Operating Manual

Ingunn Birta Ómarsdóttir

Svanfríður Júlía Steingrímsdóttir

Telma Guðmundsdóttir

B.Sc. Computer Science

B.Sc. Software Engineering

Instructor: Gunnar Sigurðsson

Examiner: Sigurjón Ingi Garðarsson

May 12ᵗʰ, 2023

Table of Contents

Table of Figures

1 Introduction

This is the operating manual created for the project Exploration of Kubernetes-based FaaS Solutions. This manual provides all the necessary information for developers who want to create their own Kubernetes-based FaaS solutions. It will go over all necessary setup along with information for running the web application created in this project both locally and through AWS, and how to create, edit and delete functions.

2 AWS

Before you can start working with the frontend and backend there are a few things that must be set up in AWS.

2.1 Create Account

First, you must create an account with [Amazon Web Services](). If you already have an account, you can move to the next step.

2.2 IAM User

To be able to make use of the Amazon Web Services Command Line Interface, or AWS CLI, you must create an IAM User and assign all needed permissions and policies to it. To make the permission and policy assigning easier, create a user group that includes all necessary permissions. That way, when the user is created, they can be assigned to the group and get all permissions associated with that group.

1. Navigate to the IAM console.

2. Click on "User groups".

3. Click "Create group" and provide a descriptive name for the user group.

4. Attach the following permissions to the group:

    a. AmazonDynamoDBFullAccess

    b. AmazonEC2ContainerRegistryFullAccess

    c. AmazonEC2FullAccess

       d. AmazonEKSClusterPolicy

       e. AmazonEKSServicePolicy

       f. AmazonEKSWorkerNodePolicy

       g. AmazonVPCFullAccess

       h. AWSCloudFormationFullAccess

       i. IAMFullAccess

       j. SecretsManagerReadWrite

5. Click "Create group".

6. Navigate to the "Users" tab and click "Add users".

7. Provide a name for the user and add the user to the newly created user group.

8. Navigate to the user and then to the "Security credentials" tab and click "Create access key".

9. Choose "Command Line Interface" and then click "Create access key".

10. Download the CSV file including the access key and secret access key. Keep track of this file as it will be needed later.

2.3 AWS Command Line Interface

2.3.1 Installation

Download AWS CLI. To see if you already have the AWS CLI installed or if you want to see whether the installation was successful you can use the terminal to run: "aws --version". This will show you what version of the AWS CLI you have. You should install the latest version.

2.3.2 Configuration

Using the AWS CLI requires authentication and configuration to ensure access to your AWS resources.

To authenticate to the CLI, run the command: "aws configure". The command will prompt you to input the access key, secret access key, region, and output format. Input the access key and secret access key that was stored in the CSV file from the previous steps found in the chapter IAM User. For region, insert your region, i.e., "eu-west-1". It is very important to specify the

correct region and have the same region as your resources, otherwise you won't be able to access them. For output, specify the output format you prefer, for example, "json" or "table". To see whether the configuration was successful you can run the command: "aws iam list-access-keys". You should see that the result matches the user you just created.

2.4   Monitoring

The main monitoring of the Kubernetes EKS cluster is done by using the Kubernetes command line tool, kubectl. Please refer to the kubectl installation referenced in the OpenFaaS Installation chapter. To be able to use kubectl on an existing EKS cluster, this command is used: "aws eks update-kubeconfig --region region-code --name <cluster name>". Replace "<cluster name>" with the name of your cluster.

This sets the current context of kubectl to the EKS cluster. As mentioned before, kubectl is used for monitoring cluster resources like nodes and pods. Kubernetes has a handy cheat sheet including all monitoring and log commands.

3   Frontend

3.1   Installation

Download NodeJS to get access to npm. To see if you already have npm installed or if you want to see whether the installation was successful you can use the terminal to run: "npm -v". This will show you what version of npm you have. You should install the latest version.

Download Git to get access to git commands. To see if you already have git installed or if you want to see whether the installation was successful you can use the terminal to run: "git version". This will show you what version of git you have. You should install the latest version.

Clone the git repository. To clone using HTTPS, open the terminal and run the command:

"git clone https://github.com/Svanfridurjulia/Exploration-of-Kubernetes-based-FaaS-solutions.git".

Install all required dependencies stored in the package.json file. Navigate to the *react_frontend/web_app* directory and run: "npm install".

3.2   Running Locally

Once you have installed the required dependencies as described in the above chapter, the frontend can be run. Navigate to the *react_frontend/web_app* directory and run: "npm start".

3.3   Running on AWS

3.3.1 Build and Push the Image

Before continuing here, you must run the Terraform file. For this go to the Terraform chapter.

These instructions refer to how to manually build and push the image for the web application to the ECR repository. This can also be done automatically using GitHub Actions.

Please refer to the Docker Installation chapter and make sure you are running Docker to be able to perform the Docker commands. Please also refer to the kubectl installation referenced in the OpenFaaS Installation chapter.

In the following steps, there are some instances where part of the command is set in angle brackets. These parts should be replaced with your own values.

