ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA MAGISTRALE

in

Infrastructures for cloud computing and big data M

Architetture serverless per algoritmi massicciamente paralleli in ambito Industria 4.0

Candidato:

Edoardo Tentarelli

Relatore:

Chiar.mo Prof. Antonio Corradi

Correlatori:

Dott. Ing. Luca Foschini
Dott. Andrea Sabbioni
Dott. Lorenzo Patera
Giacomo Lorenzo

Anno Accademico 2018-2019

Sessione III

Abstract

L’edge computing permette di distribuire l’elaborazione di servizi direttamente sulle macchine di produzione, evitando di inviare la richiesta verso data center esterni all’organizzazione, con vantaggi evidenti in termini di latenza e sicurezza. Questo modello di esecuzione, molto diffuso in industria manufatturiera, sta portando ad una migrazione dei servizi verso ambienti edge, ma la quantità limitata di risorse rende difficile il deployment di servizi computazionalmente onerosi verso questo modello.

Ultimamente, sono state rilasciate sul mercato piattaforme che garantiscono la completa gestione dell’ambiente di esecuzione, sollevando lo sviluppatore da qualsiasi pratica operazionale. Ogni allocazione di risorse è ottimizzata, trasparentemente, dalla piattaforma, garantendo elevati gradi di disponibilità e tolleranza dei servizi. Questo modello di esecuzione viene definito serverless, e molte organizzazioni stanno migrando i propri servizi verso queste soluzioni.

L’obiettivo di questo studio è stato quello di valutare le prestazioni del serverless nell’elaborazione di funzioni di image processing in ambienti edge. In particolare, lo studio è stato effettuato su algoritmi massicciamente paralleli, per cui è stato possibile parallelizzare il carico in task indipendenti. Le sperimentazioni hanno confrontato una soluzione serverless, in cui parti di immagini sono state ruotate in parallelo, ed una soluzione sequenziale, in cui la rotazione è stata effettuata sull’intera immagine.

I risultati ottenuti mostrano evidenti benefici verso la soluzione serverless, in quanto offre parametri di scalabilità maggiori. Inoltre, i consumi di risorse sono decisamente più limitati, garantendo una soluzione più idonea ad ambienti edge e adatta al caso d’uso applicativo preso in esame. Per queste considerazioni, è consigliata la migrazione di servizi CPU intensive verso architetture serverless, per poter beneficiare dei risparmi e dei vantaggi offerti da questo tipo di soluzioni.

Sommario

1 Introduzione

Dall'avvento di Internet, e soprattutto dalla nascita del Web, il numero di organizzazioni che ha iniziato ad offrire servizi remoti è cresciuto costantemente, diventando oggi un requisito fondamentale se si vuole essere competitivi sul mercato. Questa opportunità, semplice oggi ma impressionante fino a qualche decennio fa, è resa possibile dagli sviluppi nel campo dei sistemi distribuiti, dove diversi tipi di risorse cooperano per fornire servizi ad un numero indefinito di utenti. Le architetture distribuite hanno permesso ad Internet di diventare protagonista assoluto della società in cui viviamo, permettendo la fruizione di immense quantità di dati, in tempo reale e da qualsiasi posizione.

Nel 2002, *Amazon* introduce sul mercato *Amazon Web Service* [1], servizio che permette l'affitto da remoto di risorse computazionali, come virtual machine dove configurare ed eseguire sistemi distribuiti. Il cloud computing diventa una realtà diffusa, e le aziende iniziano a migrare i propri servizi dai server proprietari verso le risorse remote messe a disposizione dai provider cloud. Le tecnologie di virtualizzazione presenti in queste infrastrutture permettono alle organizzazioni di generare nodi in pochi minuti, ottimizzando l'utilizzo delle risorse e risparmiando soldi per l'acquisto ed il mantenimento di hardware costosi. Negli ultimi anni, la rivoluzione del cloud computing ha toccato tutti i settori economici, pervadendo ogni organizzazione e società interessata alla distribuzione di servizi online.

L'industria manufatturiera ha sfruttato le potenzialità offerte dal cloud computing, riconoscendo però l'importanza di ospitare questi servizi all'interno del perimetro industriale, in modo da abbattere i tempi di latenza e favorire la sicurezza delle informazioni. Per queste ragioni, molte organizzazioni hanno iniziato a migrare i propri servizi su ambienti edge, così chiamati perché serviti ai confini della rete, ma hanno trovato difficoltà nell'esecuzione di applicazioni complesse, a causa delle risorse limitate tipiche dei nodi industriali, quali macchine, sensori, ed attuatori.

La nascita di nuove soluzioni, chiamate serverless, promette una gestione completamente automatizzata ed ottimizzata delle risorse, rimettendo in discussione le possibilità computazionali offerte da ambienti edge. A tal proposito, la seguente tesi cerca di fare luce sulle possibilità del serverless come scelta architetturale per il deployment di servizi computazionalmente onerosi in macchine con quantità limitate di risorse.

Per queste ragioni, la trattazione segue una descrizione dei sistemi distribuiti, andando a far luce su architetture e middleware offerti, per poi proseguire sul cloud computing, trattando i servizi ed i modelli di deployment, oltre che gli svantaggi e le limitazioni che questa soluzione comporta. Al termine di questa panoramica, verrà introdotto il concetto di serverless, con i relativi vantaggi proposti da questa soluzione, prendendo in

considerazione le proposte open source più popolari presenti su Internet ed andando a confrontarle in base all'architettura, all'utilizzo ed al grado di configurazione offerta. Alla trattazione qualitativa delle piattaforme, seguiranno test per confrontare le prestazioni di ogni singola soluzione. Infine, la trattazione si concluderà con un confronto tra una classica soluzione a microservizi ed una soluzione serverless, mettendo in luce i vantaggi e gli svantaggi di un'architettura rispetto all'altra.

2 I sistemi distribuiti

Nel corso degli anni sono state date numerose definizioni di sistema distribuito, molte delle quali non soddisfacenti, o in completa contraddizione con le altre. Una definizione generica potrebbe essere:

> *Un sistema distribuito è una collezione di computer indipendenti che si mostrano all'utente come un singolo sistema coeso [2]*. (Van Steen, 2016)

Questa definizione porta alla luce le prime caratteristiche di un sistema distribuito: la composizione del sistema in diversi componenti indipendenti, e l'astrazione di mostrarsi come un sistema unico, compatto ed atomico. Quindi, si può dire che il punto principale di un sistema distribuito è l'interazione che avviene tra i diversi componenti separati, dove i meccanismi che rendono possibile tutto questo sono alla base nello sviluppo di un buon sistema distribuito. Inoltre, l'utente che interagisce con il sistema è ignaro dello spazio e del tempo in cui il sistema è eseguito [2].

L'obiettivo principale dei sistemi distribuiti è di garantire l'accesso a risorse remote, e permetterne la condivisione. La necessità di voler condividere le risorse è innanzitutto economica, data la possibilità di avere una risorsa, come una stampante o un dispositivo di storage, condivisa, invece di acquistare nuovo hardware per ogni utente che lo richiede. La condivisione permette di sfruttare al massimo ogni risorsa, e migliora la collaborazione e lo scambio di informazioni tra utenti. Inoltre, la completa diffusione di Internet ha permesso l'implementazione di sistemi distribuiti a basso costo, data la possibilità di avvalersi di una rete globale già pronta, accessibile da chiunque. Ovviamente, con il diffondersi di soluzioni distribuite, la sicurezza è diventata una preoccupazione importante, e sono necessari sforzi per garantire la riservatezza e l'integrità dei dati condivisi.

La trasparenza è una caratteristica fondamentale, soprattutto per la rappresentazione dei dati, in quanto ogni nodo del sistema distribuito condivide le proprie informazioni in comune accordo con gli altri nodi del cluster, astraendo completamente dalle implementazioni di ogni nodo, come ad esempio il filesystem adottato o il sistema operativo in uso. In un sistema distribuito, le risorse sono libere di migrare per motivi di sicurezza ed ottimizzazione, senza che questo sia visibile esternamente, permettendo la generazione di nuove repliche al fine di garantire latenze minori e migliorare il grado di tolleranza ai guasti.

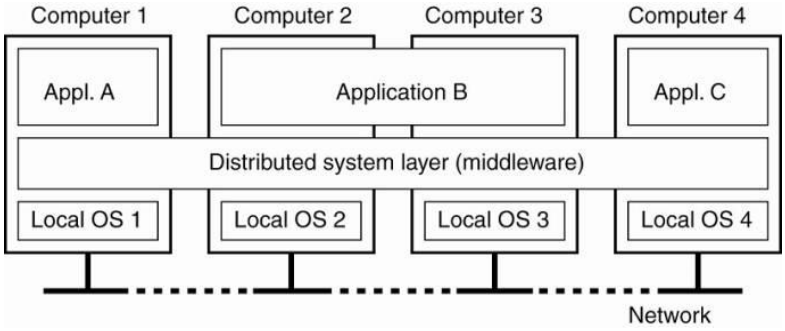

Figura 1 – Architettura generale di un sistema distribuito.

Guardando lo schema in Figura 1, ogni sistema distribuito si avvale di un set di middleware, i quali cooperano per garantire l'astrazione richiesta nella definizione precedente.

> *Per middleware si intende ogni software che si interpone tra sistema operativo e logica applicativa [3].* (Microsoft Azure)

I middleware possono essere raggruppati nelle seguenti categorie:

- Remote Procedure Call (RPC) – permettono di invocare procedure remote facendole sembrare locali. Il middleware genera un meccanismo di collegamento con il nodo remoto, rendendo le procedure trasparenti al consumatore. Tradizionalmente, questo tipo di middleware veniva usato per programmi procedurali, mentre oggi viene usato per applicazioni object-oriented. Molti linguaggi di programmazione possiedono API standard per interazioni RPC, come Java RMI [4], package rpc [5] (Go), o RPyC [6] (Python);
- Object Request Broker (ORB) – permettono la distribuzione di oggetti in applicazioni object-based verso una rete di nodi eterogenei. Lo standard ORB più conosciuto è CORBA (Common Object Request Broker Architecture) [7], il quale sfrutta a sua volta meccanismi RPC per permettere l'interazione di oggetti diversi, comunicanti tramite la definizione di interfacce che definiscono il comportamento di ogni componente;
- Message Oriented Middleware (MOM) – permettono alle applicazioni distribuite di comunicare tramite scambio di messaggi. Come per gli altri middleware, esistono diversi standard tra cui AMQP (Advanced Message Queuing Protocol) [8] o MQTT (MQ Telemetry Transport) [9], quest'ultimo utilizzato soprattutto per contesti IoT data la natura lightweight del protocollo.

In definitiva, i middleware permettono l'interazione tra i diversi componenti eseguiti sul sistema distribuito, modificando il loro comportamento a runtime [10]. Nel seguente paragrafo si discuteranno le principali architetture presenti in letteratura.

2.1 Architetture MVC e microservizi

Tipicamente, le architetture distribuite sono basate su logica client-server, dove un client richiede delle risorse a un server e rimane in attesa finché il server non risponde all'invocazione. L'interazione è sincrona, e la computazione è completamente eseguita sul server e nascosta al client, secondo i principi di trasparenza dei sistemi distribuiti. Le architetture client-server offrono all'utente servizi di computazione verso dati presenti nel server. Per queste ragioni, l'architettura può essere divisa in tre livelli imprescindibili:

- Logica di presentazione – contiene le informazioni necessarie per la presentazione del servizio all'utente, come per esempio l'interfaccia Web;
- Logica di business – anche chiamato livello applicativo, contiene le funzionalità dell'applicazione con cui vengono elaborati i dati;
- Logica di accesso e modellazione dei dati – gestisce il ciclo di vita dei dati da elaborare.

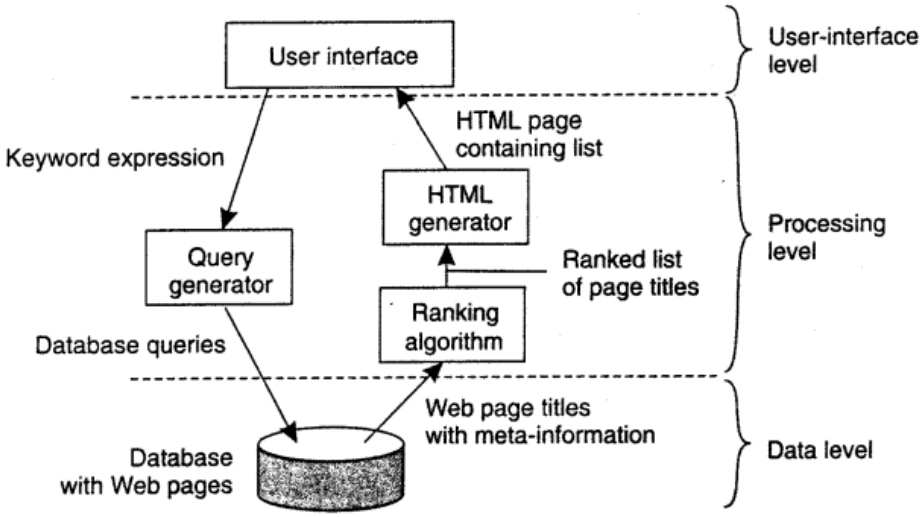

Figura 2 - Architettura semplificata di un motore di ricerca diviso nei tre livelli principali.

