1- ON-PREMISES INFRASTRUCTURE

1.1 Introduction:

As we already said, to improve and optimize the architecture implemented on AWS Lambda and Azure Functions, we need to switch to an on-premises infrastructure in order to have full control on the underlying components that allow to run serverless applications. The starting infrastructure we have available is basically composed of two components: message-oriented middleware (MOM) and invoker. So, its architecture can be represented as follows:

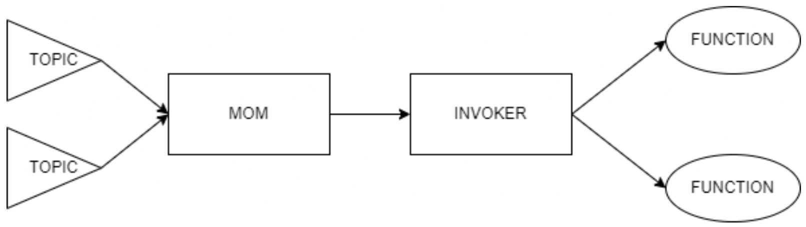


Figure 1.1: On-premises infrastructure architecture

The idea behind the infrastructure is to use the topics as triggers for our functions. In this way, we can associate a specific invoker to make it execute the correspondent function in response to the publication of a message on that topic. In the following paragraphs we will discuss about the technologies used to implement the infrastructure.

1.2 NATS:

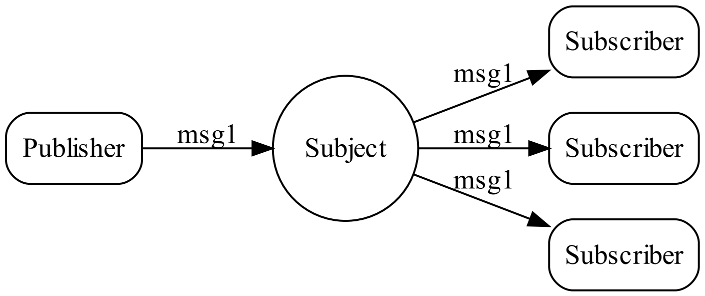
NATS is a high performance, open-source MOM designed to facilitate real-time communication between distributed systems, applications and devices. The NATS services are provided by one or more NATS server processes that are configured to interconnect with each other and provide a NATS service infrastructure. The NATS service infrastructure can scale from a single NATS server process running on an end device all the way to a public global super-cluster of many clusters spanning all major cloud providers and all regions of the world.

NATS is a subject-based messaging system that allows to publish and listen for messages on named communication channels called `Subjects`. At its simplest, a subject is just a string of characters that form a name the publisher and subscriber can use to find each other. More commonly subject hierarchies are used to scope messages into semantic namespaces. Through subject-based addressing, NATS provides location transparency across a cloud of routed NATS servers and a default many-to-many (M:N) communication pattern.

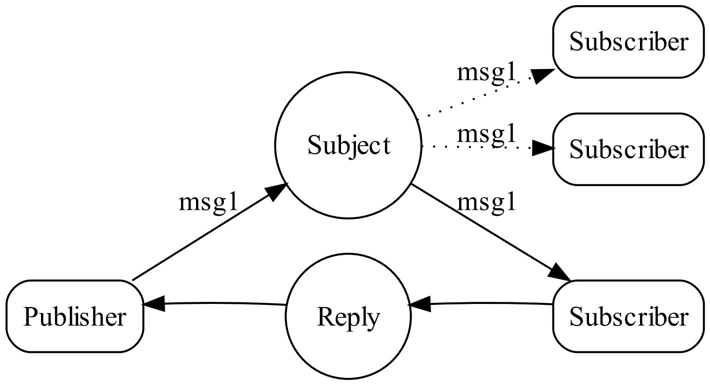
- Subject subscriptions are automatically propagated within the server cloud.
- Messages will be automatically routed to all interested subscribers, independent of location.
- Messages with no subscribers to their subject are automatically discarded.

The foundational functionality of NATS system is implemented in Core NATS. It operates in a publish-subscribe model using subject-based addressing. Starting from this base model, Core NATS offers three ways to exchange messages:

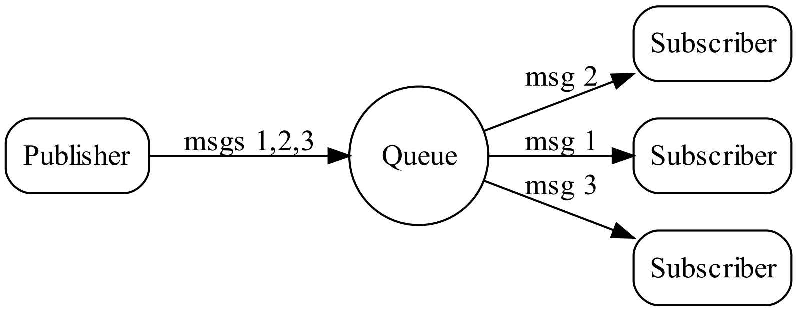
- Traditional publish-subscribe message distribution model: A publisher sends a message on a subject and any active subscriber listening on that subject receives the message.



- Request-reply: A request is published on a given subject using a reply subject. Responders listen on that subject and send responses to the reply subject. Reply subjects are called "inbox". These are unique subjects that are dynamically directed back to the requester, regardless of the location of either party.



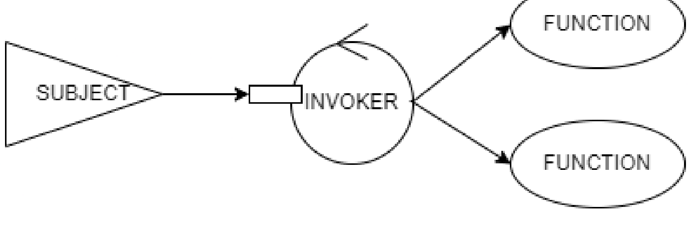
- Queue groups: NATS provides an additional feature named "queue", which allows subscribers to register themselves as part of a queue. Subscribers that are part of a queue form the "queue group". In this way, when a message is published on a subject, only a randomly chosen subscriber of that subject receives the message.



Core NATS offers only a best-effort quality of service, in particular an at-most-once semantics for the delivery of messages. Anyway, it is possible to enable the built-in persistence engine, called JetStream, to reach higher levels of service quality.

1.3 Invoker:

The invoker is the component designed for the association of the functions with specific subjects, which are the triggers of our infrastructure. So, an invoker is basically a NATS client that subscribes to a specific subject in a group queue and waits for messages (events) to be published on it. For every message, it runs the associated function in a separated process, passing the message as input and publishing the output to a target output topic.



The invoker is implemented in Rust and uses the client SDK provided by NATS to connect to a NATS server and deliver messages. It also uses the Tokio library to manage concurrency in an asynchronous way. In fact, we don’t want to block the invoker process during the execution of each instance of the function, but to let it continuously read new messages from the subject. So, the invoker basically acts as a control loop that delegates the messages to new tasks which will execute the function code synchronously in other processes. The code structure can be summarized as follows (pseudo-code):

src/invoker.rs

```
/* get environment variables (NATS server URL, subject
name, command name and arguments, etc…) */

let nc = nats::connect(&nats_server)?;

let sub = nc.queue_subscribe(&trigger_topic, &group)?;

for msg in sub.messages{

     tokio::spawn( async {

          /* execute function code and publish result on
          the output topic */

     });

}
```

In this way, it is possible to replicate the behavior of traditional FaaS platforms. In fact, if we keep in mind the traditional architecture of a FaaS platform, it is possible to notice that we have just used NATS as controller and its subjects as triggers, and our invoker acts as a standard invoker that associates to incoming events the execution of a certain function. Starting from this infrastructure, we will concentrate on how to implement dependency injection and use it in our functions.

