4- ON-PREMISES SOLUTION

4.1 Introduction:

Now that we tested the first architecture on the main public clouds, we want to bring it on an on-premises infrastructure. The main reason is that public vendors' FaaS solutions don't offer the possibility to modify the underlying infrastructure in order to upgrade and optimize the distributed dependency injection architecture. Furthermore, the use of third-party services (like external databases and SDKs) is disincentivized given that every vendor has its own standard ecosystem for application development and execution.

In this chapter, we will examine some of the main installable open-source platforms that allow to deploy and execute serverless functions on our private infrastructure. Given that all of them are based on Kubernetes, we will first introduce it. This will help the comprehension of the platforms' architecture. The objective is to find out which of those platforms is most suitable to be modified to implement the architecture presented in paragraph [3.2] by changing less source code as possible. So, we want to find a platform that naturally allows us to insert a component that will perform the role of injector.

4.2 Kubernetes:

Kubernetes is an open-source platform written in Go and designed to automate the deployment, scaling, and management of containerized applications on a cluster. Originally developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes provides a robust framework to run distributed systems resiliently. Traditionally, once a service is containerized, it has to be deployed manually or using `docker-compose`. The problem is that there wasn't an automatized way to manage the load-balancing, the dynamic allocation of resources, the rollbacks and rollouts. Kubernetes simplifies these operations.

4.2.1 Architecture:

Kubernetes has a modular and distributed architecture that is designed to manage large-scale, containerized applications efficiently. Its architecture follows a master-worker model, consisting of a control plane (master) and one or multiple worker nodes. The worker nodes can be composed by one or more machines (either physical or virtual) and they represent the endpoint of containers deployment.

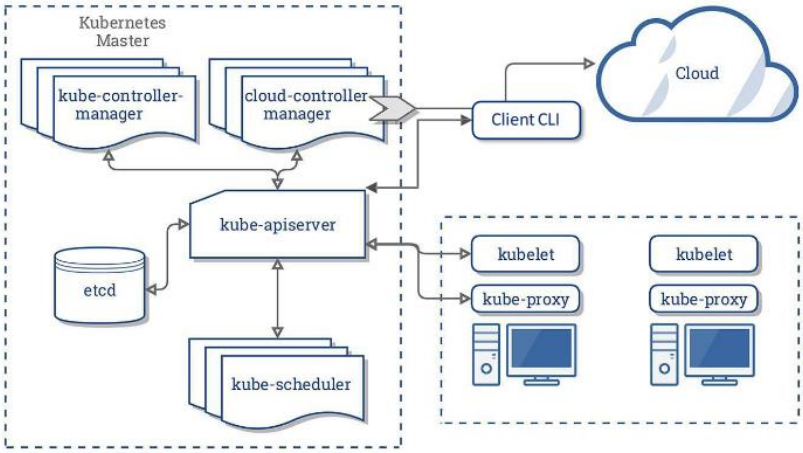


Figure 4.1: Kubernetes architecture

The control plan is responsible for the actual orchestration, and it is composed by the following components:

- **Kube-apiserver**: it is the central hub of the Kubernetes cluster. It exposes the API to interact with the platform. It is responsible for authentication, authorization and input validation. All interactions with the cluster go through the API server, which serves as the primary interface for users and other components (via RESTful APIs). There can't be a direct access to the nodes or other components.
- **Etcd**: it is an open-source key-value store, highly consistent and distributed. Data is stored in as a hierarchy of directories that forms a filesystem.
- **Kube-scheduler**: it is the component responsible for the scheduling of pods on the nodes, like the scheduler of an operating system. When a client sends an API request of creating a new pod, it has to specify which resources should be allocated to that pod (CPU, memory, volumes, priority, etc). The scheduler's task is to identify the best node on which to create the pod. To do this, it performs a scoring phase, in which it assigns a score to each node by running

certain functions on them. Afterward, it assigns the pod to the node with the highest score (binding phase).