Un esempio di logica client-server è visibile in Figura 2, dove l'architettura di un motore di ricerca è formata dalla logica di presentazione, fornita dalla pagina Web, ed è possibile scrivere la stringa da cercare. Il backend è formato da funzioni che richiedono i dati da un unico database, elaborandoli e restituendo le pagine più pertinenti alla stringa cercata. Inoltre, le pagine vengono restituite secondo un preciso ordine, stabilito da algoritmi di

ranking, ed il risultato complessivo viene trasformato in codice HTML e restituito all'utente [2].

L'architettura appena descritta viene anche chiamata architettura MVC, ovvero Model-View-Controller, ed è molto diffusa per lo sviluppo di applicazioni Web. I tre livelli di un'architettura client-server sono stati separati in tre diversi attori, ognuno con la sua responsabilità. Il client offre il livello di presentazione (view), spesso realizzato tramite interfaccia grafica, con cui comunica verso il livello di business. Quest'ultimo, anche chiamato livello applicativo (controller), effettua operazioni computazionali per raggiungere risultati da restituire al client. Infine, il livello di modellazione dei dati (model) offre meccanismi di persistenza e di tolleranza ai guasti, oltre alle interfacce di accesso, utilizzate dal livello di business per poter elaborare le richieste.

Nonostante la divisione in livelli, la crescente complessità degli attuali sistemi distribuiti richiede livelli di granularità maggiori, capaci di replicare funzionalità specifiche, oppure di poter lavorare su tabelle diverse, invece che sull'intero database. Per esempio, in una architettura MVC, l'application server risulta essere un unico blocco di funzioni, le quali devono essere contemporaneamente replicate su più nodi, nonostante molte di queste non richiedano un intervento del genere. Il database server, allo stesso modo, presenta dati che non vengono acceduti allo stesso ritmo di altri, e dunque molte tabelle non richiedono di essere migrate o duplicate al contrario di altre [11].

Per far fronte a queste problematiche, sono nate le architetture a microservizi:

> *Per architettura a microservizi si intende lo sviluppo di una singola applicazione come insieme di piccoli servizi, eseguiti in processi separati e interagenti con protocolli leggeri di comunicazione.* (Martin Fowler, 2014)

I benefici delle architetture a microservizi riguardano principalmente la modularità e l'indipendenza di ogni servizio. La modularità aiuta l'implementazione di sistemi complessi, in quanto i team di sviluppo possono lavorare con maggiore elasticità ed in maniera autonoma. Inoltre, il deployment di nuovi servizi ha meno probabilità di portare guasti al sistema, dato il disaccoppiamento offerto, ed è possibile lavorare con più linguaggi di programmazione differenti, sfruttando i vantaggi di ogni semantica. Il costo di questo tipo di architettura è da trovarsi nella distribuzione granulare dei servizi, che richiede maggiori competenze per la gestione dell'intero sistema. Inoltre, la difficoltà principale è la consistenza, parametro fondamentale in ogni database, che genera overhead troppo significativi per poter essere trascurati.

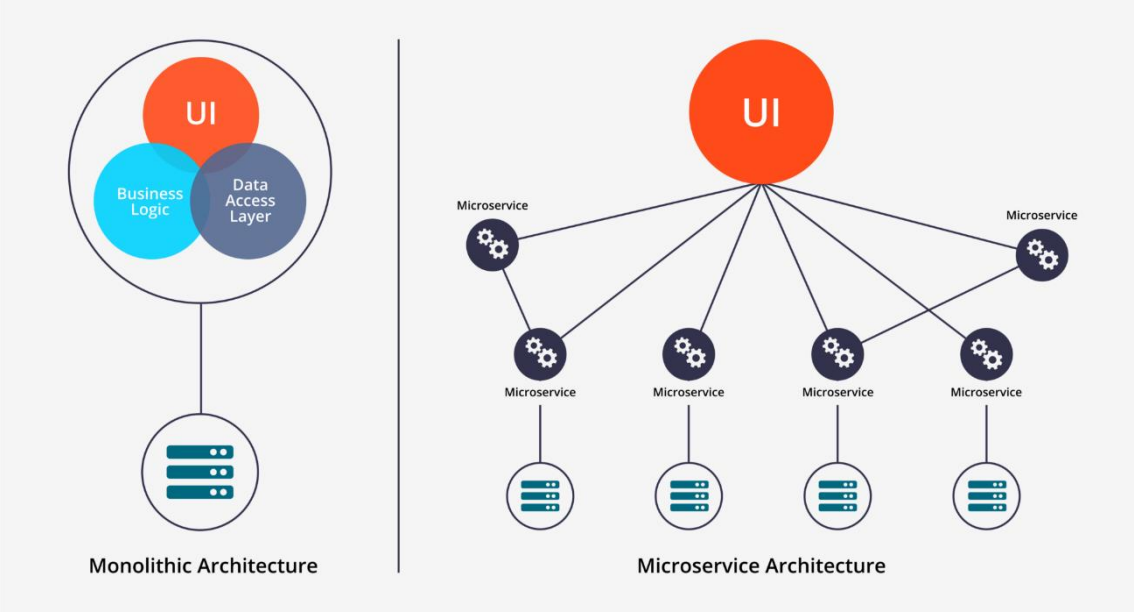

Figura 3 - Confronto tra architettura three-tier ed architettura a microservizi.

Per capire i vantaggi derivanti dalle architetture a microservizi, è possibile compararle alle architetture monolitiche client-server descritte precedentemente. In quest'ultima, il lato server presenta un'architettura monolitica, ovvero una singola istanza di esecuzione, ed ogni modifica al sistema richiede il deployment dell'intero server. Al contrario, come è visibile in Figura 3, la granularità dei microservizi permette la replicazione mirata del singolo servizio, garantendo un miglior grado di scalabilità a costo di una complessità gestionale maggiore, dovuta al numero superiore di componenti in gioco.

L'importanza dei microservizi in grandi organizzazioni è dovuta alla possibilità di dividere i team di lavoro, permettendo ad ognuno di questi di focalizzarsi sul singolo servizio, ottimizzando e riducendo tempi e costi di sviluppo. Oltre ai gruppi di sviluppo e testing, i microservizi richiedono maggiori competenze nel deployment e gestione a runtime dei componenti, dato il maggior numero di rilasci di nuove versioni, e richiedono la presenza di team operazionali, chiamati DevOps, che garantiscano la corretta esecuzione di ogni componente nel sistema distribuito [12].

2.2 API Gateway

Durante lo sviluppo di un'architettura a microservizi, è possibile avere numerosi componenti che necessitano di comunicare direttamente con la logica di presentazione offerta al cliente. In un'architettura monolitica questo non rappresenta un problema, in quanto il numero di endpoint a cui il cliente può accedere è limitato ad un unico set di

API. Invece, nelle architetture a microservizi ogni componente possiede un set di API proprietario, e risulta difficile gestire diverse logiche di presentazione che comunicano con questi servizi.

È possibile permettere al client di comunicare direttamente con ogni microservizio, e, in questo caso, ogni microservizio avrebbe un endpoint pubblico del tipo:

```
https://serviceName.api.company.name
```

L'URL andrebbe ad invocare il load balancer del servizio, il quale smisterebbe la richiesta verso le istanze disponibili. Questo approccio presenta diverse limitazioni, soprattutto in contesti distribuiti di grandi dimensioni, in quanto l'utente potrebbe trovarsi ad invocare diverse richieste separate per portare a termine l'operazione. Inoltre, i protocolli utilizzati dai diversi microservizi possono essere poco adatti alla comunicazione via Web, in quanto rischiano di essere bloccati dai firewall di rete presenti nel dispositivo del client. Infine, la comunicazione diretta rende difficile il refactoring del microservizio, in quanto diversi client possono essere strettamente accoppiati con quel servizio, impedendo modifiche al codice ed ai protocolli di comunicazione implementati. Per queste ragioni, si preferisce centralizzare le richieste verso un API Gateway, il quale fa da ponte tra i client ed i microservizi.

L'API Gateway, come mostrato in Figura 4, è un server che offre un singolo punto di accesso al sistema. Questo riceve le richieste dai client e le indirizza correttamente al microservizio desiderato, permettendo di aggregare le richieste provenienti da protocolli diversi, di smistarle se queste sono troppo numerose, o di restituire risultati aggregati se più utenti richiedono un servizio dallo stesso endpoint. Inoltre, un API Gateway può offrire servizi di autenticazione, monitoring, load balancing o caching, per ottimizzare l'utilizzo delle risorse richieste dai diversi client.

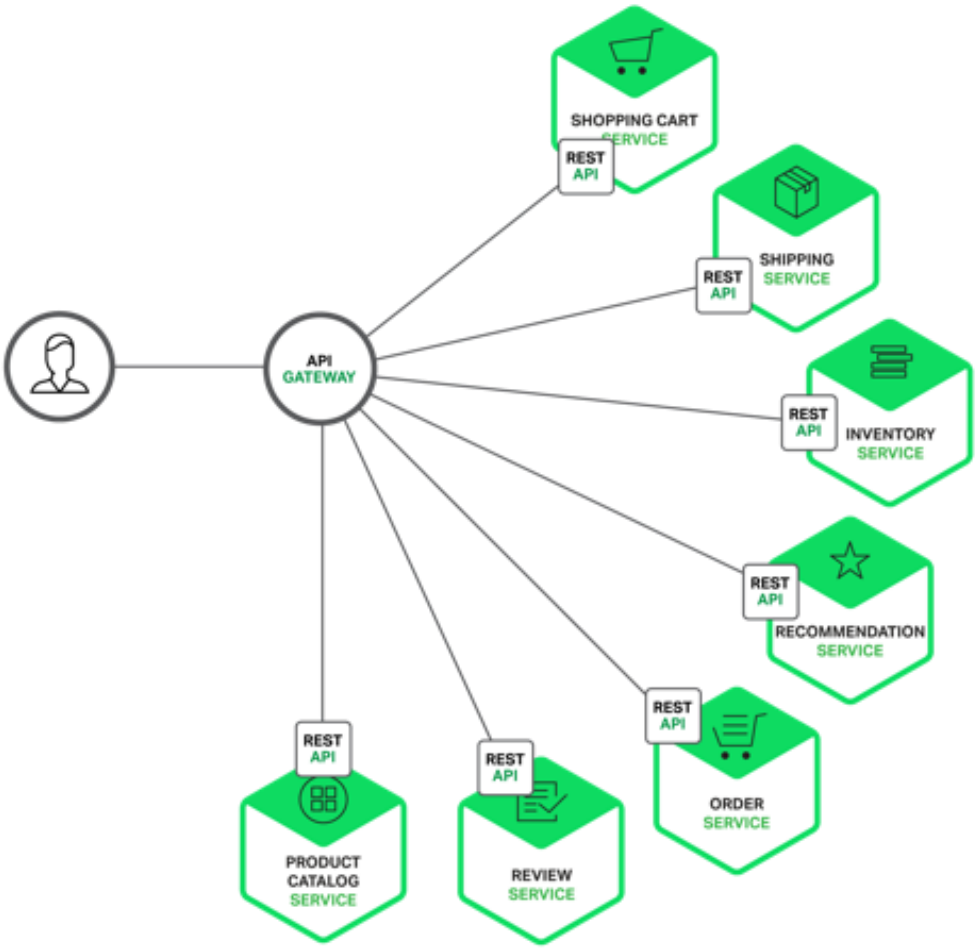

Figura 4 - Architettura di un sistema distribuito con API Gateway.

L'aspetto principale offerto dall'API Gateway è l'aggregazione di risultati ricevuti da più microservizi. Nella comunicazione diretta, il client è responsabile delle richieste verso ogni singolo servizio, mentre in questo caso il client necessita di un'unica invocazione, e l'API Gateway si fa carico di invocare i servizi, aspettare i risultati e restituire i dati aggregati. Inoltre, l'API Gateway nasconde la struttura interna dell'applicazione, permettendo un maggiore disaccoppiamento tra client e server, rispettando gli obiettivi principali richiesti da un sistema distribuito.

