ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Analisi di piattaforme Function as a Service per l'implementazione di sistemi distribuiti su larga scala

Tesi di laurea in:
PROGRAMMAZIONE AD OGGETTI

Relatore
Prof. Mirko Viroli

Presentata da
Filippo Pracucci

Correlatore
Dott. Nicolas Farabegoli

Sessione ottobre 2024
Anno Accademico 2023-2024

Sommario

L'infrastruttura serverless, ovvero un modello di cloud computing per applicazioni stateless ed event-driven, ha come scopo principale quello di realizzare applicazioni senza la preoccupazione legata alla gestione dell'infrastruttura di back-end; in questo modo lo sviluppatore pone la propria attenzione interamente verso la business logic. La forma principale di serverless computing è la Function as a Service (FaaS), che viene implementata da diversi framework, la quale consiste nella realizzazione di applicazioni suddivise in funzioni stateless. All'interno del documento si considerano brevemente i framework proprietari, mentre si pone l'attenzione sulle proposte Open Source, le quali consentono di evitare le limitazioni imposte da una soluzione privata e personalizzare maggiormente l'architettura; nello specifico si analizzano OpenFaaS, Knative e Apache OpenWhisk, mentre si effettua un breve accenno riguardo Kubeless. Un ulteriore vantaggio della sopracitata architettura, è la riduzione delle risorse richieste; per questo motivo tale infrastruttura trova molto interesse nell'ambito dell'Internet of Things (IoT) e di sistemi distribuiti, anche con dispositivi at the edge. Inoltre, all'interno del documento vengono riportati dei test effettuati tramite il tool JMeter, per osservare il comportamento dei vari framework in situazioni di aumento di carico; prendendo in considerazione come metriche di tipo quantitativo: throughput, tempo di risposta medio e tasso di successo. Infine, si mostrano i risultati ottenuti dai test effettuati, tramite anche l'ausilio di grafici, in modo da ottenere una panoramica più precisa sui framework analizzati.

Indice

Sommario **ii**

1 Introduzione **1**

2 Frameworks **4**
2.1 Framework proprietari 4
2.1.1 AWS Lambda 5
2.2 Framework Open Source 6
2.2.1 Kubeless 6
2.2.2 Apache OpenWhisk 7
2.2.3 OpenFaaS 16
2.2.4 Knative 25

3 Setup sperimentale **30**
3.1 Metriche qualitative 31
3.2 Metriche quantitative 31

4 Creazione sistema di test **33**

5 Risultati sperimentali **36**
5.1 Grafici risultati sperimentali Node.js 36
5.2 Grafici risultati sperimentali Python 36
5.3 Grafici risultati sperimentali Java 38
5.4 Analisi risultati 38

6 Conclusioni **44**

46

Bibliografia **46**

Elenco delle figure

1.1 Servizi cloud computing 2

2.1 Modello Apache OpenWhisk 7
2.2 Architettura Apache OpenWhisk 8
2.3 Architettura OpenFaaS 17
2.4 OpenFaaS Gateway 21
2.5 OpenFaaS Provider 23
2.6 Architettura Knative Serving 26
2.7 Architettura Knative 27

5.1 Grafici risultati sperimentali throughput (transazioni/s) Node.js 38
5.2 Grafici risultati sperimentali tempo di risposta medio (ms) Node.js . . . 39
5.3 Grafici risultati sperimentali tasso di successo (%) Node.js 40
5.4 Grafici risultati sperimentali throughput (transazioni/s) Python 41
5.5 Grafici risultati sperimentali tempo di risposta medio (ms) Python . . . 41
5.6 Grafici risultati sperimentali tasso di successo (%) Python 42
5.7 Grafici risultati sperimentali throughput (transazioni/s) Java 42
5.8 Grafici risultati sperimentali tempo di risposta medio (ms) Java 43
5.9 Grafici risultati sperimentali tasso di successo (%) Java 43

List of Listings

2.1 `manifest.yaml` 9
2.2 `log.json` 13
2.3 `hello.js` 14

Capitolo 1

Introduzione

L'infrastruttura serverless nasce con lo scopo di portare avanti lo sviluppo che era iniziato con la transizione di molte applicazioni monolitiche in applicazioni a microservizi. Infatti, l'architettura serverless evolve il concetto di suddivisione dell'applicazione in funzioni, per questo la forma principale di implementazione della suddetta infrastruttura è la cosiddetta **FaaS**. Come si nota dalla Figura 1.1, un servizio FaaS richiede allo sviluppatore la gestione esclusivamente della parte applicativa. Il modello serverless ha i seguenti principi fondanti:

- non ci sono server da gestire;
- scalabilità;
- disponibilità e "fault tolerance" sono già integrati;
- pagamento delle sole risorse utilizzate, in caso di piani a pagamento.

Lo scopo principale di questa architettura è consentire allo sviluppatore di concentrarsi solo sulla "business logic" dell'applicazione, lasciando le preoccupazioni legate alla creazione e alla gestione dell'infrastruttura di *back-end* (come server, container, cluster, "networking", scalabilità, "fault tolerance", etc.) a framework appositi [1]. Il serverless computing dunque, è un modello di cloud computing per applicazioni "stateless" ed "event-driven"; questo porta numerosi vantaggi come:

- miglioramento della Quality of Service (QoS), eliminando i problemi di un'infrastruttura sempre attiva, la quale causerebbe di conseguenza la presenza di container effimeri;
- accesso *on-demand* alle funzioni.

Tuttavia, è necessario considerare che non sempre è possibile optare per un approccio "cloud-based" a causa di [2]:

- alta latenza;
- problemi di privacy;
- supporto alla mobilità.

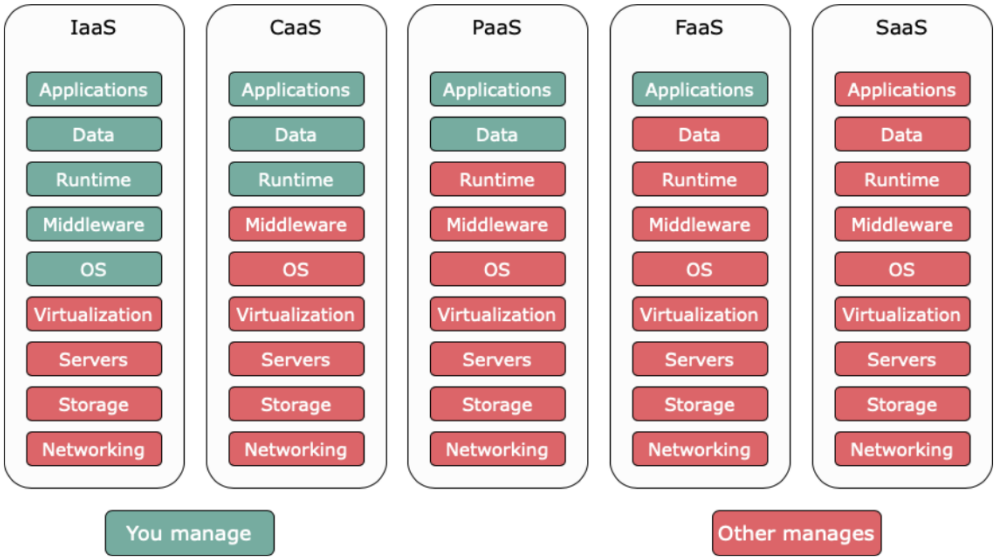

Figura 1.1: Servizi cloud computing

Il documento è organizzato nel seguente modo: il capitolo 2 presenta i principali framework, specialmente Open Source; nel capitolo 3 si individuano le metriche qualitative e quantitative, le quali verranno poi utilizzate per valutare i framework. Il capitolo 4 effettua una panoramica sulla configurazione dell'ambiente all'interno del quale verranno

effettuati i test, oltre a presentare la loro configurazione e il loro obiettivo. All'interno del capitolo 5 vengono mostrati i grafici raffiguranti i risultati dei test e si effettua una breve analisi. Infine, nel capitolo 6 si effettua un riassunto degli aspetti più importanti individuati.

Capitolo 2

Frameworks

Grazie al grande entusiasmo legato al serverless computing dal momento della sua introduzione, sono nati numerosi framework, ovvero strumenti che si occupano di eseguire le procedure operative del server, della rete, di bilanciamento del carico e di autoscaling [2]. I framework si suddividono principalmente in due categorie:

1. **Framework proprietari**: i quali sono gestiti da venditori esterni e quindi non mostrano le implementazioni delle funzionalità, non consentendo facilmente il passaggio ad un altro fornitore.

2. **Framework Open Source**: sono stati introdotti per evitare il blocco verso un venditore ("vendor lock-in") e le restrizioni computazionali di una piattaforma cloud esterna.

2.1 Framework proprietari

I framework proprietari sono gestiti da fornitori di terze parti, i quali offrono dei servizi a pagamento per poter usufruire della loro piattaforma. Inoltre, essendo gestiti a livello aziendale, non si è a conoscenza dell'implementazione e dell'effettiva realizzazione del servizio. Dato che sono prodotti commerciali non c'è alcun interesse da parte dell'azienda nel consentire un comodo passaggio dal loro framework ad un altro, causando

il cosiddetto “vendor lock-in”. Le principali alternative sono: AWS Lambda [1], Azure Functions [2], IBM Cloud Code Engine [3] e Google Cloud Functions [4].

2.1.1 AWS Lambda

Si tratta di un servizio di elaborazione event-based, cioè esegue il codice in risposta ad eventi, gestendo automaticamente le risorse; questo consente di realizzare applicazioni serverless. Lambda ha il compito di gestire “load balancing”, autoscaling, isolamento, etc; mentre consente il controllo della memoria, contrariamente alla percentuale di core della CPU e alla capacità della rete, le quali vengono assegnate proporzionalmente ad essa. In pratica, un’applicazione serverless che utilizza questo servizio lavora nel seguente modo: la sorgente di un evento provoca l’invocazione di una funzione (*AWS Lambda functions*), la quale utilizza dei servizi esterni (es. database...). Il componente principale utilizzato dal framework è l’**API gateway** che ha le seguenti caratteristiche:

- gestisce autenticazione, autorizzazione e il routing delle applicazioni;
- crea un’Application Programming Interface (API) di *front-end* unificata;
- crea i *WebSockets*, i quali suddividono la connessione in *stateful connection* verso i client e *stateless connection* nei confronti degli altri componenti.

I *Lambda execution model*, che consistono nella modalità di invocazione delle funzioni e di gestione delle risposte, sono:

- **Synchronous (push)**: la richiesta viene fatta dall’*API gateway* e si aspetta la risposta della funzione.
- **Asynchronous (event)**: non si attende la risposta della funzione.
- **Poll-based**: avviene un costante controllo (“polling”) degli stream di messaggi da parte di un servizio che richiama la funzione.