1. Navigate to the web application folder containing the Dockerfile and run:

   a. "docker build -t my-react-app"

2. Tag the image with the URI value for the ECR repository for the web application and the version of the image you are pushing with the following command:

   a. "docker tag my-react-app:latest <URI>/react-web-app:web-app-<version>"

3. Push the image to the repository:

   a. "docker push <URI>/react-web-app:web-app-<version>"

4. Make sure the context is set to the newly created EKS. To find and set the current context:

   a. "kubectl config view"

   b. Find the context name for the newly created cluster.

   c. "kubectl config use-context <context name>"

5. Update the image for the deployment:

   a. "kubectl set image deployment/my-app my-app=<URI>/react-web-app:web-app-<version>"

 6. Check the status of the deployment:

   a. "kubectl rollout status deployment/my-app"

3.4 Programming Rules

The frontend is written in JavaScript with ReactJS, a JavaScript Syntax Extension, and CSS. The following programming rules are for the frontend of the project. These rules make sure that the code stays consistent and understandable for future developers.

The web application should be broken down into components. Each component folder and JavaScript Syntax Extension file should be named in PascalCase and should be as descriptive as possible. Components should be named in English.

Variables and functions should be named in CamelCase and should be as descriptive as possible. Longer functions should have descriptive comments to explain their purpose. Names and comments should be written in English.

3.5 Tests

For each component added to the web application there should be a test file that tests the component and its features. The test file should be located in the component's folder and named after the component. For example, with the component ExampleComponent.jsx, there should be a test file called ExampleComponent.test.js. To run all tests for the frontend, navigate to the *react_frontend/web_app* directory and run: "npm test".

4 Backend

4.1 Installation

4.1.1 Docker Installation

Download [Docker.](#) To see if the installation was successful navigate to the terminal run: "docker info". You should install the latest version.

4.1.2 OpenFaaS Installation

1. Arkade:

   a. Install <u>Arkade</u>, a Kubernetes app installer.

2. Kubectl:

   a. "arkade get kubectl"

3. Kind:

   a. Move to the directory *.arkade/bin* by running: "cd .arkade/bin".

   b. "arkade get kind"

4. Faas-cli:

   a. Move back to root by running: "cd".

   b. "arkade get faas-cli"

5. OpenFaaS:

   a. "arkade install openfaas"

6. Cron-connector (optional):

   a. To have background processing functions and to be able to trigger them on a time basis, cron-connector must be installed.

   b. "arkade install cron-connector"

4.1.3 Argo CD Installation

Download <u>Argo CD</u>. To see if you already have Argo CD installed or if you want to see whether the installation was successful you can use the terminal to run: "argocd version". You should install the latest version.

4.2   Programming Rules

The functions are written in Golang, Python, and Node. The following programming rules are for the functions so the code stays consistent and understandable for future developers.

In the folder *functions* there is a subfolder for each function with the function's name along with a YAML file with the same name. The function names should be lowercase, separated by a hyphen, and be in English.

Variables and functions should be named in CamelCase and should be as descriptive as possible. Longer functions should have descriptive comments to explain their purpose. Names and comments should be written in English.

4.3   Set Up a Local OpenFaaS Registry

Open a terminal in the folder where you want to set up your local registry. In the terminal, run:

1. "mkdir openfaas"

2. "cd openfaas/"

3. "wget https://kind.sigs.k8s.io/examples/kind-with-registry.sh"

4. "chmod +x kind-with-registry.sh"

If Docker is not already running, you should run: "systemctl start docker".

Make sure that you are inside the *openfaas* folder and create a Kubernetes cluster using a Docker container with the command: "./kind-with-registry.sh".

See the context that kubectl is currently using with the command: "kubectl config current-context".

Check if the node is ready with the command: "kubectl get nodes".

Check if the services inside the Kubernetes cluster are running with the command: "kubectl get pods -n kube-system".

Wait and watch until all OpenFaaS pods are running with the command: "kubectl get pods -n openfaas -w".

Once this is complete your registry is ready.

4.4   How to Create a Function

To create a new function in OpenFaaS you must use one of the templates or create your own. To see available templates run the command: "faas-cli template store list".

Pull one of the templates from the list. For example, if you want to use golang-middleware you run the command: "faas-cli template store pull golang-middleware".

Creating a new function creates a folder and YAML file with the function's name. You do this with the command: "faas-cli new <function name> --lang golang-middleware". Replace <function name> with the name you want to use.

If running locally, the gateway in the YAML file should match the gateway you see with the command: "kubectl get svc -n openfaas". If running on EKS cluster in AWS, please refer to the Push Functions to ECR Repository chapter.

Create a Docker image out of the configuration in the YAML file with the command: "faas-cli build -f <function name>.yml".

If the image was successfully created, it should be shown when running the command: "docker images".

Push the image to the Docker registry with the command: "faas-cli push -f <function name>.yml".

Check the port needed for the next step if you are running locally. Look for the gateway port needed with the command: "kubectl get svc -n openfaas".