È importante che l'API Gateway sia un componente altamente scalabile e performante, perché deve elaborare ogni richiesta indirizzata verso il sistema, rischiando di diventare un importante collo di bottiglia per l'intera applicazione. Inoltre, ad ogni nuovo rilascio di qualsiasi nuovo microservizio è necessario aggiornare l'API Gateway, ed è dunque importante che l'aggiornamento di quest'ultimo sia rapido ed efficiente, per evitare di tenere in sospeso il rilascio di nuovi microservizi [13].

2.3 Message-Oriented Middleware

Negli ultimi anni, la complessità dei sistemi distribuiti ha continuato ad aumentare, complice l'eterogeneità dei servizi implementati. L'enorme quantità di interazioni da gestire, in scenari di esecuzione dove sono presenti diversi linguaggi di programmazione, ognuno con diverse librerie ed implementazioni, ha richiesto sforzi notevoli, soprattutto di fronte alla necessità di gestire logica computazionale e comunicativa di ogni servizio. Inoltre, i requisiti di affidabilità e tolleranza ai guasti sono diventati più stringenti, richiedendo livelli di QoS (Quality of Service) più importanti, gradi di sicurezza più elevati e affidabilità garantita 24/7. In questi scenari, è facile notare come meccanismi classici di comunicazione, quali RPC (Remote Procedure Call), non siano più adeguati a sostenere i nuovi requisiti che ogni sistema dovrebbe possedere per garantire un deployment affidabile [14].

Per far fronte a queste necessità, nuove alternative ai classici meccanismi di comunicazione sono state introdotte nel mercato. In particolare, un nuovo meccanismo, chiamato MOM (Message-Oriented Middleware), offre un nuovo approccio all'interazione tra i servizi, ponendosi come un blocco fondamentale nel deployment di ogni sistema distribuito odierno. Per MOM si intendono tutti i servizi che offrono un'infrastruttura pronta per lo scambio di messaggi disaccoppiato tra i servizi. Brevemente, un sistema MOM può inviare e ricevere messaggi, offrendosi come intermediario tra i diversi componenti in gioco. In questo modo, viene evitata l'interazione diretta tra i servizi, in quanto si instaurata una comunicazione indiretta che comporta numerosi benefici, soprattutto in architetture a microservizi [10].

2.3.1 I limiti del meccanismo RPC

Il tradizionale modello di comunicazione RPC, in Figura 5, rappresenta un concetto fondamentale dei sistemi distribuiti. L'obiettivo è quello di garantire l'interazione tra due processi, come avviene nelle invocazioni di procedure locali, dove il controllo è passato da una procedura all'altra in maniera sincrona.

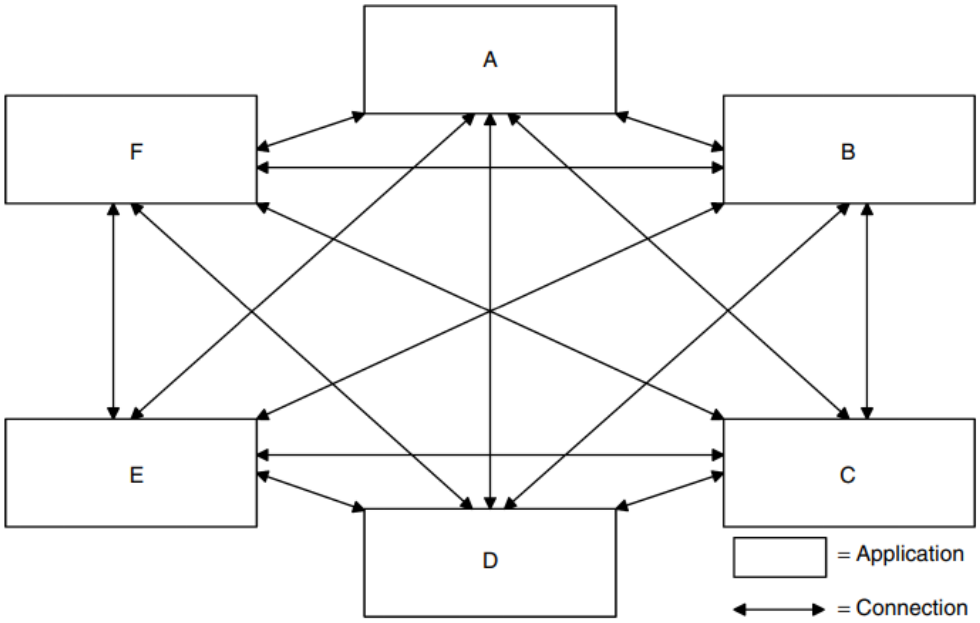

Figura 5 - Esempio di deployment con comunicazione RPC

Il meccanismo RPC necessità dell'implementazione di interfacce per permettere la comunicazione tra i diversi componenti, creando forti accoppiamenti tra i diversi servizi in quanto ogni modifica alle interfacce deve essere propagata ad entrambi i componenti comunicanti. La scalabilità è fortemente limitata dall'interazione sincrona obbligata da RPC, soprattutto in contesti dove i servizi richiedono importanti livelli di throughput. Infine, i nodi non scalano in maniera equa a causa delle operazioni di I/O che bloccano l'esecuzione dei servizi, portando alla luce la necessità di generare nuovi processi per gestire nuove risorse, con conseguente dispendio di risorse [10] [14].

2.3.2 I vantaggi del meccanismo MOM

Il meccanismo di comunicazione MOM, mostrato in Figura 6, offre un sistema di comunicazione centralizzato, in quanto quest'ultimo si fa carico dello smistamento di tutte le richieste. Le interazioni verso il MOM sono asincrone, dunque i servizi non devono preoccuparsi di attendere le risposte, affidando il recapito del messaggio ad un servizio terzo affidabile. Inoltre, l'utilizzo di un broker come intermediario tra i servizi permette di adottare politiche di affidabilità e di persistenza nello scambio dei messaggi, dando la possibilità a due servizi di comunicare in maniera disaccoppiata nello spazio e nel tempo.

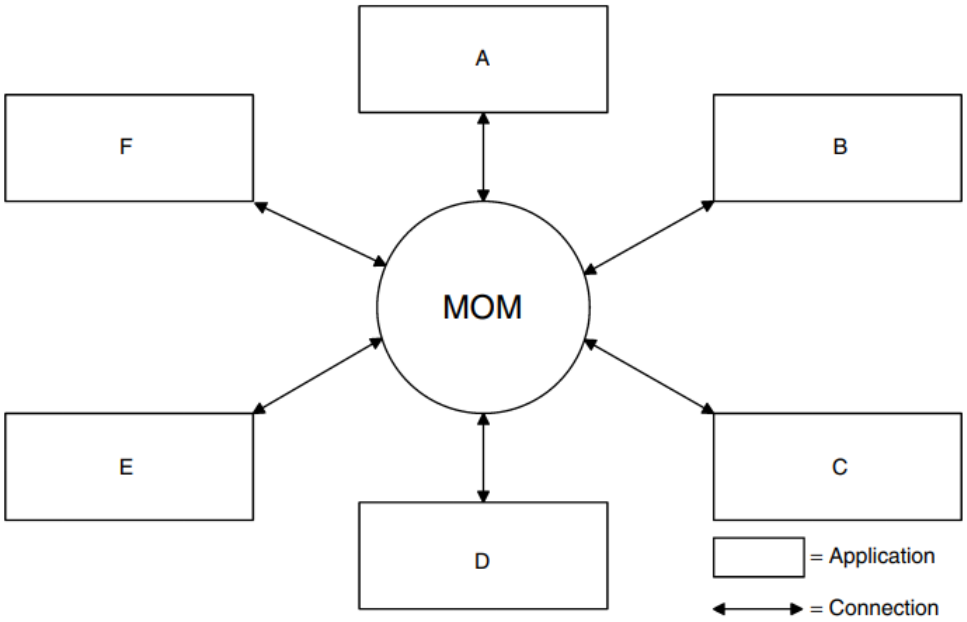

Figura 6 - Esempio di deployment con comunicazione MOM

Al contrario del meccanismo RPC dove, se mittente e destinatario sono correttamente collegati, l'interazione avviene in pochi millisecondi, nel meccanismo MOM è difficile conoscere il tempo di consegna del messaggio, in quanto il destinatario potrebbe non essere disponibile, allungando i tempi di recapito anche per diversi giorni. Il MOM inserisce un nuovo livello di astrazione tra mittente e destinatario, offrendo un nuovo livello di disaccoppiamento non raggiungibile da RPC, necessario per raggiungere i vincoli di scalabilità richiesti da sistemi distribuiti odierni.

Ogni servizio può modificato, scalato ed interrotto senza influenzare minimamente gli altri servizi. Inoltre, il broker MOM permette il dispatching di messaggi secondo politiche di load balancing, evitando di forzare il recapito del messaggio che potrebbe portare al congestionamento del servizio, ma scegliendo di recapitare prima alle applicazioni più scariche. Infine, il broker può essere gestito indipendentemente come ogni altro framework del sistema, ed è possibile replicare il numero di istanze per offrire maggiore tolleranza ai guasti e latenze ridotte [14].

2.3.3 Code

Le code sono un concetto fondamentale nei middleware MOM. Queste permettono il salvataggio dei messaggi nel broker, e rappresentano il punto di raccolta dove i clienti possono inviare e recapitare i messaggi. I messaggi vengono inseriti ed estratti dalla coda

secondo un ordine, dove solitamente la politica adottata di default è la FIFO (First-In First-Out): il primo messaggio inviato è il primo messaggio ad essere estratto. Il deployment delle code richiede la configurazione di un certo numero di parametri: il nome della coda, la dimensione, le soglie relative al carico della coda, o le politiche di ordinamento.

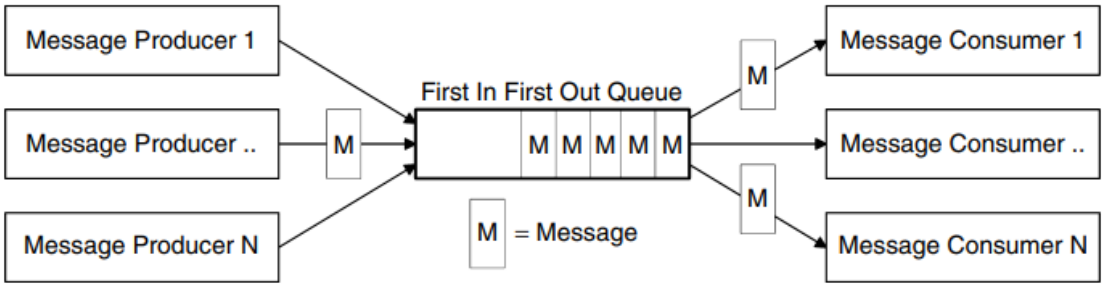

Figura 7 – Esempio di coda con politica FIFO.

In architetture a scambio di messaggi, come mostrato in Figura 7, i mittenti vengono chiamati Producer, ovvero produttori, mentre i destinatari vengono chiamati Consumer, ovvero consumatori. Le code possono essere implementate secondo due modelli di scambio di messaggi: Queue e Topic. Le Queue, o semplicemente code, offrono un'interazione point-to-point, ovvero le applicazioni comunicano grazie al broker del middleware MOM, il quale si occupa di recapitare i messaggi ai servizi interessati, ma soltanto un Consumer riceve il messaggio, nonostante altri Consumer siano in ascolto su quella Queue. Questa politica permette di implementare politiche di load balancing, evitando che lo stesso messaggio venga recapitato a più Consumer, in modo da assegnare singole richieste a singoli consumatori.

Invece, i Topic implementano un modello publish/subscriber, ovvero più produttori pubblicano messaggi nel Topic, ed i messaggi vengono recapitati a i consumatori che si sono iscritti a quel Topic. Il modello publish/subscribe permette a centinaia di migliaia di Consumer di ricevere contemporaneamente lo stesso messaggio, implementando meccanismi importanti di broadcasting nei sistemi distribuiti [14].