2- FIRST ARCHITECTURE

2.1 Introduction:

As we have already done for the preliminary tests on public clouds, we want to implement our first dependency injection architecture on the new available infrastructure. Basically, the previous solution consisted of four elements on top of the FaaS platform:

- Client: the function that needs dependencies to be injected.
- Dependency provider: the dependency that should be injected.
- Injector: SDK that provides an API to perform CRUD operations (registration and retrieval of services) on a registry.
- Registry: storage for the dependency declarations.

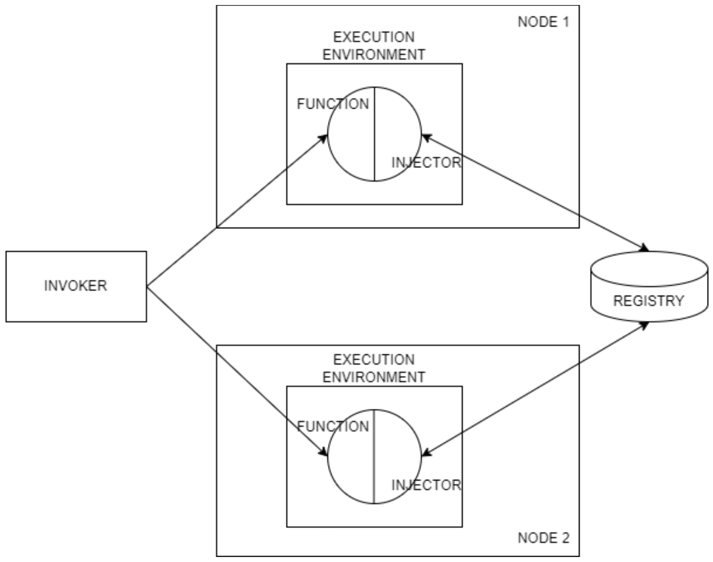


Figure 2.1: first architecture on public clouds.

For the previous tests, we decided to implement the Injector SDK in three different languages and to use the already integrated NoSQL database services as registries. We also created a simple "hello world!" Go function to use as dependency provider and we exploited the existing monitoring and logging services to collect performance parameters.

In this case, we don't have any kind of integrated feature available, but we need to build the resources that we need starting from the infrastructure presented in the previous section. Furthermore, we will use more significant dependency providers in order to replicate a more realistic use case. In particular, we will inject two functions that implement access control and logging features. In this chapter, we will discuss about the technologies used to implement the first architecture and how we put together all the components that we need. We will finally evaluate the performance of such a solution and try to optimize it.

2.2 Architecture:

Given all the consideration made in chapter 1, and keeping in mind Figure 2.1, we need to make some considerations before we start. As we already said, the invoker acts as an infinite loop that waits for messages to be published on a specific topic and creates a new process that executes the code of the associated function every time a message arrives. So, we will need to run an invoker for every function in our architecture, with the possibility to run more instances of the same invoker in order to load balance the upcoming messages. In fact, thanks to the "queue groups" feature of NATS, the balancing is offered for free by the MOM, since that the invokers of a same function will subscribe to the same group.

To enable communication between functions it is possible to use the messages published on NATS. So, it is possible to uniform the message payloads in order to make them serializable and deserializable for all the invokers. We decided to format the payloads as JSON objects, so that they can be easily managed with all the programming languages. In fact, we will implement the Injector SDK with the same languages used on public clouds (Java, Javascript and Go).

With regard to the registry, we decided to implement it using MongoDB, a popular NoSQL database designed to handle large volumes of unstructured, semi-structured, or structured data efficiently. In this way, we can represent dependencies as a tuple composed of three elements: identifier, name and topic. In fact, in this case the triggers are the topics our functions subscribe to, through their invokers, not the web URLs.

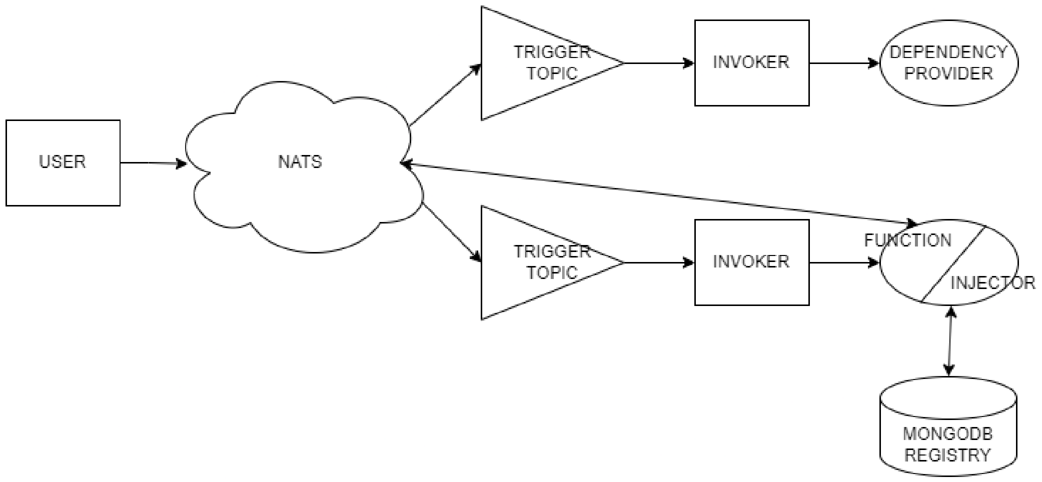


Figure 2.2: first architecture on the new infrastructure

In this way, we can replicate the same behavior of the tests conducted on the public clouds, in order to evaluate the performance differences and make further improvements.

2.3 Implementation:

For the initial implementation, we consider a situation in which all the components run locally on the same Windows machine. In this way, we can concentrate on the interconnection between them and on the business logic, leaving the distribution to a later time.