Open a new terminal to port forward OpenFaaS Gateway Service for localhost with the command: "kubectl port-forward -n openfaas svc/gateway 8080:8080". This step can be skipped if creating a function to run on an EKS cluster in AWS.

Deploy the function with the command: "faas-cli deploy -f <function name>.yml".

To see the status of the function run the command: "kubectl get pods -n openfaas-fn".

If running locally, the function should now be available on the OpenFaaS Dashboard. It can be used with the dashboard by pressing the basego function, writing some request body, and pressing the "invoke" button. The function can also be invoked in the terminal by running the command: "echo <request body> | faas-cli invoke <function name>".

4.4.1 Background Processing

To create an OpenFaaS function for scheduled background processing, cron-connector must be installed, see the OpenFaaS Installation chapter. The corresponding YAML file for the function must be edited to be able to run the function on a scheduled basis. Under the image line, add the following code:

annotations:

topic: cron-function

schedule: "* * * * *"

The text in the schedule line can be changed to fit the wanted schedule. In the example code, the function runs every minute. See more information on how to write the CRON expressions here.

Figure 1: Adding scheduling to function

```
1    version: 1.0
2    provider:
3      name: openfaas
4      gateway: http://127.0.0.1:8080
5    functions:
6      get-weather:
7        lang: golang-middleware
8        handler: ./get-weather
9        image: telmagud/get-weather:latest
10        annotations:
11          topic: cron-function
12          schedule: "* * * * *"
13
```

4.5   The OpenFaaS Dashboard

Open a browser and go to "localhost:8080". If everything is working correctly, there should be a pop-up to sign in. The username is "admin" and the password is a secret that must be fetched with kubectl.
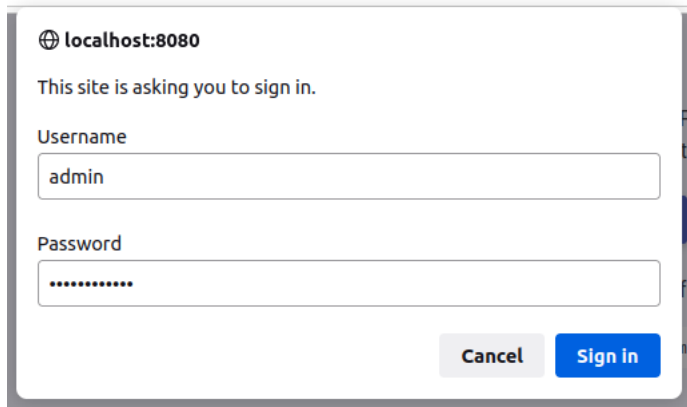
To see the secret, open a terminal and run the command: "kubectl get secret -n openfaas basic-auth -o json".

To decode and store the password run the command: "PASSWORD=$(kubectl get secret -n openfaas basic-auth -o jsonpath="{.data.basic-auth-password}" | base64 --decode; echo)".

To print the password for the OpenFaaS Dashboard to the terminal run: "echo $PASSWORD".

You must authenticate with faas-cli to communicate through the OpenFaaS service gateway. To do this run the command: "echo -n $PASSWORD | faas-cli login -s".

Figure 2: OpenFaaS Dashboard login

Figure 3: OpenFaaS Dashboard

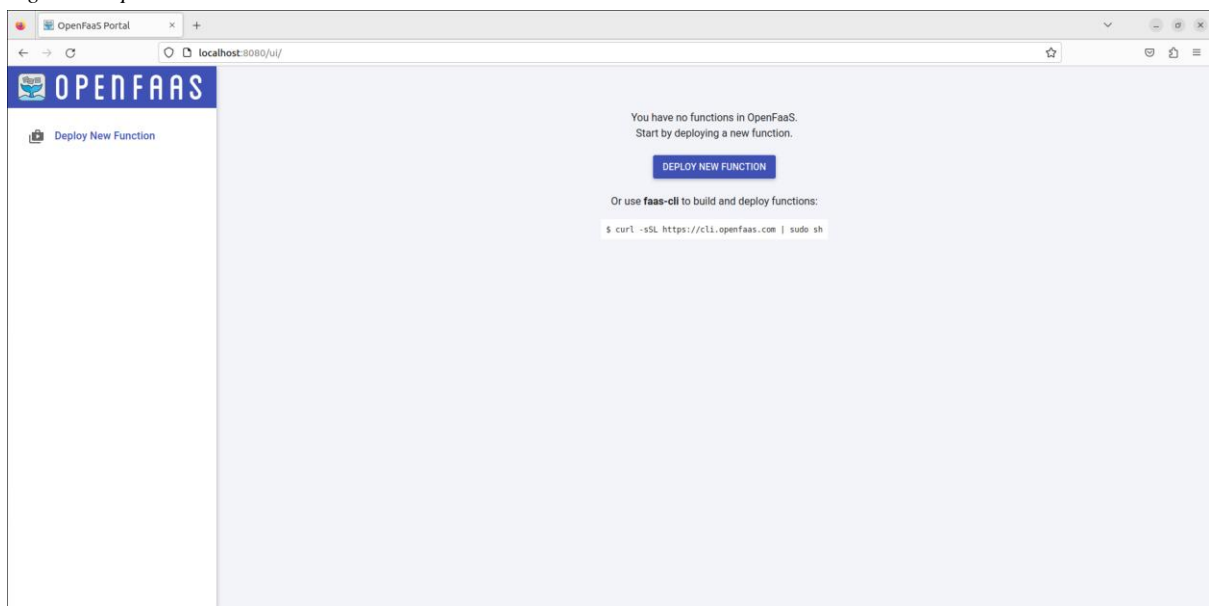## 4.6   Push Functions to ECR Repository

