By now you might be wondering how do developers deal with so much complexity around containers, micro-services, schedulers, service meshes, etc… on top of their core knowledge about programming languages and software architectures. It sounds like way too much, huh? That’s exactly how they feel and they main reason why everyone looks for ways to let them focus just on their code.

A picture containing shirt

Description automatically generated

A very interesting approach to solve this challenge is **serverless (or Function-as-a-Service, aka FaaS)**, and it is based on the idea of having someone manage the required infrastructure for you. By decoupling software code from underlying infra you can focus on the software you are developing and not in how your containers or load balancers need to grow, or if you need to update your k8s cluster with the latest security patches. Now you only need to worry about the upper layers of the shared responsibility model.

A screenshot of a cell phone

Description automatically generated

Figure : AWS shared responsibility model

Most Cloud providers have an offering in the serverless arena (ie. [AWS Lambda](https://aws.amazon.com/lambda/), [Google Cloud Functions](https://cloud.google.com/functions), [Microsoft Azure Functions](https://azure.microsoft.com/services/functions/)) where you can just submit your code in one of the supported languages and they will take care of the rest. They will handle all the required microservices, have the orchestrator auto-scale them as needed, manage load-balancers, security, availability, caching, etc. And you will only pay for the number of times your code gets executed: if nobody uses your software you don’t pay a dime. Sounds really cool, huh? And it is. But *everything comes at a price*, and here you need to consider something called *lock-in*.

If you have worked with native offerings from your own Cloud provider, probably you have noticed that it is really easy to bring your data in and build your application there. But it is not that easy to migrate it out to a different environment when you need to. The main reason is that many of the service constructs you will use to implement your application are native to the specific provider you chose. So when the moment comes to move your workloads somewhere else you basically need to rebuild your app with similar constructs available from your new favourite provider. And that’s exactly the moment when everybody wonders: “wouldn’t it be cool to have a way to transparently migrate my code to a new environment?”. That’s what we call **portability**.

In the world of containers and microservices, portability is one of the main benefits of running your workloads on Kubernetes (aka k8s). No matter what provider you use, **k8s is k8s**. Whatever you build in a certain environment will be easily migrated to a different one. So why not using that same approach to serverless? And in fact, as long as k8s is a de-facto standard for DevOps practices, why not run a FaaS engine on top of it? That way it would benefit from k8s portability natively.

Well, there are a number of initiatives driving solutions exactly in this direction (FaaS on top of k8s) so in this blog I would like to review some of them with you and see how they compare. And as per the goal of this series we’ll evaluate them by getting our hands dirty to get some hands-on experience. You will have the final word on what works best for your environment!

A person with his mouth open

Description automatically generated

In this series we will explore 3 different FaaS over k8s solutions and actually put them to work in our evaluation environment:

* OpenFaaS
* Fission
* Kubeless

A screenshot of a cell phone

Description automatically generated

For our tests you can choose your favourite managed k8s offering, I will go with [Google Kubernetes Engine](https://cloud.google.com/kubernetes-engine) but please feel free to use the one you prefer. That’s exactly the point of using FaaS over k8s… portability. You may learn more about how to use GKE in [one of my past posts](https://blogs.cisco.com/developer/leverage-public-cloud).

It’s time to start testing some of the most interesting FaaS engines available out there.

[**OpenFaaS**](https://www.openfaas.com/)

First thing you will need to do is to [install OpenFaas CLI](https://github.com/openfaas/faas-cli) in your own workstation, so you can use it to build and deploy functions. In OSX for example you would install it with:

brew install faas-cli

The easiest way to install OpenFaaS is to use [arkade](https://github.com/alexellis/arkade), for example in OSX:

sudo curl -SLsf https://dl.get-arkade.dev/ | sudo sh

arkade install openfaas --load-balancer

Using the ‘--load-balancer’ option will give us an externally accessible IP address for the ‘gateway-external’ service (it might take a couple of minutes):

$ kubectl get svc gateway-external -n openfaas

NAME TYPE CLUSTER-IP **EXTERNAL-IP** PORT(S) AGE

gateway-external LoadBalancer 10.31.244.182 **35.241.131.142** 8080:32433/TCP 114s

For convenience let’s assign that IP address to the required URL variable:

export OPENFAAS\_URL=http://**35.241.131.142**:8080

You might also want to check all your pods in the *openfaas* namespace are running and readily available:

kubectl get pods -n openfaas

Time to login from your local workstation into the OpenFaaS deployment:

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

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

The OpenFaaS runtime engine is now setup and you are ready to start deploying your functions in it! For reference, function pods will be deployed in a different namespace named *openfaas-fn*.