[1] `https://aws.amazon.com/it/lambda/`
[2] `https://azure.microsoft.com/en-us/products/functions`
[3] `https://cloud.ibm.com/codeengine/overview`
[4] `https://cloud.google.com/functions`

2.2 Framework Open Source

I framework Open Source non sono gestiti da una singola entità, ma da una serie di contributori che partecipano al miglioramento del codice, il quale è pubblico e consultabile da chiunque. Il vantaggio di questi framework è che nella maggior parte dei casi sono gratuiti, o quantomeno offrono un piano senza costi, con delle limitazioni nelle risorse concesse. Principalmente sono stati introdotti per consentire il facile passaggio tra varie soluzioni, aggirando quindi il "vendor lock-in" e tutte le limitazioni di una piattaforma proprietaria [3]. Questi sono formati da vari componenti, che spesso sono altri prodotti Open Source, i quali vengono assemblati per fornire un'infrastruttura completa. I framework Open Source di serverless computing sono numerosi, ma i principali sono:

1. **Kubeless**[5]
2. **Apache OpenWhisk**[6]
3. **OpenFaaS**[7]
4. **Knative**[8]

2.2.1 Kubeless

Si tratta di un framework serverless nativo di Kubernetes [9].
Le sue primitive sono:

- **Functions**: consistono nel codice che deve essere eseguito.
- **Triggers**: sono le sorgenti degli eventi e possono essere associati a una funzione o ad un gruppo di funzioni.
- **Runtime**: sono il linguaggio e l'ambiente dove verrà eseguita la funzione.

Il componente principale è:

[5] `https://github.com/vmware-archive/kubeless/tree/maste`
[6] `https://openwhisk.apache.org/`
[7] `https://www.openfaas.com/`
[8] `https://knative.dev/`
[9] `https://kubernetes.io/`

- **CRD controller**: è colui il quale osserva continuamente i *Function objects* in cerca di cambiamenti ed effettua di conseguenza le azioni necessarie (es. creazione o eliminazione di un *Function object*).

2.2.2 Apache OpenWhisk

Consiste in un servizio serverless di programmazione event-driven, il cui modello è rappresentato nella Figura 2.1.
Le sue primitive sono:

- **Actions**: sono *stateless function* che eseguono codice. È possibile comporne molteplici per creare una pipeline detta *sequence*.
- **Triggers**: consistono in classi di eventi che possono avere origine da varie fonti.
- **Rules**: associano un *Trigger* ad una *Action*.

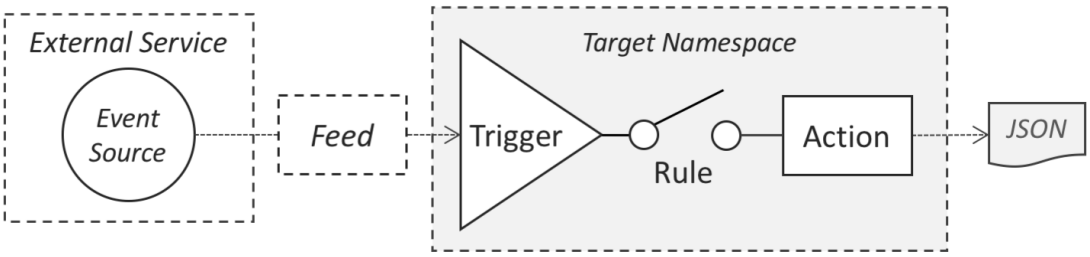

Figura 2.1: Modello Apache OpenWhisk

I componenti principali che lo compongono, come si può osservare nella Figura 2.2, sono:

- **Nginx webserver**: agisce da server proxy ed è il primo *entrypoint*.
- **Controller**: si occupa di autenticazione, autorizzazione e instradamento di ogni richiesta. In pratica traduce la richiesta ricevuta da *Nginx* nell'invocazione di un'azione. Inoltre, contiene un *Load Balancer* che si occupa di scegliere a quale *Invoker* affidare la richiesta in base al carico.

- **Apache Kafka**: gestisce le connessioni tra il *Controller* e l'*Invoker*, infatti si tratta di un sistema di messaggistica *publish-subscribe* distribuito e ad alta portata.
- **Invoker**: si occupa di copiare il codice da *CouchDB* ed iniettarlo in un container Docker[10]; questo viene sfruttato per consentire l'esecuzione delle azioni in modo sicuro ed isolato.
- **CouchDB**: consiste in un database, all'interno del quale vengono memorizzati anche i risultati delle invocazioni. Si occupa di fornire al *Controller* l'azione richiesta ed inoltre mantiene tutte le attivazioni collezionate per id.

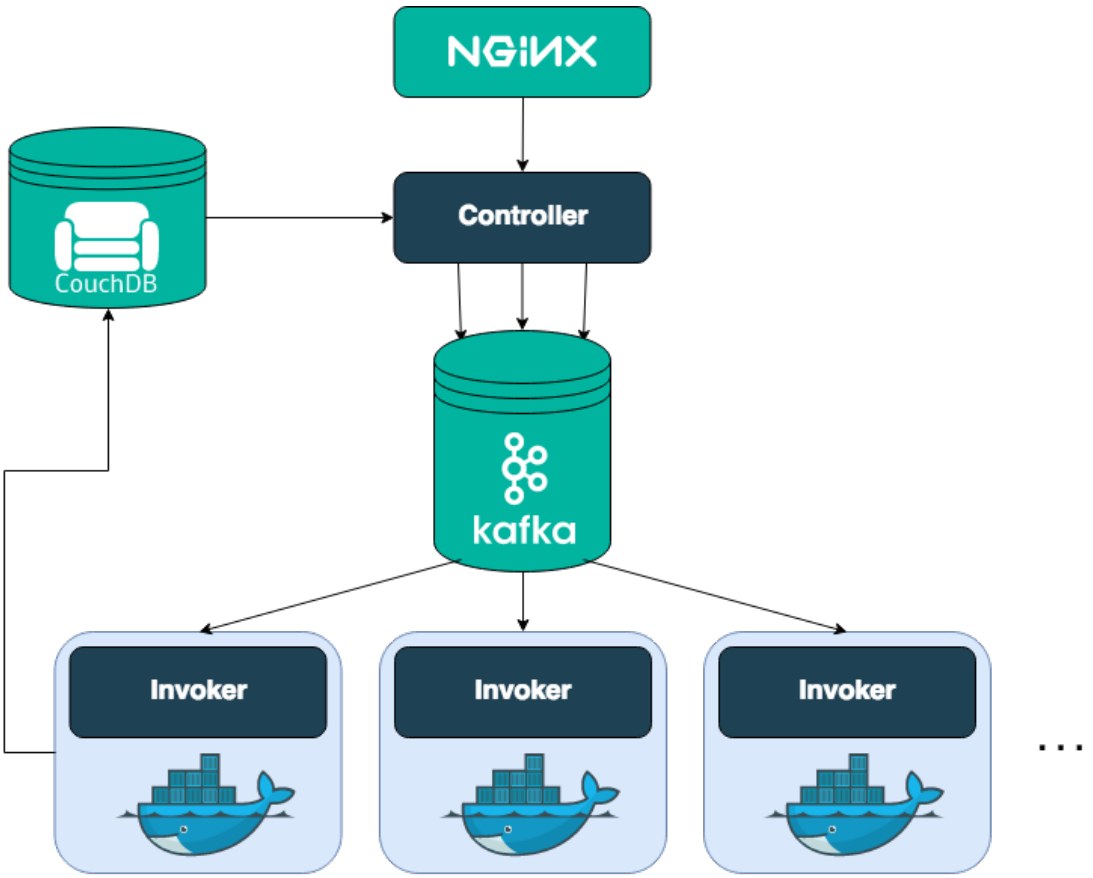

Figura 2.2: Architettura Apache OpenWhisk

[10] https://www.docker.com/

```
packages:
  default:
    actions:
      nomeAction:
        function: nomeFile
```

Listing 2.1: `manifest.yaml`

2.2.2.1 Creazione e invocazione di una Action

È innanzitutto necessario scegliere un linguaggio e successivamente attenersi ai seguenti passi:

1. creazione di un file con il codice che l'azione deve eseguire (es. in Java creazione di un file `MyAction.java` contenente la classe `MyAction`);
2. *[opzionale]* svolgimento dei passaggi specifici del linguaggio scelto (es. in Java è richiesta la compilazione del file e la creazione del `jar`);
3. creazione dell'azione con il comando:

   ```
   wsk action create myAction myAction.jar --main MyAction
   ```

   dove con il flag `--main` si specifica l'entrypoint;
4. invocazione dell'azione ed ottenimento del risultato grazie al flag `--result`, mediante il comando che segue:

   ```
   wsk action invoke myAction --result
   ```

5. *[opzionale]* il `deploy` con `wskdeploy` è realizzabile tramite i passi che seguono:
   - creazione del file `manifest.yaml`, eg: listing 2.1;
   - `deploy` sfruttando il file creato in precedenza:

     ```
     wskdeploy -m manifest.yaml
     ```

Questi passaggi potrebbero subire leggere variazioni in base al linguaggio scelto.

L'invocazione di una *Action* può essere effettuata, in alternativa al comando `wsk action`

`invoke myAction`, citato in precedenza, inviando richieste HyperText Transfer Protocol (HTTP) (per esempio tramite il comando `curl`) all'indirizzo fornito a seguito del `deploy` della funzione, che corrisponde all'*API HOST*, il quale è configurabile a piacimento.

2.2.2.2 Package

I **packages** servono per impacchettare un insieme di azioni correlate, permettendo anche la loro condivisione. Ci sono vari *package* pubblici messi a disposizione dal framework all'interno del namespace `/whisk.system`, i quali offrono diverse funzionalità e accesso a servizi esterni, come *GitHub*[11], *Slack*[12], etc.
Il sottocomando per ottenere informazioni sui *package* è:

```
wsk package
```

Ci sono due modalità di invocazione di un'azione da un *package*:

1. direttamente dal suo interno, nella seguente modalità:

   - recapito della descrizione dell'azione:

   ```
   wsk action get --summary /whisk.system/samples/greeting
   ```

   - invocazione dell'azione con parametri opzionali:

   ```
   wsk action invoke --result /whisk.system/samples/greeting --param name
       Bernie --param place Vermont
   ```

2. creazione di un *package binding*, con successiva invocazione dell'azione nel "binding":

```
wsk package bind /whisk.system/samples valhallaSamples
wsk action invoke --result valhallaSamples/greeting --param name Odin
```

2.2.2.3 Trigger

Servono ad automatizzare *Actions* in risposta ad eventi provenienti da *Event Sources*.

[11] `https://github.com/`
[12] `https://slack.com/`

Creazione ed esecuzione Trigger

- creazione di un *Trigger*:

```
wsk trigger create myTrigger
```

- esecuzione del *Trigger* specificando il nome e facoltativamente alcuni parametri:

```
wsk trigger fire myTrigger --param <key> <value>
```

Associazione tra Trigger e Action tramite Rule

Si crea una *Rule* che associa un *Trigger* ad una *Action*, entrambi già esistenti:

```
wsk rule create myRule myTrigger myAction
```

Invocazione Action mediante Trigger

```
wsk trigger fire myTrigger --param <key> <value> --param <key> <value>
```

2.2.2.4 Flow del processo

Si mostra una sequenza ordinata di fasi che descrive il processo esecutivo generale, ognuna caratterizzata da un componente dell'architettura:

1. **Nginx**
   Si tratta del primo entrypoint del sistema. Il comando inviato tramite la Command-Line Interface (CLI) `wsk` è essenzialmente una richiesta HTTP e si traduce in:
   `POST /api/v1/namespaces/$userNamespace/actions/myAction`
   `Host: $openwhiskEndpoint`

2. **Controller**
   Consiste in un'implementazione basata su Scala[13] della Representational State Transfer (REST) API, che funge da interfaccia per tutto ciò che l'utente può fare. Il *Controller* chiarisce ciò che vuole fare l'utente, basandosi sul metodo HTTP specificato nella richiesta (es. la richiesta, che sfrutta il metodo POST, di un'azione esistente viene tradotta nell'invocazione di una *Action*).

