OPENFAAS

Part 4 – Serverless Computing and OpenFaas

GIANLUCA REALI

Evolution of software developmenet

Traditional software development

Software development in early cloud computing

Software development with virtualization

Software development with containerization

Software development with serverless

With serverless applications, all low-level concerns are outsourced and managed, and the focus is on the last step : **Run the code.**

Serverless is a confusing term. It means leaving the responsibility of servers to third-party organizations. In other words, it means not getting rid of servers but server operations. When you run serverless, someone else handles the procurement, shipping, and installation of your server operations. This decreases your costs since it lets you focus on the application logic.

Serverless Compute Manifesto (2006 AWS developer conference)

- **Functions as the building blocks**
- **No servers, VMs, or containers**
- **No storage**
- **Implicitly fault-tolerant functions**
- **Scalability with the request**
- **No cost for idle time**
- **Bring Your Own Code (BYOC)**

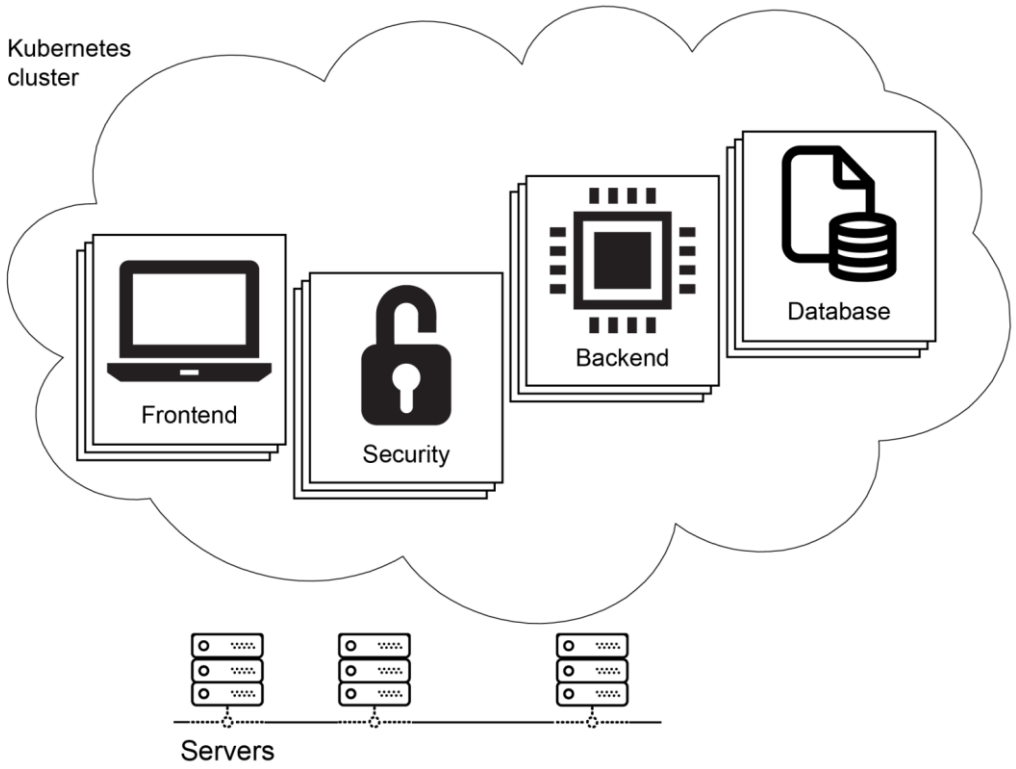
Sample Serverless Use Cases

- **Data processing**
- **Webhooks**
- **Check-out and payment**
- **Real-time chat applications**

In synthesis, applications tolerant to latency, short lived, avoiding vendor lock-in and ecosystem dependencies

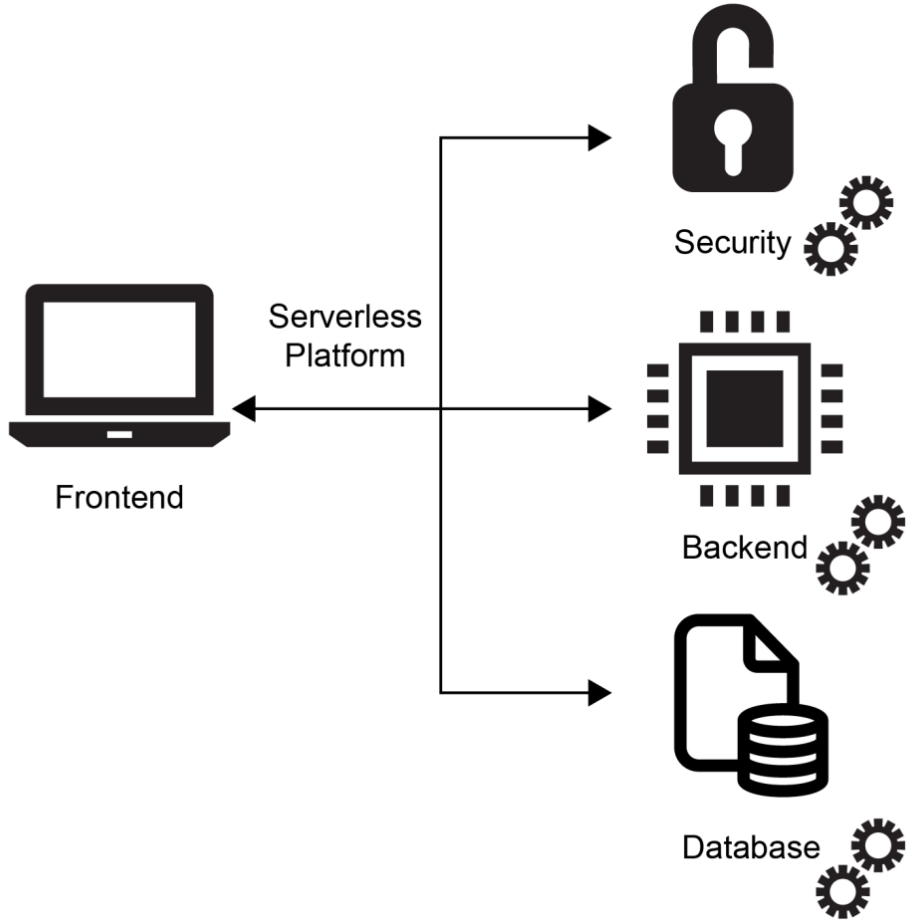
Example (legacy µservices approach)

- For the same **e-commerce system**, you have frontend, backend, database, and security components, but they would be isolated units. These components would be packaged as containers and would be managed by a container orchestrator such as Kubernetes. This enables the installing and scaling of components independently since they are distributed over multiple servers.
- Microservices are deployed to the servers, which are still managed by the operations teams.