First of all, we need to download and install the last version of NATS server (v 2.0.0). For our purposes, we just need the core features of NATS, so we don’t need to install the JetStream extension. Once it is installed, we can run the server by running the following command:

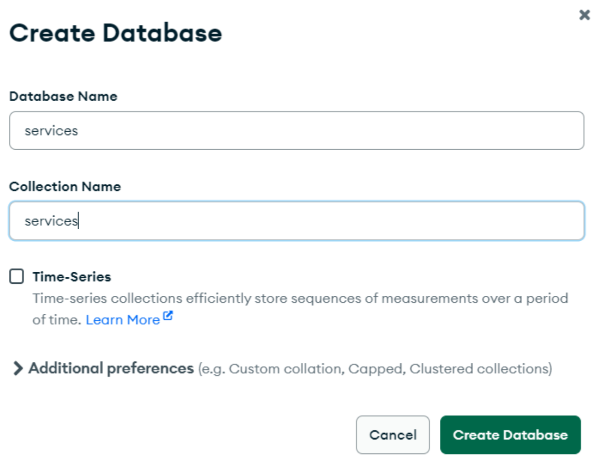
```
$    nats-server
```

We can test it by subscribing and publishing to specific subjects. To do it, we can use the following commands:

```
$    nats sub ‘some_topic’

$    nats pub ‘some_topic’ ‘message’
```

With regard to MongoDB, it is possible to download and install the community edition (v 8.0) for free. During the installation, it is possible to download MongoDBCompass, a useful GUI to interact with the MongoDB server in a simple way. So, we can easily create a new database to use as registry in the corresponding section:


Create Database

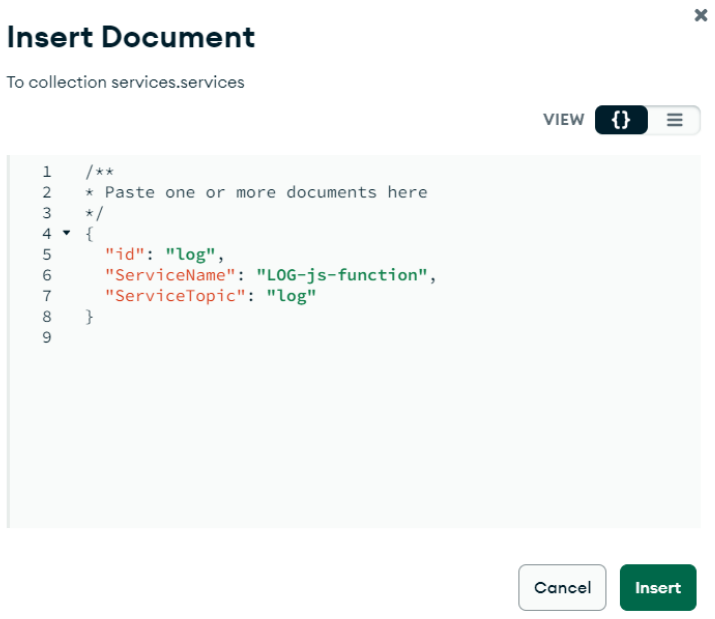
Database Name: services

Collection Name: services

☐ Time-Series
Time-series collections efficiently store sequences of measurements over a period of time. Learn More

> Additional preferences (e.g. Custom collation, Capped, Clustered collections)
Cancel
Create Database

Once the collection is created, it is possible to insert new documents as key:value couples. So, we can add the entries containing the information about our dependency providers, two Javascript functions implementing access control and logging features. As we have already said, we can represent those functions with three attributes: id, name and topic:


Insert Document

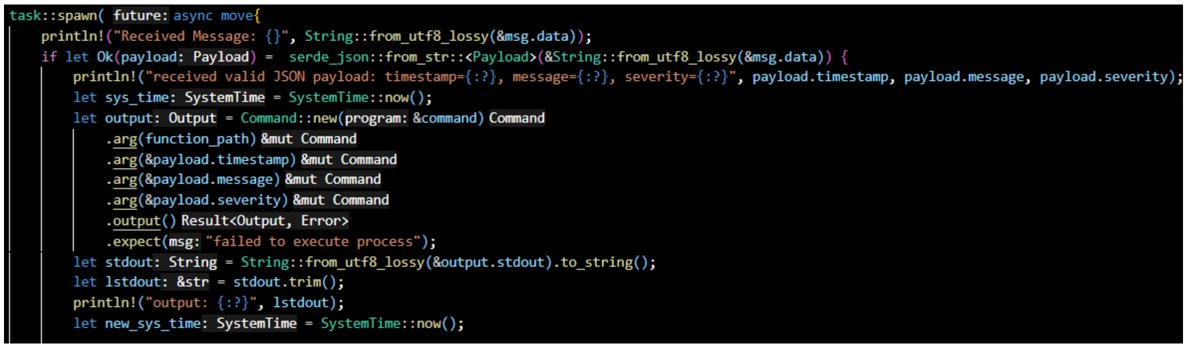
To collection services.services

```
/**
* Paste one or more documents here
*/
{
  "id": "log",
  "ServiceName": "LOG-js-function",
  "ServiceTopic": "log"
}
```

Now that the registry is initialized, we can concentrate on the invokers. First of all, let's consider the invokers related to the dependency providers. As said in chapter 1, they are implemented in Rust and consist of an infinite loop in which a new thread is started whenever a new message is published on the subscription topic. The invoker needs some parameters to run:

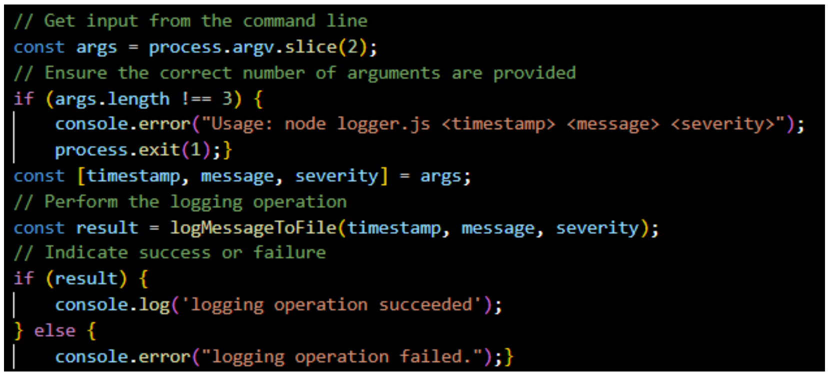
- `command`: the command to execute in order to run the function code. In this case, given that the logging function is written in Javascript, the command will be '`node`'.
- `function_path`: the path to the file that implements the function (`path/to/logger.js`).
- `trigger_topic`: the topic the invoker should subscribe to. It represents the trigger of the function.
- `output_topic`: the topic the invoker should send the output to.
- `nats_server`: the IP address of the NATS server.
- `group`: the name of the group to which subscribe. In this way, different instances of the same invoker can subscribe to a single queue group.

Thanks to the Tokio library, it is possible to run asynchronous tasks that contain the business logic needed to parse the message payloads, run the Javascript function and publish the output on the specified topic.



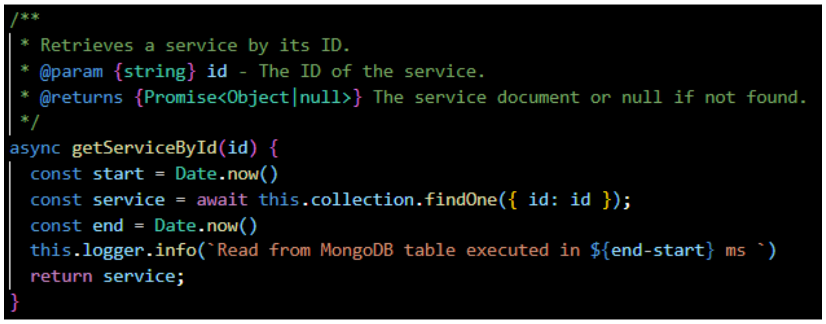
```
task::spawn( future: async move{
    println!("Received Message: {}", String::from_utf8_lossy(&msg.data));
    if let Ok(payload: Payload) =  serde_json::from_str::<Payload>(&String::from_utf8_lossy(&msg.data)) {
        println!("received valid JSON payload: timestamp={:?}, message={:?}, severity={:?}", payload.timestamp, payload.message, payload.severity);
        let sys_time: SystemTime = SystemTime::now();
        let output: Output = Command::new(program: &command) Command
            .arg(function_path) &mut Command
            .arg(&payload.timestamp) &mut Command
            .arg(&payload.message) &mut Command
            .arg(&payload.severity) &mut Command
            .output() Result<Output, Error>
            .expect(msg: "failed to execute process");
        let stdout: String = String::from_utf8_lossy(&output.stdout).to_string();
        let lstdout: &str = stdout.trim();
        println!("output: {:?}", lstdout);
        let new_sys_time: SystemTime = SystemTime::now();
```

In this way, the Javascript functions are decoupled from the infrastructure, and they will contain only the code that represents their business logic. In fact, if we consider the logging function, we can notice that it could easily run as a stand-alone application. The only one constraint is that it should take as input the parameters passed as arguments on the command line. Furthermore, the invoker will consider as output produced by the function all the text printed on the standard output of the process running the function.