[13] `https://www.scala-lang.org/`

3. **CouchDB: autenticazione e autorizzazione**
   Il *Controller* effettua alcune verifiche di autenticazione e autorizzazione tramite il controllo delle credenziali presenti nel database *CouchDB*.

4. **CouchDB: ottenere l'azione**
   Il *Controller* carica l'*Action* dal database *Whisks* in *CouchDB*. Il record dell'azione contiene il codice da eseguire, completato con i parametri passati nella richiesta; inoltre possiede le restrizioni delle risorse imposte per l'esecuzione.

5. **Load Balancer**
   Si tratta di una parte del *Controller* che ha una visione globale degli esecutori (*Invokers*) disponibili, controllando continuamente il loro stato di salute ("health status"). Il *Load Balancer* sceglie in base al carico un *Invoker* per l'azione richiesta.

6. **Kafka**
   Si tratta di un sistema di messaggistica *publish-subscribe*, distribuito e ad elevato rendimento. Si occupa della gestione dei possibili errori, come i seguenti:

   - il sistema va in "crash", perdendo l'invocazione;
   - il sistema viene sovraccaricato e quindi l'invocazione deve aspettare che ne finiscano altre prima.

   Si occupa inoltre, di rendere persistente e "bufferizzata" la comunicazione tra *Controller* e *Invoker*:

   (a) il *Controller* pubblica un messaggio a *Kafka* che contiene l'azione e i parametri;

   (b) il messaggio viene indirizzato all'*Invoker* scelto;

   (c) dopo che *Kafka* ha confermato la ricezione del messaggio, la richiesta viene risposta con un **ActivationId** (usato in seguito per accedere ai risultati dell'invocazione).

7. **Invoker**
   Il suo compito è quello di invocare un'azione. La sua implementazione è in Scala

```
{
    "activationId": "31809ddca6f64cfc9de2937ebd44fbb9",
    "response": {
        "statusCode": 0,
        "result": {
            "hello": "world"
        }
    },
    "end": 1474459415621,
    "logs": [
        "2016-09-21T12:03:35.619234386Z stdout: Hello World"
    ],
    "start": 1474459415595
}
```

Listing 2.2: `log.json`

e per l'esecuzione isolata e sicura di *Actions* sfrutta Docker, il quale viene usato per creare un container per ogni azione invocata, in modo veloce, sicuro e isolato. Funzionamento:

(a) creazione di un container Docker per ogni invocazione;

(b) il codice dell'azione viene iniettato al suo interno;

(c) il codice viene eseguito utilizzando anche i parametri;

(d) una volta ottenuto il risultato, il container viene distrutto.

Viene anche effettuata un'ottimizzazione delle performance per ridurre l'"overhead" e diminuire i tempi di risposta.

8. **CouchDB: archiviare i risultati**
   Il risultato ottenuto dall'*Invoker* viene archiviato nel database *activations* (all'interno di *CouchDB*) per `ActivationId`. Il risultato consiste in un oggetto `JSON` ottenuto dall'azione, unito al log scritto da Docker, eg: listing 2.2.

   A questo punto si può ricominciare dal passo 1 sfruttando nuovamente la REST API per ottenere l'attivazione e quindi il risultato dell'azione:

```
wsk activation result activationId
```

```
function main({name:name='Serverless API'}) {
    return {payload: `Hello world ${name}`};
}
```

Listing 2.3: `hello.js`

2.2.2.5 API gateway

L'*API gateway* agisce da proxy per **Web Actions** e fornisce loro funzionalità aggiuntive, come metodo di routing HTTP, limitazioni di rate, etc.

Creazione Web Action (uso di JavaScript)

1. Creazione di un file JavaScript (es. `hello.js` listing 2.3).
2. Creazione di una *Web Action* dalla seguente funzione Javascript:

```
wsk action create hello hello.js --web true
```

3. Creazione di un'API con percorso base `/hello`, percorso `/world` e metodo `get` con tipo di risposta `JSON`:

```
wsk api create /hello /world get hello --response-type json
```

   In questo modo è stato generato un nuovo Uniform Resource Locator (URL) che espone l'azione `hello` tramite metodo HTTP GET.

4. Si può verificare inviando una richiesta HTTP all'URL generato in precedenza, per esempio sfruttando il comando `curl`:

```
curl https://${APIHOST}:9001/api/${GENERATED_API_ID}/hello/world?name=
    OpenWhisk
```

2.2.2.6 Opzioni di deployment

L'opzione di deployment consigliata è Kubernetes, perché è supportata dalla maggior parte delle piattaforme ed offre molte possibilità sia per lo sviluppo locale che per produzione su larga scala. Ad ogni modo si mostrano tutte le possibilità di deployment che OpenWhisk mette a disposizione.

Kubernetes

OpenWhisk può essere dispiegato usando gli *Helm*[14] *Charts* su qualsiasi Kubernetes fornito localmente o da un "cloud provider" pubblico.

Standalone OpenWhisk Stack

Si tratta di uno stack completo che viene eseguito come un processo Java. Le **serverless functions** vengono eseguite dentro container Docker; quindi è necessario aver installato localmente Docker, Java e Node.js[15]. I passaggi richiesti sono:

- clonazione del repository e creazione dello stack:

```
$ git clone https://github.com/apache/openwhisk.git
$ cd openwhisk
$ ./gradlew core:standalone:bootRun
```

- Una volta che lo stack è stato creato si aprirà il browser in un *functions playground*, solitamente all'indirizzo `http://localhost:3232`, che permette la creazione e l'esecuzione delle funzioni direttamente nel browser.

Docker Compose

Docker Compose[16] è la scelta giusta nel caso in cui si voglia utilizzare direttamente Docker.

```
$ git clone https://github.com/apache/openwhisk-devtools.git
$ cd openwhisk-devtools/docker-compose
$ make quick-start
```

Ansible

Il deployment sfruttando Ansible[17] consiste in un approccio più imperativo, basato su script. Gli **OpenWhisk playbooks** sono strutturati in modo da consentire il "cleaning", "deploying" o "re-deploying" di un singolo componente come dell'intero stack.

[14] `https://helm.sh/`
[15] `https://nodejs.org/`
[16] `https://docs.docker.com/compose/`
[17] `https://www.ansible.com/`

Vagrant

Vagrant[18] per essere utilizzato richiede l'installazione di VirtualBox[19], per gestire la virtualizzazione dell'hardware.

2.2.3 OpenFaaS

Questo framework si appoggia su Kubernetes e Docker, infatti ogni funzione è impacchettata in un container Docker isolato dal resto dell'architettura, la quale viene mostrata nella Figura 2.3. Ogni container inoltre contiene un **watchdog**, ovvero una guardia.
Le sue primitive sono:

- **Functions**: lo sviluppatore ha il compito di fornire la funzione ed un `Handler`.

I principali componenti che lo formano sono:

- **API gateway**: fornisce accesso alle funzioni, colleziona le metriche e fornisce autoscaling, interagendo con l'*orchestration engine* (es. Kubernetes).
- **AlertManager[20] o Kubernetes Horizontal Pod Autoscaler (HPA)**[21]: nel primo caso è combinato con **Prometheus**[22], mentre nel secondo si tratta dell'HPA di Kubernetes.

2.2.3.1 Workflow

- Tutti i servizi e le funzioni prendono una route esposta di default, ma possono essere usati domini personalizzati per ogni *endpoint*;
- cambiando l'URL legato ad una funzione da `/function/NAME` a `/async-function/NAME`, un'invocazione può essere eseguita in una coda usando il componente **NATS Streaming** (mostrato nella Figura 2.3);
- **faas-netes** è l'*orchestration provider* più popolare. I provider sono costruiti con il *faas-provider Software Development Kit (SDK)*.

[18] `https://www.vagrantup.com/`
[19] `https://www.virtualbox.org/`
[20] `https://github.com/prometheus/alertmanager`
[21] `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`
[22] `https://prometheus.io/`

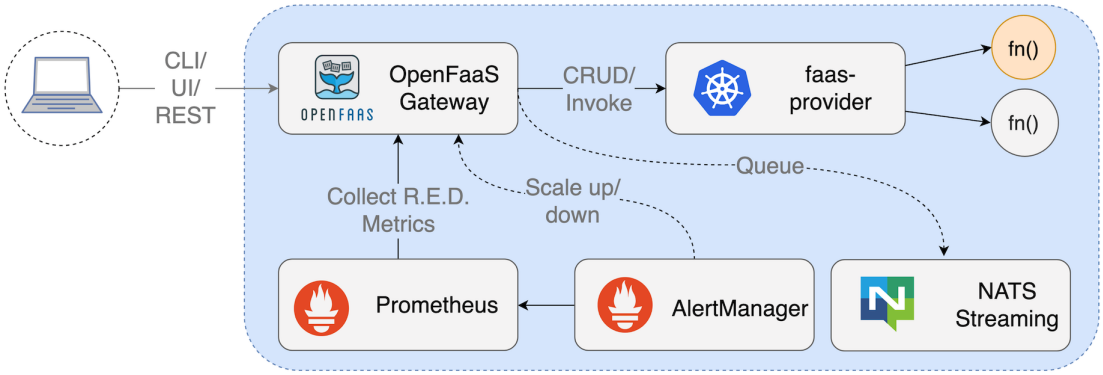

Figura 2.3: Architettura OpenFaaS

2.2.3.2 Deployment

In questa sezione si fa riferimento alla versione **OpenFaaS CE**.

1. Creazione di un cluster Kubernetes, il quale deve avere sempre completo accesso ad Internet.
   Alcune opzioni per cluster locali sono:

   - KinD[23]: effettua l'"upstream" di Kubernetes in un container con Docker;
   - K3s[24]: distribuzione di Kubernetes leggera, ideale per lo sviluppo, IoT e dispositivi *at the edge*;
   - K3d[25]: come K3s, ma in un container Docker; infatti consiste in un *light-weight wrapper* per eseguire K3s;
   - minikube[26]: creazione di cluster Kubernetes nella macchina locale tramite una Virtual Machine (VM) separata.

   Opzioni per cluster remoti:

[23] `https://kind.sigs.k8s.io/`
[24] `https://k3s.io/`
[25] `https://k3d.io/`
[26] `https://minikube.sigs.k8s.io/`

- K3sup[27]: costruzione di cluster a nodo singolo o multiplo usando "cloud VMs".

2. Installazione della CLI **faas-cli** per la gestione ed il `deploy` delle funzioni;

3. installazione di **OpenFaaS CE** tramite Arkade[28] (consigliato) oppure Helm;

4. ogni volta che una funzione è dispiegata o scalata verso l'alto, Kubernetes farà il `pull` di una copia potenzialmente aggiornata dell'immagine dal registry, a meno che la relativa opzione non sia stata configurata a `Never`, in quel caso funzioneranno solo le immagini locali.

2.2.3.3 Creazione e gestione funzione

Creazione funzione

Il framework mette a disposizione un archivio da cui è possibile scaricare vari template:

```
$ faas-cli template store list
$ faas-cli template store pull <template-name>
```

Una volta fatto questo si può creare una funzione utilizzando il template scaricato:

```
faas-cli new --lang <template-name> <function-name>
```

Il codice della funzione può essere modificato all'interno del file `Handler`.

Costruzione funzione

Le funzioni sono costruite come immagini di container compatibili con il formato Open Container Initiative (OCI), che di solito contengono il *watchdog* come proxy. Quando si esegue il seguente comando usando un template dal negozio, l'immagine del container sarà costruita nella libreria locale:

```
faas-cli build
```

I template tendono ad astrarre il *Dockerfile* e l'*entrypoint HTTP server*, permettendo di concentrarsi su un **HTTP/functions handler**.