3 Il cloud computing

Fra le buzzword più popolari dell'ultimo decennio troviamo quelle riguardanti il cloud computing. Il NIST – dipartimento americano per l'innovazione tecnologica – offre una definizione standard di cloud:

> Il cloud rappresenta l'accesso ad un insieme di risorse condivise, adoperabili con il minimo sforzo *[15]*. (NIST, 2011)

I provider cloud offrono diversi servizi, principalmente divisi per livello di astrazione offerto, ed è possibile scegliere in base alle diverse necessità di controllo ed ottimizzazione che le aziende richiedono. Il cloud offre numerosi benefici, tra cui la riduzione dei costi di amministrazione IT, la maggiore flessibilità nella gestione dei servizi offerti, e le garanzie di qualità offerte dai provider cloud. Per queste ragioni, il mercato dei provider cloud è tra i più floridi in assoluto, con ricavi fino a 182.4 miliardi di dollari nel 2018, ed una crescita stimata del 17.5 percento nel 2019 [16]. Un esempio importante di investimento nel cloud è *Netflix*, che in sette anni ha trasferito la propria infrastruttura su *AWS*, provider cloud di *Amazon*, migliorando l'affidabilità dei propri servizi ed abbattendo i costi di gestione IT [17].

Il NIST standardizza i servizi offerti dai provider cloud, come descritto in Figura 8, in base al livello di astrazione offerto. Ovviamente, è importante sottolineare che da una maggiore astrazione deriva un minor controllo dell'ambiente di esecuzione. I servizi On-Premises sono gestiti completamente dalle aziende, le quali devono farsi carico di ogni aspetto relativo all'infrastruttura del sistema distribuito, mentre gli altri servizi nascondono dettagli implementativi importanti, in quanto i provider cloud garantiscono un certo livello di astrazione: dai servizi IaaS che offrono un'infrastruttura di macchine virtuali (*AWS EC2*) fino ad applicazioni complete SaaS gestite interamente dai provider cloud (*Google Docs, Gmail*).

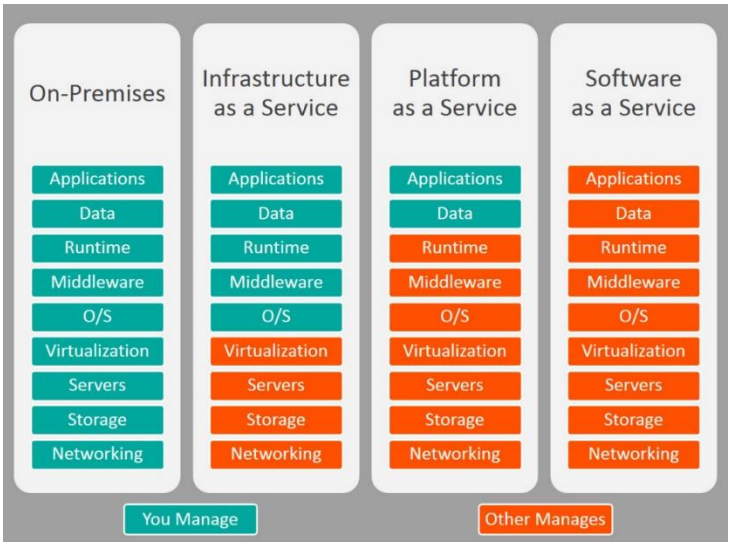

Figura 8 - Modelli di servizio cloud

Il cloud offre la possibilità di eseguire i servizi descritti precedentemente in differenti ambienti di esecuzione. Oltre alle comodità offerte dal cloud, è importante offrire un certo livello di sicurezza, diverso per ogni organizzazione. Per queste ragioni, è possibile effettuare il deployment dei servizi sia su infrastrutture pubbliche condivise da più utenti, come nel caso dei più comuni servizi di *AWS* o *Google Cloud* offerti sul Web, sia in infrastrutture private di proprietà dell'azienda, spesso gestite da enti terzi o dalla società stessa [18].

3.1 Caratteristiche essenziali

Il NIST definisce le caratteristiche essenziali del cloud, elencando cinque aspetti imprescindibili che ogni infrastruttura deve possedere:

- On-demand self-service – il consumatore può accedere a qualsiasi risorsa cloud senza richiedere alcun intervento umano;
- Broad network access – le risorse cloud sono disponibili sulla rete e accessibili a chiunque, grazie all'utilizzo di protocolli standard per permettere la fruizione a qualsiasi client;
- Resource pooling – le risorse sono assegnate secondo politiche di frequenza delle richieste ricevute, e sono condivise a più utenti secondo un modello a multi-tenant. Tipicamente, il consumatore non ha conoscenza della locazione delle risorse, e non possiede alcun controllo sull'allocazione di quest'ultime, anche se esistono astrazioni per definire il data center in cui i servizi vengono eseguiti;

- Rapid elasticity – le risorse possono essere assegnate e rilasciate rapidamente, spesso anche automaticamente, per scalare di fronte a qualsiasi quantità di richieste ricevute; inoltre, per l'utente il cloud presenta una quantità illimitata di risorse, accessibile ovunque ed in qualsiasi momento;
- Measured service – il cloud ottimizza l'utilizzo di risorse grazie alla rilevazione di metriche, che permettono inoltre il monitoring ed il reporting delle risorse consumate [15].

3.2 I modelli di deployment

Il cloud offre tre diversi modelli di esecuzioni:

- Private cloud – le risorse cloud sono di uso esclusivo di una singola organizzazione. L'infrastruttura può essere gestita internamente all'azienda, può essere controllata da organizzazioni di terze parti, o entrambe le cose. Il cloud privato è spesso una scelta per chi possiede un data center interno, quindi ha sviluppato delle conoscenze IT, oppure per le organizzazioni che necessitano di importanti gradi di sicurezza e performance. La scelta di un cloud privato permette di trasformare il vecchio data center in un efficiente dispatcher di servizi, grazie ai vantaggi offerti da automatismi e virtualizzazione;
- Public cloud – i servizi cloud sono accessibili da chiunque. L'infrastruttura cloud viene gestita da un'organizzazione, un provider, un'accademia o una combinazione di queste cose. Il cloud pubblico offre un modello di business, dove un insieme di risorse viene reso disponibile al pubblico, e chiunque può pagare per averne accesso. I provider cloud che gestiscono l'infrastruttura possiedono competenze IT che le organizzazioni fanno fatica a possedere, con conseguente risparmio di risorse, ed offrono sistemi robusti, con risorse computazionali praticamente infinite. In ogni caso, i dati e i processi non sono controllati direttamente dall'organizzazione, suscitando preoccupazioni relative alla sicurezza delle informazioni;
- Hybrid cloud – la struttura del cloud è composta da più ambienti di esecuzione, privati e pubblici. La scelta di cloud ibridi permette di cogliere i vantaggi di entrambe le soluzioni: potenza ed off loading del cloud pubblico, e sicurezza del cloud privato. Nuove soluzioni stanno offrendo astrazioni capaci di connettere trasparentemente i diversi ambienti cloud, garantendo nuovi livelli di portabilità e offrendo nuove opportunità di sviluppo e deployment. Lo stato dell'arte vede le aziende adottare soluzioni ibride, le quali però sono ancora in sviluppo, soprattutto

nell'ambito di standardizzazioni relative la portabilità delle applicazioni sviluppate in cloud [19].

3.3 I modelli di servizio

Come già accennato, i servizi del cloud computing divergono per livello di astrazione offerto:

- Infrastructure-as-a-Service – permettono all'utente di generare, tramite tecniche di virtualizzazione, risorse computazionali basilari, quali virtual machine, storage e networking. I servizi IaaS sono pensati per utenti esperti, a causa delle importanti competenze richieste nell'ambito dei sistemi distribuiti. Con questo tipo di servizi, il customer può realizzare una rete di virtual machine in completa autonomia, avendo la possibilità di scegliere: il sistema operativo, i middleware relativi il deployment del servizio da eseguire, nonché la configurazione relativa al networking presente tra le macchine virtuali;
- Platform-as-a-Service – offrono gli strumenti necessari per lo sviluppo di applicazioni cloud native, in quanto offrono all'utente librerie, linguaggi di programmazione, servizi ed utility pronti all'utilizzo. L'utente non gestisce la piattaforma sottostante, in quanto non possiede i diritti per scegliere il sistema operativo in cui vengono eseguiti i servizi, bensì possiede l'accesso alle API delle librerie disponibili per poter controllare le funzionalità offerte. Tipicamente, i servizi PaaS offrono componenti di supporto, quali database o message broker, per facilitare il cliente nello sviluppo di applicazioni modulari e complesse. Le risorse computazionali sono astratte, e l'utente non ha potere in relazione alle quantità di risorse da allocare, anche se è possibile trovare parametri di configurazione con cui andare ad ottimizzare, anche se limitatamente, l'utilizzo di quest'ultime;
- Software-as-a-Service – rappresentano il più alto livello di astrazione offerto all'utente, in quanto consistono in applicazioni complete e pronte all'utilizzo. L'adozione di queste applicazioni è molto diffusa in ambiente industriale, e la sottoscrizione verso questi servizi richiede spesso il pagamento tramite abbonamento periodico. I SaaS sono i servizi più popolari tra quelli offerti in ambiente cloud, in quanto l'utilizzo non è limitato ad architetti e programmatori, bensì sono accessibili da chiunque, basti pensare ad applicazioni quali *Office 365* di *Microsoft*, oppure *Google Drive* di *Google*. L'intera applicazione è gestita dal provider, e per questa ragione l'utente non deve preoccuparsi di effettuare nessuna

operazione IT, offrendo potere computazionale praticamente illimitato a fronte di costi molto limitati [20].

3.4 Oltre le virtual machine: i container

I servizi IaaS offrono al cliente la possibilità di generare macchine virtuali, lasciando la libertà di scegliere e di configurare il sistema operativo e i middleware adottati da ognuna di queste. Le virtual machine offrono un elevato grado di isolamento, in quanto in ogni macchina viene istanziato un nuovo sistema operativo, con tutto l'overhead che questo comporta. Si può dire che la virtual machine è la completa simulazione di un computer, soprattutto perché ad ogni computer è richiesta la presenza di un sistema operativo capace di comunicare con l'hardware sottostante. In ogni caso, le macchine virtuali sono componenti pesanti, che produco un importante overhead e che richiedono diversi minuti per potersi avviare.

Per queste ragioni, nel corso degli anni si è notata la mancanza di flessibilità offerta da questa tecnologia, e nuove soluzioni sono arrivate in soccorso per superare le limitazioni appena citate. L'alternativa più importante alle macchine virtuali sono i container, ovvero unità di esecuzione, dove all'interno sono presenti codice, dipendenze ed ambiente dell'intera applicazione. La presenza di ogni dipendenza richiesta dal software rende i container estremamente portatili, in quanto l'esecuzione di ognuno di questi non è influenzata in nessun modo dal sistema in cui viene eseguito, né dai componenti con cui interagisce. Un container è un package leggero, standalone, ed eseguibile, e l'overhead generato dal deployment è estremamente meno influente rispetto a quello di una virtual machine. La principale motivazione di questa differenza è dovuta dalla possibilità di istanziare più container sopra lo stesso sistema operativo, in quanto questi necessitano unicamente di eseguire librerie e dipendenze, e non di avvalersi di un sistema operativo personale [21].

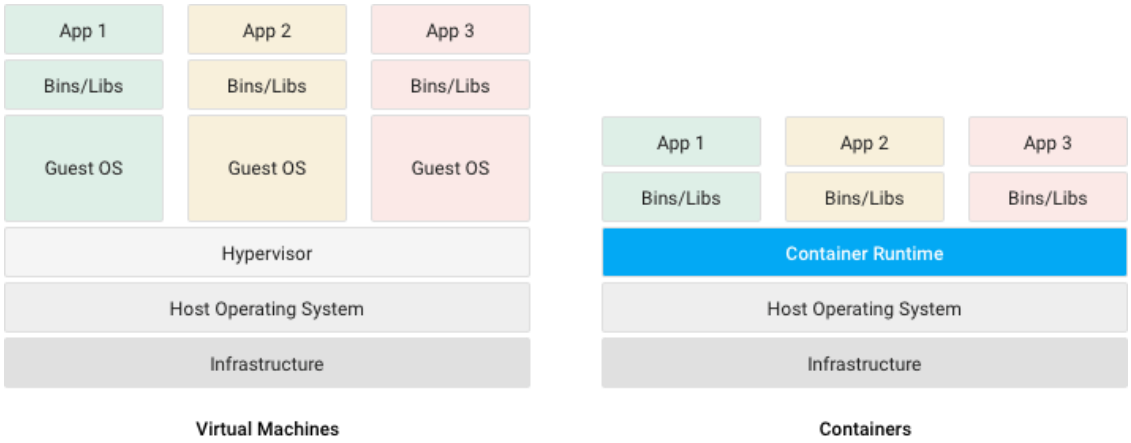

Figura 9 - Differenze tra virtual machine e container.