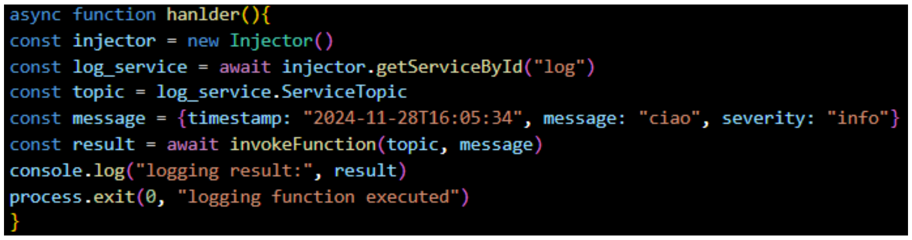
```
// Get input from the command line
const args = process.argv.slice(2);
// Ensure the correct number of arguments are provided
if (args.length !== 3) {
    console.error("Usage: node logger.js <timestamp> <message> <severity>");
    process.exit(1);}
const [timestamp, message, severity] = args;
// Perform the logging operation
const result = logMessageToFile(timestamp, message, severity);
// Indicate success or failure
if (result) {
    console.log('logging operation succeeded');
} else {
    console.error("logging operation failed.");}
```

With regard to the injector, we decided to implement it in three different languages: Javascript, Java and Go. Basically, it is an SDK that exposes methods to register and retrieve services from MongoDB. So, it uses the clients provided by MongoDB for the different programming languages to interact with the server.



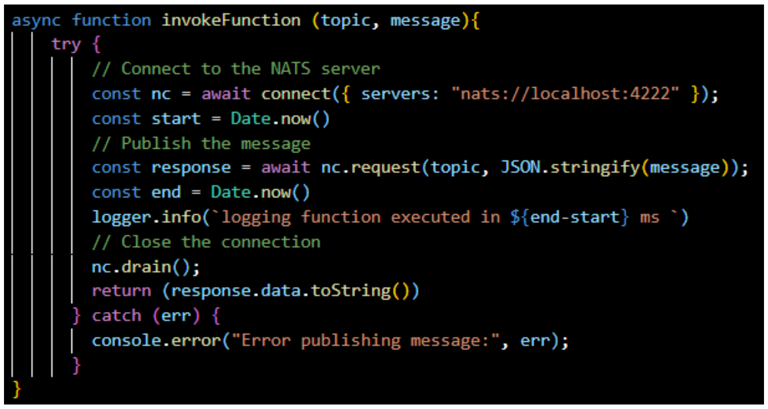
```
/**
 * Retrieves a service by its ID.
 * @param {string} id - The ID of the service.
 * @returns {Promise<Object|null>} The service document or null if not found.
 */
async getServiceById(id) {
  const start = Date.now()
  const service = await this.collection.findOne({ id: id });
  const end = Date.now()
  this.logger.info(`Read from MongoDB table executed in ${end-start} ms `)
  return service;
}
```

We created an example function that uses this injector to retrieve and invoke the Javascript functions.



```
async function hanlder(){
const injector = new Injector()
const log_service = await injector.getServiceById("log")
const topic = log_service.ServiceTopic
const message = {timestamp: "2024-11-28T16:05:34", message: "ciao", severity: "info"}
const result = await invokeFunction(topic, message)
console.log("logging result:", result)
process.exit(0, "logging function executed")
}
```

We have to remember that, for the moment, the only one way to make two functions communicate is that one publishes a message on the topic to which the other's invoker is subscribed. To do so, our example function has to use the provided NATS client and to pass the topic retrieved by the injector as parameter.



```
async function invokeFunction (topic, message){
    try {
        // Connect to the NATS server
        const nc = await connect({ servers: "nats://localhost:4222" });
        const start = Date.now()
        // Publish the message
        const response = await nc.request(topic, JSON.stringify(message));
        const end = Date.now()
        logger.info(`logging function executed in ${end-start} ms `)
        // Close the connection
        nc.drain();
        return (response.data.toString())
    } catch (err) {
        console.error("Error publishing message:", err);
    }
}
```

Obviously, also the example function is invoked by another associated invoker.

Doing so, it is possible to run locally all the components that realize the infrastructure (NATS server, MongoDB and invokers) and make them communicate to obtain the behavior of traditional FaaS platforms. Publishing a message on the subscription topic of the example function will trigger its execution, causing the retrieval of the dependency provider from the registry and its invocation.

2.3.1 Distributing the infrastructure:

To distribute the infrastructure on more nodes we used three Ubuntu 22.04 virtual machines. On the first VM (VM1), we installed NATS server, version 2.0, by running the following commands:

```
wget https://github.com/nats-io/nats-server/releases/
download/v2.0.0/nats-server-v2.0.0-linux-amd64.tar.gz
```

```
tar -xvzf nats-server-v2.0.0-linux-amd64.tar.gz
```

```
sudo mv nats-server-v2.0.0-linux-amd64/nats-server/
usr/local/bin/
```

On the same VM we have also installed MongoDB version 8 by running the following commands:

```
curl -fsSL https://www.mongodb.org/static/pgp/
server-8.0.asc | sudo gpg -o /usr/share/keyrings/
mongodb-server-8.0.gpg --dearmor
```

```
echo "deb [ arch=amd64,arm64 signed-by=/usr/share/keyrings/mongodb-server-8.0.gpg]https://repo.mongodb.org/apt/ubuntu noble/mongodb-org/8.0 multiverse" | sudo tee /etc/apt/sources.list.d/ mongodb-org-8.0.list

sudo apt-get update

sudo apt-get install -y mongodb-org
```

Once it is installed, it is possible to interact with the `mongod` service through the dedicated shell by running the command `mongosh`. In this way, we can create a new database containing our collection, as we already have done on the Windows machine. We can also initialize the table with the two service providers with the command `db.collection.insertOne( … )`, passing a JSON formatted string containing the data.

