of:

Prof. Dr. Christian Baun

Table of content

1 Introduction

OpenFaaS makes it easy for developers to deploy event-driven functions and microservices to Kubernetes without repetitive, boiler-plate coding. Package your code or an existing binary in a Docker image to get a highly scalable endpoint with auto-scaling and metrics.[1]

1.1 Continuous Integration and Continuous Delivery

CI/CD stands for Continuous Integration, Continuous Delivery and Continuous Deployment. Continuous integration is the process of testing and integrating new code automatically into an existing code base. Continuous delivery describes the automatic release of the code. The automatic deployment of the build code to the different environments is called Continuous Deployment. The process from start to finish has the term pipeline.
Our project focuses more on the deployment aspects of the pipeline. The terraform language is not a typical programming language, so testing and integrating is handled. Although the amount of lines of code can not be compared to a typical programming language.[2]

1.2 Github Actions

Github is the biggest version-control and collaboration platform. It is based on the git program, which is an Open-Source code management tool. Github Actions is their CI/CD pipeline product. If there are changes in a specified area of the repository, a program is started with step by step instructions of tasks. The possible tasks and steps depend only on the wished outcome.
Our pipeline deploys a Kubernetes cluster into the Google and Amazon Cloud. Then it installs OpenFaas into the cluster and uploads an example function in the cluster.[3]

1.3 Terraform

With Terraform it is possible to declare the wanted infrastructure as a script. With these scripts it is possible to automate the creation of the infrastructure and have the same configuration every time. The usage of different cloud providers is also possible.[4]
We use Terraform for the description of our Kubernetes cluster and configuration for a public IP or domain for the accessibility of our service.

---

[1] https://docs.openfaas.com/
[2] https://www.redhat.com/en/topics/devops/what-is-ci-cd#overview
[3] https://www.techtarget.com/searchitoperations/definition/GitHub
[4] https://www.terraform.io/

1.4 Kubernetes

Kubernetes is a tool for automatic deployment, scaling and management of containern. The tool is designed and developed by Google and is now the standard for container orchestration. The configuration demand for a response to demand changes or pushing a new update to the containers is low.[5]
Our OpenFaas service will run in a Kubernetes cluster for the reasons mentioned above.

1.5 Cloud Solution

Unlike a private or shared cloud solution, the public cloud is accessible to businesses and the general public. Because the cloud platform is also intended to appeal to private individuals, the offered services have a wide price range.There are a few free education offers or trials available, although they have their limitations on the resources that can be used. On the other side, the available services offer easy scalability to fulfill the needs of bigger companies or government organizations as well.

We have chosen the two largest providers, Google Cloud and Amazon Web Services (AWS), because they either provide a generous starting credit upon registration or because they are relatively inexpensive. The two cloud service providers have also already made a name for themselves in the area of cloud solutions and are also used by companies or government organizations, so the experience gained with these cloud service providers has a certain practical relevance.

[5] https://kubernetes.io

2 Architecture

The goal of the project is to create a CI/CD pipeline which deploys OpenFaas Functions on a given Kubernetes cluster. The pipeline gets triggered by a change in a Github repository. OpenFaas will be deployed into a Kubernetes Cluster, which will be hosted in a public cloud. In our case the public clouds from Google or Amazon (AWS) are used. For a reliable installation process, the most of the configuration will be scripted in Terraform files or short Tf files. In the files is declared how the cluster shall be configured. For the use case of Google Cloud, in the Terraform files are the configurations for the Google Kubernetes Engine (GKE) Cluster. In the use case of AWS, the Terraform files contain the configuration for the Elastic Kubernetes Service (EKS).
The Terraform state contains information about the current state of the deployed cluster and is stored in a remote storage, because the virtual machines that execute the Github Actions do not store files permanently.