Example (serverless)

- The best option for the backend logic is to convert it into functions and deploy them into a serverless platform such as AWS Lambda or Google Cloud Functions. Finally, the frontend could be served by storage services such as AWS Simple Storage Service (S3) or Google Cloud Storage.
- With a serverless design, it is only required to define these services for you to have scalable, robust, and managed applications running in harmony.



Function as a Service (FaaS)

- **FaaS** is the most popular and widely adopted implementation of serverless architecture.
- All major cloud providers have FaaS products, such as AWS Lambda, Google Cloud Functions, and Azure Functions.
- As its name implies, the unit of deployment and management in FaaS is the function. Functions in this context are no different from any other function in any other programming language. They are expected to take some arguments and return values to implement business needs.
- FaaS platforms handle the management of servers and make it possible to run event-driven, scalable functions.

Properties of a FaaS

- **Stateless**: Functions are designed to be stateless and ephemeral operations where no file is saved to disk and no caches are managed. At every invocation of a function, it starts quickly with a new environment, and it is removed when it is done.
- **Event-triggered**: Functions are designed to be triggered directly and based on events such as cron time expressions, HTTP requests, message queues, and database operations. For instance, it is possible to call the startConversation function via an HTTP request when a new chat is started. Likewise, it is possible to launch the syncUsers function when a new user is added to a database.
- **Scalable**: Functions are designed to run as much as needed in parallel so that every incoming request is answered and every event is covered.
- **Managed**: Functions are governed by their platform so that the servers and underlying infrastructure is not a concern for FaaS users.

Kubernetes and Serverless

A strong connection between Kubernetes and serverless architectures exists:

- Serverless and Kubernetes arrived on the cloud computing scene at about the same time, in 2014.
- Kubernetes gained dramatic adoption in the industry and became the de facto container management system. It enables running **both stateless applications**, such as web frontends and data analysis tools, **and stateful applications**, such as databases, inside containers. running microservices and containerized applications is a crucial factor for successful, scalable, and reliable cloud-native applications.
- **No vendor lock-in**: If you use a Kubernetesbacked serverless platform, you will be able to quickly move between cloud providers or even on-premises systems.
- **Reuse of services**: Kubernetes offers an opportunity to deploy serverless functions side by side with existing services. It makes it easier to operate, install, connect, and manage both serverless and containerized applications.

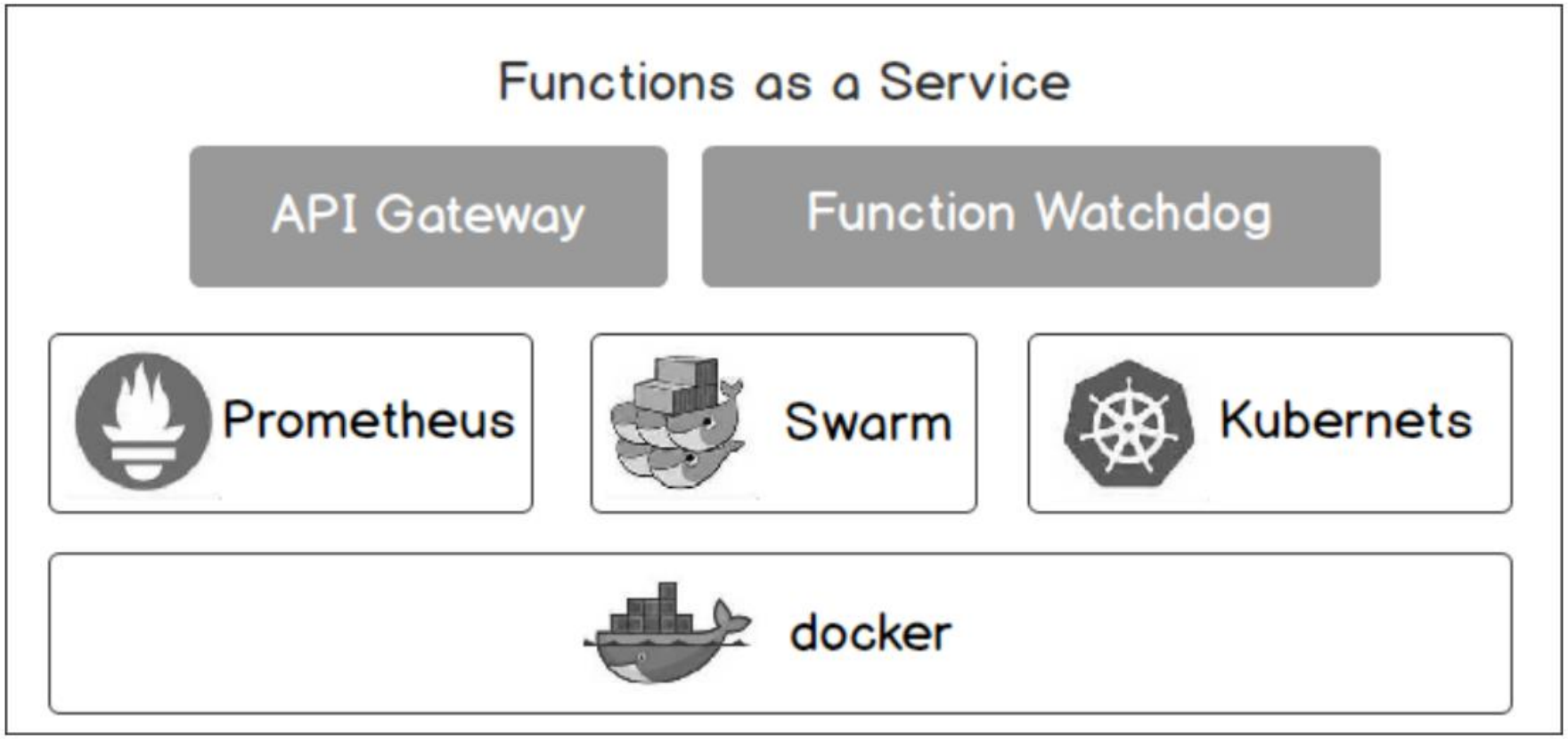
Overview of OpenFaas

- OpenFaas is an **open source (kind of…)** framework written in Go and used to build and deploy serverless functions on top of containers
- OpenFaaS was originally designed to work with Docker Swarm, which is the clustering and scheduling tool for Docker containers. Later, the OpenFaaS framework was rearchitected to **support also Kubernetes**.
- OpenFaaS comes with a built-in **UI named OpenFaaS Portal**, which can be used to create and invoke the functions from the web browser. This portal also offers a **CLI named *faas-cli*** that allows us to manage functions through the command line.