[27] `https://github.com/alexellis/k3sup`
[28] `https://github.com/alexellis/arkade`

Risultato della creazione di una funzione

Ogni funzione che ha subito `deploy`, creerà un oggetto separato *Kubernetes Deployment*; esso ha un valore **replicas**, il quale corrisponde al numero di *Pods* creati nel cluster. Viene anche creato un oggetto *Kubernetes Service*, il quale è usato per accedere all'endpoint HTTP della funzione sulla porta 80 dentro il cluster. Di norma le funzioni hanno un numero minimo di repliche uguale a uno, in modo da prevenire la **cold start**, infatti in questo modo non deve essere creato ogni volta un *Pod* per gestire una richiesta. Per *cold start* si intende il processo che richiede l'avvio di una nuova istanza di una funzione, ma nel caso di un framework FaaS questo potrebbe anche comportare la necessità della creazione e configurazione di un nuovo container, con il conseguente `deploy` della funzione; dunque nel suddetto contesto i tempi di gestione di una richiesta aumenterebbero notevolmente [4]. Per quanto riguarda invece il limite delle invocazioni che possono essere effettuate nei confronti di una funzione, è possibile impostarlo tramite la variabile d'ambiente `max_inflight: N`, altrimenti di default non è presente alcun limite.

Deploy funzione

Per eseguire il `deploy` di una funzione, si può utilizzare il comando `up`, oppure eseguire i singoli comandi che esso racchiude, singolarmente:

- in caso di comando singolo:

```
faas-cli up -f <function-name>.yml
```

- In caso di separazione dei comandi:

```
$ faas-cli build -f <function-name>.yml
$ faas-cli push -f <function-name>.yml
$ faas-cli deploy -f <function-name>.yml
```

2.2.3.4 Invocazioni

Le funzioni possono essere invocate tramite richieste HTTP all'**OpenFaaS gateway** oppure sfruttando il comando:

```
faas-cli invoke
```

Ogni funzione è dispiegata sotto forma di *Kubernetes Deployment and Service*, con un determinato numero di repliche; quindi può scalare verso l'alto e il basso, oltre a gestire più richieste concorrenti.

Invocazioni sincrone e asincrone

1. Invocazioni sincrone: si può invocare il *gateway* con una richiesta HTTP all'indirizzo `http://<GATEWAY_IP>:<PORT>/function/NAME`, dove `NAME` è il nome della funzione, `GATEWAY_IP` e `PORT` sono l'indirizzo e la porta dell'*API gateway*. È possibile avere anche uno strato intermedio, come un *reverse proxy*. La connessione tra il chiamante e la funzione rimane sempre attiva, finché l'invocazione non è completata o scaduta.

2. Invocazioni asincrone: le richieste HTTP sono messe in una coda del **NATS Streaming**, seguite da un *header* `accepted` e un *call-id* ritornati al chiamante. Successivamente un **queue-worker** separato, prende il messaggio dalla coda ed invoca la funzione in modo sincrono. Non c'è mai una connessione diretta tra chiamante e funzione.

2.2.3.5 Events

Per ogni *event trigger* un **long-running daemon** è sottoscritto ad un topic o ad una coda, poi quando riceve dei messaggi, cerca le funzioni pertinenti e le invoca in maniera sincrona o asincrona.

2.2.3.6 Autenticazione

La **REST API** è usata per gestire le funzioni ed implementa di default le tecniche di autenticazione basiche; è possibile inoltre abilitare la crittografia tramite Transport Layer Security (TLS) e un *reverse proxy*. Si possono gestire le funzioni con la *Function Custom Resource Definition (CRD)*, che è osservata da un operatore, il quale può creare risorse in Kubernetes, oltrepassando la REST API del *gateway*; quest'ultima viene utilizzata comunque per le invocazioni. Per quanto riguarda le funzioni spetta invece allo sviluppatore fornire un meccanismo di autenticazione.

2.2.3.7 Gateway

Ogni funzione è costruita in un'immagine Docker immutabile, prima di essere dispiegata con *faas-cli*, User Interface (UI) o REST API. Quando si effettua il `deploy` di una funzione si creano da uno a molti *Pods/containers*, in base al numero minimo e massimo specificati nei parametri di scalabilità. Nella Figura 2.4 si notano i componenti di OpenFaaS, tra cui il *gateway*, all'interno di un diagramma concettuale della piattaforma.

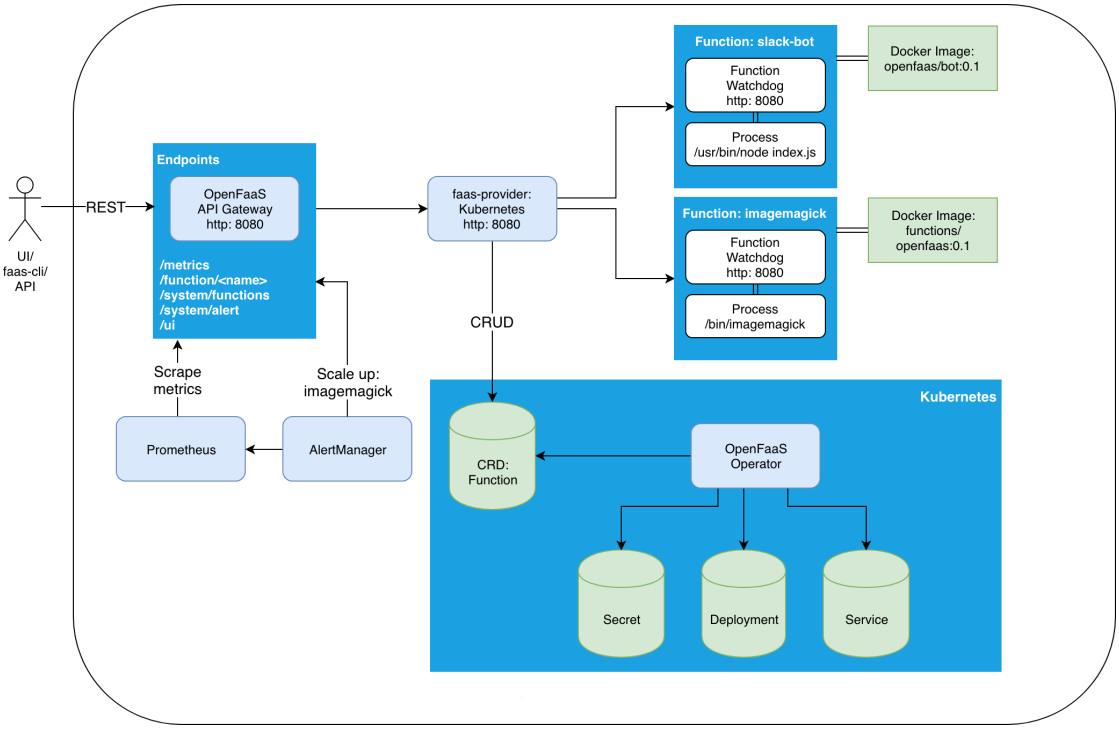

Figura 2.4: OpenFaaS Gateway

2.2.3.8 Watchdog

Si tratta del responsabile dell'avvio e del monitoraggio delle funzioni. Ogni file binario con un **watchdog** può diventare una funzione; questo diventa un *init process* con un *embedded HTTP server* scritto in Golang[29], il quale può supportare richieste concorrenti, timeout e controlli dello stato.

[29] `https://go.dev/`

Watchdog classico

Fornisce un'interfaccia tra l'esterno e la funzione; inoltre avvia un processo per ogni richiesta e usa `stdio` per la comunicazione. Ogni funzione deve incorporare questo binario e usarlo come entrypoint, cioè come *init process*. Una volta che il processo è stato "forked", il *watchdog* passa alla richiesta HTTP via `stdin` e legge una risposta via `stdout`; quindi non è necessario che il processo sappia qualcosa del web o di HTTP.

of-watchdog

Si tratta di un progetto complementare a quello classico. Fornisce un'alternativa a `stdio` per la comunicazione tra il watchdog e la funzione. La principale differenza è che consente di mantenere il processo della funzione "caldo" tra le invocazioni. Questo watchdog infatti abilita una modalità `http`, dove lo stesso processo può essere riusato ripetutamente per compensare la latenza del *forking*; inoltre abilita il riutilizzo della memoria e la risposta rapida alle richieste. Il componente implementa un server HTTP in ascolto sulla porta 8080 e agisce da *reverse proxy* per eseguire funzioni e microservizi. Ci sono varie modalità di interazione con il microservizio o la funzione:

- `HTTP`: è l'opzione di default e quella più efficiente se il target ha un server HTTP;
- `serializing`: quando non c'è l'implementazione di un server HTTP, allora `stdio` è letto dalla memoria e poi inviato in un processo "forked";
- `streaming`: analoga alla modalità antecedente, ma la richiesta e la risposta sono in streaming, invece di essere "bufferizzate" completamente in memoria prima che la funzione si avvii.

2.2.3.9 Autoscaling

Nel file di configurazione per il componente *AlertManager*, viene definita un'unica regola di autoscaling, usata per tutte le funzioni: l'*AlertManager* legge le metriche di uso (richieste per secondo) da *Prometheus*, per sapere quando scatenare un avviso all'*API gateway*. Quest'ultimo gestisce gli *alerts* nella route `/system/alert`. L'autoscaling fornito può essere disabilitato eliminando il deployment dell'*AlertManager* o scalando il deployment a zero repliche. Questo metodo di autoscaling viene utilizzato sia per le invocazioni sincrone che asincrone.

Si può inoltre impostare il numero minimo e massimo di repliche a tempo di deployment, aggiungendo una delle seguenti *label* alla funzione:

- `com.openfaas.scale.min`: è 1 di defualt;
- `com.openfaas.scale.max`: il valore massimo di default è 5 per 5/5 *Pods*;
- `com.openfaas.scale.factor`: di default è il 20%, in generale è compreso tra 0 e 100.

2.2.3.10 OpenFaaS Provider

Il **faas-provider**, mostrato nella Figura 2.5, fornisce la Create, Read, Update, and Delete (CRUD) API per le funzioni, così come le capacità di invocazione. Si tratta di un SDK scritto in Go, che si conforma alla HTTP REST API del provider.
Ogni provider implementa i seguenti comportamenti:

- CRUD per le funzioni;
- invocazione delle funzioni via proxy;
- scalabilità delle funzioni.

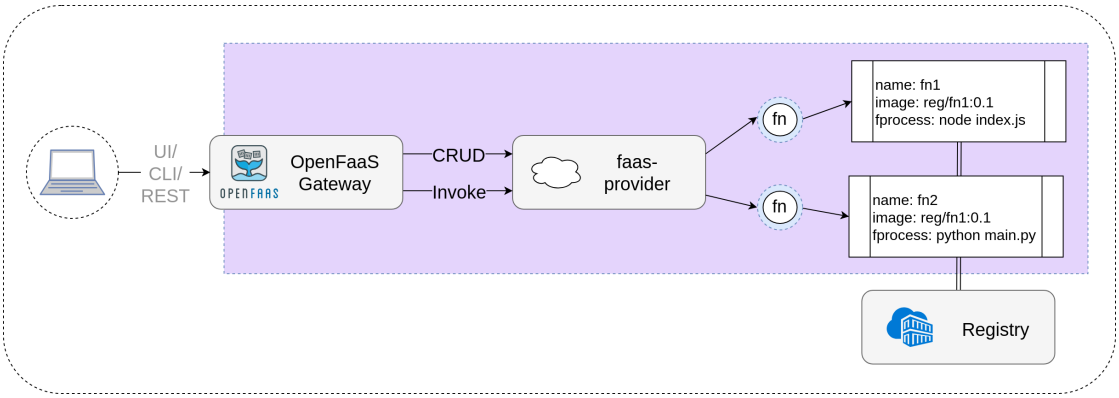

Figura 2.5: OpenFaaS Provider

Kubernetes Provider: **faas-netes** è il provider ufficiale per Kubernetes, impacchettato con *Helm Charts*. Fornisce anche una *Function CRD* tramite il flag `-operator=true`.