Edit the image value of the YAML file to point to the URI of the ECR repository to be used, "<URI>/faas-funciton-repository:<function name>-<version>". The gateway value should be the URL of the function's domain record configured as well as the correct port. This can be done using a code editor of your choice.

To push the image to the repository run the command: "faas-cli up -f <function name>.yml".

4.7   How to Edit a Function

It is possible to edit both the YAML file and the handler in the function's folder. After a function has been edited it has to be deployed again. Use the terminal and run:

1. "faas-cli build -f <function name>.yml"

2. "faas-cli push -f <function name>.yml"

3. "faas-cli deploy -f <function name>.yml"

These commands can also be run as one by using the command: "faas-cli up -f <function name>.yml".

4.8 How to Delete a Function

To list all the OpenFaaS functions go to the terminal and run the command: "faas-cli list".

To delete a function run the command: "faas-cli remove <function name>".

4.9 Tests

For each added function, unit tests should be created to make sure that the function works as intended. Unit tests and how they are run are different depending on what programming language is being used. As the backend is made up of many different functions of various programming languages, the way the tests are executed may differ. For example, with a Golang function you have to navigate to the function's folder and run: "go test ./". However, you can test all Python functions simultaneously by running: "pytest". The position of the test file for each function might also vary slightly, but the Golang and Python tests are located in each function's folder.

4.10 Possible Errors

When running locally, there is a possibility that when pushing a function, the following error will appear:

"Unable to push one or more of your functions to Docker Hub:

- <function name>

You must provide a username or registry prefix to the Function's image such as user1/function1"

To resolve this error, change the image line in the corresponding YAML file. This can be done using a code editor. You must provide a username or registry prefix to the function's image such as "user/function:latest". When running OpenFaaS locally, it is necessary to be logged in

on Docker, and "user" refers to the Docker username. When using OpenFaaS with AWS, "user" is replaced by the ECR registry URI, the function is replaced by the repository name for the function and a tag is added. This would look like: "<URI>/<repository name>:<function name>-<version>".

5 Terraform

5.1 Installation

Download the Terraform CLI. To see if you already have Terraform installed or if you want to see whether the installation was successful you can run: "terraform -help". You should install the latest version.

Follow the steps of the AWS chapter and make sure all resources specified there have been created.

5.2 Build Infrastructure

By using Terraform, it is possible to write code to achieve some wanted end state for the infrastructure.

Create a directory where the Terraform files should be. Add a *main.tf* file to the directory. There are many blueprints online that show what can be added to the *main.tf* file, depending on what you want Terraform to do. Terraform will consider all files in the directory with the ending .tf.

Once a *main.tf* file is created, Terraform can be initialized. That can be done by running the command: "terraform init".

If the initialization is successful, you should run the command: "terraform plan". This will show what changes will be made and will insert the needed input variables.

If the changes presented by "terraform plan" are as expected, run: "terraform apply".

For this project we also set up the file *variables.tf* which is used to hold the input variables for the *main.tf* file. When running "terraform apply" you can specify the variables like so: "terraform apply -var '<variable name>=<variable value>'".

You can also do "terraform plan" and insert the values there and then run: "terraform apply".

After running "terraform apply", Terraform will show what resources will be created and you must manually confirm the creation of the resources.

After "terraform apply" has finished executing, Terraform will create a state file which holds information about all resources created and works as a dictionary to be able to keep track if any of the resources are deleted.

5.3 Change Infrastructure

To make changes to the existing infrastructure created by Terraform you can simply change the code in the Terraform file and run the Terraform init, plan, and apply commands.

5.4 Destroy Infrastructure

To destroy the resources that have been created, run the command: "terraform destroy".

Disclaimer: It is possible to run into errors when running "terraform destroy". This can happen because of stuck dependencies that Terraform is unable to delete. Luckily, Terraform is well documented and by using a search engine these errors should be resolved fairly quickly.

5.5 Terraform in This Project

For this project we set up a Terraform file which sets up the whole infrastructure in AWS for running our web application and functions. The file includes multiple providers, resources and modules which all have different configurations.

5.5.1 Walkthrough of the Code

5.5.1.1 Providers

The file describes three providers: AWS, Kubernetes, and Helm. These providers are used to create the resources since they communicate with external APIs and services. Each provider block sets up the needed configuration for this communication.