Overview of OpenFaas

- **Main Features**
  - Portable functions platform - run functions on any cloud or on-premises without fear of lock-in
  - Write functions in any language and package them in Docker/OCI-format containers
  - Easy to use - built-in UI, powerful CLI and one-click installation
  - Scaling capabilities, able to scale up a function when there is increased demand, and it will scale down when demand decreases, or even scale down to zero when the function is idle
  - Ecosystem - community marketplace for functions and language templates

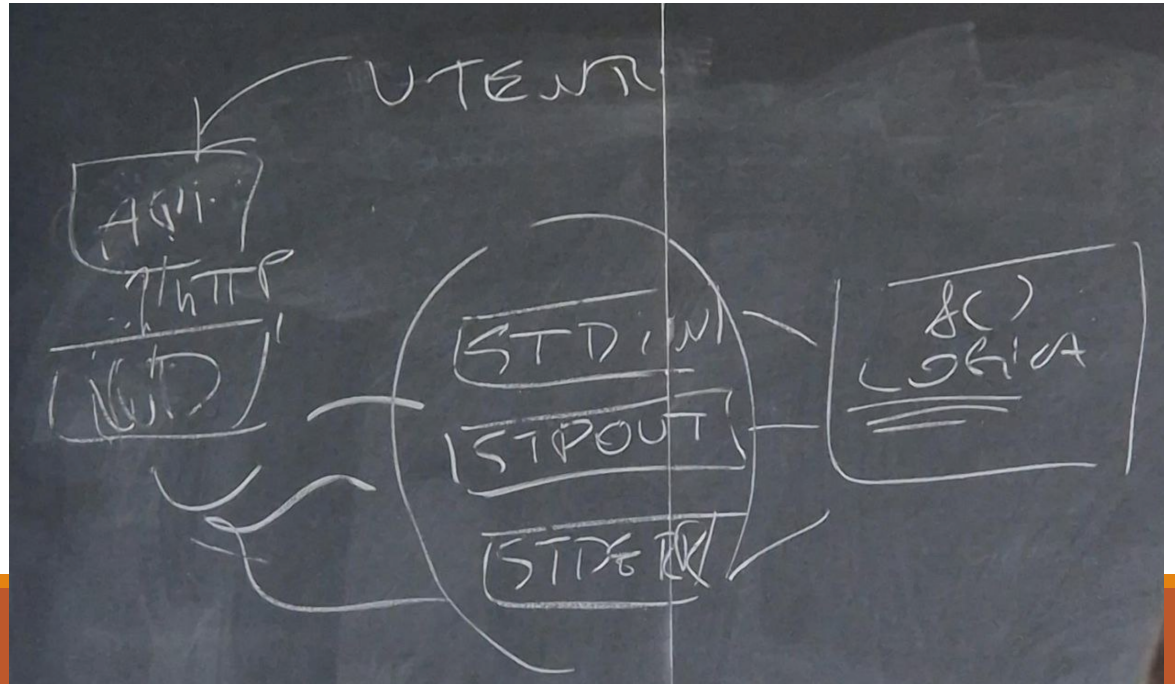
OpenFaaS main components



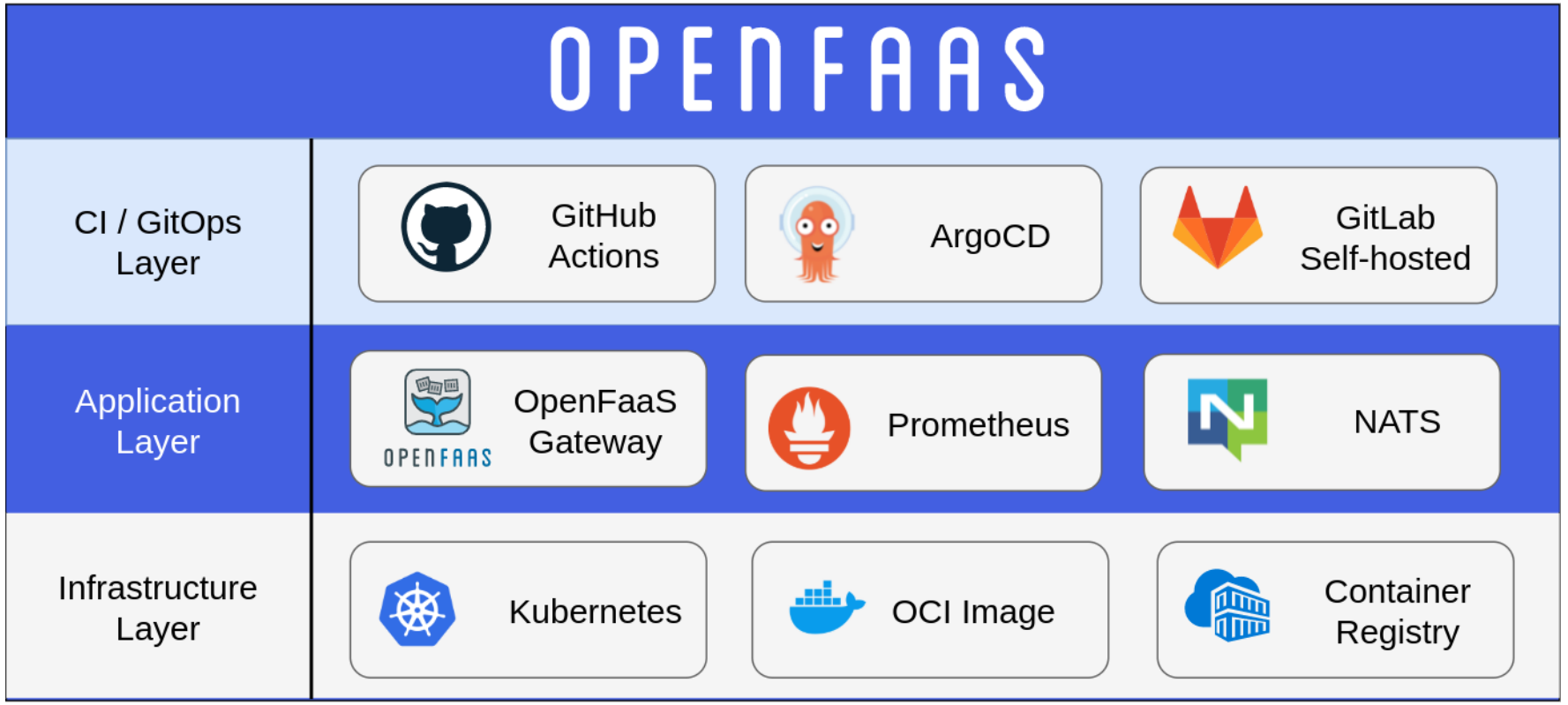
Versions

ci sono diverse versioni di OpenFaaS, noi usiamo la seconda

- OpenFaaS Standard/For Enterprises for commercial use & production, deployed to Kubernetes
- Community Edition - free for personal use only, 60-day limit for commercial use, deployed to Kubernetes
- faasd - Free to use on any cloud or on-premises, works on a single VM (no clustering or Kubernetes)



Conceptual layers of the OpenFaaS stack



OpenFaas functions

- OpenFaaS functions can be written in **any language** supported by Linux or Windows, and they can then be converted to a serverless function using Docker containers. This is a major advantage of the OpenFaaS framework compared to other serverless frameworks that support only predefined languages and runtimes

- This requires to follow some specific steps, as to **create the function code**, **add any dependencies**, and **create a Dockerfile** to **build the Docker image**. It requires a certain amount of understanding of the OpenFaaS platform in order to be able to perform all the needed tasks.