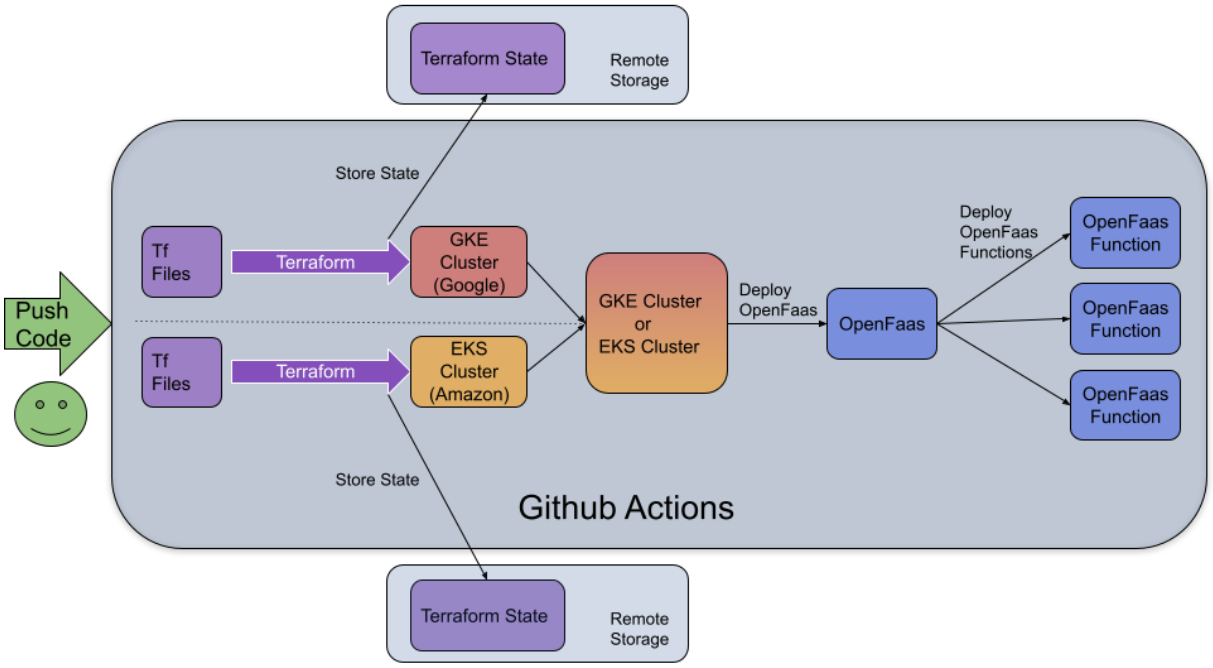


Figure 1: Architecture diagram of the used components for the CI/CD pipeline.

3 Installation

In this section the manual steps for the deployment of a GKE and EKS Cluster as well as for the deployment of OpenFaas and OpenFaas Functions on the created cluster are displayed. Code Blocks with a dark background represent terminal commands, whereas code blocks with a light background is expressing other code from Terraform files, Github Actions or a python function.

3.1 Google Cloud

3.1.1 Configure Google Cloud SDK

Install Google SDK

```
$ sudo apt install google-cloud-sdk
```

Connect Google SDK with Google Account

```
$ gcloud init
```

Add Google Account to ADC Application Default Credentials, so that Terraform can access the credentials

```
$ gcloud auth application-default login
```

3.1.2 Terraform

Our kubernetes setup is based on the example from Terraform.[6]
We are using this a base and extend it into usage for Github Actions and OpenFaas

```
$ git clone https://github.com/hashicorp/learn-terraform-provision-gke-cluster
```

To start working with the example and later execute terraform commands, we have to change the working directory.

```
$ cd learn-terraform-provision-gke-cluster
```

In the terraform.tfvars file, we have to enter our project id as well as the region in which we want the cluster to be created. In this case we choose europe-west3, this region has three zones respectively europe-west3-a, europe-west3-b and europe-west3-c.
All available regions and their zones can be seen at the following link.[7]

```
project_id = "crypto-parser-350713"
region     = "europe-west3"
```

[6] https://learn.hashicorp.com/tutorials/terraform/gke
[7] https://cloud.google.com/compute/docs/regions-zones?hl=en

The project id can be retrieved with the following command from the command line or from the google cloud console website at the dashboard.

```
$ gcloud config get-value project
```

The following two blocks of code show the content of the file gke.tf. In here we declare the resources that we want to deploy on google cloud to create our cluster. First the variable "gke_num_nodes" is created with a default value of 1. With this variable we can declare how many nodes we want to deploy in a single zone. This is the first thing you have to change if you are using the free google cloud trial, because the default value from the terraform example was 2, which exceeds the limit of the free trial.
Next a resource called "google_container_cluster" is created. Therefore you have to provide the name and location for the cluster, in this case the values from the terraform.tfvars file we set earlier are used. As recommended we want to use a separately managed node pool, because this has more flexibility in customizing the kubernetes cluster. Therefore we immediately delete the default node pool by setting the remove_default_node_pool to true.

```
variable "gke_num_nodes" {
  default     = 1
  description = "number of gke nodes"
}

# GKE cluster
resource "google_container_cluster" "primary" {
  name     = "${var.project_id}-gke"
  location = var.region
  remove_default_node_pool = true
  initial_node_count       = 1

  network    = google_compute_network.vpc.name
  subnetwork = google_compute_subnetwork.subnet.name
}
```

Now the separately managed nodel pool is declared with the resource "google_container_node_pool". Here we again set the name and location of the node-pool, then we have to assign the predefined cluster from above to the cluster argument. The variable node_count is set to our predefined variable with the default value of 1. Another argument to pay attention to is the machine_type, because this determines the machine_type of the single nodes and in this case has a great impact on cost. In this case the machine_type "n1-standard-1" is chosen which is a low cost machine. More information on available machine_types and their specifications here.[8]

```
# Separately Managed Node Pool
resource "google_container_node_pool" "primary_nodes" {
  name       = "${google_container_cluster.primary.name}-node-pool"
  location   = var.region
  cluster    = google_container_cluster.primary.name
```

[8] https://cloud.google.com/compute/docs/general-purpose-machines?hl=en#n1_machines

```
  node_count = var.gke_num_nodes

  node_config {
    oauth_scopes = [
      "https://www.googleapis.com/auth/logging.write",
      "https://www.googleapis.com/auth/monitoring",
    ]
    labels = {
      env = var.project_id
    }
    # preemptible  = true
    machine_type = "n1-standard-1"
    tags         = ["gke-node", "${var.project_id}-gke"]
    metadata = {
      disable-legacy-endpoints = "true"
    }
  }
}
```