5.5.1.2 Locals

The *Locals* block defines the local variables that will be used within the configuration file. In this block you can define the name of the cluster to be created and set the region you are using in AWS.

```
locals {
 name   = "FinalEKSCluster"
 region = "eu-west-1"


 cluster_version = "1.24"


 vpc_cidr = "10.0.0.0/16"
 azs      = slice(data.aws_availability_zones.available.names, 0, 3)


 tags = {
  Blueprint   = local.name
  GithubRepo = "github.com/aws-ia/terraform-aws-eks-blueprints"
 }
}
```

5.5.1.3 Global Data

The global data is made up of three data blocks which describe values that are used multiple times throughout the file. Here you can set the value of the domain you have registered and want to use.

```
data "aws_route53_zone" "your_domain" {
 name = "fabulousasaservice.com"
}
```

5.5.1.4 Individual Resources

Individual resources include all the stand-alone or smaller resources being created. Both of the ECR repositories are created here and you can change the name of the repositories to fit your wants.

```
resource "aws_ecr_repository" "faas_function_repository" {
 name = "faas-function-repository"
}

resource "aws_ecr_repository" "react_web_app" {
 name = "react-web-app"
}
```

This also includes setting up the *admin-email-password* secret to be stored in the AWS Secrets Manager. The secret is accessed in the *send-email* function, but this part can be commented out if you do not feel the need to create a secret for that function. First, the secret is created and then the value of the secret is set. The *recovery_window_in_days* flag is set to 0 which means that the AWS Secrets Manager will keep a copy of the secret for 0 days after it has been deleted. This value can be changed.

```
resource "aws_secretsmanager_secret" "email_password" {
 name = "admin-email-password"
 recovery_window_in_days = 0                    19


}

resource "aws_secretsmanager_secret_version" "email_password" {
 secret_id     = aws_secretsmanager_secret.email_password.id
 secret_string = jsonencode({
  password = var.email_password
 })
}
```

The permissions for the worker nodes are also set in this part. First, the policy is created and then the policy is assigned to the worker node group. To add more permissions to the policy you can add another object to the *Statement* list which describes the permissions. The permissions that are already described give the worker nodes full access to DynamoDB, allow the fetching of secrets in the AWS Secrets Manager and gives EC2 permissions which allow communications with other EC2 resources.

```
resource "aws_iam_policy" "worker_node_permissions" {
 name        = "worker-node-permissions"
 description = "Policy for EKS worker nodes to access DynamoDB and Secrets Manager"

 policy = jsonencode({
  Version = "2012-10-17"
  Statement = [
   {
    Action = [
     "dynamodb:*"
    ]
    Effect   = "Allow"
    Resource = "*"
   },
   {
    Action = [
     "secretsmanager:GetSecretValue"
    ]
    Effect   = "Allow"
    Resource = "*"
   },
```

```
    {
      Action = [
        "ec2:Describe*",
        "ec2:AttachVolume",
        "ec2:DetachVolume",
      ]
      Effect   = "Allow"
      Resource = "*"
    }
  ]
})
}
resource "aws_iam_role_policy_attachment" "worker_node_permissions_attachment" {
  policy_arn = aws_iam_policy.worker_node_permissions.arn
  role = replace(module.eks.eks_managed_node_groups["initial"].iam_role_arn, "arn:aws:iam::112172658395:role/",
"")
}
```

5.5.1.5   VPC

The VPC module creates the needed network resources to be able to deploy resources on AWS using Terraform. The current configuration is minimal and provides reasonable default values, however, this can of course be changed to meet the needed requirements.

```
module "vpc" {
 source  = "terraform-aws-modules/vpc/aws"
 version = "~> 4.0"

 name = local.name
 cidr = local.vpc_cidr

 azs            = local.azs
 private_subnets = [for k, v in local.azs : cidrsubnet(local.vpc_cidr, 4, k)]
 public_subnets  = [for k, v in local.azs : cidrsubnet(local.vpc_cidr, 8, k + 48)]

 enable_nat_gateway = true
 single_nat_gateway = true

 public_subnet_tags = {
   "kubernetes.io/role/elb" = 1
 }

 private_subnet_tags = {
   "kubernetes.io/role/internal-elb" = 1
 }

 tags = local.tags
}
```

5.5.1.6 EKS

This module creates and configures the EKS cluster with managed node groups in the specified VPC and subnets. This configuration sets the cluster endpoint as public as setting it as private can include more complicated configurations and authorizations. The instance type of the worker nodes is set to *t3.medium* and specifies the *min_size*, *max_size* and *desired_size*. This can be changed to meet the needed requirements but for running our project the *t3.medium* instance type is the minimum instance type required to be able to run everything. The *min_size* describes the minimum number of nodes that should be running, the *max_size* the maximum number of nodes that are allowed to be running and the *desired_size* the desired number of nodes that should be running. By setting the *min_size* to 0 we were able to stop all running instances to make sure they were not running unnecessarily and racking up avoidable costs.

```
module "eks" {
 source  = "terraform-aws-modules/eks/aws"
 version = "~> 19.12"

 cluster_name    = local.name
 cluster_version = local.cluster_version

 vpc_id     = module.vpc.vpc_id
 subnet_ids = module.vpc.private_subnets

 cluster_endpoint_public_access = true
 cluster_endpoint_public_access_cidrs = ["0.0.0.0/0"]

 eks_managed_node_groups = {
  initial = {
    instance_types = ["t3.medium"]

    min_size     = 0
    max_size     = 7
    desired_size = 5

  }
 }

 tags = local.tags
}
```