2.2.3.11 faasd

Si tratta di OpenFaaS senza la complessità ed il costo di Kubernetes. Questa soluzione richiede bassi requisiti, infatti esegue su un singolo host, ed è ampiamente compatibile con OpenFaaS su Kubernetes. Sfrutta **containerd** e **Container Networking Interface (CNI)**, insieme ai componenti principali del progetto centrale.
I casi d'uso principali sono:

- esecuzione di codice localmente, usando container;
- creazione di automazione, portali web, bots, etc;
- ricerca di un modo per fare deployment remoti sopra REST API;
- non si ha a disposizione larghezza di banda sufficiente per gestire Kubernetes;
- "embedded apps" in IoT e *at the edge*;
- costi ridotti;
- quando servono solo pochi microservizi o funzioni, senza il costo di un cluster.

Non comporta lo stesso onere di manutenzione che si ha con l'aggiornamento, il mantenimento e la gestione della sicurezza di un cluster Kubernetes.

Deployment
faasd è un binario statico che risiede in un sistema Linux configurato con `systemd` e richiede risorse di sistema minime.
In base al sistema operativo ci sono delle differenze:

- su Windows e MacOS si può usare *multipass* per dispiegare *faasd* in una VM;
- su Linux si può fare `deploy` direttamente o usare comunque *multipass*.

L'utilizzo per scopi di produzione richiede il `deploy` in uno dei seguenti modi:

- uso dello script bash di installazione;
- utilizzo dello script `cloud-init` fornito;
- sfruttando Terraform[30].

[30] `https://www.terraform.io/`

2.2.4 Knative

Si tratta di un framework costruito sopra Kubernetes e Istio[31]; quindi estende la piattaforma di Kubernetes utilizzando le CRDs per abilitare un livello di astrazione superiore. Tuttavia, non si può considerare una piattaforma serverless completa, dato che alcune implementazioni non vengono gestite, ma vengono lasciate allo sviluppatore, il quale talvolta può affidarsi ad altri framework per completare l'architettura. Il framework, che viene mostrato nella Figura 2.7, è formato principalmente dai seguenti componenti:

- **Serving**: definisce un insieme di oggetti come *Kubernetes CRDs*. Queste risorse sono usate per definire e controllare come si comporta il carico di lavoro serverless nel cluster. Come rappresentato nella Figura 2.6, i componenti principali che lo compongono sono:
  - **Services**: gestiscono automaticamente il ciclo di vita del carico di lavoro.
  - **Routes**: gestiscono il traffico.
  - **Configurations**: mantengono lo stato desiderato per il deployment.
  - **Revisions**: sono oggetti immutabili che consistono in "snapshots" del codice e della configurazione.

- **Eventing**: è una collezione di API che consentono di usare un'architettura event-driven nelle applicazioni. Le principali caratteristiche sono le seguenti:
  - le API possono essere usate per creare componenti che instradano gli eventi dalle sorgenti (*event producers*) ai consumatori (*sinks*);
  - si tratta di una piattaforma a sé stante che fornisce supporto per vari carichi di lavoro;
  - utilizza richieste di tipo HTTP POST per inviare e ricevere eventi tra sorgenti e *sinks*;
  - i suoi componenti non sono molto legati, quindi possono essere sviluppati e dispiegati indipendentemente tra loro.

[31] `https://istio.io/`

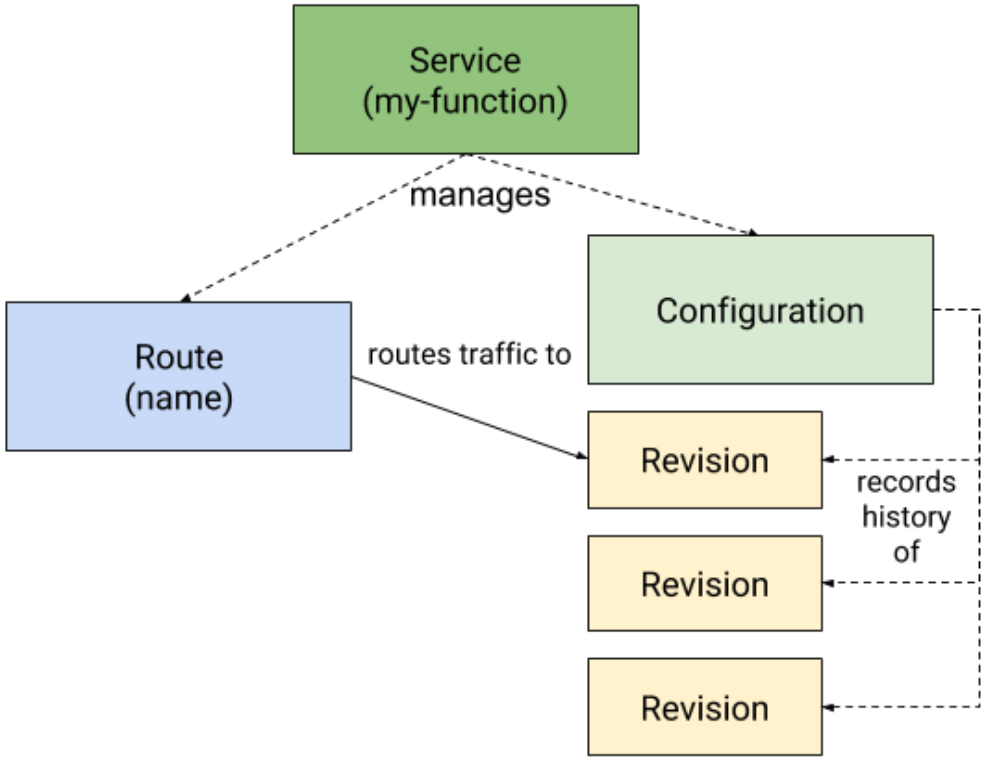

Figura 2.6: Architettura Knative Serving

2.2.4.1 Knative Functions

Knative Functions fornisce un semplice modello di programmazione per usare funzioni, senza la necessità di possedere una conoscenza approfondita di Kubernetes, *Dockerfiles*, etc. Esse inoltre abilitano la creazione, la costruzione e il `deploy` in modo semplice di funzioni stateless ed event-driven, sotto forma di **Knative Services**, sfruttando la CLI **func** o il relativo plugin **kn func**. Quando si costruisce o esegue una funzione, un'immagine di container in formato OCI viene generata automaticamente e viene archiviata in un *container registry*. Ogni volta che si aggiorna il codice e poi si esegue o viene fatto il `deploy`, allora anche l'immagine viene aggiornata.

Creazione funzione

Dopo aver installato *Knative Functions* si può creare un progetto nel seguente modo:

```
(kn) func create -l <language> <function-name>
```

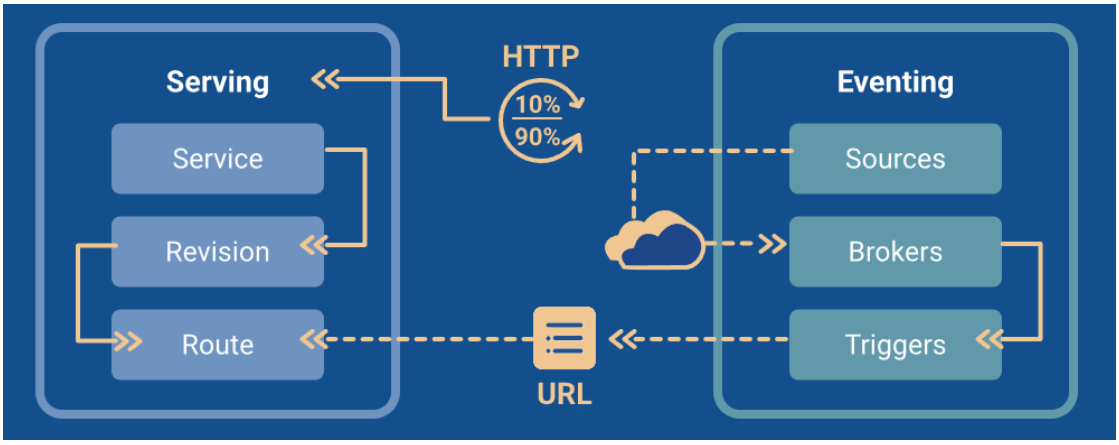

Figura 2.7: Architettura Knative

Esecuzione funzione

L'esecuzione di una funzione crea un'immagine di container in formato OCI per essa, prima di eseguirla in ambiente locale, ma non effettua il `deploy` su un cluster. Questo è utile se si vuole eseguire una funzione localmente per testarla.

Requisiti:

- si ha un **Docker daemon** in locale.

Procedura:

Il comando `run` costruisce un'immagine per la funzione richiesta e la esegue localmente. Se non è ancora stata fatta la `build` della funzione allora è necessario il flag `--registry`. Per eseguire la funzione localmente è necessario lanciare il seguente comando dall'interno della directory del progetto:

```
(kn) func run [--registry <registry>]
```

Deploy funzione

Fare il `deploy` di una funzione, come anticipato precedentemente, crea un'immagine di container OCI per la funzione ed effettua il `push` nell'*image registry*. La funzione è dispiegata sul cluster come un *Knative Service*. In caso di reiterazione del `deploy` si aggiorna l'immagine ed il corrispondente *Service* che è in esecuzione nel cluster. Le

funzioni sul cluster sono accessibili come i servizi.

Requisiti:

- è presente un **Docker daemon** localmente;
- si ha accesso ad un *container registry* ed è possibile fare il `push` delle immagini in esso.

Procedura:
Il comando `deploy` usa il nome del progetto come nome del *Knative Service*. Quando la funzione è costruita, il nome del progetto e il nome del registry sono usati per costruire un nome pienamente qualificato per l'immagine della funzione.

```
(kn) func deploy --registry <registry>
```

Build funzione
Costruire una funzione causa la creazione di un'immagine di container OCI per essa, la quale può subire poi il `push` nel registry. Questo comando non effettua il `deploy` e non esegue la funzione, il che è utile se si vuole costruire un'immagine localmente, ma non eseguirla o fare il `deploy` automaticamente al cluster.

1. **Build locale**
   Requisiti:
   - si possiede un **Docker daemon** in locale.

   Procedura:
   Il comando `build` usa il nome del progetto e dell'*image registry* per costruire un nome pienamente qualificato per l'immagine della funzione. Se il progetto non era già stato costruito bisogna fornire un *image registry*.

```
(kn) func build
```

2. **Build on-cluster**
   Requisiti:

- la funzione deve esistere in un repository Git[32];
- si deve configurare il cluster per usare *Tekton Pipelines*.

Procedura:
La prima volta che si utilizza tale comando è indispensabile specificare l'URL di Git per la funzione:

```
(kn) func deploy --remote --registry <registry> --git-url <git-url> -p hello
```

Subscribe funzione a CloudEvents
Requisiti:

- **Knative Eventing** installato nel cluster.