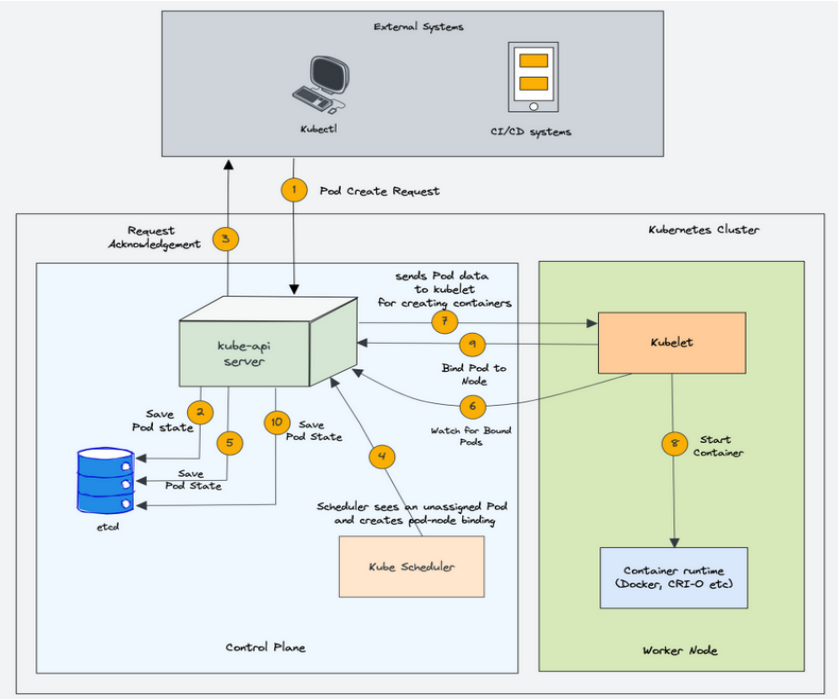


Figure 4.2: Kube-scheduler workflow

- **Controller-manager**: it is the component that runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process. So, this component logically works as a control loop that continuously observes the cluster state and makes adjustments to reconcile the current state with the desired state specified in the pod specs stored in etcd.
- **Cloud-controller manager**: it is the component that embeds cloud-specific control logic. The cloud controller manager allows to link the cluster into a specific cloud provider's API and separates out the components that interact with that cloud platform from components that only interact with the cluster. If Kubernetes is running on premises, the cluster does not have a cloud controller manager.

On the other side, on the worker nodes run the following components:

- **Kubelet**: It is the first agent that runs on every node of the cluster. It is responsible for registering the node via the API server and acts as an intermediary between the control plane and the pods within its node. The

Kubelets receive information about the containers from the API server through PodSpecs, which are YAML or JSON objects that describe a pod. The Kubelets ensure that the containers specified in the PodSpecs are running and conform to the specifications.

- **Pod**: it is an abstraction that represents a group of tightly coupled containers and the resources they share (storage, network, configuration information). Containers within a pod share the same IP address and are always distributed and scheduled together, executing within the same context. Pods are the atomic unit of deployment in Kubernetes and remain on the same node until their execution is complete (if the node they reside on fails, they are rescheduled on another available node). Pods are designed to host a single instance of an application. To scale horizontally, you must use multiple pods (one for each instance). Pods are ephemeral and unreliable (they frequently terminate, and when they do, they are not restarted but recreated from scratch).
- **Service**: A service is an abstraction that defines a logical set of pods and a policy for accessing them, creating a network of pods. Each service defines a set of endpoints (usually one endpoint for each pod) and an access policy (IP and logical name of the service, with the respective routing between pods). Its main purpose is to create decoupling between dependent pods (you cannot rely on the IP of the pods since they are ephemeral and a new IP is assigned every time they are recreated), as well as to expose the pods to the outside of the cluster. Services have a frontend that exposes a logical name, an IP, and a port, and a backend that acts as a load balancer for the pods.
- **Kube-proxy**: It is an agent installed on all nodes (like Kubelet). The task of kube-proxy is to remap the networking within the services every time a pod terminates and is recreated. It is responsible for keeping track of the IP addresses of the services and the pods they contain by modifying the IPtables within the kernel space.

