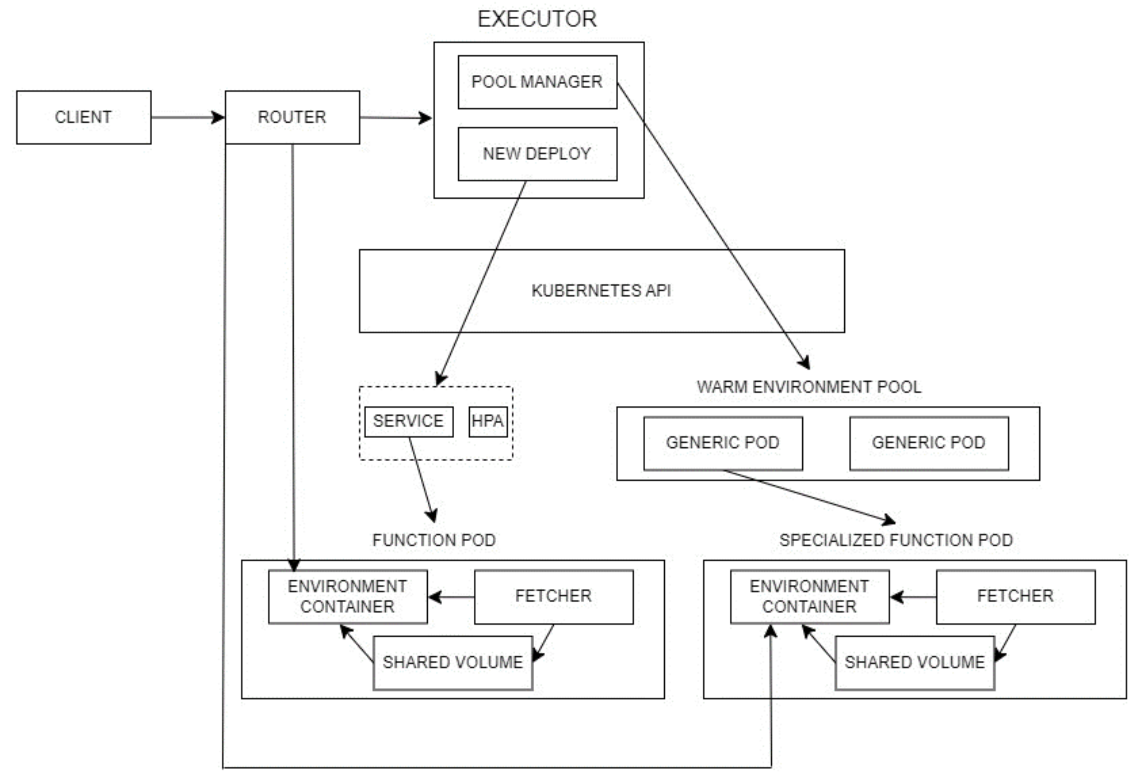
FAAS PLATFORMS EVALUATION

Knative:

- Written in Go and based on Kubernetes
- It supports functions written in most any language
- It doesn't use a function invoker component: functions live in pods that are managed by Kubernetes. Knative only offers a high-level way to deploy and run serverless applications, but lower-level actions are performed by Kubernetes.
- Given that there isn't a modifiable invoker that could be used as injector, there are only two ways to achieve dependency injection in Knative: adding environment variables and binded services in our Custom Resource Definitions (CRDs) or using specific libraries and frameworks in our function codes.
- Nevertheless, Kubernetes offers additional mechanisms that can be used to inject dependencies. One of them is the so called "sidecar pattern": it is possible to deploy container images that contain cross-cutting and infrastructure concerns to the same pods where our functions are instantiated. So, it is possible to use those sidecar containers as injectors and they will be accessible to the functions in the same pod at 'localhost'. In this way, we can implement a specific injector for every pod.
  Another way to obtain the same objective is to use Kubernetes Service Mesh: it is possible to use sidecar containers as proxies to enable secure and observable communication between kubernetes' services. In other terms, a service mesh is a specialized application layer network that offers centralized system services, so they don't need to be embedded within the functions. Doing so, we can obtain a centralized dependency injection container hosted on the control plane of the service mesh tool that we decide to use. In particular, Istio permits to all the deployed functions to be aware of and to reference each other in a transparent way.