5.5.1.7 DynamoDB

Two DynamoDB tables are configured and are used to create users and posts. Here you might want to change the *billing_mode* to something that fits you. The *hash_key* describes the primary key for the table.

```
resource "aws_dynamodb_table" "table1" {
 name        = "users"
 billing_mode  = "PAY_PER_REQUEST"
 hash_key     = "user_id"

 attribute {
  name = "user_id"
  type = "N"
 }

 tags = {
  Environment = "dev"
 }
}
```

5.5.1.8 OpenFaaS

To set up and configure OpenFaaS on the cluster multiple blocks are used. To keep the cluster cleaner and easier to manage two namespaces are created which will hold the OpenFaaS deployments and services. The first namespace is *openfaas*, which will include all the management services and deployments. This includes metrics collection, a gateway to invoke the functions, alert handling, and task queueing. The second namespace is *openfaas-fn*, which will hold all the functions that have been deployed.

To deploy the OpenFaaS platform to the cluster the resource *helm_release* is used. In this configuration the namespaces are specified and the load balancer for invoking the functions is created.

```
resource "helm_release" "openfaas" {
depends_on = [
  kubernetes_namespace.openfaas  ]
 name      = "openfaas"
 repository = "https://openfaas.github.io/faas-netes/"
 chart     = "openfaas"

 namespace = "openfaas"

 set {
  name  = "faasnetes.imagePullPolicy"
  value = "IfNotPresent"
 }

 set {
  name  = "functionNamespace"
  value = "openfaas-fn"
 }
```

```
set {
  name  = "generateBasicAuth"
  value = "true"
}

set {
  name  = "serviceType"
  value = "LoadBalancer"
}

}
```

To be able to configure the routing of the domain for invoking the functions a few blocks are needed. First, the external hostname of the load balancer created is extracted and written to a text file. The null resource is used to execute the command needed. Then a data block is used to be able to extract the contents of the file that was written to. Lastly, the *aws_route53_record* is used to create a new CNAME record which points to the external hostname of the load balancer.

```
resource "null_resource" "openfaas_lb" {
 depends_on = [
   helm_release.openfaas,
   kubernetes_namespace.openfaas,
 ]

 provisioner "local-exec" {
   command = <<-EOC
     kubectl get svc gateway-external -n openfaas \
       --token="${data.aws_eks_cluster_auth.this.token}" \
       --server="${module.eks.cluster_endpoint}" \
       --insecure-skip-tls-verify=true \
       -o jsonpath='{.status.loadBalancer.ingress[0].hostname}' > lb_hostname.txt
   EOC
 }
}


data "local_file" "lb_hostname" {
 depends_on = [null_resource.openfaas_lb]
 filename   = "lb_hostname.txt"
}

resource "aws_route53_record" "openfaas_cname" {
 zone_id = data.aws_route53_zone.your_domain.id
 name    = "functions.fabulousasaservice.com"
 type    = "CNAME"
 ttl     = "300"
 records = [data.local_file.lb_hostname.content]
}
```

5.5.1.9   Argo CD

Before the Argo CD add-on is set up a random password is generated, and a secret is created to store the password. This password will be used for the admin account of Argo CD. The module for the Argo CD set up includes quite a bit of configuration. This includes the URL of the GitHub repository, which branch it should monitor, the path to the folder it should monitor, the namespace, synchronization options and additional add-ons.

```
module "eks_blueprints_kubernetes_addons" {
 source = "git::https://github.com/aws-ia/terraform-aws-eks-blueprints.git//modules/kubernetes-addons"


 eks_cluster_id      = module.eks.cluster_name
 eks_cluster_endpoint = module.eks.cluster_endpoint
 eks_oidc_provider   = module.eks.oidc_provider
 eks_cluster_version  = module.eks.cluster_version

 enable_argocd = true

 argocd_helm_config = {
 set_sensitive = [
  {
    name  = "configs.secret.argocdServerAdminPassword"
    value = random_password.argocd.result
  }
 ]
 set = [
   {
    name  = "controller.repoServer.timeoutSeconds"
    value = "60" # Set the desired timeout value in seconds
   }
  ]
}

 keda_helm_config = {
  values = [
   {
    name  = "serviceAccount.create"
    value = "false"
   }
  ]
 }
 argocd_manage_add_ons = true # Indicates that ArgoCD is responsible for managing/deploying add-ons
 argocd_applications = {
  addons = {
   path          = "chart"
   repo_url       = "https://github.com/aws-samples/eks-blueprints-add-ons.git"
   add_on_application = true
  }
```

```
  workloads = {
    path            = "envs/dev"
    repo_url          = "https://github.com/aws-samples/eks-blueprints-workloads.git"
    add_on_application = false
  }
  openfaas = {
    path            = "functions/charts/openfaas-functions/"
    repo_url          = "https://github.com/Svanfridurjulia/Exploration-of-Kubernetes-based-FaaS-solutions.git"
    target_revision    = "main"
    add_on_application = false
    project_name       = "openfaas"
    namespace          = "openfaas-fn"
    sync_policy        = "automated"
    sync_options       = {
      validate = true
      prune    = true
    }
  }
}

# Add-ons
enable_amazon_eks_aws_ebs_csi_driver = false
enable_aws_for_fluentbit            = false

aws_for_fluentbit_create_cw_log_group = false
enable_cert_manager              = false
enable_cluster_autoscaler          = false
enable_karpenter               = false
enable_keda                 = false
enable_metrics_server            = false
enable_prometheus               = true
enable_traefik               = false
enable_vpa                 = false
enable_yunikorn               = false
enable_argo_rollouts             = true

tags = local.tags
}
```