So, thanks to Kubernetes it is possible to configure a cluster in a declarative way, specifying a set of configuration files containing the specifics of the pods to create. The master plane will automatically instantiate the pods and ensure that the desired state is reached in every moment.

4.3 Serverless platforms evaluation:

Now that we described how Kubernetes works, we can analyze some of the main open-source serverless platforms in order to identify the one that better fits for our goal. In particular, we will discuss about OpenFaaS, Fission and Knative.

4.3.1 OpenFaaS:

OpenFaaS is an open-source framework for building serverless functions on top of container orchestrators like Kubernetes or Docker Swarm. It simplifies deploying, managing, and scaling functions in a cloud-agnostic way, allowing users to run stateless functions as microservices while handling the operational overhead like scaling, monitoring, and networking.

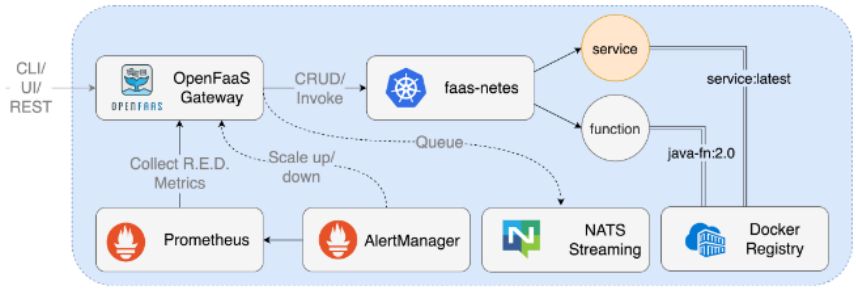


Figure 4.3: OpenFaaS' main components

The two main components of OpenFaaS architecture are the Gateway and faas-netes:

- OpenFaaS Gateway: it represents the access point to the system, and it is responsible for exposing the functions via REST APIs. It handles requests and forwards them to the appropriate function, providing authentication and load balancing across replicas of functions. The Gateway is also responsible for scaling the functions up or down based on demand (auto-scaling) based on the metrics provided by the monitoring system (Prometeus).
- Faas-netes: it is the core OpenFaaS service responsible for managing the lifecycle of functions. It interacts with the underlying orchestrator (like Kubernetes or Docker Swarm) to schedule, deploy, and scale the functions. Once that it receives a request from the Gateway, faas-netes executes that request using a Docker register to retrieve the right function's image to invoke.

Each function, when deployed, in addition to the process responsible for executing the code, has a watchdog process exposed on HTTP port 8080 to accept connections from the OpenFaaS provider. The watchdog behaves in the following way:

1) Receives an event from the controller
2) Generates a child process to execute the function's code (a new process for each request)
3) Executes the code passing the event as argument
4) Waits for the child to terminate
5) Kills the child process

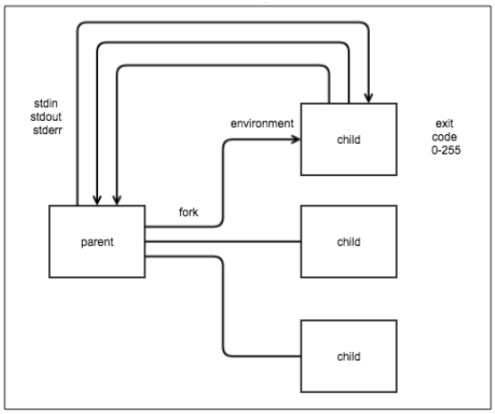


Figure 4.4: standard type of watchdog

It is possible to use a different type of watchdog that exploits HTTP channels to communicate with the child process. In fact, when the watchdog is started, it immediately generates a single child process that will serve all the requests. This violates the definition of FaaS (the watchdog keeps a non-ephemeral container that can maintain state), but the performance is better because cold starts effect is softened. The child process can also create threads to work concurrently.