Another important aspect when working with terraform is considering the state. The state contains the necessary information for terraform to know which resources are deployed at the moment. For our use case we need a remote state, because the machines that execute the Github Actions does not have permanent storage. Therefore the state is stored in google cloud storage, as declared in backend.tf.

```
terraform{
      backend "gcs" {
            bucket = "cloudprojekttest"
            prefix = "terraform/state"
      }
}
```

With the command terraform init, terraform initializes the needed state and providers

```
$ terraform init

Initializing the backend...

Successfully configured the backend "gcs"! Terraform will automatically
use this backend unless the backend configuration changes.

Initializing provider plugins...
- Finding hashicorp/google versions matching "3.52.0"...
- Installing hashicorp/google v3.52.0...
- Installed hashicorp/google v3.52.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the
```

```
provider selections it made above.

Terraform has been successfully initialized!
```

The command terraform apply then installs the defined structure in GCP.

```
$ terraform apply
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

..

Plan: 4 to add, 0 to change, 0 to destroy.
```

3.1.3 Kubernetes

For managing Kubernetes clusters, their CLI is needed. So we will install kubectl first.

```
$ sudo apt install kubectl
```

As the second step, we need the credentials to connect to our cluster. With the following command, we will get the credentials from Google Cloud.

```
$ gcloud container clusters get-credentials $(terraform output -raw
kubernetes_cluster_name) --region $(terraform output -raw region)

Fetching cluster endpoint and auth data.
kubeconfig entry generated for crypto-parser-350713-gke.
```

3.2 Setting up Amazon Web Service EKS EC2 Cluster

3.2.1 Create a AWS Account

In order to use Amazon Web Services (AWS), you first need an account, as with Google Cloud. After creating the aws account, we still need to create a user, which we need to interact with aws. The Identity and Access Management (IAM) dashboard is responsible for managing users, groups and permissions. Once in the IAM, select Users on the left-hand side and then Add users. Now you can think about a username and in *Select AWS credential type* you must select *Access key - Programmatic access*.After we have created a new user, we select it on the overview page and in the first tab Permissions we add to the user all the permissions it needs to be able to create and manage the cluster. To give the user the necessary permissions, we first click on *Add Permissions* and then on *Attach Existing Policies Directly* from the top three buttons. In the lower table we now mark three