For our first test let’s use something simple as [figlet](http://www.figlet.org/), a simple program to make large letters out of a provided message. Deploying it is as simple as running the following command:

faas-cli store deploy figlet

You may check it has been deployed with:

faas-cli list

Now let’s see it working:

echo "Hello Cisco" | faas-cli invoke figlet

It’s working! But… what happened? Well, basically the *deploy* command created a k8s deployment with a single replica in the *openfaas-fn* namespace:

$ kubectl get deployment -n openfaas-fn

NAME READY UP-TO-DATE AVAILABLE AGE

figlet 1/1 1 1 40m

And everytime you run a message through it the number of invocations grow:

$ faas-cli list

Function Invocations Replicas

figlet 2 1

You can also see the number of pod replicas based on the workload.

Please feel free to explore other available apps:

faas-cli store list

Hopefully you are now excited and want to start deploying your own code as functions! If that's the case you may use the OpenFaaS CLI to find templates for the most common programming languages, running the following command:

faas-cli template store list

To download them to a local *template* folder you just need to run:

faas-cli template pull

With that you can start creating your functions and see the available template options:

$ faas-cli new --list

Languages available as templates:

- csharp

- dockerfile

- go

- java11

- python

- node

Let’s create a simple one using the *node* template:

faas-cli new callme --lang node

This will create a *callme.yml* manifest and a new folder named *callme* with the template for your new function. Before anything else let’s edit the manifest and include your Dockerhub user-id before the resulting image name, so that it can be published correctly later. It should look similar to this:

image: juliocisco/callme:latest

If you take a look at the *handler.js* file in the *callme* folder you will notice that the template code just returns a *status: “done”* message. For your function you would include here your own code, but this is good enough for our demo.

First thing you will need to do is to build the container image that includes your code. Please make sure you have Docker running locally in your workstation, as the build process will be run locally.

faas-cli build -f callme.yml

With that you can now publish the image to your repo (Dockerhub by default):

faas-cli push -f callme.yml

And finally you need to deploy a new pod in your k8s cluster using the published image:

faas-cli deploy -f callme.yml

Your new function is now deployed in OpenFaaS! Let’s invoke it and see if it works, we will pass it any input (today’s date in our case) and it should answer with the *status: “done”* message.

$ date | faas-cli invoke -f callme.yml callme

{"status":"done"}

It works!

This function is available now to be consumed from the outside world using the HTTP endpoint accessible via the LoadBalancer IP, let’s give it a try:

$ curl -X GET $OPENFAAS\_URL/function/callme

{"status":"done"}

Nice!

You can of course use your browser as well:

A screenshot of a cell phone

Description automatically generated

As you can see OpenFaaS is easy to deploy, very k8s friendly with its own namespaces & functions deployments, and a great starting point with templates to deploy your own code.

[**Kubeless**](https://kubeless.io/)

Same as with other FaaS engines you will need to install first its own CLI. For example with OSX:

export OS=$(uname -s| tr '[:upper:]' '[:lower:]')

curl -OL https://github.com/kubeless/kubeless/releases/download/$RELEASE/kubeless\_$OS-amd64.zip && \

unzip kubeless\_$OS-amd64.zip && \

sudo mv bundles/kubeless\_$OS-amd64/kubeless /usr/local/bin/

Now you need to create a new namespace that will be used specifically for Kubeless:

kubectl create ns kubeless

And finally deploy the latest Kubeless release in the new namespace:

export RELEASE=$(curl -s https://api.github.com/repos/kubeless/kubeless/releases/latest | grep tag\_name | cut -d '"' -f 4)

kubectl create -f https://github.com/kubeless/kubeless/releases/download/$RELEASE/kubeless-$RELEASE.yaml

Check the deployment is 0k:

$ kubectl get deployment -n kubeless

NAME READY UP-TO-DATE AVAILABLE AGE

kubeless-controller-manager 1/1 1 1 4m51s

Kubeless is now fully deployed in your k8s cluster… that was easy!

So let’s create our first function now, we will do it in python this time. Create a file called *test.py* and include this content:

def hello(event, context):

print event

return event['data']

As you can see it is a very simple program that returns the same data it receives.

Let’s deploy it with:

kubeless function deploy hello --runtime python2.7 \

--from-file test.py \

--handler test.hello

Wait until it shows up as READY:

$ kubeless function ls hello

NAME NAMESPACE HANDLER RUNTIME DEPENDENCIES STATUS

hello default test.hello python2.7 1/1 READY

Your function is now deployed on Kubeless!