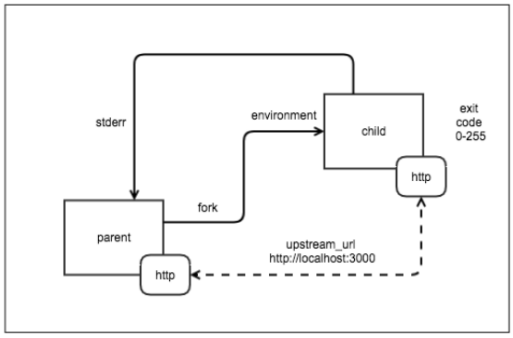


Figure 4.5: HTTP watchdog

In this scenario, we can think to use the watchdog as injector to implement our architecture, given that this component is started before the execution of each function and it shares the same container with them. So, we can extend the behavior of the watchdog modifying its source code by adding the logic of registration and retrieval of services.

4.3.2 Fission:

Fission is an open-source, Kubernetes-native, written in Go serverless framework that allows users to deploy, manage, and run functions without the need to manage the underlying infrastructure. Its architecture can be resumed as shown below:

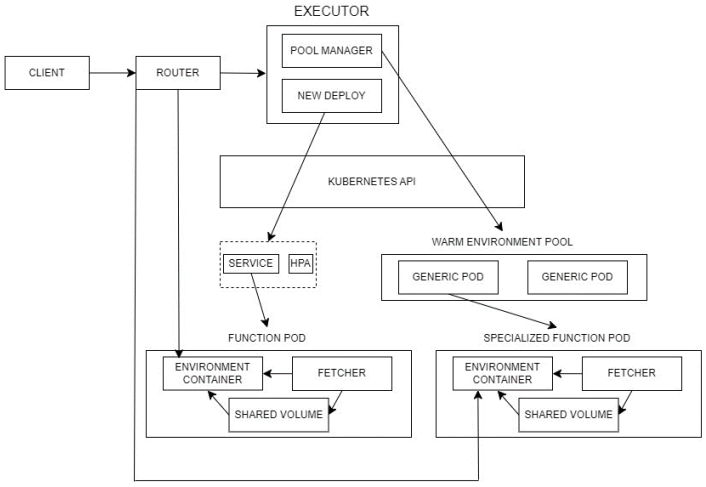


Figure 4.6: Fission architecture

We can explain it describing a typical workflow:

1) A client sends an HTTP request to a certain URL to trigger the execution of a function.
2) The router checks if a trigger for that URL already exists 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, the router asks the Executor to provide the service address of the function.
3) The Executor retrieves function information from Kubernetes' Custom Resource Definitions (CRDs) and invokes one of the executors 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. Fission maintains a pool of “warm” pods that contain a small dynamic loader that are used to reduce cold starts time.

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. Also in this case, we would need to modify the fetcher’s source code to implement the solution.

4.3.3 Knative:

Knative is an open-source platform built on Kubernetes that provides powerful abstractions for building, deploying, and managing serverless applications. So, this platform appears quite different from all the others, as it wasn't created with the purpose of directly providing FaaS, but to extend Kubernetes with a set of features and APIs that can be used to build, among other things, serverless functions.

The architecture is divided into two logical blocks independent from each other and placed on top of Kubernetes: Serving and Eventing. We will only discuss about Serving in this moment, given that the Eventing component isn’t interesting for our goal.

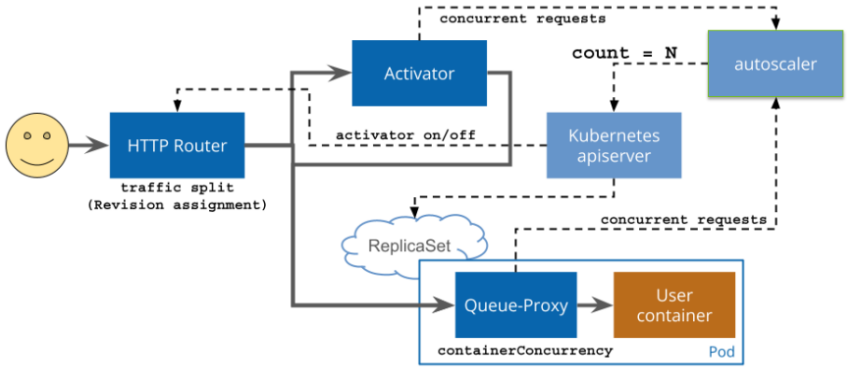
Serving:

Knative Serving defines a set of objects as Kubernetes Custom Resource Definitions. These resources are used to define and control how the serverless workload behaves on the cluster. The four main resource types are:

- **Route**: a Route provides a network endpoint for a user's service (which consists of a series of software and configuration Revisions over time). A Kubernetes namespace can have multiple routes. The Route provides a long-lived, stable, named, HTTP-addressable endpoint that is backed by one or more Revisions.
- **Revision**: a Revision is an immutable snapshot of code and configuration. A Revision references a container image. Revisions are created by updates to a Configuration.

- **Configuration**: A Configuration describes the desired latest Revision state, and creates and tracks the status of Revisions as the desired state is updated. A configuration will reference a container image and associated execution metadata needed by the Revision. On updates to a Configuration's spec, a new Revision will be created.
- **Service**: a Service encapsulates a Route and Configuration which together provide a software component. A Service exists to provide a singular abstraction which can be access controlled, reasoned about, and which encapsulates software lifecycle decisions such as rollout policy and team resource ownership. A Service acts only as an orchestrator of the underlying Route and Configuration (much as a Kubernetes Deployment orchestrates ReplicaSets).

The architecture can be presented describing the typical flow of an HTTP request:



1) An HTTP request that targets a specific Knative service arrives at the ingress gateway (the entry point of the system).
2) The gateway examines the request to determine which service should handle it.
3) Knative Serving keeps the service deployed at all times. When a request arrives and no instances are running, it promptly activates a new instance by spinning up a pod.
4) Knative Serving checks the current load and decides how many instances of the service need to be running to handle incoming requests efficiently.
5) For the first-time request, it goes to the activator. The activator asks the auto scaler to scale up one pod to serve the initial request, ensuring rapid response and availability.
6) The request is then forwarded to one of the instances of the service, where the application code processes it.

7) Within each pod, there are two containers: one User container that hosts the application code, and a Queue container that monitors metrics and observes concurrency levels.
8) When the concurrency exceeds the default level, the autoscaler spins up new pods to handle the increased concurrent requests, ensuring optimal performance.
9) After processing the request, your service generates a response, which is sent back through the same flow to the user who made the initial request.

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 implement a dependency injector. First of all, it is possible to use the sidecar pattern to deploy container images that contain cross-cutting and infrastructure concerns to the same pods where our functions are instantiated. It is the same pattern used to implement the Queue-Proxy container in Knative, and it could be used to implement an injector service that will be deployed in all the pods on the cluster, making it accessible in the functions' code at `localhost`.

Furthermore, it is possible to use Kubernetes Service Mesh in combination with sidecar containers to build an infrastructure layer that manages and optimizes the communication between functions and other distributed services. With a service mesh, a proxy container is injected into all the pods and it intercepts all the incoming and outcoming traffic in order to add cross-cutting concerns common to all the services. In this way, it is possible to automatically add sidecar containers to all the pods that host our functions and create a network that allows every service to reference others in a transparent way. Istio is one of the most popular service mesh solutions for Kubernetes and it is also well integrated with Knative.

So, thanks to sidecar pattern and service mesh, it is possible to inject containers that act as dependency injectors into all the pods of the Kubernetes cluster and, given that Knative is already based on Kubernetes, it is possible to implement the architecture presented in [3.2] without modifying any part of source code. For this reason, we chose Knative as FaaS platform to implement distributed dependency injection.

References:

https://kubernetes.io/docs

https://docs.openfaas.com/

https://fission.io/docs/

https://knative.dev/docs/