permissions: *AmazonEC2FullAcces*, *IAMFullAccess*, *AmazonEKSServicePolicy*. Then we review the selected permissions again and confirm them. Now we add another permission, but instead of selecting it from the table we click on the Create policy button above it and click on the JSON tab. Here you can program permissions instead of selecting them from the predefined pattern.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "eksadministrator",
            "Effect": "Allow",
            "Action": "eks:*",
            "Resource": "*"
        }
    ]
}
```

Alternatively, we can also distribute the rights via groups. For this we still click in the IAM on User groups and then on create group. Here we give our new group a name and select all users who should belong to this group. Below that we select the same three permissions that are needed for the administration of the cluster.

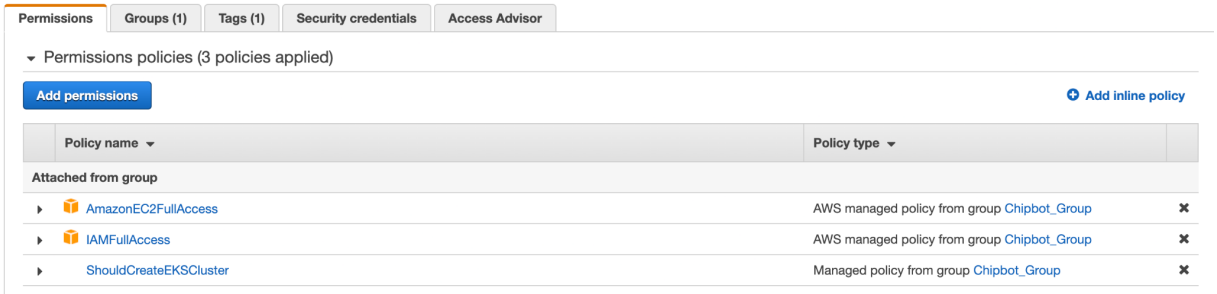
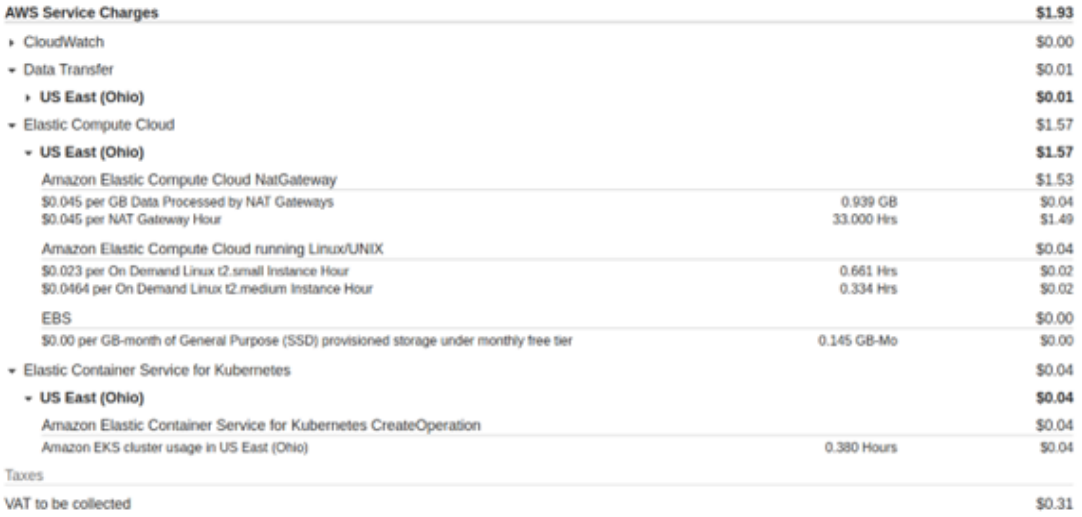


Figure 2: Figure shows the overview page of newly created user and his permissions, which are managed by the group he belongs to.

We confirm this self-created authorization again. Back on the User Profile page, we now select *Security credentials* in the upper tabs where we now generate our access key so that our user has access to aws systems. For this we click on *Create access key* and write down the two number and letter combination *Access key ID* and the *Secret access key*.

Although AWS offers several free services, unfortunately an EKS cluster is not free. You will

be charged $0.045 per GB Data Processed by NAT Gateways, $0.045 per NAT Gateway Hour, $0.0464 per On Demand Linux t2.medium Instance Hour, and an additional fee depending on the location of the server. To prevent these costs from skyrocketing unattended, you should create a new budget in the *Billing Dashboard -> Budgets* and select that all services should be switched off as soon as the set limit is exceeded. Alerts can also be set to notify you either when certain amounts are exceeded or when a percentage is reached.



AWS Service Charges		**$1.93**
▸ CloudWatch		$0.00
▾ Data Transfer		$0.01
▸ **US East (Ohio)**		**$0.01**
▾ Elastic Compute Cloud		$1.57
▾ **US East (Ohio)**		**$1.57**
Amazon Elastic Compute Cloud NatGateway		$1.53
$0.045 per GB Data Processed by NAT Gateways	0.939 GB	$0.04
$0.045 per NAT Gateway Hour	33.000 Hrs	$1.49
Amazon Elastic Compute Cloud running Linux/UNIX		$0.04
$0.023 per On Demand Linux t2.small Instance Hour	0.661 Hrs	$0.02
$0.0464 per On Demand Linux t2.medium Instance Hour	0.334 Hrs	$0.02
EBS		$0.00
$0.00 per GB-month of General Purpose (SSD) provisioned storage under monthly free tier	0.145 GB-Mo	$0.00
▾ Elastic Container Service for Kubernetes		$0.04
▾ **US East (Ohio)**		**$0.04**
Amazon Elastic Container Service for Kubernetes CreateOperation		$0.04
Amazon EKS cluster usage in US East (Ohio)	0.380 Hours	$0.04
Taxes		
VAT to be collected		$0.31

Figure 3: Image from the billing dashboard after the cluster ran for about 1 hour.

Now that we have set up everything important in the AWS Dashboard, it's time to set up and install AWS and Terraform. In order to interact with AWS Services, we first need the AWS Command Line Interface (AWS CLI).

```
$ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install
```

After the successful AWS CLI installation, which can be verified with `$ aws -v`, we run `$ aws configure` to configure AWS CLI. Here we first enter the AWS *generated access key ID* and the *secret access key*. Afterwards we can decide for a region in which our cluster should run later and for the default output format we decide for json.

```
$ aws configure
AWS Access Key ID [None]: YOUR_AWS_ACCESS_KEY_ID
AWS Secret Access Key [None]: YOUR_AWS_SECRET_ACCESS_KEY
Default region name [None]: YOUR_AWS_REGION
Default output format [None]: json
```

In earlier versions of AWS CLI (before 1.16.156 approx. summer 2019), the AWS IAM Authenticator was additionally required to be able to authenticate against the system. In the

meantime, this function is integrated in AWS CLI and can also be queried manually with the command `$ aws eks get-token` .

After all the tools required by AWS have been installed, we then move on to setting up Terraform in the next Chapter.

3.2.2 Terraform

After we have created an AWS account in the last chapter, set up the cost limit and assigned all necessary rights to our new user, we can now start with the installation of Terraform.

For this we need kubectl, which we can create using

```
$ curl -LO https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl
```

and then install it with.

```
$ sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

We also need Wget, for which we need the following command:

```
$ sudo apt-get install wget
```

After that we can install Terraform. To do this, we first clone the corresponding repository:

```
$ git clone https://github.com/hashicorp/learn-terraform-provision-eks-cluster
```

And then go into the freshly created directory with

```
$ cd learn-terraform-provision-eks-cluster
```

In the directory there are several *.tf* among others to change terraform settings, but for this example we can take the default values.