Fission:

- Written in go and based on Kubernetes
- The architecture can be summarized as shown below:



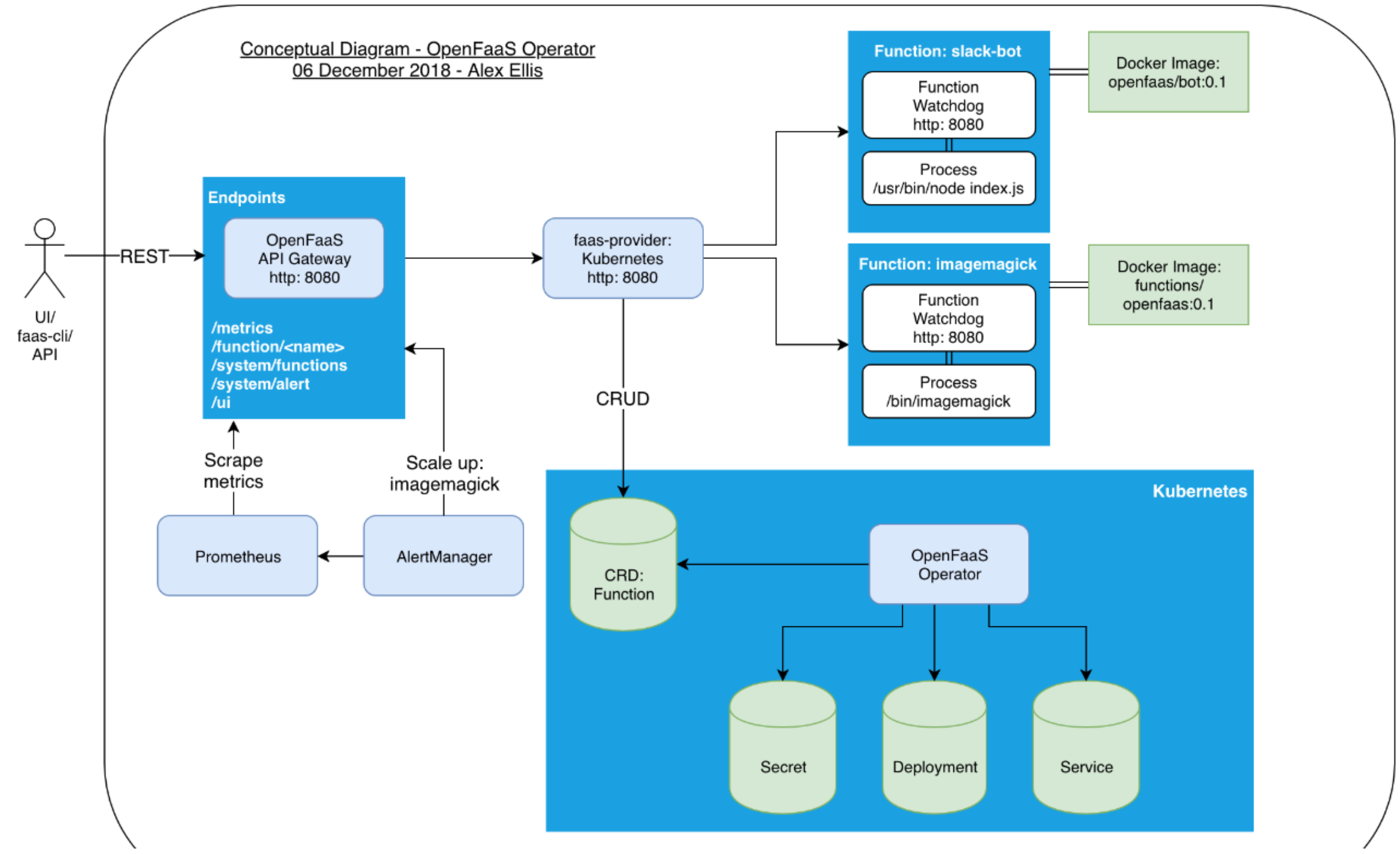
A typical workflow is:

1) A client sends an HTTP request at a certain URL for the execution of a function
2) The router checks if already exists a trigger for that URL and if there is an associated service in its cache: if yes, the request is redirected to the address of that service and the function is executed. Otherwise, router asks the Executor to provide the service address of the function
3) The Executor retrieves function information from CRD and invokes one of executor type to get the address. If there isn't an existent service for that function, Pool Manager and New Deploy create corresponding Kubernetes resources and return their address to the Router.

Independently from the type of Executor, our functions will run inside an Environment Container that is language specific.