La tecnologia a container è resa possibile da alcuni moduli presenti nel sistema operativo Linux, che permettono di astrarre una certa quantità di risorse, andando ad isolare i diversi ambienti richiesti dal deployment dei container. I vantaggi derivati dall'adozione di questa tecnologia sono numerosi:

- Distribuzione immediata – è possibile fare il deployment di container in pochi secondi, se non addirittura in millisecondi. Prestazioni così importanti permettono di liberare le risorse nel cloud, e, quando necessario, generare un nuovo container ed offrire il servizio, cosa impossibile da effettuare con le virtual machine;
- Alta disponibilità – i servizi eseguiti su container possono essere replicati con estrema semplicità su più nodi, mentre le macchine virtuali richiedono una quantità di risorse non trascurabili;
- Elevato grado di scalabilità – i container rendono semplice la possibilità di scalare orizzontalmente i servizi, permettendo di generare milioni di container;
- Performance migliori – la granularità offerta dai container rispetto alle virtual machine permette di scalare in maniera più efficiente ed ottimizzata.

Nonostante i numerosi vantaggi presentati dai container, le virtual machine sono una tecnologia ancora apprezzata in scenari industriali, soprattutto per l'isolamento offerto, dato che più container condividono lo stesso sistema operativo, al contrario delle macchine virtuali. Inoltre, alcune applicazioni funzionano meglio con sistemi operativi dedicati, soprattutto nel caso di applicazioni stateful dove è necessario aggregare importanti quantità di dati [22].

3.5 Function-as-a-Service

Le FaaS rappresentano un tipo di servizio cloud che permette all'utilizzatore di eseguire codice in risposta ad eventi, senza la necessità di dover gestire complesse infrastrutture, sforzo tipicamente richiesto durante lo sviluppo e l'esecuzione di applicazioni a microservizi. Con i servizi FaaS: hardware, sistemi operativi, web server ed altri middleware sono gestiti completamente dal provider cloud, permettendo allo sviluppatore di focalizzarsi unicamente sullo sviluppo del codice applicativo.

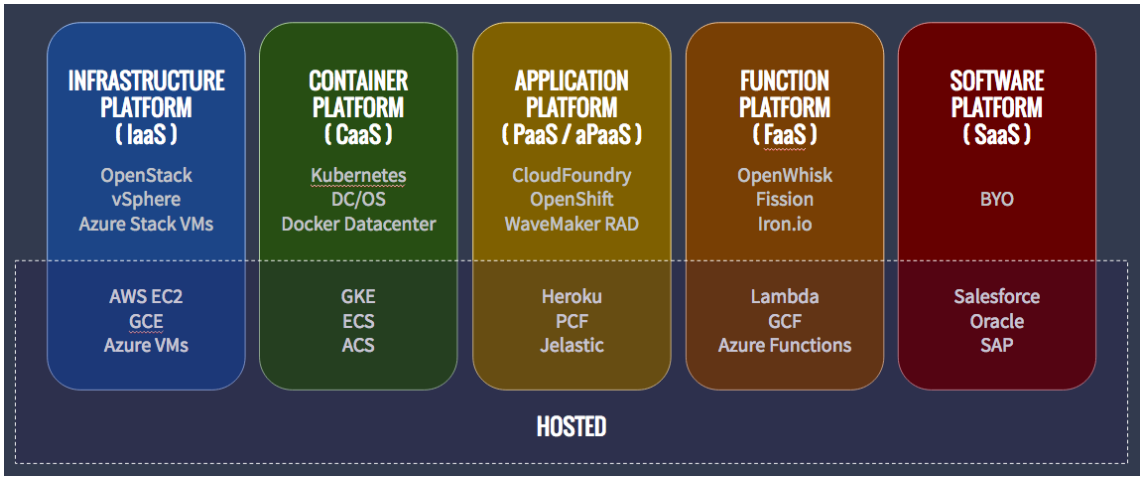

Figura 10 - Modelli di servizio cloud con CaaS e FaaS

L'utilizzo di questi servizi porta importanti benefici, soprattutto relativi al risparmio economico derivante dall'adozione delle funzioni:

- Più codice, meno infrastruttura – con i servizi FaaS, è possibile dividere i server in funzioni, le quali sono scalate automaticamente ed indipendentemente dalla piattaforma, evitando di dover gestire le risorse allocate;
- Addebito costi in base al tempo di esecuzione – il costo dell'adozione di servizi FaaS è calcolato unicamente in base al tempo di esecuzione delle funzioni. Non esiste nessun server in idle da pagare, in quanto la piattaforma genera e termina le funzioni invocate. Per queste ragioni, il costo complessivo derivante da queste soluzioni può essere molto più basso rispetto a quello di altri servizi;
- Scalabilità automatica – i servizi FaaS scalano automaticamente, senza alcun intervento richiesto;
- Affidabilità delle infrastrutture cloud – le funzioni vengono replicate automaticamente tra i diversi data center offerti dal provider cloud, garantendo disponibilità e tolleranza ai guasti.

Le funzioni, per garantire le massime prestazioni, dovrebbero essere leggere e limitate, in modo da generarsi ed eseguirsi nel minor tempo possibile. Infatti, l'idea generale alla base dell'adozione di queste soluzioni è quella di migrare i propri servizi in funzioni isolate, leggere ed indipendenti, per offrire gradi di scalabilità non raggiungibili dalle soluzioni a microservizi.

Le FaaS sono soluzioni adatte principalmente per grandi carichi di lavoro e per algoritmi massicciamente paralleli, data la facilità di questi servizi di generare migliaia di repliche in pochi secondi. Questi servizi possono essere utilizzati per sviluppare servizi di backend mantendo i costi relativamente bassi, ad esempio per lo sviluppo di servizi per stream e data processing, o soluzioni per dispositivi IoT [23]. Infatti, *AWS Lambda*, servizio FaaS di *AWS*, e *IBM Cloud Functions*, servizio FaaS di *IBM*, addebitano i costi di esecuzione delle funzioni ogni cento millisecondi, permettendo notevoli risparmi rispetto ai servizi basati su virtual machine o a semplici container [24] [25].

Le funzioni possono incrementare significativamente le prestazioni di un'applicazione. Per esempio, distribuendo il carico su funzioni, due studenti, insieme ad ingegneri della *IBM*, sono riusciti ad eseguire una simulazione di Monte Carlo in 90 secondi invocando circa 1000 funzioni in parallelo. Lo stesso problema, in locale, avrebbe richiesto su un singolo portatile circa quattro ore ed il completo utilizzo della CPU [23].

3.6 Svantaggi e limitazioni

Le organizzazioni possono incontrare diverse difficoltà quando migrano i propri servizi on-premise su infrastrutture cloud. La scelta di adottare soluzioni cloud è vantaggiosa dal punto di vista economico, dati i risparmi conseguenti all'abbandono di infrastrutture auto gestite, ma presenta svantaggi per quanto concerne la sicurezza, l'affidabilità e la qualità dei servizi richiesti dai provider cloud.

Il primo aspetto riguarda la sicurezza e la privacy, in quanto le informazioni vengono affidate a provider esterni, e queste vengono diffuse e replicate in nodi sparsi per la rete. I cloud pubblici condividono le proprie risorse con un numero indefinito di utenti, mettendo a rischio dati sensibili se vengono scoperte falle nell'infrastruttura da parte di utenti che hanno accesso ai servizi. La scelta di adottare cloud pubblici è naturale in piccole organizzazioni, dove mancano le risorse per mettere in sicurezza un'infrastruttura cloud privata, mentre, in grandi organizzazioni, i dati sensibili vengono spesso elaborati in cloud privati, evitando di mantenere le informazioni più importanti all'esterno del perimetro aziendale.

Un altro problema che sorge dall'adozione di una specifica tecnologia cloud è quella del vendor lock-in, ovvero l'impossibilità di trasferire i propri servizi in altre tecnologie senza impiegare importanti risorse, come eseguire il completo refactoring del codice implementativo dei servizi. Ogni vendor utilizza software e hardware differenti, e manca una standardizzazione per la portabilità delle applicazioni e dei dati mantenuti tra i diversi cloud provider. Per esempio, *Google* mantiene i propri dati sul framework *BigTable*, mentre *Facebook* li mantiene sulla piattaforma *Cassandra*, e *Amazon* utilizza *Dynamo*. La mancanza di interfacce obbliga l'utente a fidarsi completamente del provider, nonostante questo possa perdere valenza tecnologica nel tempo, o nuove soluzioni più allettanti vengano lanciate sul mercato.

La disponibilità dei servizi è un punto cruciale nella scelta del cloud provider. Esistono organizzazioni che richiedono una disponibilità 24/7 dei propri servizi, e non accettano guasti all'infrastruttura. I provider, nonostante le ingenti risorse a disposizione, non possono evitare di mandare in down le loro infrastrutture, nonostante questo accada per poche ore nel corso dell'intero anno. Per far fronte alla paura delle organizzazioni nei confronti di possibili guasti, si definiscono clausole di qualità del servizio, chiamate Service Level Agreement (SLA), che offrono tempi e misure garantite dal provider verso i servizi offerti, e che, se violati, possono essere strumento di denuncia e di risarcimento monetario. Per esempio, i maggiori provider cloud offrono up-time garantito del 99,99% l'anno, che corrisponde a dieci minuti di downtime massimo in ben dodici mesi di servizio eseguito.

Altri svantaggi sono legati all'instabilità infrastrutturale dovuta all'importante numero di utenti che, ogni giorno, utilizza i servizi cloud, portando a possibili picchi di carico con conseguenti colli di bottiglia nelle performance. Inoltre, fattore importante che verrà approfondito successivamente, è la latenza, in quanto i diversi data center cloud sono pochi e, spesso, distanti dalla località in cui viene lanciata la richiesta. La distanza porta ritardi considerevoli, e, in applicazioni real time, sono inaccettabili, dunque sono richieste soluzioni alternative per far fronte agli stringenti vincoli di tempo imposti [26].

4 Serverless

Il primo utilizzo di questo termine è comparso nel 2012, in un articolo sul futuro delle applicazioni, scritto da Ken Fromm. Solo nel 2015 il termine diventa popolare, grazie al lancio sul mercato di due soluzioni serverless: *AWS Lambda* di Amazon nel 2014 e *AWS API Gateway* nel luglio 2015. A metà del 2016, il termine serverless è già una buzzword, visibile dalla nascita di conferenze dedicate a questo nuovo tipo di modello, e all'adozione del termine da parte di molte soluzioni offerte dai maggiori provider cloud presenti sul mercato [27].

Ad oggi, diverse definizioni sono state date relativamente al serverless, ognuna delle quali evidenzia caratteristiche fondamentale di questa architettura.

> *A Serverless solution is one that costs you nothing to run if nobody is using it [28]*. (Johnston, 2017)

> *Roughly speaking [Serverless is] it's an event driven, utility based, stateless, code execution environment in which you write code and consume services [29]*. (Wardley, 2018)

> *[Serverless means] managed services that scale to zero [30]*. (Stephens, 2018)

Le definizioni di serverless riportate descrivono alcuni punti fondamentali nella distinzione tra un'architettura serverless ed una serverful. In particolare, si citano event driver, stateless, e scale to zero, caratteristiche fondamentali che contraddistinguono questo tipo di soluzioni. La Tabella 1 riassume le principali differenze tra architetture serverless ed architetture serverful.