Now that we have everything we need for Terraform installed, let's initialize the Terraform workspace next.

```
$ terraform init
```

```
Initializing modules...

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/cloudinit from the dependency lock file
- Reusing previous version of hashicorp/local from the dependency lock file
- Reusing previous version of hashicorp/null from the dependency lock file
- Reusing previous version of hashicorp/kubernetes from the dependency lock file
- Reusing previous version of hashicorp/aws from the dependency lock file
- Reusing previous version of hashicorp/random from the dependency lock file
- Reusing previous version of terraform-aws-modules/http from the dependency lock
file
- Using previously-installed hashicorp/null v3.1.0
- Using previously-installed hashicorp/kubernetes v2.7.1
- Using previously-installed hashicorp/aws v3.71.0
- Using previously-installed hashicorp/random v3.1.0
- Using previously-installed terraform-aws-modules/http v2.4.1
- Using previously-installed hashicorp/cloudinit v2.2.0
- Using previously-installed hashicorp/local v2.1.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Figure 4: After the successful initialization of Terraform, the settings should be correct and dependencies should be present. Terraform is now ready to be executed to create a cluster.

And then we run the planned actions.

```
$ terraform apply
```

Terraform will now display a list of actions to be performed. These have to be confirmed again with a yes before execution.

Terraform now creates a cluster, which takes about 10 to 15 minutes.

After the cluster has been created, we continue in the following chapter with the setup of Kubernetes.

3.2.3 Kubernetes

Now that our cluster is live we need to set up kubectl. How kubectl gets installed is described under 3.1.3. For this we used the following command:

```
$ aws eks --region $(terraform output -raw region) update-kubeconfig
--name $(terraform output -raw cluster_name).
```

Before we look at our cluster through the Kubernetes dashboard, we can view additional metric values using the Kubernetes metrics server.
To do this, we download the metrics server using the following command.

```
$ wget -O v0.3.6.tar.gz https://codeload.github.com/kubernetes-sigs/metrics-server/tar.gz/v0.3.6 && tar -xzf v0.3.6.tar.gz
```

And run it with the following command

```
$ kubectl apply -f metrics-server-0.3.6/deploy/1.8+/
```

Then we check if the deployment of the metric server was successful.

```
$ kubectl get deployment metrics-server -n kube-system
```

To use the dashboard, we need to run this first, through the following command:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-beta8/aio/deploy/recommended.yaml
```

Now we need a proxy, which allows us to access the dashboard conveniently through the browser.

```
$ kubectl proxy
```

Now it is possible to access the Kubernetes Dashboard with the following link.

```
http://127.0.0.1:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/
```

To be able to authenticate to the dashboard we have to give admin rights to the cluster.
To do this, in a new terminal window, since we do not want to terminate the proxy, we need to enter the following command:

```
$ kubectl apply -f https://raw.githubusercontent.com/hashicorp/learn-terraform-provision-eks-cluster/main/kubernetes-dashboard-admin.rbac.yaml
```

And then another command to generate an authentication token:

```
$ kubectl -n kube-system describe secret $(kubectl -n kube-system get secret | grep service-controller-token | awk '{print $1}')
```

We copy out the token we get now and select the dashboard token from the freshly generated token below it.
We are then greeted by the Kubernetes Dashboard displaying all of our cluster nodes.