- As a solution, OpenFaaS has a **template store** that includes prebuilt templates for a set of supported languages. This means that you can download these templates from the template store, update the function code, and then the CLI does the rest to build the Docker image.

Create functions

Once you've installed the **faas-cli** you can start creating and deploying functions via the faas-cli up command or using the individual commands:

- *faas-cli build* - build an image into the local Docker library
- *faas-cli push* - push that image to a remote container registry
- *faas-cli deploy* - deploy your function into a cluster

The *faas-cli up* command automates all of the above in a single command.

Templates

The OpenFaaS CLI has a **template engine** built-in which can create new functions in a given programming language. The way this works is by reading a list of templates from the *./template* location in your current working folder.

Before creating a new function you can pull in the official OpenFaaS language templates from GitHub via the templates repository.

$ *faas-cli template pull*

This way it possible to generate functions in the most popular languages and explains how you can manage their dependencies too.

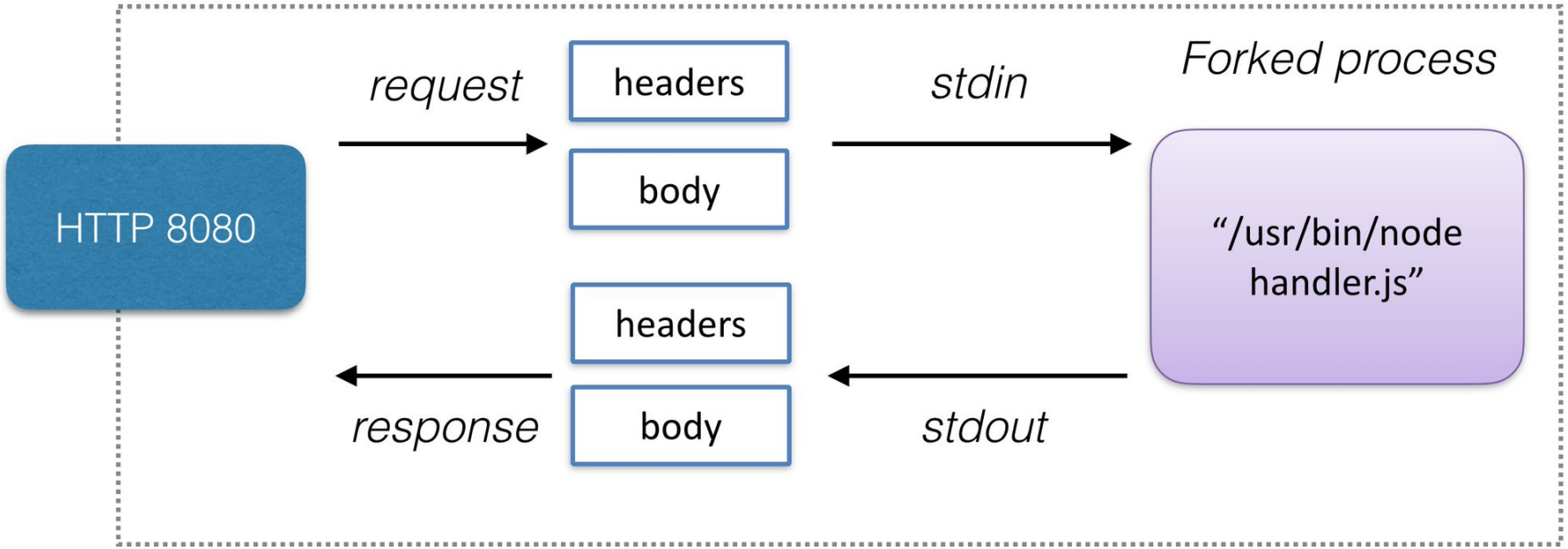
Classic watchdog vs. of-watchdog templates

- The **Classic Templates** are held in the openfaas/templates repository and are based upon the **Classic Watchdog** which uses **STDIO** to communicate with your function.
- The **of-watchdog** uses **HTTP** to communicate with functions and most of its templates are available in the openfaas organisation in their own separate repositories on GitHub and in the store.