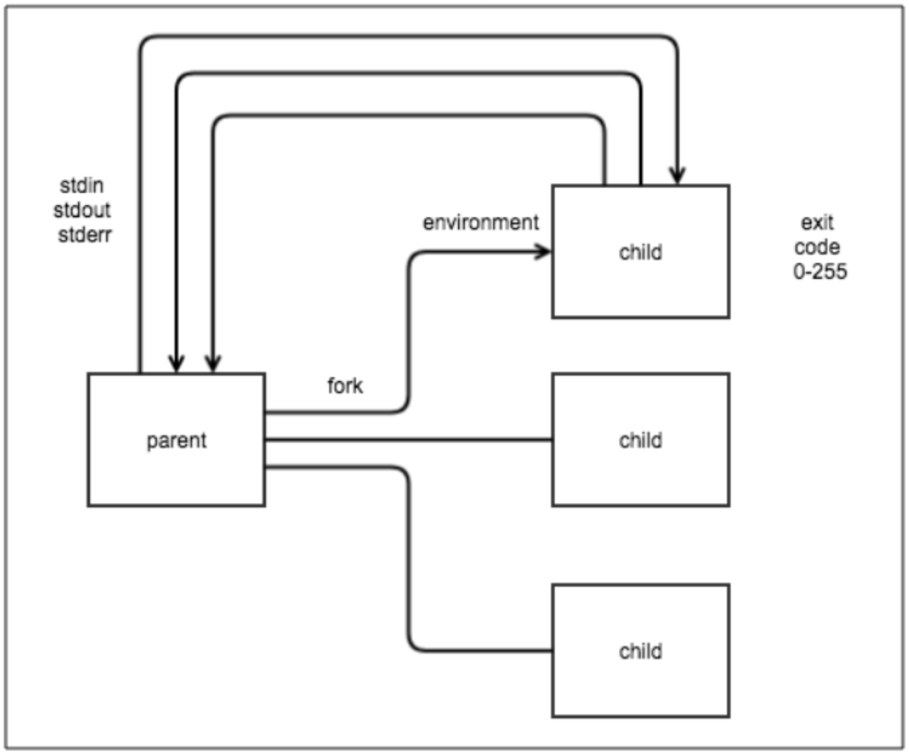
During the initialization of function pods, two containers are instantiated: the fetcher and the environment container. The fetcher takes care of fetching the function compiled source-code from Fission StorageSVC and storing it in the shared volume. At this point, the environment container can load that code and run it to execute the function's business logic. In this scenario, the fetcher is a suitable candidate for the role of dependency injector. Its lifetime is the same as that of the function and they share the same pod, so, they can refer each other easily.

OpenFaaS:
serverless framework to deploy and run functions on Kubernetes.


Conceptual Diagram - OpenFaaS Operator
06 December 2018 - Alex Ellis
UI/ faas-cli/ API
REST
Endpoints
OpenFaaS API Gateway http: 8080
/metrics
/function/<name>
/system/functions
/system/alert
/ui
faas-provider: Kubernetes http: 8080
Function: slack-bot
Function Watchdog http: 8080
Process /usr/bin/node index.js
Docker Image: openfaas/bot:0.1
Function: imagemagick
Function Watchdog http: 8080
Process /bin/imagemagick
Docker Image: functions/ openfaas:0.1
CRUD
Scrape metrics
Scale up: imagemagick
Prometheus
AlertManager
Kubernetes
CRD: Function
OpenFaaS Operator
Secret
Deployment
Service

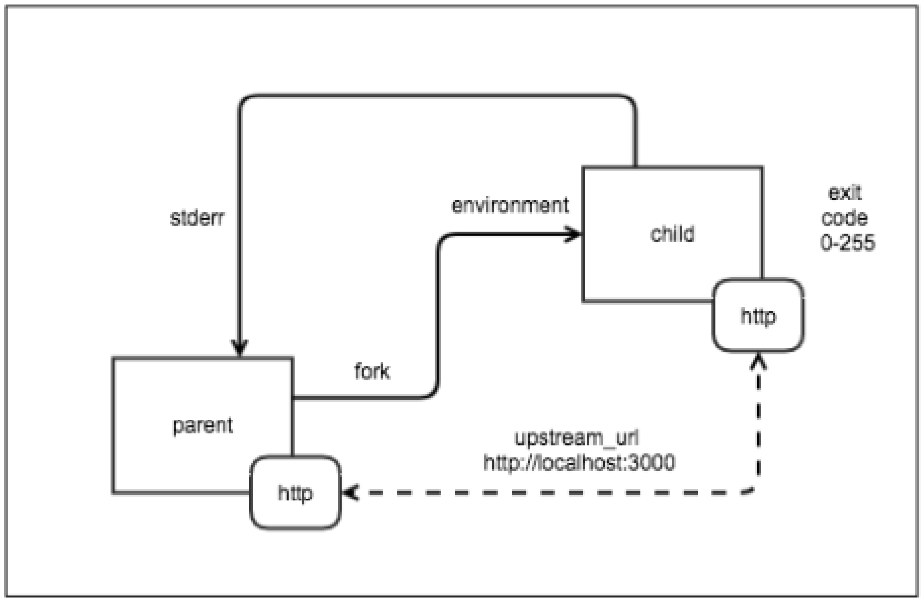
Containers that contain function include a watchdog process too: this process is responsible for starting and monitoring the functions. Essentially, it is an HTTP server that provides an interface between the outside world and the function. There are two types of watchdogs:

- Classic watchdog: it starts a new process for every request and communicate with them via STDIO



Each child process terminates at the end of the function.