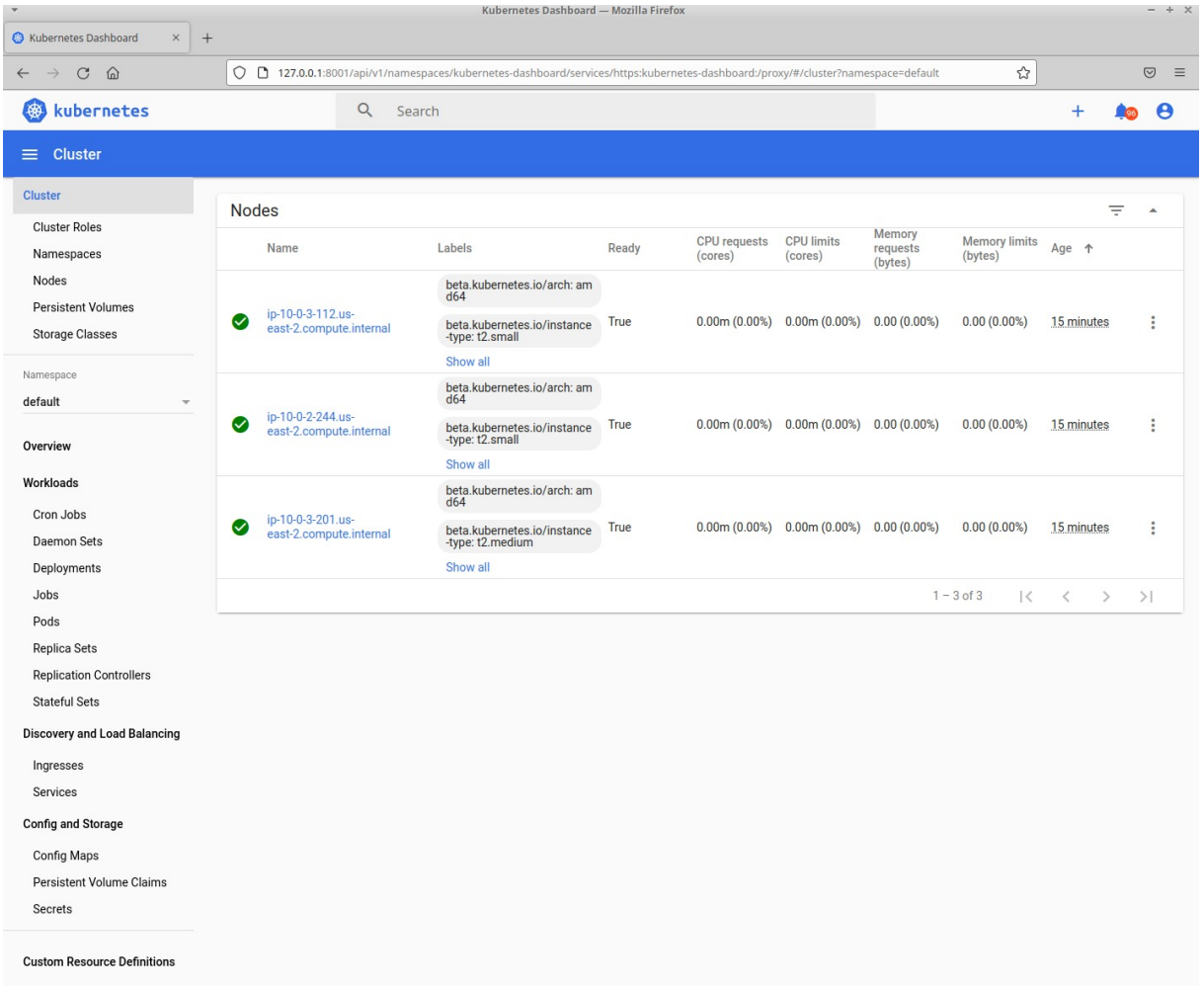


Figure 5: Kubernetes dashboard interface, after successfully logging in while running an AWS cluster as previously described.

When we are done with the cluster and want to turn it off again, this is easily done with the command.

```
$ terraform destroy
```

3.3 Installation of OpenFaas

Arkade is an Open-Source tool for installing programs to the Kubernetes cluster. Arkade makes it relatively easy for the developer and is also recommended by OpenFaas. For installing Arkade you need the following command.

```
$ curl -sLS https://get.arkade.dev | sudo sh
```

The next step is to install OpenFaas in the Kubernetes Cluster. That can be done with only one command:

```
$ arkade install openfaas -load-balancer
Using Kubeconfig: /home/runner/.kube/config
Client: x86_64, Linux
```

```
..

Release "openfaas" does not exist. Installing it now.
NAME: openfaas
LAST DEPLOYED: Tue Jun  7 09:09:58 2022
NAMESPACE: openfaas
STATUS: deployed
REVISION: 1
TEST SUITE: None

..
```

3.4 Possible connection to OpenFaas

OpenFaas has in the standard installation configuration two different gateways. One is for internal calls and the other one for external calls. If we use the port-forwarding capabilities from Kubernetes, we use the gateway and if we use a public IP or Domain from a public cloud provider, we use the external-gateway.

The first step in this process is to download the OpenFaas-CLI, which is needed not only for the login, but also for uploading a function. For downloading is the following command needed:

```
$ curl -sSL https://cli.openfaas.com | sudo sh
```

3.4.1 Port-forwarding

For the connection from a local computer, it is possible to forward the gateway port to your local one. With the first command, you can see the deployment status of the gateway. Only if the gateway is completely deployed, a port-forward is possible.

```
$ kubectl rollout status -n openfaas deploy/gateway
```

The following command will forward the port from the gateway to your local computer. A server will be started, so that you also can connect with a browser on your localhost domain to the cluster.

```
$ kubectl port-forward -n openfaas svc/gateway 8080:8080 &
```

The next step is getting the password for openfaas and saving it into a variable.

```
export PASSWORD=$(kubectl get secret -n openfaas basic-auth -o
jsonpath="***.data.basic-auth-password***" | base64 --decode; echo)
)
```

As the last step, you can login into OpenFaas via the command line and the saved password.

```
echo -n $PASSWORD | faas-cli login --username admin --password-stdin
```

As mentioned above, it is also possible to access OpenFaas via the browser on a local port. If you enter the domain localhost:8080 in the browser, a login window will pop. The user is admin and the password is in the saved password variable.

3.4.2 Connection over public IP

For accessing the cluster over the public IP, we need two pieces of information. The first one is the IP-address, which can change from each upload to the cloud config provider and the password of OpenFaas. The first command saves the IP of the cluster and the second command saves the password from OpenFaas into a variable.

```
$ export GATEWAY_IP=$(kubectl get service gateway-external -n openfaas -o
jsonpath="{.status.loadBalancer.ingress[0].ip}")
```

```
$ export PASSWORD=$(kubectl get secret -n openfaas basic-auth -o
jsonpath="{.data.basic-auth-password}" | base64 --decode; echo)
```

The last step is to login into Openfaas. For that we are using the IP as gateway address and the password as password.

```
$ echo -n $PASSWORD | faas-cli login --username admin --password-stdin
--gateway http://$GATEWAY_IP:80$ export PASSWORD=$(kubectl get secret -n
openfaas basic-auth -o jsonpath="{.data.basic-auth-password}" | base64
--decode; echo)
```

3.5 OpenFaas functions

3.5.1 Add function from store

The function store is a collection of templates, which are designed and maintained by the OpenFaas community. These functions are hosted on public docker repositories and can be installed per UI or console.[9] For our test example, we are installing the NodeInfo function. The NodeInfo function gives information about the host system. The returned values are Count of CPUs, hostname, OS and Uptime.

```
faas-cli store deploy 'NodeInfo' --gateway http://$GATEWAY_IP:8080
```

```
curl $GATEWAY_IP/functions/NodeInfo
```

3.5.2 Creating new function

For creating a new function it is necessary to have docker installed.