Serverless	Serverful
Applicazioni eseguite soltano quando invocate	Applicazioni continuamente in esecuzione
Stato mantenuto in storage (stateless)	Stato mantenuto ovunque (stateful o stateless)
Massima memoria utilizzabile ~3GB	Massima memoria utilizzabile >10TB

Massimo tempo di esecuzione in minuti	Nessun limite al tempo di esecuzione
Sistema operativo e macchine scelte dal provider	Sistema operativo e istanze scelte dall'utente
Scaling effettuato dal provider	Scaling effettuato dall'utente

Tabella 1 - Differenze tra applicazioni serverless e serverful [31]

Date le caratteristiche del serverless, è necessario sviluppare applicazioni tenendo in considerazione i vincoli imposti da questo modello. Per queste ragioni, di seguito vengono riportati i principi fondamentali del serverless, a cui tutti gli sviluppatori devono attenersi nello sviluppo di architetture di questo tipo:

- Scrivere codice come funzioni – le funzioni sono piccoli blocchi di codice con un unico scopo, eseguibili dinamicamente. Le funzioni sono definibili tramite servizi cloud chiamati FaaS, che richiedono all'utente solamente la scrittura del codice e la definizione degli eventi per scatenarlo. Nonostante molti scambino i termini "serverless" e "function", i due non corrispondono alla stessa cosa, infatti le funzioni rappresentano la dimensione computazionale delle applicazioni serverless, ma è possibile scrivere soluzioni serverless senza eseguire funzioni. Inoltre, è possibile eseguire funzioni su piattaforme non serverless.
- Utilizzare architetture event-driven – i sistemi event-driven eseguono il proprio codice in risposta ad eventi. Ogni evento rappresenta un cambiamento di stato interno al sistema, come la ricezione di un messaggio, il completamento di un download, oppure l'inserimento di un record nel database. Tipicamente, le funzioni vengono scritte per rispondere ad eventi, infatti le soluzioni FaaS spesso integrano connettori verso applicazioni di message broker e di storage.
- Connettere più servizi serverless insieme – a causa della natura stateless delle funzioni, tutti gli stati e le configurazioni necessarie alle function devono essere salvati in servizi terzi, quali database, message broker, o API gateway. In questo caso, i servizi devono essere tutti serverless, in modo da rendere l'intera infrastruttura trasparente allo sviluppatore, ed utilizzare le funzioni come collante tra i diversi servizi serverless offerti dal provider.
- Scale-to-zero – questo è un concetto fondamentale del serverless. Ogni servizio, quando non richiesto, libera tutte le risorse occupate, al contrario di soluzioni stateful dove i server restano sempre in ascolto di nuove richieste. La piattaforma, ricevuta una nuova richiesta, si fa carico di ripristinare istanze del servizio per gestire correttamente la richiesta [32].

Un paper della Berkley definisce il serverless come:

Put simply, serverless computing = FaaS + BaaS [31] (Berkley, 2019)

Questa definizione rispecchia i principi elencati precedentemente, dove per BaaS (Backend-as-a-Service) si intendono tutti i servizi cloud serverless, quali database o message broker, e per FaaS tutte le piattaforme che permettono l'esecuzione di funzioni, le quali rappresentano il core di ogni architettura serverless.

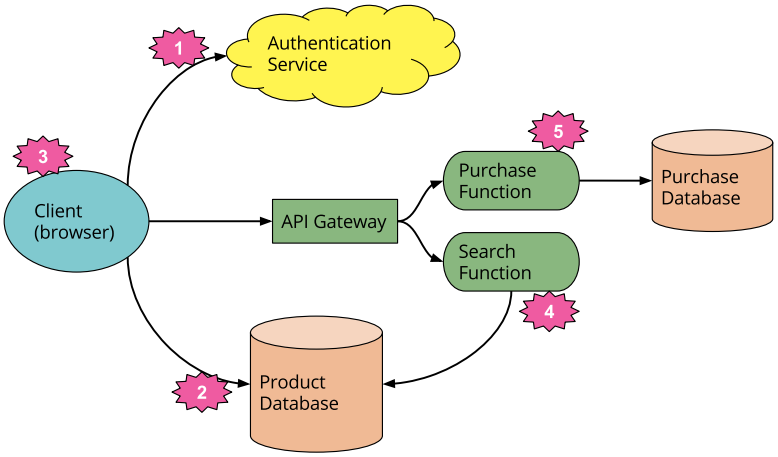

Figura 11 – Esempio di architettura serverless

In Figura 11, le funzioni descrivono le operazioni di vendita e ricerca dei prodotti, e vengono invocate direttamente dall'API Gateway. Per permettere la vendita dei prodotti, è presente un sistema di autenticazione, ed ogni funzione comunica direttamente con i database per salvare nuove vendite e ricercare i prodotti richiesti. I database, data la natura serverless delle funzioni, sono necessari per mantenere lo stato in ogni architettura di questo tipo. Tutti i servizi descritti, a parte le funzioni, sono serverless.

Il tempo di creazione ed esecuzione di una funzione è un argomento importante nella valutazione di un framework FaaS. Ogni funzione è diversa, e conseguentemente è diverso il tempo richiesto per generare un'istanza e servire la richiesta. Questa differenza è dovuta a numerosi fattori, e possono cambiare la latenza di avvio da pochi millisecondi a diversi secondi [27]. Inoltre, lo scale-to-zero, caratteristica fondamentale di tutte le piattaforme serverless, oltre ai vantaggi riportati precedentemente, impone tempi di latenza nella creazione e nella distruzione delle istanze del servizio. Questo problema non è presente nelle soluzioni stateful, in quanto i server sono sempre in esecuzione ed in

attesa delle richieste da elaborare. Per queste ragioni, i tempi di avviamento delle istanze sono argomento di grande interesse e ricerca per le piattaforme serverless.

Figura 12 - Lifecycle di una funzione

In caso di istanze eseguite in container, in Figura 12 sono mostrate le fasi del ciclo di vita di una funzione, la quale, prima dell'effettiva esecuzione del codice, necessita di recuperare il materiale necessario per permettere alla funzione di gestire la richiesta. Se ogni istanza della funzione è stata terminata, allora è necessario pagare l'overhead del cold start, dove la funzione deve riscaricare il codice, istanziare un nuovo container ed avviare l'ambiente di esecuzione, e, infine, eseguire la funzione. Altrimenti, se è passato poco tempo tra gli eventi ricevuti, è possibile avere ambienti di esecuzione già pronti, e l'unico overhead da pagare è quello del warm start, che consiste nella semplice esecuzione del codice [33].

Per garantire un buon trade-off tra startup latency e costi di idle time (costi dovuti all'esecuzione di container in attesa di invocazioni), i provider FaaS pubblici mantengono le funzioni in warm state per alcuni minuti. Per esempio, *AWS Lambda* mantiene le function in warm state per circa dieci minuti dopo la prima invocazione in cold state, *Azure Functions* per circa venti minuti mentre *Google Cloud Functions* adotta strategie più sofisticate, basate sulla frequenza di invocazioni verso il servizio in un determinato intervallo di tempo, mantenendo le funzioni in warm state dai tre minuti alle cinque ore o più [34].

La differenza tra i diversi cold start è dovuta principalmente alla scelta del linguaggio di programmazione: un linguaggio di programmazione compilato comporta un significativo overhead, dovuto anche all'istanziazione dell'ambiente di esecuzione, come nel caso di Java con la JVM. Anche la dimensione del package, compresa di relative dipendenze, favorisce l'aumento della latenza in quanto le librerie devono essere caricate prima dell'esecuzione della funzione. Infine, l'aumento delle risorse allocate ai container accelera i tempi di avviamento delle funzioni, soprattutto nel setup di ambienti virtuali come la JVM [34] [35].

L’adozione di tecnologie serverless garantisce all’azienda più tempo per lo sviluppo e il testing di nuove applicazioni, sollevando gli sviluppatori dalla gestione dei nodi in produzione. Inoltre, le operazioni offerte dai servizi serverless vengono addebitate al cliente in base al tempo di esecuzione o al throughput generato, e non più in base alle risorse allocate [36]. Adam Pash, ingegnere informatico presso Postlight, scrive di aver migrato i propri servizi verso tecnologie serverless, con un risparmio pari a due ordini di grandezza: 39 milioni di richieste servite al mese al costo di 370 dollari invece di 10000 [37].

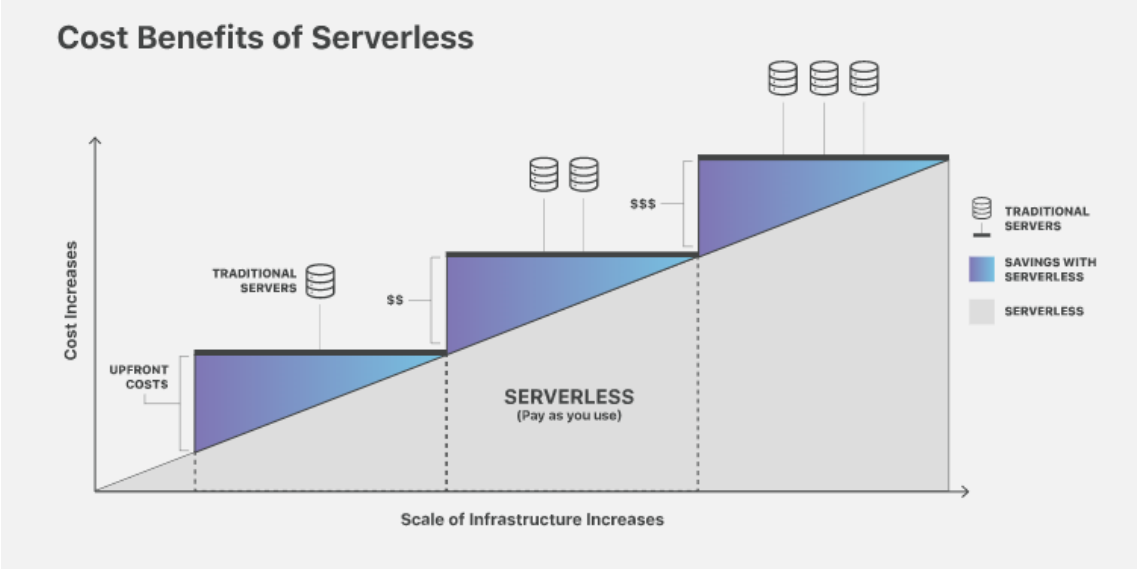

Figura 13 - Incremento dei costi all'aumentare delle risorse presenti nell'infrastruttura

In Figura 13 è evidente il vantaggio economico offerto dalle soluzioni serverless rispetto a quelle a microservizi. Il pagamento delle risorse è adattivo, in quanto si paga unicamente in base all’utilizzo delle risorse, e non in base al tempo di allocazione. Questo permette di risparmiare sul tempo di inattività delle risorse, garantendo importanti risparmi economici a grandi organizzazioni che migrano i propri servizi verso soluzioni serverless. Inoltre, il risparmio economico è accentuato dalla riduzione di sforzi operazionali per il deployment di nuovi servizi, automatizzato dalla piattaforma serverless, e permette agli sviluppatori di focalizzarsi sullo sviluppo ed il testing dell’applicazione.

Oltre al risparmio derivante dall’adozione di soluzioni serverless, un aspetto fondamentale è la possibilità di lanciare sul mercato nuovi servizi in pochi istanti. Per il rilascio di un nuovo servizio non è richiesto nessun intervento operazionale, offrendo nuovi gradi di libertà nello sviluppo di applicazioni cloud. Ogni servizio può essere

creato, aggiornato ed eliminato indipendentemente dagli altri, ed ognuno di questi è sempre pronto a ricevere un qualsiasi numero di richieste, adattandosi ad ogni scenario senza alcuno spreco di risorse.

Nonostante i benefici descritti precedentemente, il serverless presenta un insieme di limitazioni che minano le possibilità di adozione verso questo tipo di architettura. Il primo problema riguarda la difficoltà di debugging, presente a causa dell'astrazione offerta verso il backend, in quanto manca la visibilità dei processi in esecuzione sui server. Inoltre, la sicurezza rappresenta una grande preoccupazione, in quanto i provider sono padroni dell'esecuzione di ogni istanza, le quali possono essere eseguite su server in cui sono eseguite istanze di altri servizi. In caso di vulnerabilità, i servizi possono accedere a dati sensibili di altri utenti, compromettendo la sicurezza dell'intero sistema. Infine, una differenza sostanziale rispetto ai microservizi è che in idle non sono presenti istanze pronte ad essere eseguite, il che comporta tempi di avviamento non trascurabili, soprattutto se le richieste sono distanti tra loro [38].

4.1 Stato dell'arte delle piattaforme serverless

Da quando *Amazon* nel 2014 ha presentato *AWS Lambda* come primo servizio FaaS in commercio, gli investimenti verso tecnologie di questo tipo hanno continuato a crescere in maniera costante, complice anche la competizione supportata da compagnie quali *Google*, *Microsoft* e *IBM* [39]. Nonostante i notevoli sforzi impiegati in questa tecnologia, sono ancora molti i difetti presenti che non permettono di fidarsi completamente di questi servizi. A tal proposito, un paper della UC Berkley ha analizzato alcune delle limitazioni presenti nei principali framework FaaS, prendendo come riferimento il servizio *AWS Lambda*:

- Ciclo di vita limitato - le funzioni *Lambda* sono disattivate dopo 15 minuti di inattività, soffrendo di un'evidente latenza in scenari dove le funzioni non sono continuamente invocate;
- Collo di bottiglia I/O - la natura stateless delle funzioni obbliga lo sviluppatore all'utilizzo di BaaS per lo storage e la sincronizzazione tra i diversi componenti in gioco, con evidenti colli di bottiglia nel trasferimento dei dati: i risultati hanno dimostrato come il throughput raggiunto da una singola funzione *Lambda* raggiunga circa 538Mbps, un ordine di grandezza inferiore rispetto alla velocità offerta da un singolo SSD;
- Lenta comunicazione tra funzioni - due funzioni *Lambda* possono comunicare unicamente tramite un servizio intermedio (BaaS), estremamente più lento di una comunicazione diretta point-to-point;

- Hardware non ottimizzato - i middleware FaaS offrono la possibilità di gestire la quantità di vCPU e memoria richiesta dalle funzioni, ma non esiste alcun meccanismo per sfruttare intelligentemente l'hardware sottostante [40].