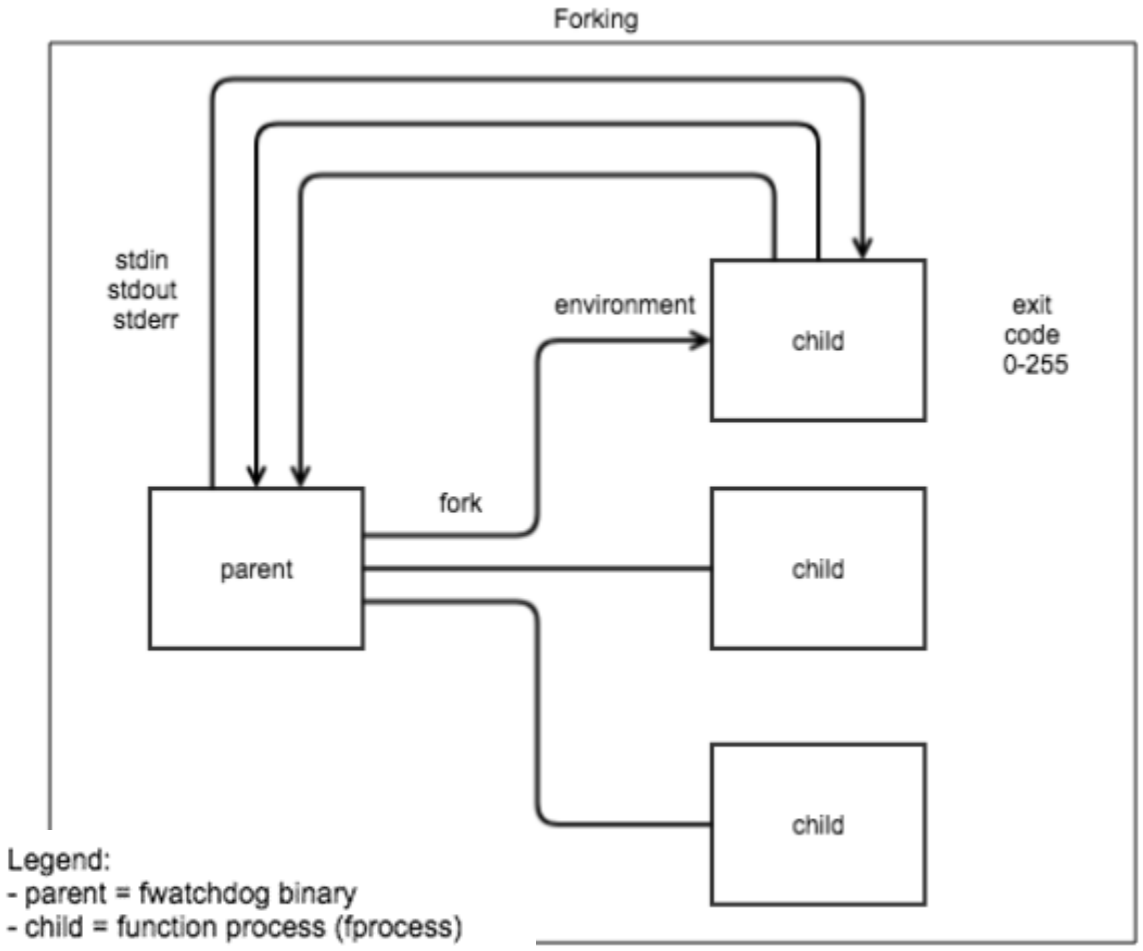
Classic Whatchdog

Every function needs to embed this binary and use it as its **ENTRYPOINT or CMD**, in effect it is the init process for your container. Once your process is forked the watchdog passses in the HTTP request via stdin and reads a HTTP response via stdout. This means your process does not need to know anything about the web or HTTP.

a tiny web-server or shim that forks your desired process for every incoming HTTP request



Classic Whatchdog mode



Behave as defined by the FaaS definition

1. Receive an event from the controller
2. Create a child process to execute the function code
3. Run the function code in the child
4. Pass the event as argument to the function
5. Waits for the end of execution of the function
6. Terminates the child

Whatchdog HTTP methods

- The HTTP methods supported for the watchdog are:
  - With a body: **POST, PUT, DELETE, UPDATE**
  - Without a body: **GET**

- The API Gateway currently supports the POST route for functions.

- **Content-Type of request/response**
  - By default the watchdog will match the response of your function to the "Content-Type" of the client.
    - If your client sends a JSON post with a Content-Type of application/json this will be matched automatically in the response.
    - If your client sends a JSON post with a Content-Type of text/plain this will be matched automatically in the response too

of-whatchdog

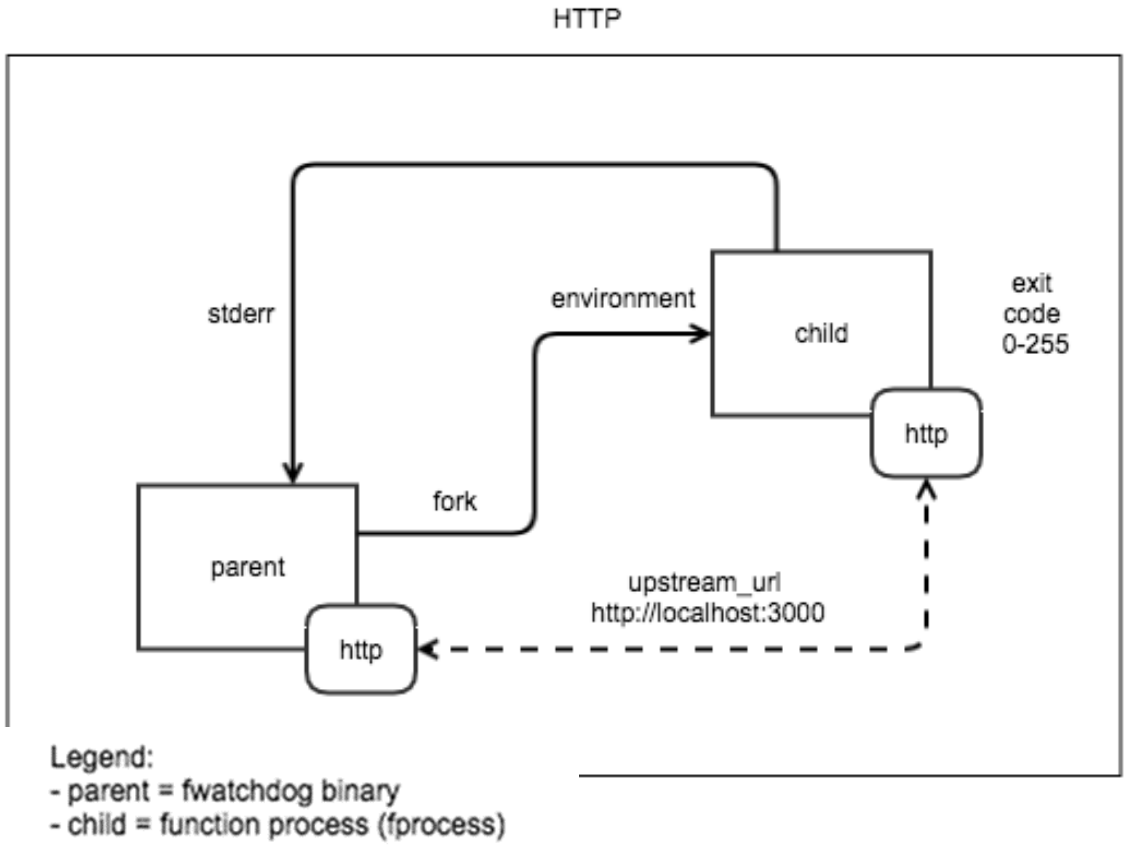
- It introduced an **improved control over HTTP responses**, "**hot functions**", **persistent connection pools** or to **cache a machine-learning model in memory**.
- The of-watchdog implements an HTTP server listening on port 8080, and acts as a reverse proxy for running functions and microservices.
- It does not aim to replace the Classic Watchdog, but offers another option for those who need the additional functions introduced.