[9] https://github.com/openfaas/store

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable"
```

The next step is to login via docker into a docker hub account or if no account exists, create a new one. One important remark is to be sure that the docker hub and the following created repositories are public.

```
$ docker login --username user --password password
```

The easiest way is to use a template from the OpenFaas store. With the following it is possible to pull every template on the local machine.

```
$ faas-cli template pull
```

After all templates are pulled, it is possible to code a new function. With the command

```
$ faas-cli new function-name --lang template
```

openfaas cli will create the needed yml file and the corresponding folder. The yml file holds the configuration for the docker repository, gateway ip, programming language and the folder name. The files in the corresponding folder vary from template and programming language. After the wanted code is added to the function, the next step is to upload the function to the OpenFaas cluster. This will do the command

```
$ faas-cli up -f function-name.yml
```

The uploading process is divided into the three steps: build, push and deploy. The first step is, that faas-cli will build a docker file, which includes the code files. Then it will save the docker file to the local docker repository. The next is uploading the docker file to a remote docker repository. Therefore it is necessary to have for example a docker hub account and to have it connected to your console.

The last step of the upload process is the deployment to the OpenFaas cluster. All the steps are included in the up command, but can be executed as separate commands.[10]

```
$ faas-cli build
```

```
$ faas-cli push
```

```
$ faas-cli deploy
```

3.5.3 Example function

Our test function is created with the python template and is therefore a python function. The function was created with the following command.

[10] https://docs.openfaas.com/cli/templates/#get-started

```
$ faas-cli new test --lang python3
```

The command will create a main yaml file and in a folder two python and one text file.
The created yml file has as provider name openfaas and provider gateway the url of the kubernetes cluster. The cluster url is given at the upload, because the IP can change from every upload of the cluster. In the function part of the yml file is the programming language Python3, the corresponding folder test with the python files and the public docker repository.
test.yml

```
version: 1.0
provider:
  name: openfaas
  gateway: http://${URL:-exampleco}:8080
functions:
  test:
    lang: python3
    handler: ./test
    image: saibot101/test:latest
```

The __init__.py will be created, will remain unchanged, so will not be further discussed.
test/__init__.py

```
empty
```

The handler.py file contains the python code, which will be executed, when the endpoint will be called. Our test example is very simple and will return a string.
test/handler.py

```
def handle(req):
    """handle a request to the function
    Args:
        req (str): request body
    """

    return "test function"
```

In the requirements.txt must be all used pip modules mentioned with their version. In our example no module is mentioned, because we don't use one in our example function.
test/requirements.txt

```
empty
```

4 Pipelines

With knowledge from Chapter 3 we created 2 different pipelines. The first one is creating the kubernetes cluster, installing OpenFaas into it and uploading a preinstalled test function. The second pipeline takes the credentials from an existing kubernetes cluster, connects to it and

uploads a self-written function. The pipelines can be found on our github page https://github.com/cloud-computing-projekt.

The part of the pipeline is the initialization. There will be the pipeline set up, necessary packages loaded and for different providers the authorization handled. Because the initialization phase for both pipelines is similar, it will be explained in the beginning.

In the first block will be defined, on which specific events will the pipeline be triggered. Our pipeline will be triggered on every push to the main repository or on every pull request. For our proof of concept the settings are not important and become more relevant, when the pipeline is used for production use cases.
The workflow dispatch setting lets us run the pipeline independently from push or pull request triggers, which can be important for testing.
In the next part we define the environment, in which the pipeline will run and which permissions the will have.

```
name: CI

# Controls when the workflow will run
on:
  # Triggers the workflow on push or pull request events but only for the
main branch
  push:
    branches: [ main ]
  pull_request:


  # Allows you to run this workflow manually from the Actions tab
  workflow_dispatch:

jobs:
  setup-and-deploy:
    name: Setup and Deploy
    runs-on: ubuntu-latest

    # Add "id-token" with the intended permissions.
    permissions:
      contents: 'read'
      id-token: 'write'
    steps:
```

Github Checkout is in our use case not relevant. This package was used in the example pipeline on the github account from Google[11].

[11] https://github.com/google-github-actions/setup-gcloud/blob/main/example-workflows/gke/.github/workflows/gke.yml

The terraform package is used in the first pipeline. It can be included with the uses key. In the later stages of the pipeline a waiting function is needed and because there is no built-in, we use the wait-action package from Jake Jarvis.

```
  - name: Checkout
    uses: actions/checkout@v3
  - name: Terraform
    uses: hashicorp/setup-terraform@v2

   - name: Sleep
     uses: jakejarvis/wait-action@master
```

For our pipeline is a connection to Google Cloud needed. The installation of GCP is split in the part authentication and setup. For the setup are the credentials needed, which are stored as a secret in Github. The credentials can be downloaded via the IAM section in Google Cloud. For the setup part are no credentials needed.