- Of-watchdog: only one child process is created at the start of the watchdog. They can communicate either via HTTP messages or STDIO.



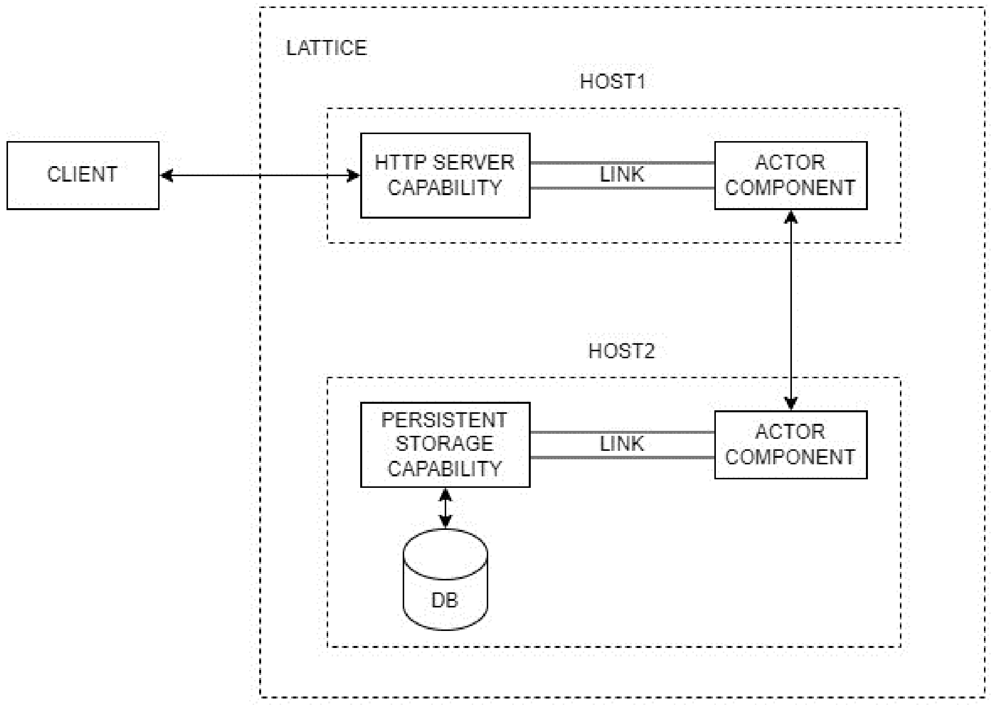
In this way, we break the FaaS paradigm. In fact, the process that executes the function never terminates, so the function isn't ephemeral. This leads to many advantages, like better performances, the possibility to reuse DB connections and the possibility to have local state.

The watchdog process can be used to implement dependency injection inside containers, since it is started before functions and they share the same container.

WasmCloud:

Universal application platform that helps build and run globally distributed WebAssembly applications everywhere. It is written in Rust.

An example of WasmCloud application architecture is shown below:



The architecture is based on 5 principal components:

- A lattice, that is WasmCloud mesh network. It provides a unified, flat topology across any number of environments. Hosts inside the same lattice are automatically aware of each other
- Hosts, that represent the execution environments of our WebAssembly components
- Actor components, that are the actual services. "Actors" means that they are stateless by design, single-threaded, reactive and communicate with messages. They only implement functional requirements. To achieve non-functional requirements (accessing DBs, sending notifications, ...) they have to call Capability Providers by using capability contracts

- Capability Providers, that are processes that implement capability contracts (APIs) to achieve non-functional requirements.
- Links, that are connections between actors and specific capability providers. They are defined by an actor's ID, provider's ID and a link name.

Actors only have to know the contract interface to use a certain type of provider. The information about the actual implementation of the provider to be used is stored in the link definition. So, dependency injection of providers is obtained by design. Furthermore, Actors inside the same lattice can communicate with each other by using a simple RPC protocol, given that actors are univocally defined by their ID.

There are open-source adaptors to map WasmCloud hosts to Kubernetes pods.