5.5.1.10 Web Application

To deploy the web application to the cluster we need both a Kubernetes deployment and
Kubernetes service. These blocks specify the configurations of the web application, such as
how many replicas should be created and the URL of the image. You should update the image
tag in the Kubernetes deployment to match the URI of your ECR registry and then set the name
of the ECR repository that was created. The Kubernetes service sets up a load balancer for the
web application which will be used to fetch the web application.

```
resource "kubernetes_deployment" "my-app" {
 metadata {
  name = "my-app"
 }

 spec {
  replicas = 3

  selector {
   match_labels = {
    app = "my-app"
   }
  }

  template {
   metadata {
    labels = {
     app = "my-app"
    }
   }

   spec {
    container {
     name  = "my-app"
     image = "112172658395.dkr.ecr.eu-west-1.amazonaws.com/react-web-app:web-app-v1"
     port {
      container_port = 3000
     }
    }
   }
  }
 }
}

resource "kubernetes_service" "my-app" {
 metadata {
  name = "my-app"
  annotations = {
   "service.beta.kubernetes.io/aws-load-balancer-backend-protocol" = "http"
  }
 }
```

```
spec {
  selector = {
    app = "my-app"
  }

  type = "LoadBalancer"

  port {
    protocol    = "TCP"
    port        = 80
    target_port = 3000
  }
}
}
```

To be able to configure the routing of the domain for accessing the web application three blocks are needed. First, the external hostname of the load balancer created is extracted and written to a text file. The null resource is used to execute the command needed. Then a data block is used to be able to extract the contents of the file that was written to. Lastly, the aws_route53_record is used to create a new CNAME record which points to the external hostname of the load balancer.

```
resource "null_resource" "my_app_lb" {
  depends_on = [
    kubernetes_deployment.my-app
  ]

  provisioner "local-exec" {
    command = <<-EOC
      kubectl get svc my-app \
        --token="${data.aws_eks_cluster_auth.this.token}" \
        --server="${module.eks.cluster_endpoint}" \
        --insecure-skip-tls-verify=true \
        -o jsonpath='{.status.loadBalancer.ingress[0].hostname}' > my_app_lb_hostname.txt
    EOC
  }
}

data "local_file" "my_app_lb_hostname" {
  depends_on = [null_resource.my_app_lb]
  filename   = "my_app_lb_hostname.txt"
}
resource "aws_route53_record" "app_cname" {
  zone_id = data.aws_route53_zone.your_domain.id
  name    = "web-app.fabulousasaservice.com"
  type    = "CNAME"
  ttl     = "300"
  records = [data.local_file.my_app_lb_hostname.content]
}
```

5.5.1.11   Grafana

A separate namespace is created for Grafana and used for the deployment of the helm resource. The helm resource creates a load balancer which will be used to access Grafana.

```
resource "kubernetes_namespace" "grafana" {
 metadata {
  name = "grafana"
 }
}
resource "helm_release" "grafana" {
 name      = "grafana"
 namespace  = "grafana"
 repository = "https://grafana.github.io/helm-charts"
 chart     = "grafana"

 values = [
  <<-EOT
  service:
    type: LoadBalancer
  EOT
 ]
}
```

To be able to configure the routing of the domain for accessing Grafana three blocks are needed. First, the external hostname of the load balancer created is extracted and written to a text file. The null resource is used to execute the command needed. Then a data block is used to be able to extract the contents of the file that was written to. Lastly, the aws_route53_record is used to create a new CNAME record which points to the external hostname of the load balancer.

```
resource "null_resource" "grafana_lb" {
 depends_on = [
  helm_release.grafana,
  kubernetes_namespace.grafana,
 ]

 provisioner "local-exec" {
  command = <<-EOC
    kubectl get svc grafana -n grafana \
     --token="${data.aws_eks_cluster_auth.this.token}" \
     --server="${module.eks.cluster_endpoint}" \
     --insecure-skip-tls-verify=true \
     -o jsonpath='{.status.loadBalancer.ingress[0].hostname}' > grafana_lb_hostname.txt
  EOC
 }
}
```

```
data "local_file" "grafana_lb_hostname" {
 depends_on = [null_resource.grafana_lb]
 filename   = "grafana_lb_hostname.txt"
}


resource "aws_route53_record" "grafana_cname" {
 zone_id = data.aws_route53_zone.your_domain.id
 name    = "grafana.fabulousasaservice.com"
 type    = "CNAME"
 ttl     = "300"
 records = [data.local_file.grafana_lb_hostname.content]
}
```