Procedura:
Il comando `subscribe` connette la funzione ad un gruppo di eventi, rispettando una serie di filtri per *Cloud Event metadata* e *Knative Broker*, come fonti di eventi.
Quando si iscrive una funzione ad alcuni eventi per un *broker*, se non specificato, si considera quello di default:

```
(kn) func subscribe --filter type=com.example --filter extension=my-extension-
    value [--source my-broker]
```

Invocazione funzione
Con il seguente comando si invia una richiesta di test per invocare una funzione localmente o nel cluster:

```
(kn) func invoke
```

Questo comando viene usato per testare che una funzione non abbia problemi e sia in grado di ricevere correttamente richieste HTTP e **CloudEvents**.
Ci sono vari tipi di flag per simulare diverse tipologie di richieste, per esempio aggiungendo il flag `--data` si possono inviare dati di test alla funzione.
In alternativa, è possibile invocare una funzione inviando richieste HTTP (per esempio tramite il comando `curl`) all'indirizzo fornito a seguito del `deploy` della funzione, messo a disposizione da *Knative Serving*.

[32] `https://git-scm.com/`

Capitolo 3

Setup sperimentale

L'analisi dei framework **Knative**, **OpenWhisk** e **OpenFaaS** è stata realizzata sopra un'architettura sperimentale, la quale è formata da due strati ("layer"):

- **strato fisico**: composto da una macchina fisica equipaggiata con una CPU `Intel(R) Core(TM) i5-11300H 4.40GHz` e `8GB di RAM DDR4 3200MHz`. Il sistema operativo della macchina è `Kubuntu 22.04.4 LTS`.

- **strato virtualizzato**: nel quale la macchina fisica ha installato Docker in versione `v27.2.0`. I container sono gestiti tramite Kubernetes, ovvero un sistema di orchestrazione e gestione di container, il quale è presente all'interno del dispositivo nella versione `v1.31.0`. È stato creato un cluster per ogni framework, formato da un unico *control plane*, tramite il tool KinD, il quale consente di eseguire cluster Kubernetes localmente, sfruttando i container Docker come nodi.

Si è utilizzata la medesima macchina fisica per inviare richieste HTTP, con lo scopo di invocare le funzioni di cui si era fatto il `deploy` su ogni framework serverless. Questo processo di test è stato realizzato mediante una procedura per verificare il comportamento dei framework in diverse situazioni di distribuzione del carico e gestito grazie all'applicazione **JMeter**[1]. Le modalità di autoscaling suddivise per framework, le quali vengono ulteriormente analizzate nel seguente articolo [5], in particolare OpenFaaS e Knative, sono:

[1] `https://jmeter.apache.org/`

- **OpenFaaS**: il componente *AlertManager* legge le metriche (richieste per secondo) dal componente *Prometheus*, in modo da sapere quando inviare un avviso all'*API gateway*.

- **Knative**: si è optato per il **Knative Pod Autoscaler (KPA)**, che è una parte di *Knative Serving*, tramite il quale si effettua un autoscaling in base alle richieste per secondo.

- **OpenWhisk**: mette a disposizione la possibilità di utilizzare l'*autoscaler* di Kubernetes, come l'HPA.

3.1 Metriche qualitative

Le metriche qualitative prese in considerazione per confrontare le diverse piattaforme, sono:

- linguaggi di programmazione supportati;

- supporto a orchestratori di container;

- supporto al monitoraggio: ovvero possedere uno strumento di monitoraggio integrato per controllare ed analizzare le performance;

- modalità di invocazione: nello specifico supporto ad invocazioni sincrone (*HTTP-based*) ed asincrone (*event-based*);

- interfaccia CLI.

3.2 Metriche quantitative

Lo scopo è quello di verificare l'efficienza nella scalabilità in base al carico; dunque si considerano le seguenti metriche:

- **tempo di risposta**: il tempo di risoluzione della richiesta. Viene presa in considerazione la sua misura media in millisecondi;

- **throughput**: il numero di richieste soddisfatte per secondo. L'unità di misura utilizzata è il numero di transazioni al secondo;
- **tasso di successo**: il rapporto tra il numero di richieste con esito positivo e il numero di richieste totali; considerato in percentuale.

Queste metriche vengono testate tramite il tool JMeter, il quale è stato configurato per eseguire 10 richieste HTTP con vari livelli di concorrenza (nello specifico 1, 5, 10, 15 e 20), ovvero il numero di utenti da simulare contemporaneamente. Quest'ultimo viene specificato in JMeter tramite il valore *Number of Threads (Users)*, richiesto durante la creazione di un *Thread Group*. Questo è stato fatto per ogni framework e a sua volta per tutte le funzioni: Node.js, Python e Java.

Capitolo 4

Creazione sistema di test

È stato creato un sistema di test per verificare l'effettivo funzionamento dei seguenti framework: **Knative**, **Apache OpenWhisk** e **OpenFaaS**. Per ogni framework si riportano i passaggi necessari per la configurazione e le modalità di testing delle funzioni. Il test è stato eseguito per tre linguaggi diversi: Python, Node.js e Java. Tutte le funzioni implementano la stessa logica, poco dispendiosa in termini di risorse: ricevono in input un file `JSON` e restituiscono una risposta nello stesso formato, che comprende:

- dati presenti nel file;
- numero di elementi presenti nel file;
- dimensione del file (in Bytes).

Le differenze nelle tre funzioni riguardano principalmente le specificità richieste dal linguaggio, come per esempio le funzioni messe a disposizione per la gestione del formato `JSON`.
La fase di test che si vuole realizzare consiste nell'osservare il comportamento del framework di fronte ad un carico di lavoro, il quale può variare in base alle richieste ricevute; inoltre realizzando la funzione in tre linguaggi diversi, si è in grado di notare se la scelta di un linguaggio rispetto ad un altro incide nelle prestazioni.

Creazione test
I test sono stati creati, come detto in precedenza, tramite JMeter, grazie al quale è stato

possibile realizzarne uno per ogni linguaggio ed ogni livello di concorrenza. Il software consente di creare dei test aggiungendo un **Thread Group** con alcuni parametri:

- **Number of Threads (Users)**: determina il numero di utenti da simulare contemporaneamente; viene utilizzato dunque per configurare il livello di concorrenza. Questo parametro in ogni test assume un valore diverso tra 1, 5, 10, 15 e 20.

- **Ramp-up period (seconds)**: specifica la durata della piena attività degli utenti, mentre JMeter avvia i thread. Questo valore viene scelto sempre uguale ad un secondo, in questo modo si dice a JMeter che in questo intervallo di tempo deve aver avviato tutti i thread.

- **Loop Count**: indica quante volte i thread eseguiranno il test, dunque questo valore viene impostato sempre uguale a 10, in modo da effettuare le richieste HTTP prefissate.

Successivamente al *Thread Group* si aggiunge una richiesta HTTP opportunamente configurata come segue:

- si inserisce il protocollo da utilizzare (HTTP/HyperText Transfer Protocol Secure (HTTPS));

- si inserisce l'URL per raggiungere la funzione da testare;

- si sceglie il metodo della richiesta; quindi in questo caso il metodo POST;

- si aggiunge il file `JSON` da passare come argomento.

I test per il framework Apache OpenWhisk richiedono anche l'aggiunta, nella richiesta HTTP, di un *Authorization Manager*; all'interno del quale si specificano l'*API Host* e l'*Authentication Key*, impostati durante la fase di installazione, per l'URL al quale si inviano le richieste. Inoltre, il suddetto framework è anche l'unico al quale vengono fatte richieste con il protocollo HTTPS, motivo per cui è necessario fornire l'autorizzazione per accedere alle risorse.

Svolgimento test

Per eseguire i test si è sfruttato il seguente comando da CLI, in questo modo si limitano al massimo le risorse occupate da altri processi, come la Graphical User Interface (GUI) di JMeter:

```
jmeter -n -t <path-to-file>/test.jmx -l <path-to-file>/test.csv -e -o
<path-to-directory>/test_report.html
```

Questo comando specifica il file `.jmx`, che consiste nel test creato con JMeter; inoltre si specificano un file `csv` e una directory, all'interno dei quali verranno inseriti i risultati del test.

Gli esperimenti effettuati e il procedimento per poterli replicare si possono trovare nell'apposito repository[1].

[1] `https://github.com/FilippoPracucci/FaaS_frameworks`

Capitolo 5

Risultati sperimentali

La Tabella 5.1 mostra le valutazioni delle metriche qualitative citate nella sezione 3.1. I risultati mostrano che i tre framework analizzati nel dettaglio offrono funzionalità comparabili. Per quanto riguarda le metriche quantitative, elencate nella sezione 3.2, i risultati vengono mostrati nelle sezioni a seguire.

5.1 Grafici risultati sperimentali Node.js

In questa sezione si mostrano i grafici con i risultati dei test eseguiti per le funzioni Node.js. I grafici vengono suddivisi per metrica valutata e in ogni grafico si confrontano i risultati di tutti i framework. Quindi nella Figura 5.1 si mostrano le misurazioni riguardanti lo throughput, inteso come transazioni al secondo; nella Figura 5.2 si riportano i risultati riguardanti la metrica del tempo di risposta medio, in millisecondi, mentre nella Figura 5.3 si visualizza il tasso di successo delle richieste. Infine, ogni grafico suddivide i risultati per livello di concorrenza del test.

5.2 Grafici risultati sperimentali Python

Si mostrano i grafici con i risultati dei test eseguiti per le funzioni Python. I grafici vengono suddivisi per metrica ed in ogni grafico si confrontano i risultati di ogni framework. Dunque nella Figura 5.4 si riporta lo throughput, inteso come transazioni al secondo; nella Figura 5.5 si mostrano i risultati riguardanti il tempo di risposta medio in

	OpenFaaS	**Knative**	**Apache OpenWhisk**
Linguaggi di programmazione supportati	Bash, Dockerfile, C#, Go, Java, Node.js, PHP, Python, Ruby, Rust	Node.js, Python, Go, Quarkus, Rust, Spring Boot, TypeScript	Go, Java, Node.js, PHP, Python, Ruby, Rust, Swift, .NET Core, Dockerfile
Supporto a Orchestratori di container	Kubernetes, OpenShift[1]	Kubernetes	Kubernetes, "Standalone OpenWhisk Stack", Docker Compose, Ansible, Vagrant
Supporto al monitoraggio	Prometheus con Grafana[2]	Prometheus con Grafana, OpenTelemetry Collector[3]	Nessuno
Modalità di invocazione	HTTP e sorgenti di eventi	HTTP e *CloudEvents*	HTTP e *Triggers*
Interfaccia CLI	faas-cli	kn, func	wsk

Tabella 5.1: Metriche qualitative

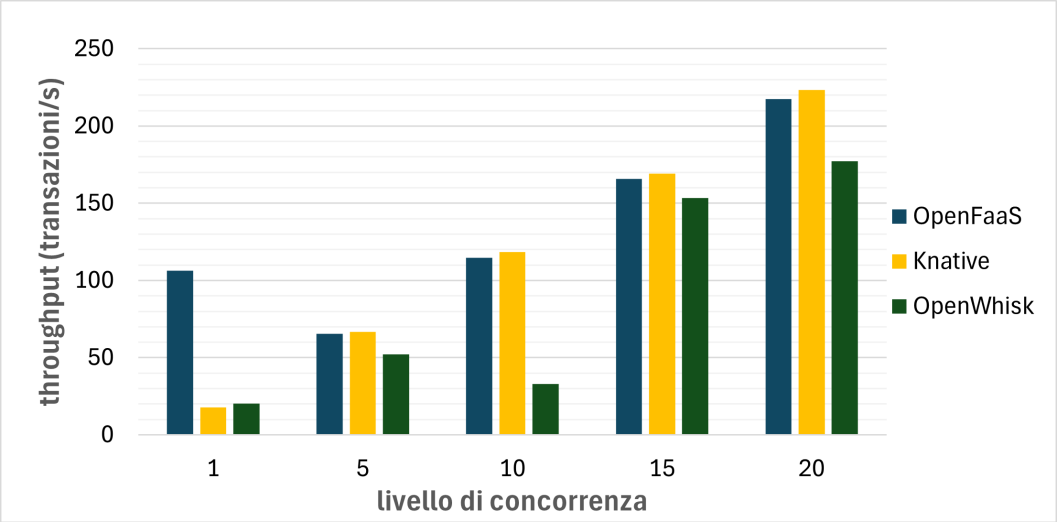

Figura 5.1: Grafici risultati sperimentali throughput (transazioni/s) Node.js

millisecondi ed infine nella Figura 5.6 si visualizza il tasso di successo delle richieste, in percentuale. Inoltre, ogni grafico suddivide i confronti per livello di concorrenza del test.