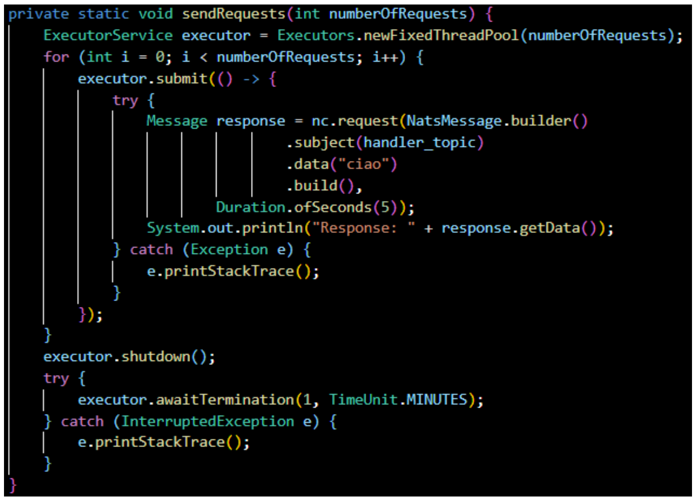
By default, MongoDB launches with `bindIp` set to 127.0.0.1, which binds to the localhost network interface. This means that the `mongod` can only accept connections from clients that are running on the same machine. To modify this behavior, it is possible to change the `bindIp` option in the MongoDB configuration file to 0.0.0.0. In this way, `mongod` will accept connections from any node.

On VM2, we decided to run the invokers related to the dependency providers, while, on VM3, the invoker related to the example function. Of course, we had to modify the code in order to update the URLs of the NATS server and the MongoDB instance. In this way, we can recreate a more realistic situation, in which we can't know on which nodes our functions will be executed a priori. On the most known FaaS platforms, like AWS Lambda and Azure Functions, even different instances of the same function can run on different nodes, in a transparent way from the point of view of the users. The injector SDK and the NATS middleware allow us to decouple the functions' code from the dependency providers locations.

2.4 Testing the infrastructure:

Once that we set up the environment as described in the previous paragraph, it is possible to start the execution of the example function by publishing a message on its trigger topic (the subject "`handler`" in this case). So, we disposed a stress test to verify the performance of the architecture and the differences between the programming languages that we chose. The stresser simulates a situation of constant number of published messages per second which is followed by a gradual increase until a peak is reached. In this way, we can test the access time to the MongoDB instance and the NATS server performance in message queuing and delivering.