of-whatchdog

Goals

- Keep function process **warm** for lower latency / caching / persistent connections through using HTTP
- Enable **streaming of large responses from functions**, beyond the RAM or disk capacity of the container
- Different modes available for the of-watchdog which changes how it interacts with your microservice or function code.

of-whatchdog HTTP mode



HTTP mode - the default and most efficient option

HTTP mode is recommended for all templates where the target language has a HTTP server implementation available.

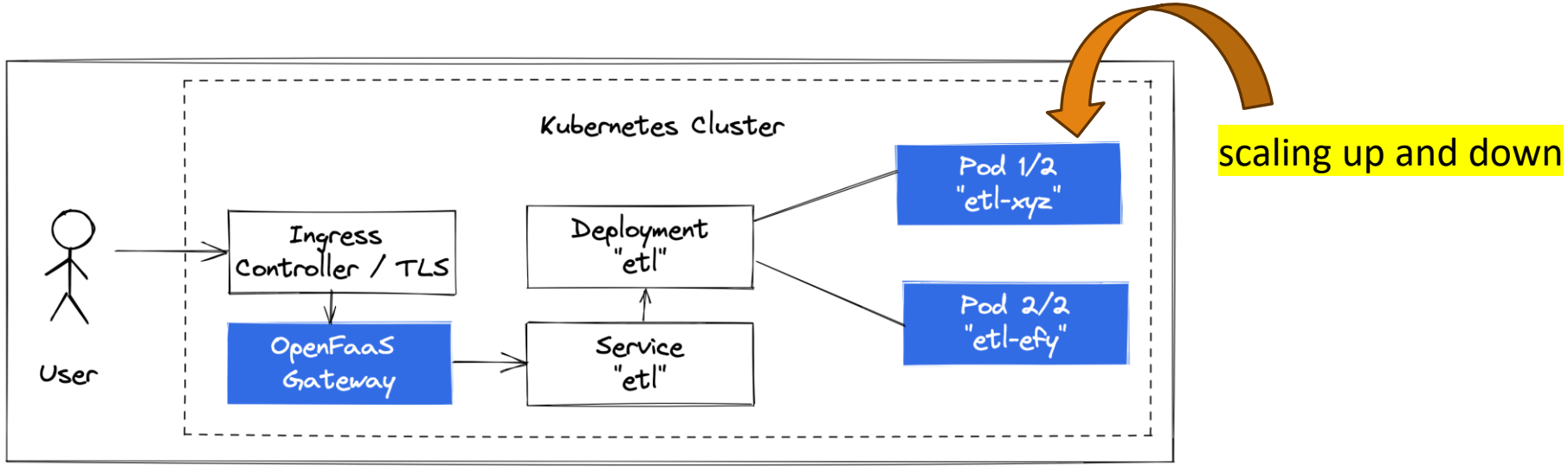
HTTP reverse proxy: A process is forked when the watchdog starts, then any request incoming to the watchdog is forwarded to to a HTTP port within the container.

Invocation of functions

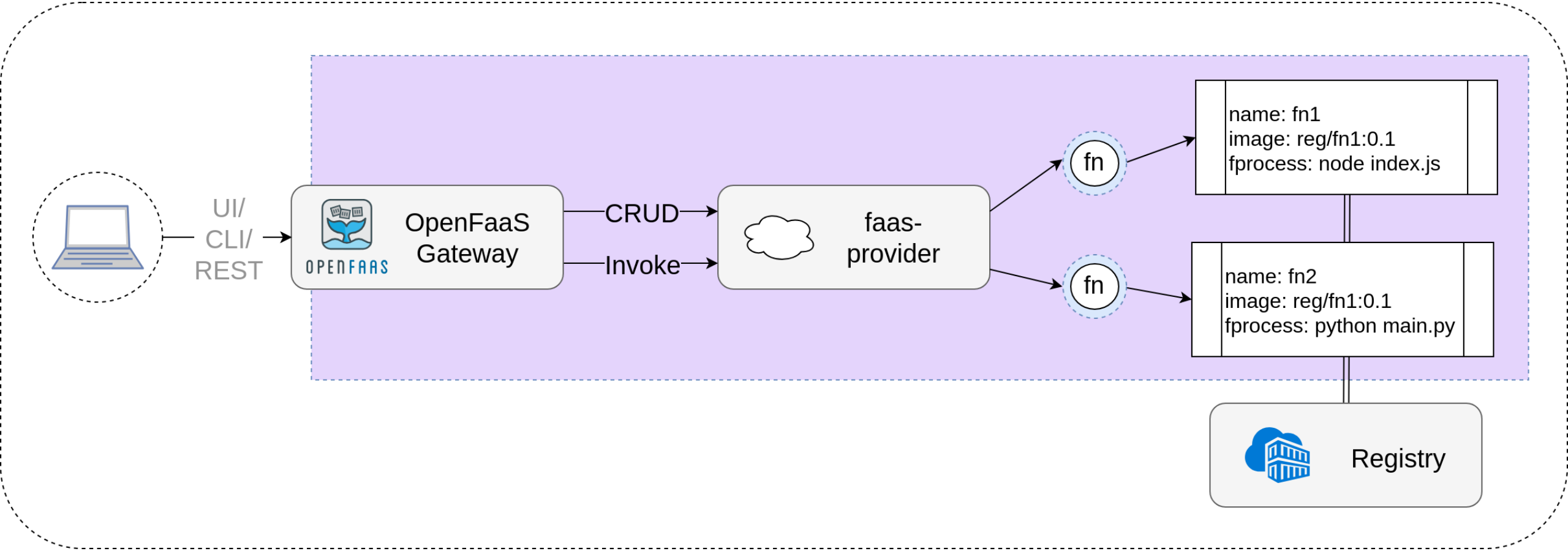
Synchronous function invocations:

In an OpenFaaS installation, functions can be invoked through **HTTP requests** to the **OpenFaaS gateway**, specifying the path as part of the URL.