```
    - id: 'auth'
      name: 'Authenticate to Google Cloud'
      uses: 'google-github-actions/auth@v0'
      with:
        credentials_json: '${{ secrets.GCP_CREDENTIALS }}'

    # Setup gcloud CLI
    - name: Set up Cloud SDK
      uses: google-github-actions/setup-gcloud@v0
    - name: 'Use gcloud CLI'
      run: 'gcloud info'
```

For our second pipeline is a connection to a public docker repository needed. Therefore will be the docker packaged with Username and Access Token initialized and automatically logged in.

```
      - name: Login to Docker Hub
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKER_HUB_USERNAME }}
          password: ${{ secrets.DOCKER_HUB_ACCESS_TOKEN }}
```

4.1 Create Cluster, install OpenFaas and upload function

The first step in the creation of the cluster is the initialization. We needed to add the lock equals false flag, because our pipeline got interrupted in the execution in later stages and the state management file could not be unlocked by the process. After the initialization, the defined components will be applied to Google Cloud. Here we use the auto-approve flag, because in the normal workflow the apply command must be approved by the user, which is not possible in a pipeline.

```
    - name: Terraform Init
      id: init
      run: terraform init -lock=false

    - name: Terraform Plan
      id: plan
      run: terraform plan -no-color -lock=false
      continue-on-error: true

    - name: Terraform Apply
      run: terraform apply -auto-approve -lock=false
```

After the cluster is created, we are setting a kubectl entry to connect to the cluster.

```
 - name: Get kubectl Connection
                  run:   gcloud   container   clusters   get-credentials
crypto-parser-350713-gke --region  europe-west1
      continue-on-error: true
```

The next step is the download of the CLI of the arkade marketplace. With the arkade marketplace is it simple to install Openfaas on the cluster. In the pipeline, we are saying arkade to install a load-balancer for Openfaas, which connects automatically to the Google Cloud Systems for an public IP.

```
    - name: Install Arkade
      run: curl -sLS https://get.arkade.dev | sudo sh

    - name: Test Arkade
      run: arkade --help

    - name: Install Openfaas
      run: arkade install openfaas --load-balancer
      continue-on-error: true
```

Before we can retrieve the public IP and the Password from the cluster, we need to wait, because the creation time in the cluster is longer than the command line presents. After that defined period we will retrieve the IP and password and save them in Github Environment Variables for later use.

```
   - name: Sleep 2 min
      run: sleep 120s

   - name: Get IP
        run: echo GATEWAY_IP=$(kubectl get service gateway-external -n
openfaas    -o    jsonpath="{.status.loadBalancer.ingress[0].ip}")    >>
```

```
$GITHUB_ENV
      continue-on-error: true
    - name: Get Password
       run: echo PASSWORD=$(kubectl get secret -n openfaas basic-auth -o
jsonpath="{.data.basic-auth-password}"  |  base64  --decode;  echo)  >>
$GITHUB_ENV
      continue-on-error: true
```

For the connection is the Openfaas CLI-tool needed and will be installed first. The connection to Openfaas will be down with the retrieved IP and password. As the last step a function from the Openfaas store will be as Proof of Concept deployed.

```
    - name: Download open-faas cli
      run: curl -sSL https://cli.openfaas.com | sudo -E sh
      continue-on-error: true

    - name: Connect to Openfaas
       run: echo -n ${{env.PASSWORD}} | faas-cli login --username admin
--password-stdin --gateway http://${{env.GATEWAY_IP}}:8080
      continue-on-error: true

    - name: Push Test function
                run:  faas-cli  store  deploy  'NodeInfo'  --gateway
http://${{env.GATEWAY_IP}}:8080
      continue-on-error: true
```

4.2 Add function to existing cluster

After the initialization phase the pipeline sets a kubectl entry for the connection.

```
      - name: Get kubectl Connection
                  run:  gcloud  container  clusters  get-credentials
crypto-parser-350713-gke --region europe-west1
        continue-on-error: true
```

After the entry the next step is to receive the IP and password.

```
      - name: Get IP
         run: echo GATEWAY_IP=$(kubectl get service gateway-external -n
openfaas    -o    jsonpath="{.status.loadBalancer.ingress[0].ip}")    >>
$GITHUB_ENV
        continue-on-error: true

      - name: Get Password
        run: echo PASSWORD=$(kubectl get secret -n openfaas basic-auth -o
jsonpath="{.data.basic-auth-password}"  |  base64  --decode;  echo)  >>
$GITHUB_ENV
```

```
        continue-on-error: true
```

The Openfaas CLI will be downloaded and with the IP and password to Openfaas in the cluster connected.
The IP will be given as a parameter in front of the upload command and will be placed in the test.yml file in the section provider and gateway. The IP changes from upload to upload in the cloud and can't be hard coded in the yaml file.

```
      - name: Download open-faas cli
        run: curl -sSL https://cli.openfaas.com | sudo -E sh
        continue-on-error: true

      - name: Connect to Openfaas
         run: echo -n ${{env.PASSWORD}} | faas-cli login --username admin
--password-stdin --gateway http://${{env.GATEWAY_IP}}:8080
        continue-on-error: true
      - name: Upload function
        run: URL=${{env.GATEWAY_IP}} faas-cli up -f test.yml
```