The stresser is implemented in Java using the `java.util.concurrent` library to instantiate parallel threads that can generate more messages at the same time, and the `io.nats.client.Nats` library to publish those messages on the NATS server.



```java
private static void sendRequests(int numberOfRequests) {
    ExecutorService executor = Executors.newFixedThreadPool(numberOfRequests);
    for (int i = 0; i < numberOfRequests; i++) {
        executor.submit(() -> {
            try {
                Message response = nc.request(NatsMessage.builder()
                                        .subject(handler_topic)
                                        .data("ciao")
                                        .build(),
                                Duration.ofSeconds(5));
                System.out.println("Response: " + response.getData());
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
    }
    executor.shutdown();
    try {
        executor.awaitTermination(1, TimeUnit.MINUTES);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

2.4.1 Results:

We kept track of the service retrieval time from MongoDB and the total duration of execution of the example function. In this section, we present the results related to the different programming languages and for both types of dependency provider.

Given that we run a single instance of every invoker before the stresser is executed, the results will not be affected by cold starts. The only bottleneck is represented by the serving time of the published messages. In fact, if the invoker can't serve all the incoming messages by triggering the execution of the example function, the messages will be placed in an in-memory queue internal to the NATS server. In this case, messages can be lost if not consumed in time, given that we are not exploiting the persistence offered by Jetstream. We could also run more instances of the same invokers in order to consume more messages at the same time.

Access control function case:

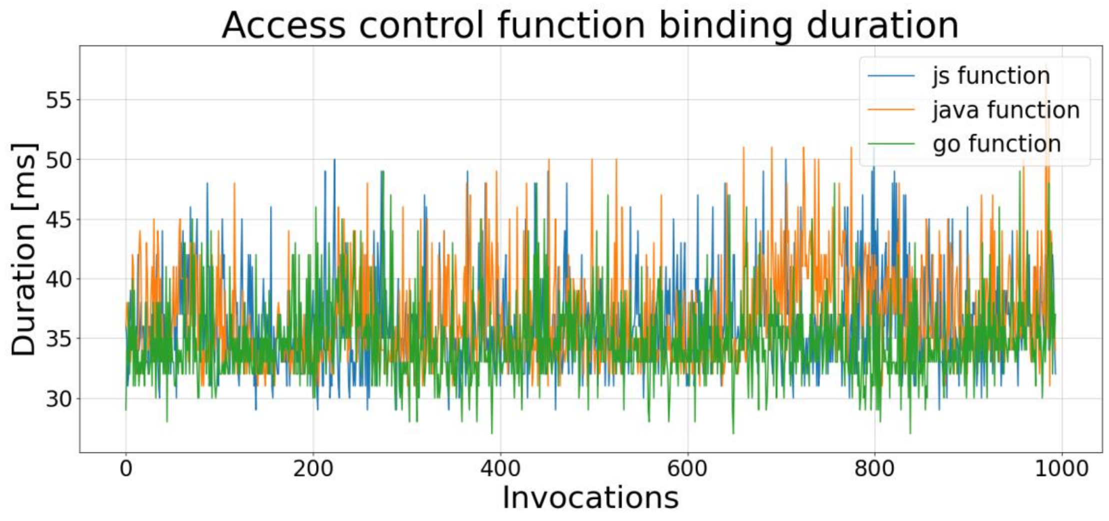


Figure 2.3: Access control function binding duration

In figure 2.3 we collected data related to the service provider binding durations. They include the time needed to invoke the access control function (by publishing a message to its trigger topic) and its execution time. As we can see, all durations are almost equal in the range between 30 and 50 ms. Only the Go example function had a slightly smaller binding time, probably caused by the faster NATS client.

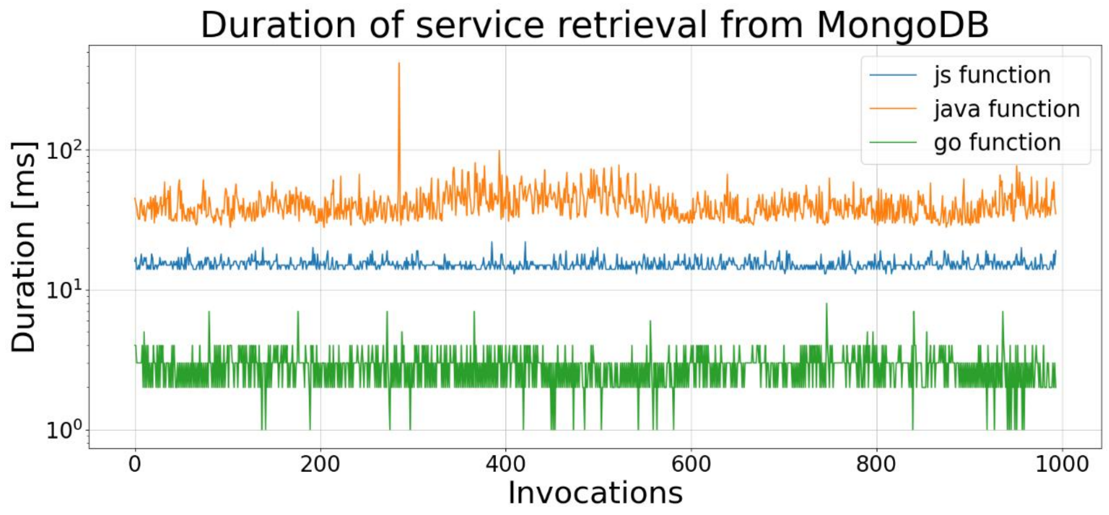


Figure 2.4: Duration of access control service retrieval from MongoDB

In figure 2.4 we can see the durations of service retrievals from MongoDB. It is evident that the Go client is faster than the other ones, registering durations of less than 10 ms.

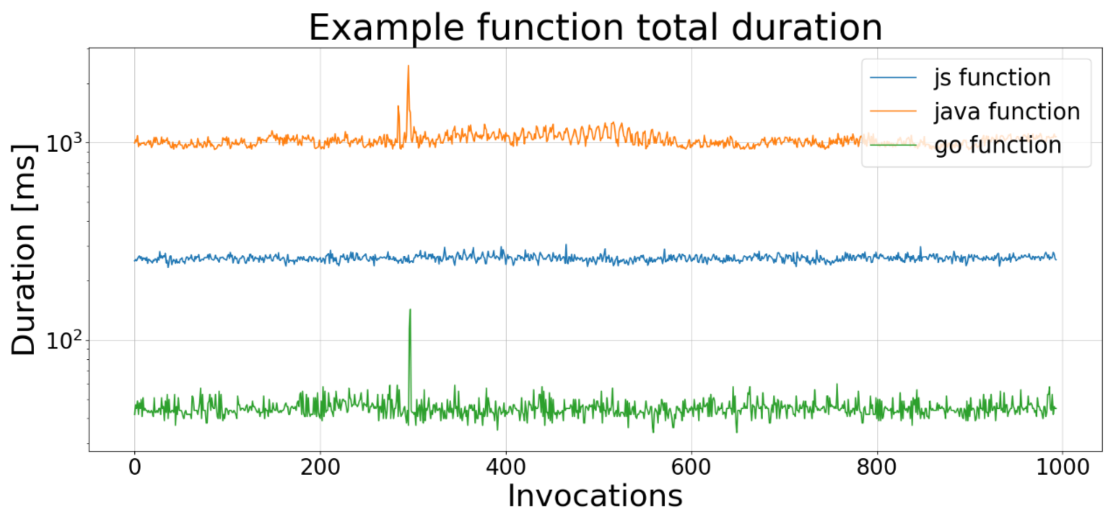


Figure 2.5: Example function total duration

In figure 2.5 we represented the total durations of execution of the example function. They include the execution time of the example function, the service retrieval from MongoDB and the binding time of the access control function. As we can see, the Go function is the fastest, while the Java function took 1 second to terminate. This is probably caused by the huge start up time of the JVM, which represents the majority of the total execution time.

Logging function case:

The metrics collected during this case are very similar to the previous ones, given that the only thing that changed is the service provider. So, we rapidly show the graphics:

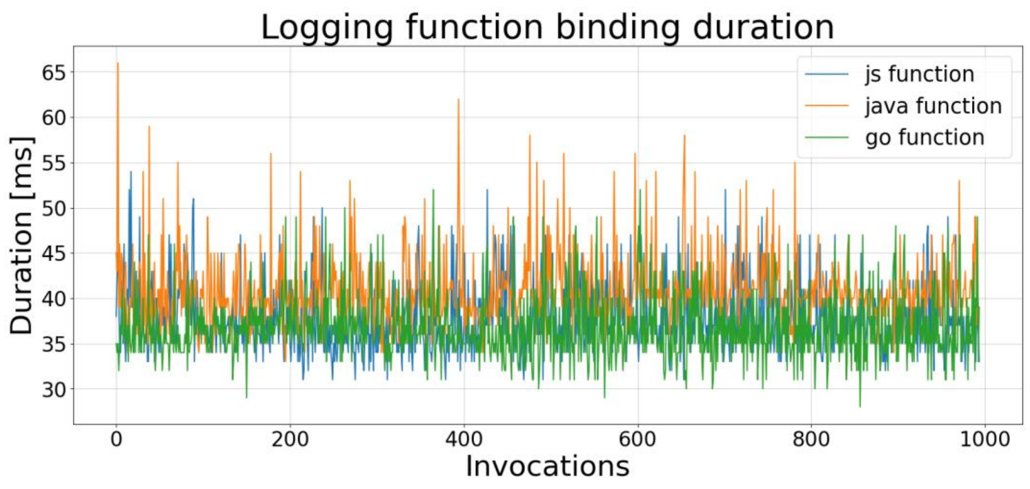


Figure 2.6: Logging function binding duration

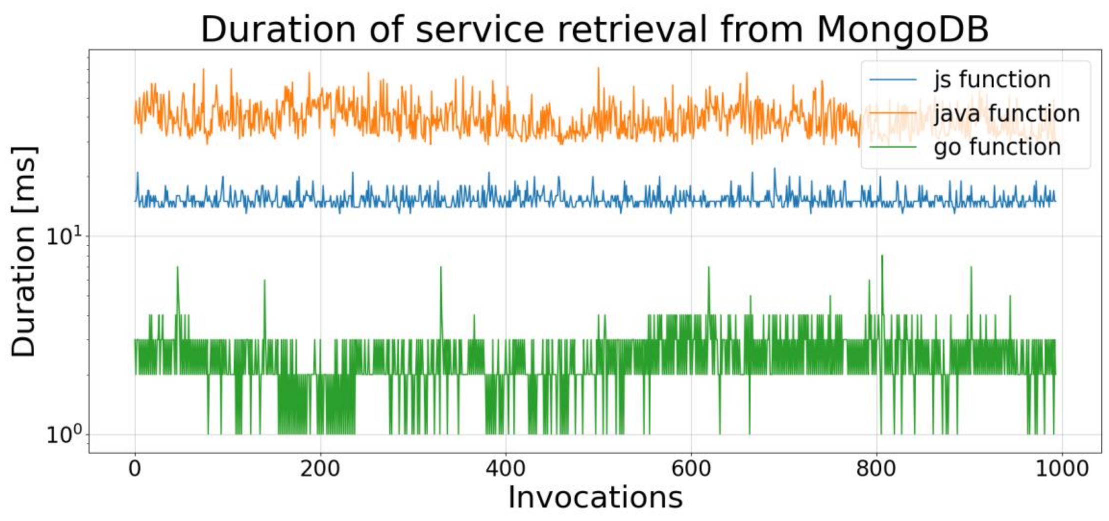


Figure 2.7: Duration of logging service retrieval from MongoDB

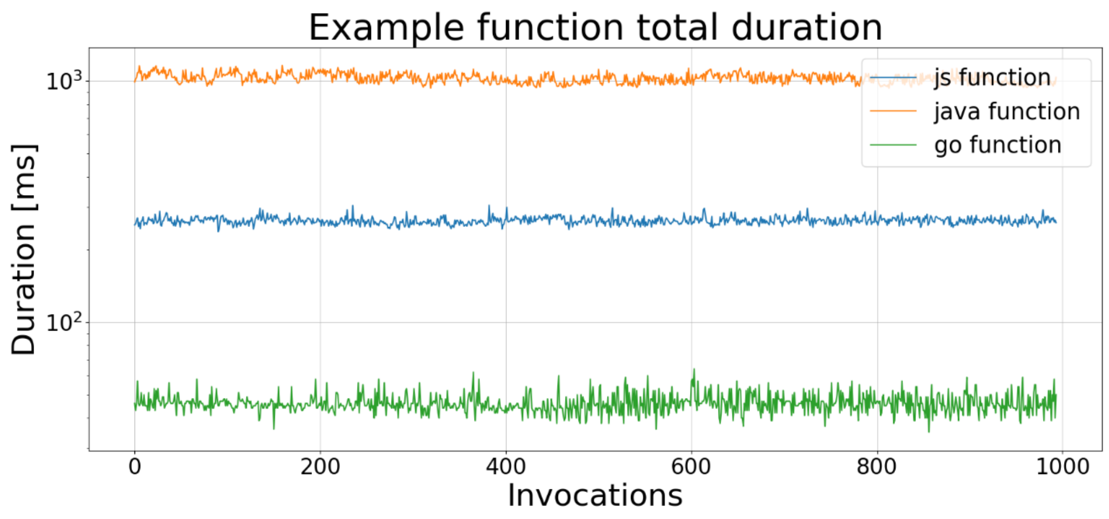


Figure 2.8: Example function total duration

2.5 Virtualization:

Virtualization is a technology that allows the creation of a virtual version of a computing resource, such as an operating system, server, storage device, or network. It enables a single physical machine to host multiple isolated virtual environments, each running its own operating system and applications, without interfering with one another.

Nowadays, virtualizations technologies form the foundation of cloud computing (and, accordingly, of serverless solutions), where virtualized resources are pooled together and delivered over the internet, enabling users to access computing resources on-demand, without the need to manage the underlying physical infrastructure. This is due to many advantages that virtualization brings in such contexts:

- Cost savings: instead of purchasing more hardware every time the needs expand, virtualization allows cloud providers to simply create a new virtual machine on the existing infrastructure.
- Scalability: virtualized resources can be easily created, cloned and scaled based on demand. Cloud providers use virtualization to dynamically allocate resources to different customers (multi-tenancy).
- Isolation and security: each virtualized resource runs independently, isolated by others. This prevents failures or security breaches in one instance from affecting others.
- Portability: virtualized resources can be moved across different servers or even cloud providers without modification.

There are two main virtualization technologies: virtual machines (VMs) and containers. They represent two different approaches to packaging computing environments that combine various IT components and isolate them from the rest of the system. A VM is essentially an emulation of a real computer that run on top of a physical machine using a hypervisor, a software layer that separates the VM from the underlying hardware and that permits to virtualize the physical available resources. In this way, VMs can host their own operative system and all the libraries and applications needed, independently from the host machine characteristics.

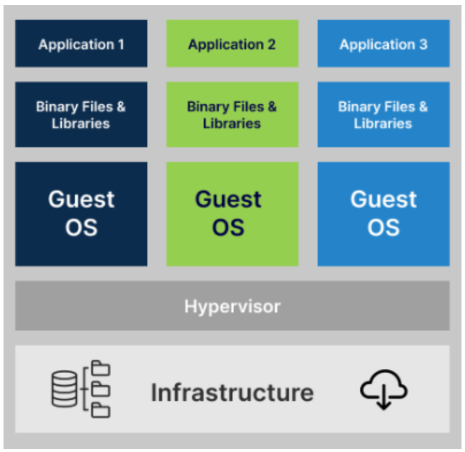


Figure 2.9: Virtual machines architecture

Unlike a VM which provides hardware virtualization, a container provides operating-system-level virtualization by abstracting the "user space". This means that many containers can run on the same host machine sharing the same OS, but with isolated namespaces. Virtualization and isolation are managed by container engines like, for example, Docker.

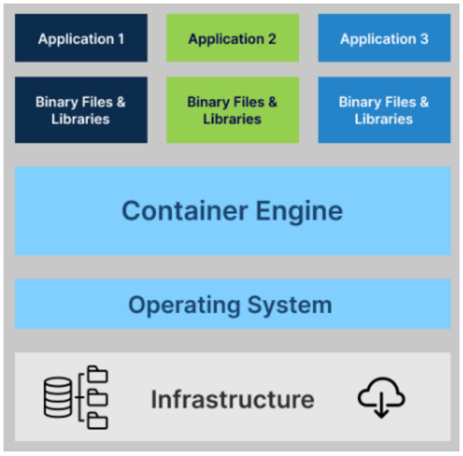


Figure 2.10: Containers architecture

In general, VMs have bigger dimensions, given that each of them has its own full OS. This leads to a series of disadvantages compared to containers, such as slower startup speed and less efficiency for scaling. For these reasons, in ephemeral and dynamic scenarios such as serverless computing systems, containers are usually preferable. So, we will virtualize the first architecture with containers. In particular, we will use Docker.

2.5.1 Docker:

Docker is an open-source engine that automates the deployment of applications into containers. Its architecture is composed of:

- Docker daemon: the core of the platform. It exposes a series of RESTful APIs to manage images, containers, networks and volumes.
- Docker client: command line client library to interact with the Docker engine.
- Docker registry: Docker images repository (DockerHub is the default registry).

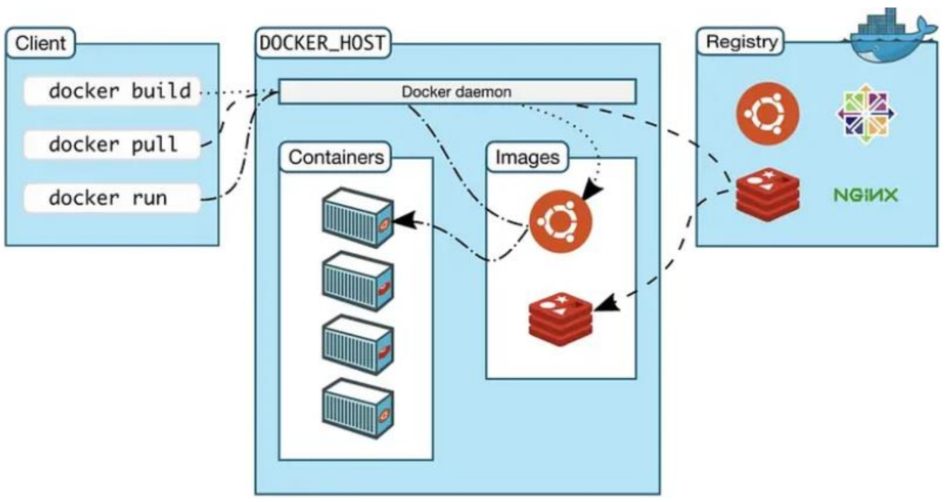


Figure 2.11: Docker architecture.

Docker images are multi-layered self-contained files that act as the template for creating containers. They can be seen as shut down containers, like turned off virtual machines. They are components related to the build time and can be obtained from a remote registry (like DockerHub) or created from zero. Images are structured as a stack of read-only layers, in which every level is univocally identified by a hash code. Each level is built on top of a base image (each layer is dependent from the previous one). They are managed with a copy-on-write approach, so, each layer only keeps track of the differences with the base image, in order to efficiently use the memory. When a container is started from an image, a thin layer which contains runtime operations is added on top of the image.

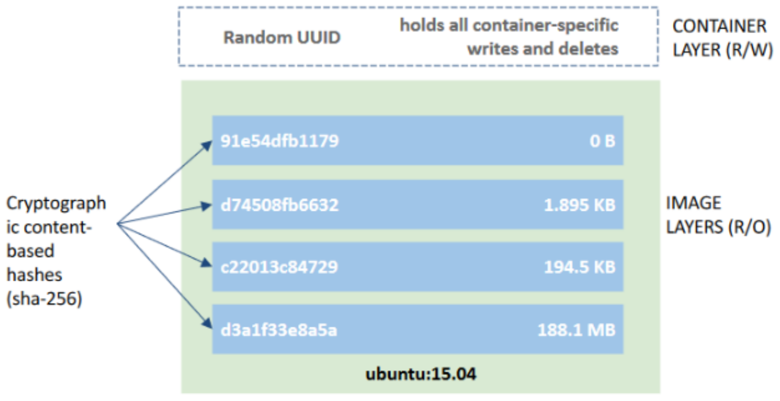


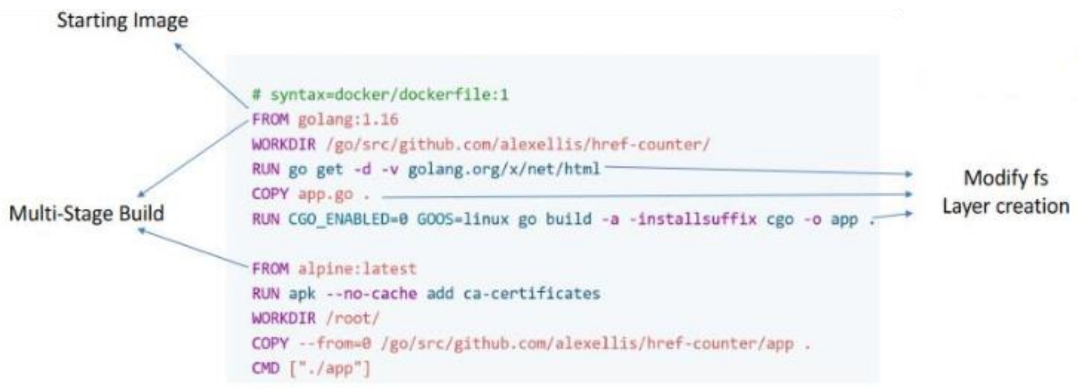
Figure 2.12: Docker image structure.

An image can be create from zero by defining a Dockerfile, a text file that, through a domain specific language, describe a Docker image by the execution of specific commands. In the past, different container engines had different image formats. But later on, the Open Container Initiative (OCI) defined a standard specification for container images which is complied by the major containerization engines out there. This means that an image built with Docker can be used with another runtime like Podman without any additional hassle.

The main instruction that can be used inside a Dockerfile are:

- FROM: to specify the base image from which to start.
- ENV: to specify environment variables in the execution context of the Dockerfile.
- RUN: to execute commands inside the image.
- WORKDIR: to set the working directory inside the image in which the commands will be executed.
- ADD and COPY: to copy files and directories from the build context (the local path to the host in which the Dockerfile is being created) to the image's filesystem.
- CMD: to define which command will be executed when the container will be started.

It is possible to use more FROM instructions to use different base images in order to create a multi-stage build. Each stage has its own set of instruction and build context. Every image generated by an intermediate stage is used as base build context for the next one. The last stage will produce the final image that will be used for the execution. The advantage of doing this is that it is possible to minimize the dimension of the final image, given that useless files and dependencies can be removed in intermediate stages.