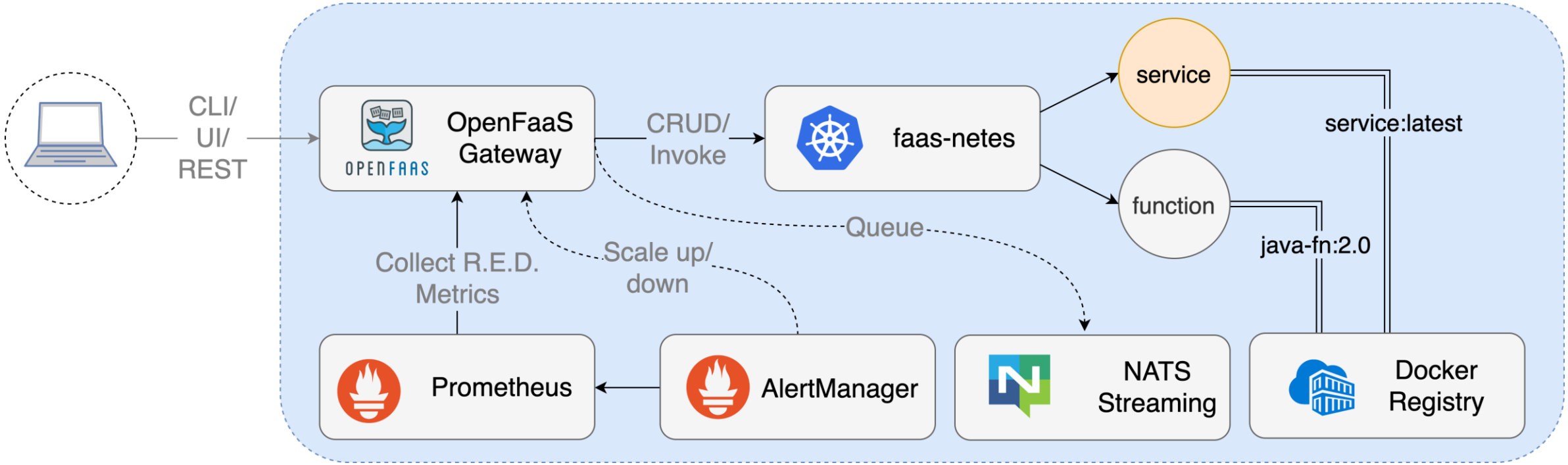
Each function is deployed as a Kubernetes Deployment and Service



Conceptual workflow



Conceptual workflow



faas-netes is an OpenFaaS provider which enables Kubernetes for OpenFaaS. It's part of a larger stack that brings a cloud-agnostic serverless experience to Kubernetes.

nah

Invocation of functions

Synchronous function invocations:

- During development you may invoke the OpenFaaS gateway using a HTTP request to *http://127.0.0.1:8080/function/NAME*, where NAME is the name of the function.
- When you move to production, you may have another layer between your users and the gateway such as a reverse proxy or Kubernetes Ingress Controller.
- The connection between the caller and the function remains connected until the invocation has completed, or times out.

Invocation of functions

nah

Asynchronous function invocations:

With an asynchronous invocation, the HTTP request is enqueued to **NATS**, followed by an "accepted" header and call-id being returned to the caller. Next, at some time in the future, a separate queue-worker component dequeues the message and invokes the function synchronously.

There is never any direct connection between the caller and the function, so the caller gets an immediate response, and can subscribe for a response via a webhook when the result of the invocation is available.