You can see the pod running in the default namespace:

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

hello-55f6478db6-d72fz 1/1 Running 0 4m49s

One big difference with Kubeless is that during the installation process it creates a new [Custom Resource Definition](https://kubernetes.io/docs/tasks/access-kubernetes-api/custom-resources/custom-resource-definitions/) (CRD) native to k8s, so that functions can be managed like standard k8s objects:

$ kubectl get functions

NAME AGE

hello 9m25s

0k, let’s see if our new function works with:

$ kubeless function call hello --data 'Hello world'

Hello world

Nice, it works fine!

For our Kubeless deployment we did not configure any externally accessible IP address via a LoadBalancer service, so we will test HTTP access using *kubectl proxy*:

kubectl proxy -p 8080 &

curl -L --data 'Hi there!' \

localhost:8080/api/v1/namespaces/default/services/hello:http-function-port/proxy/

Fantastic, our function is now readily available for everyone to access it. Well done!

[**Fission**](https://fission.io/)

Let’s start by installing fission in its own namespace:

kubectl create namespace fission

helm install --namespace fission --name-template fission \

https://github.com/fission/fission/releases/download/1.9.0/fission-all-1.9.0.tgz

The output will show you how to install the fission client CLI. For example with OSX:

curl -Lo fission https://github.com/fission/fission/releases/download/1.9.0/fission-cli-osx && chmod +x fission && sudo mv fission /usr/local/bin/

There are 3 different namespaces that have been created for fission:

* fission
* fission-builder
* fission-function

kubectl get namespace

The fission namespace includes all the different pods deployed for the framework itself:

kubectl get pods -n fission

The other 2 namespaces are empty for now as there are no functions deployed yet.

Now that the fission environment is completely set up you are now ready to start working on your first function. We will start by creating a new environment for the specific programming language you would like to use in your function. For our example we will use NodeJS:

fission env create --name nodejs --image fission/node-env

You may now take a look at the *fission-function* namespace and see now it includes 3 pods for that new environment you just deployed.

$ kubectl get pods -n fission-function

NAME READY STATUS RESTARTS AGE

poolmgr-nodejs-default-144232726-5579665886-bwkxc 2/2 Running 0 3m27s

poolmgr-nodejs-default-144232726-5579665886-qctgh 2/2 Running 0 3m27s

poolmgr-nodejs-default-144232726-5579665886-qxdmw 2/2 Running 0 3m27s

Wait a minute… I have not deployed any code yet, and there are already pods running in the system!?

A picture containing newspaper, photo, person, man

Description automatically generated

Yep, that’s an important difference when comparing fission with other FaaS engines: it pre-deploys a number of pods for a *warm* start-up when code needs to be run.

Let’s download a simple hello-world javascript app we can use for our demo, and save it to a local file called hello.js:

curl https://raw.githubusercontent.com/fission/fission/master/examples/nodejs/hello.js > hello.js

Take a look at its content and you will see it is as simple as returning a “hello, world!” message. We will use it to create a new function in the already deployed nodejs environment. In fission that means *registering* the function with the available environment (--env nodejs).

fission function create --name hello --env nodejs --code hello.js

You will see no changes in the *fission-function* namespace, as the environment pods were already deployed, but now your app code has been included there.

You may list the deployed functions with:

$ fission function list

NAME ENV EXECUTORTYPE MINSCALE MAXSCALE MINCPU MAXCPU MINMEMORY MAXMEMORY TARGETCPU SECRETS CONFIGMAPS

hello nodejs poolmgr 0 0 0 0 0 0 0

You may now invoke your function by running:

$ fission function test --name hello

hello, world!

It works fine!

In order to access this function via a HTTP endpoint let's take a look at the externally reachable IP address assigned to the *router* LoadBalancer service:

$ kubectl get svc router -n fission

NAME TYPE CLUSTER-IP **EXTERNAL-IP** PORT(S) AGE

router LoadBalancer 10.31.240.5 **104.155.113.47**  80:30668/TCP 25m

But before accessing the endpoint we need to accommodate another requirement for fission: [triggers](https://docs.fission.io/docs/triggers/). Functions in fission are invoked only when an event happens, and that's what they call a trigger. Before being able to access that new function you need to create a trigger. In our case we will create a trigger that invokes our *hello* function when the URL path */hello* is accessed.

$ fission route create --method GET --url /hello --function hello

trigger '027634d8-44a6-4eba-9354-cd28e55beb69' created

You may now access the URL path where the function resides:

$ curl http://**104.155.113.47**/hello

hello, world!

It works, well done!