```
# syntax=docker/dockerfile:1
FROM golang:1.16
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/github.com/alexellis/href-counter/app .
CMD ["./app"]
```

Figure 2.13: Dockerfile example.

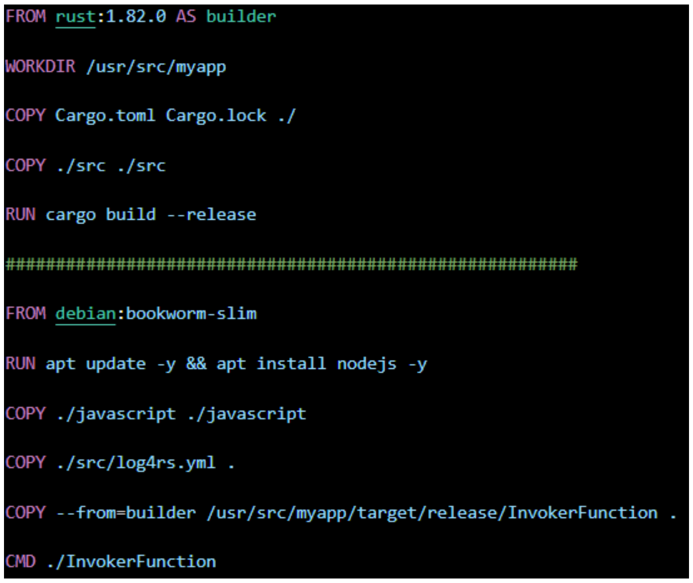
2.5.2 Containerizing the first architecture

As we said, functions in serverless systems usually run inside containers. So, we used Docker in order to containerize our invokers and run the functions inside them. Even the main public cloud FaaS solutions usually have a series of pre-prepared Docker images for every type of execution environment, in which the users' functions code is inserted when uploaded to the platform. For example, AWS Lambda maintains a docker image for every programming language supported by the infrastructure and uses Firecracker (a lightweight virtualization platform) to launch microVMs containing the function code when the function is triggered. Other services external to the containers manage the scaling, the routing, the authorization, etc.

In our case, the invoker is essentially a Rust application that, based on the programming language in which the related function is written, executes another piece of code. So, the container that will host the invoker should be able to execute both the Rust code and the function code. Taking into examination the case of the Javascript example function, we prepared a multi-stage Dockerfile to containerize the invoker. Given that the project structure of the invoker is