5.5.2 Build

To run this project's Terraform file, navigate to the Terraform folder and run the following commands:

1. "terraform init"

2. "terraform apply -var 'aws_access_key=<your aws access key>' -var 'aws_secret_key=<your aws secret access key>' -var 'email_password=<password for email>'"

The build might take some time to finish. After the Terraform build has finished you can make use of the Argo CD pipeline and GitHub Actions set up to make sure the images are pushed to the ECR repositories and the functions deployed to the cluster. Follow the instructions of the next chapter to achieve this.

6 CI/CD

6.1 Argo CD

As specified in the Terraform file Argo CD monitors the charts folder located in the GitHub Repository. This folder includes multiple YAML files which describe the functions to be deployed. To make Argo CD monitor a new function, follow these steps:

1. Make sure you have created this new function and that its code is located in the *functions* folder.

2. Navigate to the *values.yml* file and add a description block for the function to the bottom of the file.

```
<function name>:
  lang: <language>
  handler: <path to function folder>
  image: <URI>/faas-function-repository:<function name>-<version>
```

3. Navigate to the *templates* folder and add a YAML file for your function. The file should look like this:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: <function name>
 namespace: openfaas-fn
 labels:
   app: <function name>
spec:
 replicas: 1
 selector:
   matchLabels:
     app: <function name>
 template:
   metadata:
     labels:
       app: <function name>
   spec:
     containers:
     - name: <function name>
       image: <URI>/faas-function-repository:<function name>-<version>
       ports:
       - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
 name: <function name>
 namespace: openfaas-fn
spec:
 selector:
   app: <function name>
 ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
 type: ClusterIP
```

Should you want to use the predefined functions make sure all image tags are in the form: "<URI>/faas-function-repository:<function name>-<version>". To activate the Argo CD pipeline push the changes to the repository that is being monitored, as specified in the

Terraform file. Should you make changes to your code and want to deploy the new code make sure to update the version of the image. This is important so Argo CD pulls the newest image from the ECR repository.

6.2   GitHub Actions

For our project there are three workflows in place and they are located in the *.github/workflows* folder.

6.2.1 Deploying OpenFaaS Images

The first workflow builds and pushes the images for all of the OpenFaaS functions to the ECR repository. The workflow makes use of three GitHub secrets:

1. AWS access key, used to authenticate to AWS.
2. AWS secret access key, also used to authenticate to AWS.

The workflow uses "faas-cli build" and "docker push" to build and push the OpenFaaS images to the ECR repository. Before running this workflow, it is important to make sure that all image tags are in the correct format. It is also important to make sure the image tags in the function YAML files in the *functions* folder match the image tags in the *charts* folder. To run this workflow simply push to the branch specified in the workflow file.

6.2.2 Deploying the Web Application Image

This second workflow builds and pushes the image for the web application to the ECR repository. The workflow makes use of three GitHub secrets:

1. ECR Registry which is the URI of the ECR registry where the functions are being pushed.
2. AWS access key, used to authenticate to AWS.
3. AWS secret access key, also used to authenticate to AWS.

The workflow uses "docker build" and "docker push" to build and push the image to the ECR repository for the web application. Before running this workflow make sure to update the version of the image tag. To run this workflow simply push to the branch specified in the workflow file. After the image has been pushed you can run the following command to update the image being used for the web application deployment on the cluster: "kubectl set image deployment/my-app my-app=<URI>/react-web-app:web-app-<version>".

6.2.3 Tests

The last workflow runs the tests that have been implemented. Currently, there are three types of tests: Golang tests, pytests and React tests. When new code is pushed into the main branch via a pull request, each of these test types is run as a separate job, under one workflow.

7 Grafana

Once the Terraform has been run, Grafana will be up and running. You can access the dashboard through the domain specified in the Terraform file but in our case, it is grafana.fabulousasaservice.com. When you first navigate to the Grafana domain you will have to log in. The username for the account is "admin" but to get the password you can use the following command: "kubectl get secret grafana -n grafana -o jsonpath="{.data.admin-password}" | base64 –decode".

To set up the Prometheus data source in Grafana, navigate to the data source page in the administration dashboard. Click on "Add data source" and choose Prometheus.

1. Go to the terminal and type in this command to get the URL of the Prometheus service:

    a. "kubectl get service prometheus -n openfaas"

2. Copy the cluster-ip and in the URL input for the data source write:

    a. "http://<cluster-ip>:9090"

Scroll down and click "Save & test". This should save the data source and give you confirmation that the data source is configured correctly.

Navigate to the "Dashboards" page, click "New" and then "Import".

We will be using this predefined template: https://grafana.com/grafana/dashboards/3434-faas/

Fill in the needed information about the template.

After that you should be able to view the dashboard. From here you can customize the dashboard as you want and add even more data sources to get more metrics.