5.3 Grafici risultati sperimentali Java

In questa sezione si mostrano i grafici rappresentanti i risultati dei test eseguiti per le funzioni Java. I grafici vengono separati in base alla metrica presa in considerazione per poi confrontare i risultati di tutti i framework. Nella Figura 5.7 si mostrano le misurazioni riguardanti la metrica dello throughput, misurata in transazioni al secondo; mentre nella Figura 5.8 si visualizzano i risultati relativi al tempo di risposta medio, in millisecondi ed infine nella Figura 5.9 si riporta il tasso di successo delle richieste. Tutti i grafici suddividono i risultati in base al livello di concorrenza del test.

5.4 Analisi risultati

I risultati che sono stati riportati nei grafici evidenziano vari aspetti:

- OpenFaaS risulta essere il framework con prestazioni migliori sotto il punto di vista dello throughput. Infatti, come si può notare nelle Figura 5.1 e Figura 5.4,

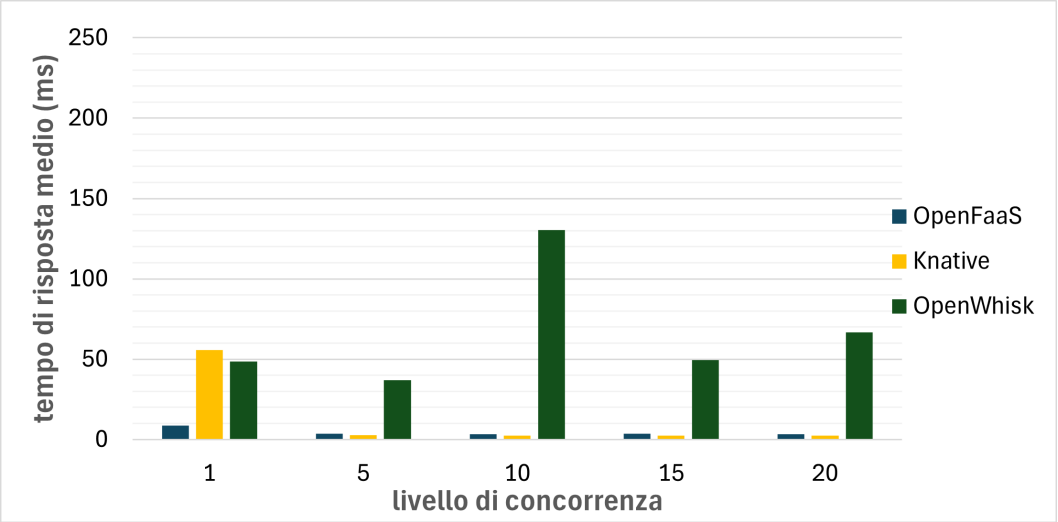

Figura 5.2: Grafici risultati sperimentali tempo di risposta medio (ms) Node.js

in corrispondenza delle funzioni realizzate in Node.js e Python, il suddetto framework mostra un numero di transazioni al secondo nettamente superiore rispetto agli altri con un livello di concorrenza uguale ad uno, per poi allinearsi ad essi all'aumentare del carico, in particolare con Knative. Il comportamento è diverso per la funzione realizzata in Java, il cui grafico è raffigurato nella Figura 5.7, il quale evidenzia che lo throughput maggiore lo si ha in Knative per tutti i livelli di concorrenza. Questo è dovuto al fatto che il framework supporta tale linguaggio tramite l'ausilio dei framework Spring Boot[4] e Quarkus[5] (in questo caso è stato sfruttato il primo). In generale, all'aumentare del livello di concorrenza, lo throughput aumenta di conseguenza per tutti i framework.

- I tempi di risposta medi per ogni linguaggio sono molto più elevati per il framework OpenWhisk rispetto agli altri, i quali presentano delle misure comparabili; questo comportamento lo si può osservare in Figura 5.2 e Figura 5.5. Analogamente allo throughput, il tempo di risposta della funzione realizzata in Java è inferiore per Knative in maniera rilevante, anche rispetto ad OpenFaaS, come possiamo notare nella Figura 5.8; la motivazione è la stessa fornita al punto precedente.

[4] `https://spring.io/projects/spring-boot`
[5] `https://quarkus.io/`

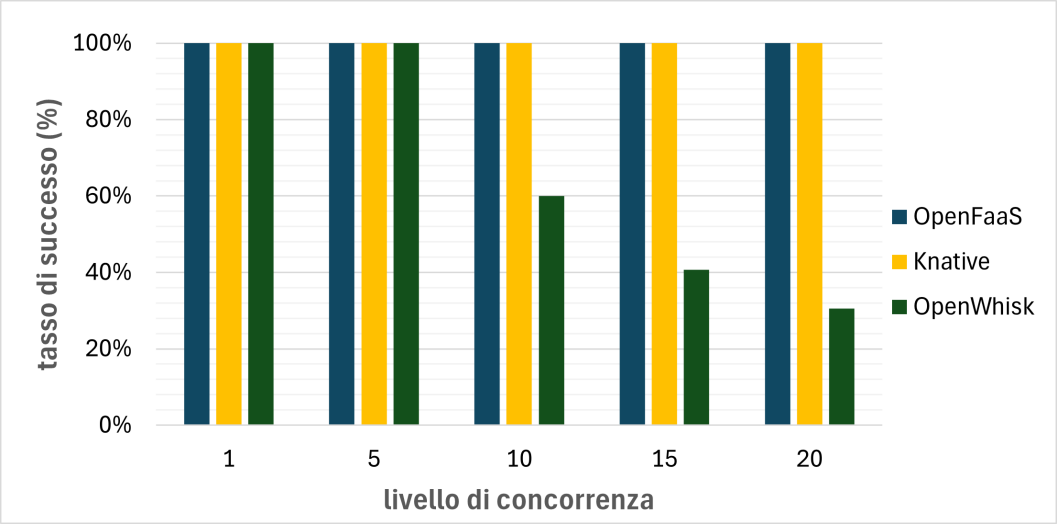

Figura 5.3: Grafici risultati sperimentali tasso di successo (%) Node.js

- Il tasso di successo delle richieste effettuate nei test di ogni funzione, presenta sempre lo stesso comportamento: il 100% delle richieste viene risposto con successo per quanto riguarda Knative e OpenFaaS, mentre diminuisce all'aumentare del livello di concorrenza per OpenWhisk. Infatti, si può notare dai grafici in Figura 5.3, Figura 5.6 e Figura 5.9 che il tasso di successo delle risposte alle richieste per quanto riguarda OpenWhisk, diminuisce all'aumentare del livello di concorrenza, a cominciare da un numero di richieste concorrenti uguale a dieci. Questo probabilmente è dovuto al fatto che il framework in questione gestisce tutte le richieste HTTP in ingresso tramite un singolo componente centralizzato, ovvero *Nginx*, creando una sorta di "collo di bottiglia" all'aumentare del numero di richieste.

I risultati ottenuti trovano un riscontro anche nel lavoro eseguito nel documento [2], all'interno del quale si giunge alla medesima conclusione, ovvero che il framework OpenWhisk offra le prestazioni peggiori, sia in termini di reattività ed efficienza, sia in termini di affidabilità; mentre Knative ed OpenFaaS siano sullo stesso livello. Nel documento sopracitato, le metriche quantitative prese in considerazione equivalgono a quelle qui valutate, motivo per cui si può dire che dal punto di vista quantitativo i risultati prestazionali ottenuti siano comparabili, quantomeno in termini di confronto tra i framework che sono stati analizzati in entrambi i lavori.