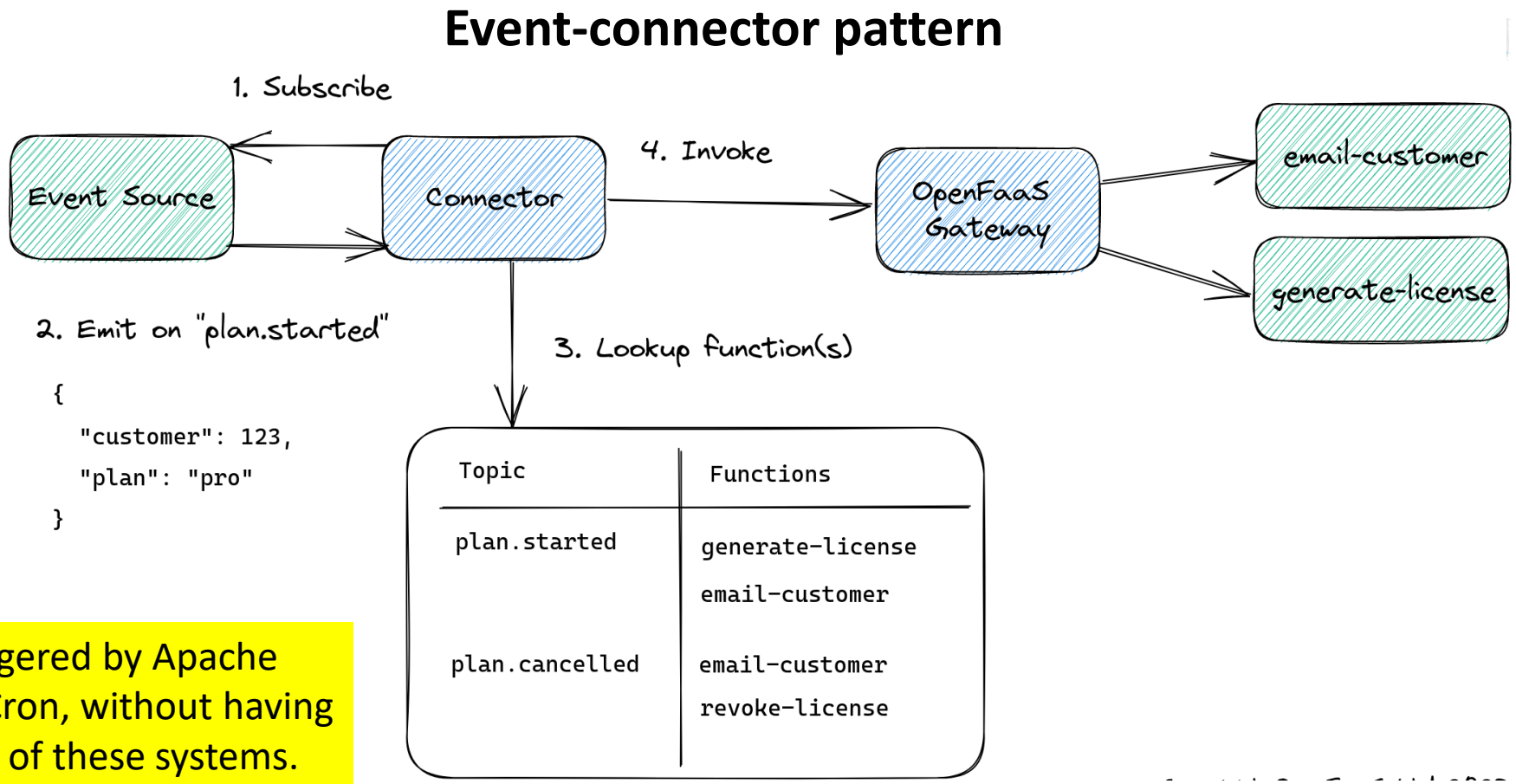
Events

- OpenFaaS functions can be triggered easily by any kind of event. The most common use-case is HTTP which acts as a lingua franca between internet-connected systems.
- A number of **event triggers** are supported in OpenFaaS. With each of them, a long-running daemon subscribes to a topic or queue, then when it receives messages looks up the relevant **functions** and invokes them synchronously or asynchronously.
- Popular event sources include: Cron, Apache Kafka and AWS SQS.

Built-in sample triggers¶
HTTP / webhooks¶
This is the default, and standard method for interacting with your Functions.

CLI¶
Trigger a function using the faas-cli by using the function name

Events



A function can be triggered by Apache Kafka, AWS SNS and Cron, without having any direct coupling to of these systems.

Autoscaling

- Autoscaling is a feature available in OpenFaaS that scales up or scales down function replicas based on demand. This feature was built using both Prometheus and the Alert Manager components available with the OpenFaaS framework.
- Alert Manager will fire alerts when the function invocation frequency exceeds the defined threshold.
- While deploying the functions, the following labels are used to control the number of minimum replicas, maximum replicas, and the increase/decrease factor of the functions:
  - **com.openfaas.scale.min** – This defines the initial number of replicas, which is 1 by default.
  - **com.openfaas.scale.max** – This defines the maximum number of replicas.
  - **com.openfaas.scale.factor** – This defines the percentage of pod replica increase (or decrease) when the Alert Manager fires the alerts. By default, this is set to 20% and should have a value between 0 and 100.

Scaling modes

There are three auto-scaling modes: **Capacity, RPS, CPU**

- When configuring auto-scaling for a function, you need to set a target number which is the **average load per replica** of your function.
- **Each mode** can be used to record a current load for a function across all replicas in the OpenFaaS cluster.
- Then, a query is run periodically to calculate the current load.
- The current load is used to calculate the new number of replicas.

desired = ready pods * (mean load per pod / target load per pod) **<- CEIL()**

Scaling modes

The target-proportion flag value, in the range [0,1], can be used to adjust scaling:

desired = ready pods * (mean load per pod / (target load per pod * **target-proportion**))

Scaling modes

- **Capacity** - Based upon inflight requests (or connections). Ideal for long-running functions or functions which can only handle a limited number of requests at once.
  A hard limit can be enforced through the ***max_inflight*** environment variable on the function.
- **RPS** - Based upon requests per second completed by the function. A good fit for functions which execute quickly and have **high throughput**. You can tune this value on a per function basis.
- **CPU** - Based upon CPU usage of the function. Ideal for CPU-bound workloads, or where Capacity and RPS are not giving the optimal scaling profile. The value configured here is in **milli-CPU**.
- **Scaling to zero -** Scaling to zero is an opt-in feature on a per function basis. It can be used in combination with any of the three scaling modes listed above.

Autoscaling Example

For example:

- sleep is running in the **capacity mode** and has a **target load of 5 in-flight requests**.
- The load on the sleep function is measured as **15 inflight requests**.
- There is only one replica of the sleep function because its **minimum range is set to 1**.
- We are assuming **com.openfaas.scale.target-proportion is set to 1.0** (100%).

mean per pod = 15 / 1

3 = ceil (1 * (15 / 5 * 1))

Therefore, 3 replicas will be set.

Autoscaling Example

With **3 replicas** and **25 ongoing requests**, the load will be spread more evenly, and evaluate as follows:

mean per pod = 25 / 3 = 8.33

5 = ceil(3 * (8.33 / 5 * 1))

When the load is no longer present, it will evaluate as follows:

mean per pod = 0 / 3 = 0 0 = ceil (3 * (0 / 5 * 1))

But the function will not be set to zero yet, it will be brought up to the minimum range which is 1.

Autoscaling Example

If you are limiting how much concurrency goes to a function, let's say for 100 requests maximum, then you may want to set the target to 100 with a proportion of 0.7, in this instance, when there are 70 ongoing requests, the autoscaler will add more replicas:

total load = 90
mean per pod = 90 / 1 = 90

2 = ceil(1 * (90 / (100 * 0.7)))

Scaling up from zero replicas

The latency between accepting a request for an unavailable function and serving the request is sometimes called a "**Cold Start**".

- The cold start in OpenFaaS is strictly optional and it is recommended that for time-sensitive operations you avoid one by having a minimum scale of 1 or more replicas. This can be achieved by not scaling critical functions down to zero replicas, or by invoking them through the asynchronous route which decouples the request time from the caller.
- The "Cold Start" consists of the following: creating a request to schedule a container on a node, finding a suitable node, pulling the Docker image and running the initial checks once the container is up and running. This "running" or "ready" state also has to be synchronised between all nodes in the cluster. The total value can be reduced by pre-pulling images on each node and by setting the Kubernetes Liveness and Readiness Probes to run at a faster cadence.

Sources:

- Onur Yılmaz Sathsara Sarathchandra: «Serverless Architectures with Kubernetes». Packt Publishing, 2019.
- https://docs.openfaas.com/
- https://docs.openfaas.com/architecture/autoscaling/
- https://github.com/openfaas/classic-watchdog/blob/master/README.md
- https://github.com/openfaas/of-watchdog/blob/master/README.md
- https://iximiuz.com/en/posts/openfaas-case-study/