```
./InvokerJSFunction
      ./javascript
      ./src
      Cargo.toml
      Cargo.lock
      Dockerfile.yml
```

the resulting Dockerfile is:



```
FROM rust:1.82.0 AS builder

WORKDIR /usr/src/myapp

COPY Cargo.toml Cargo.lock ./

COPY ./src ./src

RUN cargo build --release

#########################################################

FROM debian:bookworm-slim

RUN apt update -y && apt install nodejs -y

COPY ./javascript ./javascript

COPY ./src/log4rs.yml .

COPY --from=builder /usr/src/myapp/target/release/InvokerFunction .

CMD ./InvokerFunction
```

In the first stage, we use the standard Rust version 1.82 image as base. This image contains all the instruments needed to build and run Rust applications. So, we copy the files related to the invoker into the image and we run the command to compile the application. In this way, an executable file that contains all its dependencies is created in `/usr/src/myapp/target/release`. In the second stage, we start with a plain Debian image to install the default version of Nodejs runtime, in order to execute the Javascript example function. Then, we copy the directory containing the function code and the executable file from the previous stage. The only prerequisite to run that executable file is to have the right version of the `glibc` library installed. The base image already has it installed. With the last instruction, we configure the container to run the invoker when it starts.

We created similar Dockerfiles to containerize all the invokers related to the example functions and the dependency providers.