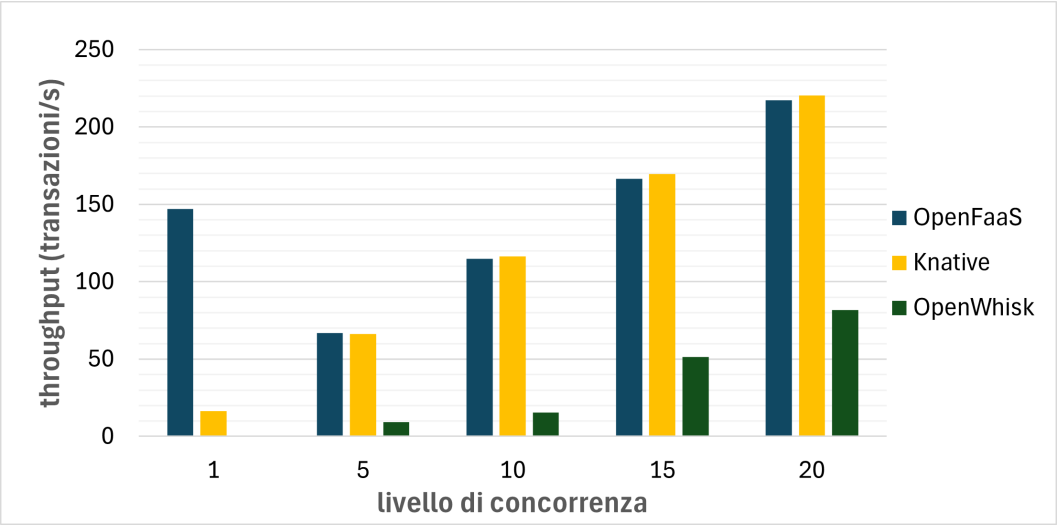

Figura 5.4: Grafici risultati sperimentali throughput (transazioni/s) Python

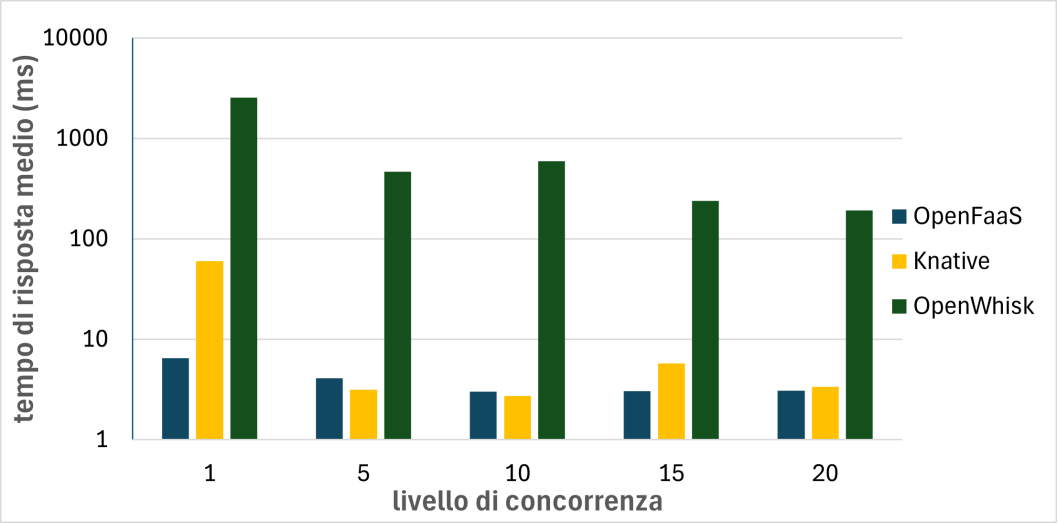

Figura 5.5: Grafici risultati sperimentali tempo di risposta medio (ms) Python

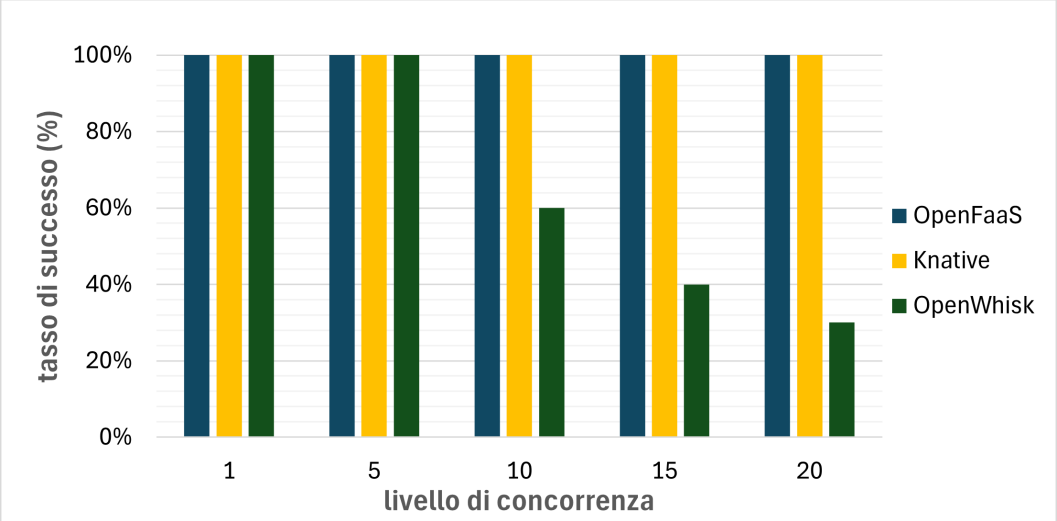

Figura 5.6: Grafici risultati sperimentali tasso di successo (%) Python

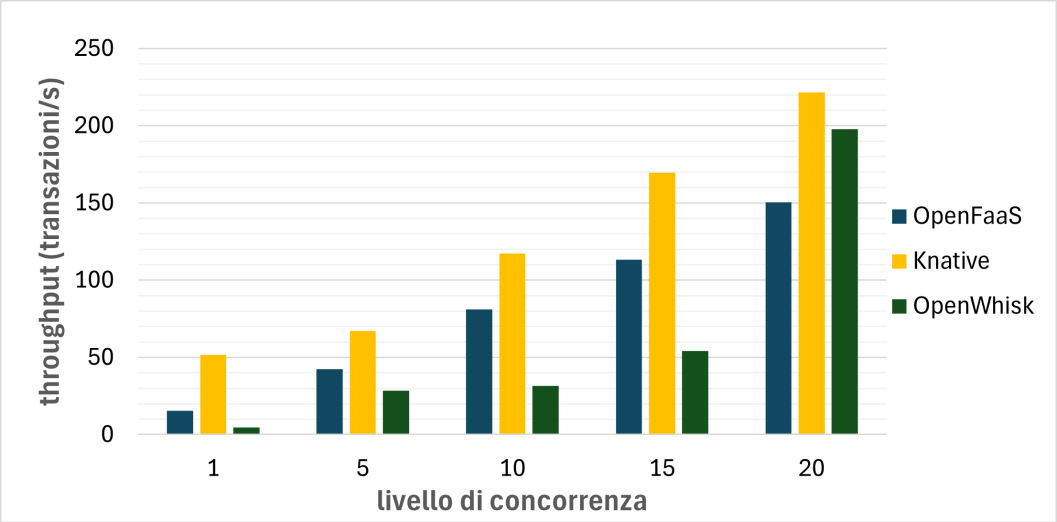

Figura 5.7: Grafici risultati sperimentali throughput (transazioni/s) Java

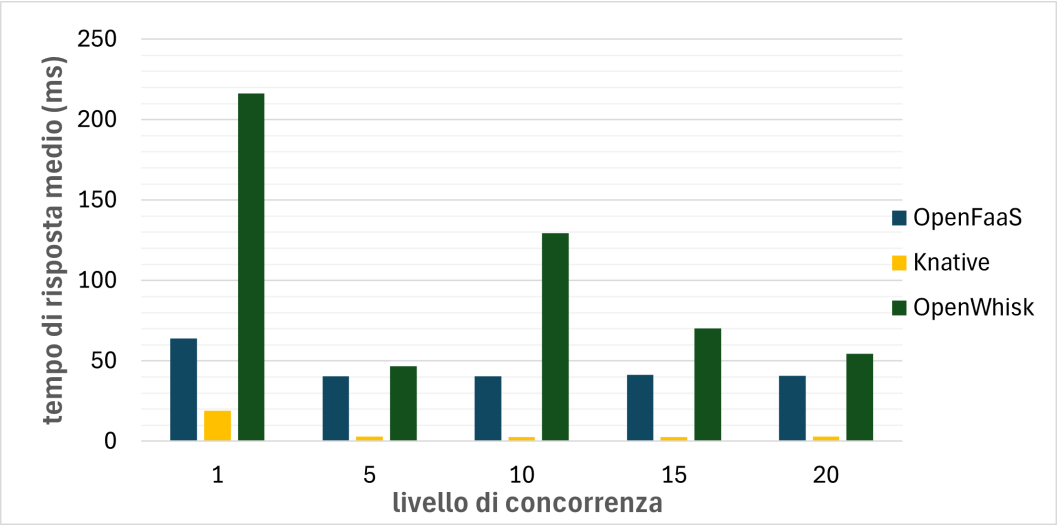

Figura 5.8: Grafici risultati sperimentali tempo di risposta medio (ms) Java

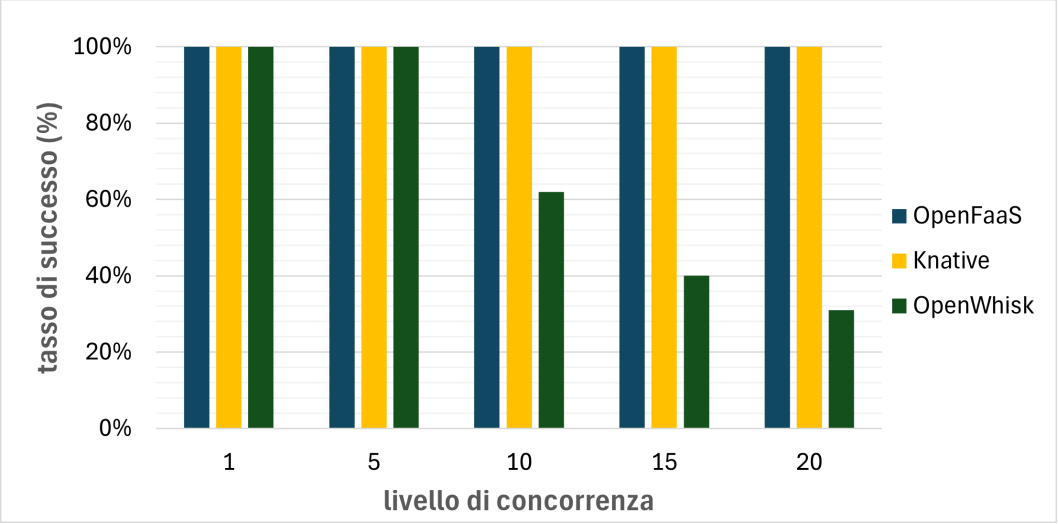

Figura 5.9: Grafici risultati sperimentali tasso di successo (%) Java

Capitolo 6

Conclusioni

L'architettura serverless si è dimostrata molto utile nel gestire varie tipologie di necessità, su tutte quella di sviluppare funzioni che operano *at the edge* di un'infrastruttura e su sistemi distribuiti; questo grazie ad una quantità di risorse richieste non eccessiva, anche per merito della possibilità di sfruttare varie architetture adattabili ai bisogni e alle strutture che si hanno a disposizione. A questo proposito sono stati eseguiti uno studio e un'analisi delle principali soluzioni che il mondo dei framework **FaaS** Open Source mette a disposizione. Nello specifico, si è optato di andare a fondo nell'analisi e nel testing dei framework **Knative**, **OpenWhisk** ed **OpenFaaS**, escludendo **Kubeless**, in quanto non è più mantenuto. L'analisi svolta e l'utilizzo di tutte e tre le piattaforme ha portato alla conclusione che ognuno ha i suoi punti di forza e di debolezza. Se si osservano da un punto di vista quantitativo, OpenWhisk offre le prestazioni peggiori in ogni aspetto, ovvero throughput, tempo di risposta medio e tasso di successo; mentre Knative e OpenFaaS si possono considerare sullo stesso livello da un punto di vista della consistenza nelle prestazioni. Infatti, come analizzato in precedenza, OpenWhisk presenta uno throughput inferiore, un tempo di risposta medio notevolmente superiore e un tasso di successo che diminuisce considerevolmente all'aumentare del livello di concorrenza, dimostrando minore efficienza prestazionale e affidabilità nel portare a termine positivamente le richieste.
In conclusione, OpenWhisk soffre l'aumento del carico maggiormente rispetto agli altri due framework. Da un punto di vista più qualitativo, ovvero di linguaggi supportati, facilità di sviluppo e materiale fornito, OpenFaaS è il framework più flessibile e perso-

nalizzabile in base alle esigenze, oltre a fornire molto materiale di supporto. Considerando i medesimi aspetti, Knative e OpenWhisk vengono valutati leggermente inferiori. In futuro potrebbe risultare utile effettuare ulteriori test, di diversa natura, su questi framework e provare ad implementarli all'interno di un sistema distribuito esistente e funzionante, in modo da osservare l'effettiva efficacia di questa architettura.

Bibliografia

[1] S. K. Mohanty, G. Premsankar, and M. di Francesco, "An evaluation of open source serverless computing frameworks," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, Dec. 2018.

[2] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," in *2019 IEEE World Congress on Services (SERVICES)*, IEEE, July 2019.

[3] V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, and F. Leymann, "Faasten your decisions: A classification framework and technology review of function-as-a-service platforms," *Journal of Systems and Software*, vol. 175, p. 110906, May 2021.

[4] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, (Boston, MA), pp. 133–146, USENIX Association, July 2018.

[5] J. Li, "Analyzing open-source serverless platforms: Characteristics and performance (s)," in *Proceedings of the 33rd International Conference on Software Engineering and Knowledge Engineering*, vol. 2021 of *SEKE2021*, p. 15–20, KSI Research Inc., July 2021.