Il paper termina con alcune proposte relative ai prossimi sviluppi per le architetture FaaS, e raccoglie idee per possibili nuovi scenari applicativi.

Una collaborazione tra le università del Wisconsin-Madison, dell'Ohio e della Cornell Tech di New York offre una profonda analisi delle principali tecnologie FaaS presenti sul mercato: *AWS Lambda* [41], *Azure Functions* [42] e *Google Cloud Functions* [43]. Le sperimentazioni condotte cercano di rilevare le risorse utilizzate dai middleware FaaS, rilevabili grazie al proc filesystem di Linux (procfs), eseguendo cinquantamila funzioni ed analizzando il comportamento risultante. La piattaforma *Amazon* mostra una marcia in più, grazie anche al maggior tempo di sviluppo del prodotto, raggiungendo tempi di latenza minori e un ottimo livello di isolamento dovuto alla strategia di eseguire virtual machine dedicate per utente. La scalabilità ottenuta da *Lambda* risulta essere estremamente più soddisfacente rispetto a *Google Cloud Functions* e *Azure Functions*, dove la prima riesce a generare un numero di istanze dieci volte superiore rispetto ai suoi competitor. I tempi di cold start analizzati premiano ancora *Amazon*, dove i valori mediani riportano 265.21 millisecondi per *Lambda*, 493.04 millisecondi per il FaaS di *Google* e ben 3640.02 millisecondi per *Microsoft*, complici alcuni bug a cui la casa di Redmond sta lavorando. I risultati ottenuti derivano da un'osservazione indiretta degli ambienti e dunque non permettono una valutazione completamente oggettiva ed una comparazione precisa [44].

Il lavoro svolto da un gruppo di ricercatori del *Trinity College Dublin* è simile a quello svolto nel paragrafo precedente, ma compara piattaforme FaaS open source. Le piattaforme selezionate sono *Apache OpenWhisk*, *OpenFaaS*, *Knative* e *Kubeless*: le piattaforme più popolari presenti sul repository di codici open source *GitHub* [45]. Il paper analizza questi middleware in scenari di edge computing ed IoT, con un grado di concorrenza di venti richieste ricevute simultaneamente. I valori riportati dalle sperimentazioni mostrano come *Kubeless* sia la piattaforma migliore con un throughput di 60 transazione al secondo (tps), percentuale di errore nullo e tempo di attesa medio di circa cento millisecondi. *OpenFaas* e *Knative* seguono la classifica con risultati leggermente inferiori, con circa 50 transazioni al secondo, nessun errore conseguito e tempi di attesa di circa cento millisecondi. Infine, *OpenWhisk* chiude la classifica con 40 transazioni al secondo, tempo di risposta di circa cento millisecondi, ed una percentuale di successo del solo 10 percento [46].

4.2 Principali piattaforme serverless

Come visto negli articoli correlati, le principali piattaforme FaaS di provider cloud pubblici disponibili sul mercato sono *AWS Lambda*, *Google Cloud Functions* e *Azure Functions*. L'affidabilità di queste soluzioni è garantita grazie alle infrastrutture proprietarie che offrono livelli di affidabilità difficilmente raggiungibili da soluzioni on-premise. Nonostante questo, le tecnologie proprietarie soffrono di difetti importanti, quali l'impossibilità di analizzare il codice sorgente, l'obbligo di appoggiarsi verso infrastrutture esterne con la paura di esporre i propri dati sensibili, e i costi derivanti da una soluzione cloud centralizzata. Le soluzioni open source, al contrario, offrono trasparenza, e permettono il deployment in qualsiasi ambiente: edge computing, hybrid cloud, private cloud, public cloud. I vantaggi relativi all'adozione di un middleware open source sono numerosi, e sono una scelta obbligata in contesti sensibili, dove si vuole avere la certezza del software che si esegue in produzione.

Ultimamente, i framework FaaS più gettonati risultano essere: *OpenWhisk* [47], *OpenFaas* [48], *Knative* [49], *Fission* [50], *Kubeless* [51], *Nuclio* [52] e *OpenLambda* [53]. Il criterio nella scelta dei middleware per il confronto di questa tesi è basato sul supporto della comunità al progetto, potendo contare il numero di contributor e star che GitHub rende pubblici per ogni repository.

Framework	Contributors	Stars
Apache OpenWhisk	**168**	**4386**
Knative	**152**	**2453**
OpenFaaS	**128**	**16099**
Fission	**90**	**4754**
Kubeless	82	5225
Nuclio	49	3036
OpenLambda	19	643

Figura 14 - contributor e star di ogni progetto su GitHub (aggiornata al 16/11/2019)

La Figura 14 evidenzia il numero di contributor come parametro principale, e, in caso di pareggio, vengono prese in considerazione il numero di star ottenute, dove per numero di star si intende gli utenti che hanno salvato la repository come preferita. La tabella riporta *OpenWhisk*, *Knative*, *OpenFaaS* e *Fission* come vincitori, per cui le sperimentazioni saranno eseguite su questi framework.

Tutte le piattaforme elencate in Figura 14 eseguono i propri componenti in container, data la portabilità e la versatilità che questa tecnologia offre. Per queste motivazioni, è sempre raccomandabile eseguire le piattaforme FaaS su orchestratori di container, capaci di semplificare il management di complesse applicazioni.

4.2.1 Orchestratori

I framework descritti necessitano di un ambiente di esecuzione capace di controllare la grande quantità di richieste che una piattaforma FaaS può ricevere durante il suo ciclo di vita. Alcuni dei framework citati, quali OpenFaaS ed OpenWhisk, permettono il deployment su più piattaforme, garantendo un buon grado di portabilità tra i diversi ambienti disponibili. Al contrario, Knative e Fission sono vincolati ad essere eseguiti su Kubernetes, offrendo una maggiore integrazione con l'orchestratore a scapito di una minore possibilità di scelta relativa all'ambiente di esecuzione. Di seguito sono riportati due orchestratori, scelti sia per l'importante successo commerciale conseguito, sia per analizzare come Kubernetes sia diventato l'orchestratore di riferimento in tutti i settori economici.

Docker Swarm

Docker Swarm è l'orchestratore di container proprietario di Docker e, in quanto tale, si occupa di gestire container eseguiti in un cluster di nodi distribuiti. L'orchestratore di Docker si occupa di tutti task operazionali richiesti nella gestione di microservizi, permettendo di gestire il numero di repliche del servizio in base al carico di lavoro richiesto, offrendo un'infrastruttura di rete capace di mettere in comunicazione i diversi servizi ospitati in nodi differenti, e garantendo un buon livello di sicurezza e monitoring dell'intero cluster. Inoltre, in caso di failover, l'orchestratore si occupa di ripristinare i servizi non più disponibili a causa di errori o guasti improvvisi [54].

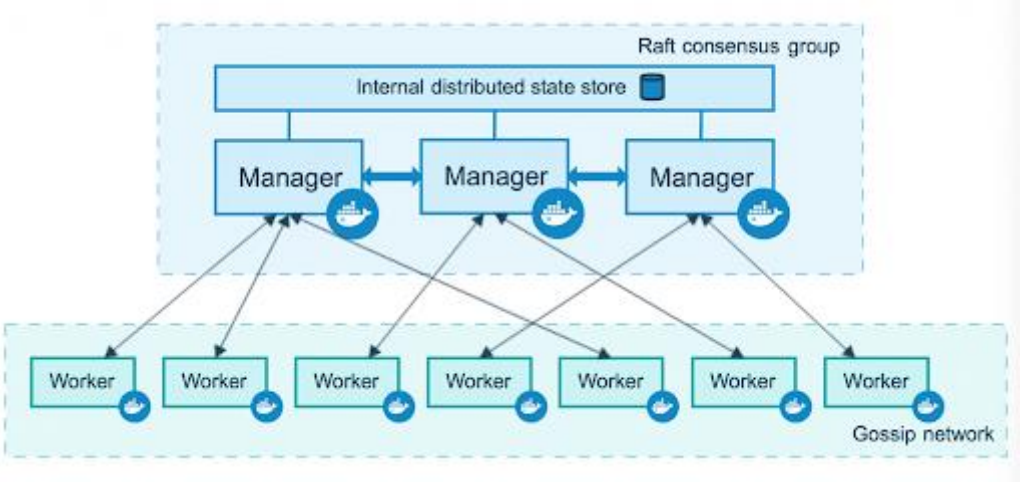

Figura 15 - Architettura di Docker Swarm

L'architettura di Docker Swarm, visibile in Figura 15, presenta una topologia master slave. Un cluster può possedere più Manager, ovvero i master dell'architettura, ma è presente sempre e solo un leader, scelto in base ad un algoritmo di elezione (algoritmo Raft) presieduto da tutti i nodi Manager, il quale gestisce tutti i task dell'orchestratore. I nodi Worker ricevono il lavoro da eseguire direttamente dal leader Manager, il quale, oltre a decidere a quale Worker assegnare la richiesta, mantiene stabile lo stato del cluster [55].

Kubernetes

Kubernetes è un orchestratore per container, sviluppato da Google, ed attualmente di proprietà della CNCF (Cloud Native Computing Foundation) come progetto open source, è attualmente la soluzione più utilizzata a livello industriale. Questo orchestratore è un framework complesso, adatto per la gestione di cluster di medie e grandi dimensioni, in quanto capace di gestire normalmente migliaia di nodi. Anche Kubernetes, come Docker Swarm, presenta un'architettura master slave, aggiungendo diversi livelli di astrazione tra la dimensione dei container e l'effettivo controllo sul cluster [56].

Per l'esecuzione e il controllo di Kubernetes, è necessario lavorare con tre componenti fondamentali:

- Kubeadm – comando per inizializzare automaticamente un cluster secondo le best practice richieste da Kubernetes. Il nodo che inizializza il cluster con kubeadm è anche il master del cluster, mentre gli altri nodi possono utilizzare kubeadm per unirsi come worker;

- Kubelet – agente di Kubernetes per l'interazione con gli altri nodi;
- Kubectl – CLI per l'interazione tra utente e orchestratore.

Tutte le risorse offerte dall'orchestratore sono descritte da file di configurazione in formato YAML, da cui è possibile constatare il tipo di risorsa, le caratteristiche e le richieste necessarie per la corretta esecuzione di ognuna di queste.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-web-server
  labels:
    app: web
spec:
  containers:
    - name: front-end
      image: nginx
      ports:
        - containerPort: 80
```

Figura 16 - Esempio di risorsa Pod per l'esecuzione del reserve proxy NGINX

Il componente base dell'infrastruttura di Kubernetes è il Pod, il quale rappresenta l'insieme di uno o più container strettamente connessi tra loro. Nella descrizione del Pod, in Figura 16, è possibile vedere i container che lo popolano, in particolare è possibile leggere l'immagine Docker richiesta, scaricabile da un registro Docker locale oppure da un registro Docker pubblico, come il Docker Hub. Ogni container possiede porte con cui interagire, e la descrizione dei Pod ne richiede la definizione esplicita per permettere a Kubernetes di comunicare con servizi specifici presenti all'interno dei container. Per permettere il deployment di componenti più complessi, Kubernetes offre la risorsa Deployment che, oltre a descrivere i Pod, offre la possibilità di stabilire il numero di repliche in esecuzione, e permette di gestire lo staged roll out per il rilascio di nuove versioni. Lo staged roll out, in fase di deployment di nuove versioni, garantisce che il traffico in ingresso non venga perduto, mantenendo in memoria le richieste ricevute mentre vengono istanziate le nuove versioni del servizio.

Altre risorse importanti, presenti in Kubernetes, sono i Volume, che rappresentano storage a cui i Pod possono accedere per leggere e scrivere dati; i Secret, risorse per il mantenimento di informazioni riservate, cifrando, per esempio, username e password; ed i Namespace, simili ai namespace offerti dal sistema operativo Linux, per isolare le risorse all'interno dello stesso sistema e garantire la stabilità.

Le risorse presenti in Kubernetes possono essere eseguite grazie all'utilizzo di package manager, che semplificano la gestione di più risorse in un unico package. Helm [57] è un gestore di pacchetti per Kubernetes, capace di istanziare automaticamente servizi complessi, senza la necessità di dover installare a mano ogni singolo componente richiesto da un determinato deployment. Le applicazioni disponibili su Helm sono distribuite sotto forma di Chart, già pronti per essere eseguiti sul cluster. Per installare un Chart è necessario scrivere sul terminale:

```
helm install [NAME] [CHART_URL]
```

I Pod rappresentano l'ambiente di esecuzione dei componenti eseguiti nell'orchestratore, ed è possibile generare interfacce che permettono l'astrazione tra i servizi ed i Pod effettivamente in esecuzione. Queste interfacce vengono chiamate Service e permettono di connettere più endpoint che rappresentano lo stesso servizio. Kubernetes offre diverse strategie per la gestione del traffico in ingresso ed in uscita all'orchestratore. I meccanismi di accesso alle risorse sono definiti dal tipo di Service eseguito, ed ogni Service offre politiche differenti per la ricezione delle richieste. La prima tipologia di Service analizzata è il ClusterIP, che permette la comunicazione al servizio soltanto ai componenti eseguiti nel cluster Kubernetes, ed è dunque adatta per componenti che non richiedono accessi esterni, ad esempio clienti connessi ad Internet. Il primo meccanismo che permette comunicazioni esterne al cluster è il NodePort, il quale apre una porta raggiungibile all'IP di uno dei nodi del cluster, e permette l'interazione con un certo servizio.

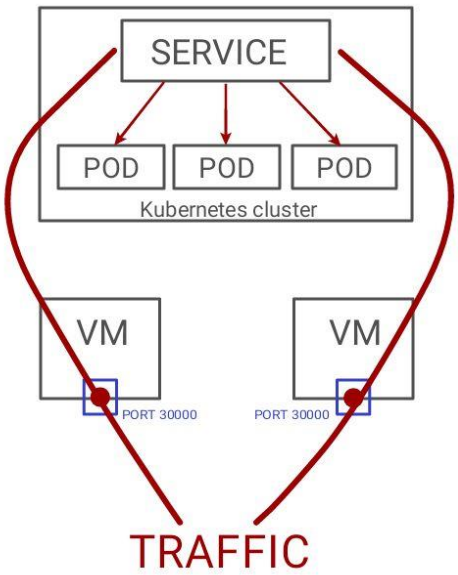

Figura 17 - Workflow del traffico in presenza di NodePort

In Figura 17 la porta offerta dal servizio NodePort è collegata direttamente alla porta del servizio, nel senso che la connessione alla porta 30000 inoltra le richieste alla porta 8080, per esempio nel caso in cui sia ospitato un Web server. Adottare la politica dei NodePort non è consigliato in produzione, in quanto non offre un sistema sicuro per l'interazione con il cluster, complice la mancanza di meccanismi di sicurezza ed autenticazione, oltre che alla limitazione del numero di porte disponibili per nodo. Inoltre, gli IP con cui accedere alle macchine sono relativi alla rete locale in cui è eseguito il cluster, e, a meno di meccanismi di port-forwarding, non è possibili accedere ai servizi esternamente alla rete LAN.

Un altro servizio offerto è LoadBalancer, che offre ad ogni servizio un nuovo IP con cui raggiungerlo. Ad ogni servizio, come mostrato in Figura 18, è associato un IP, e non più una porta, permettendo di utilizzare IP pubblici per raggiungere i servizi anche da Internet, aumentando l'accessibilità dell'intero sistema Kubernetes. Lo svantaggio di LoadBalancer è proprio la necessità di affidare un nuovo IP ad ogni servizio che si vuole rendere pubblico, in quanto si è spesso limitati dalla quantità di IP disponibili [58].

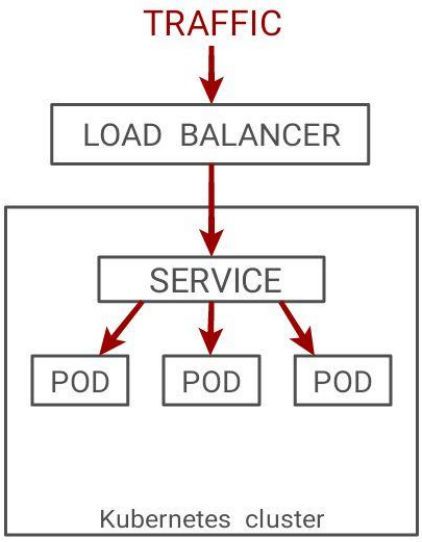

Figura 18 - Workflow del traffico in presenza di LoadBalancer

Per fronte a questa limitazione, è nata la necessità di centralizzare il servizio di LoadBalancer in API Gateway che permettono di smistare internamente i servizi, senza la necessità di utilizzare un nuovo IP per ogni servizio che lo richiede. Questi punti di accesso vengono chiamati Ingress, e consistono tipicamente in HTTP API, in cui ogni URL corrisponde ad un servizio accessibile tramite richiesta GET. Le politiche di Ingress permettono livelli di scalabilità più importanti rispetto alle altre descritte

precedentemente, e sono perciò molto diffuse implementazioni che permettono di astrarre dai file di configurazione Ingress dell'orchestratore per facilitare e gestire meglio i servizi mappati [59].

Il networking è una parte centrale per l'esecuzione di un cluster in Kubernetes. Date le astrazioni presenti, Kubernetes lascia allo sviluppatore il compito di decidere quali framework di rete adoperare per la comunicazione tra i componenti presenti nel cluster. Il livello più basso di comunicazione presente è quello tra i container comunicanti nello stesso Pod, risolto automaticamente con comunicazione a localhost. Il secondo livello, quello tra Pod comunicanti in nodi diversi, richiede di implementare interfacce di rete per container secondo il modello CNI (Container Network Interface) adottato dalla Cloud Native Computing Foundation [60]. Il disaccoppiamento offerto tra networking e orchestratore permette di scegliere l'implementazione più consona allo scenario di produzione richiesto. Data la semplicità di utilizzo, come interfaccia di rete è stata scelta *Flannel* [61], la quale istanzia una overlay network che incapsula i pacchetti in ingresso ed in entrata per ogni nodo, evitando di assegnare indirizzi IP diversi per ogni Pod presente sulla rete.

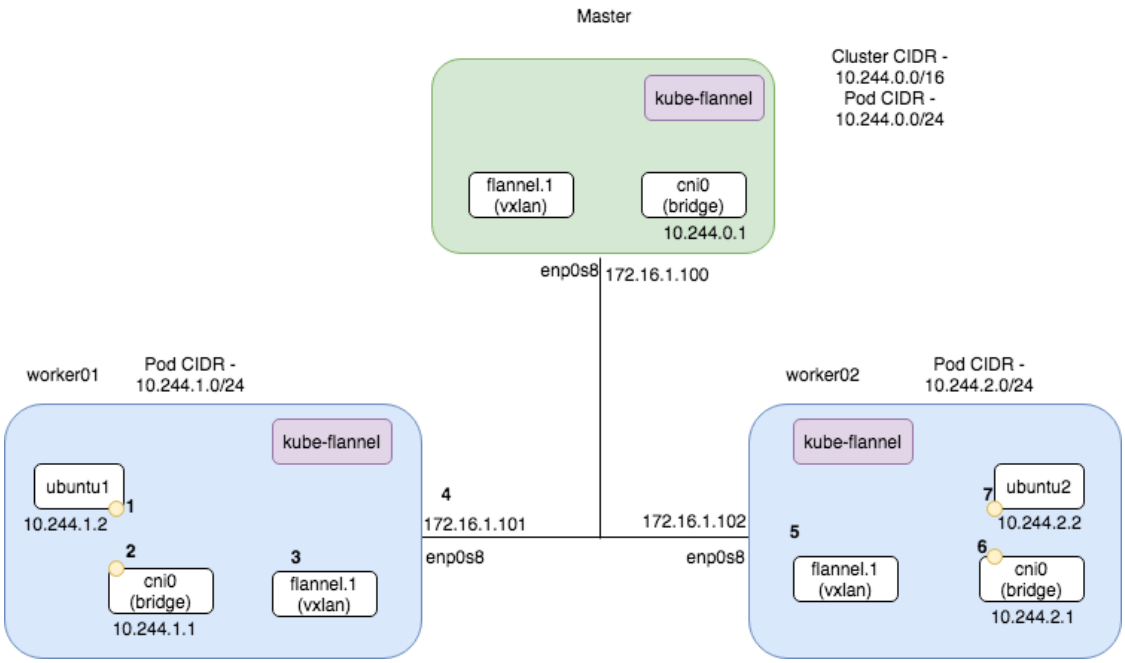

Figura 19 - Networking in Flannel

In *Flannel*, visibile in Figura 19, ogni Pod possiede un IP relativo al suo nodo e al suo cluster: Ogni volta che due Pod comunicano, il pacchetto IP viene incapsulato dal

componente kube-flannel, per poi essere de capsulato sempre da kube-flannel nel nodo del destinatario.

Per inizializzare il cluster in modo da poter utilizzare la overlay network di *Flannel*, è necessario passare come parametro la sottorete adottata dal middleware di rete:

```
kubeadm init --pod-network-cidr=10.244.0.0/16
```

Una volta terminata la creazione del cluster, è necessario eseguire il deployment dei componenti *kube-flannel* per eseguire effettivamente la overlay network:

```
kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/2140ac876ef134e
0ed5af15c66e414cf26827915/Documentation/kube-flannel.yml
```

Infine, è necessario aprire le porte UDP 8285 e 8472 su ogni nodo, per permettere a *Flannel* di comunicare correttamente [62].

Confronto tra orchestratori

Nonostante l'approfondimento più accurato riservato a Kubernetes, è necessario analizzare i due orchestratori per capirne le differenze, nonché i vantaggi e gli svantaggi di ognuno. Partendo dalla fase di installazione, Docker Swarm è sembrato sicuramente più facile da installare in quanto si presenta come un modulo built-in di Docker, permettendo di passare direttamente alla fase di creazione del cluster. Invece, Kubernetes richiede l'installazione di diversi componenti, tra cui un container runtime per l'effettiva esecuzione dei container da parte dell'orchestratore.

In fase di generazione di un nuovo cluster, Docker Swarm ha dimostrato ancora la sua estrema semplicità, data la necessità di scrivere una singola riga di codice per essere pronti all'utilizzo. Diverso il discorso per Kubernetes, in quanto è necessaria una certa comprensione delle dinamiche relative al networking, le quali sono lasciate allo sviluppatore, e che presentano un'elevata curva di apprendimento.

La semplicità intrinseca dell'orchestratore di Docker permette l'esecuzione di piccoli cluster, data l'impossibilità di scalare il prodotto con funzionalità necessarie al controllo di migliaia di componenti. Al contrario, la complessità offerta da Kubernetes offre importanti opportunità di sviluppo, soprattutto in contesti industriali dove è necessario lavorare con migliaia di Pod, complici anche i moduli di produzione offerti dall'orchestratore di Google. Infatti, oltre all'interfaccia di rete CNI richiesta per la comunicazione pod-to-pod, Kubernetes offre interfacce di storage, nonché interfacce di container runtime, rendendo il prodotto estremamente potente e modulabile.

Kubernetes offre un’ambiente completo, capace di far fronte a tutte le esigenze richieste in scenari importanti di produzione, mentre Docker Swarm risulta interessante per il testing e lo sviluppo di nuove piattaforme, ma non per il deployment industriale. Le seguenti ragioni hanno portato alla scelta di Kubernetes come ambiente di esecuzione per le sperimentazioni di tutte le piattaforme.

La trattazione relativa agli orchestratori è stata necessaria al fine di introdurre il lettore alle piattaforme serverless, in quanto i framework serverless sono stati eseguiti e testati su orchestratori di container. I prossimi paragrafi tratteranno le piattaforme serverless riportate precedentemente in Figura 14.

4.2.2 Apache OpenWhisk

Apache OpenWhisk è una piattaforma serverless open source, fondata sulla programmazione ad eventi e adottata da IBM come infrastruttura ufficiale per il servizio IBM Cloud Functions. Le funzioni, chiamate qui Action, possono essere scritte in qualsiasi linguaggio di programmazione, e queste possono essere invocate sia in seguito ad eventi esterni emessi da sorgenti esterne, sia direttamente da richieste HTTP. La qualità dei componenti in gioco, più la capacità espressiva offerta dalla piattaforma, pone OpenWhisk come una delle principali soluzioni FaaS in sistemi ad alta scalabilità.

Architettura

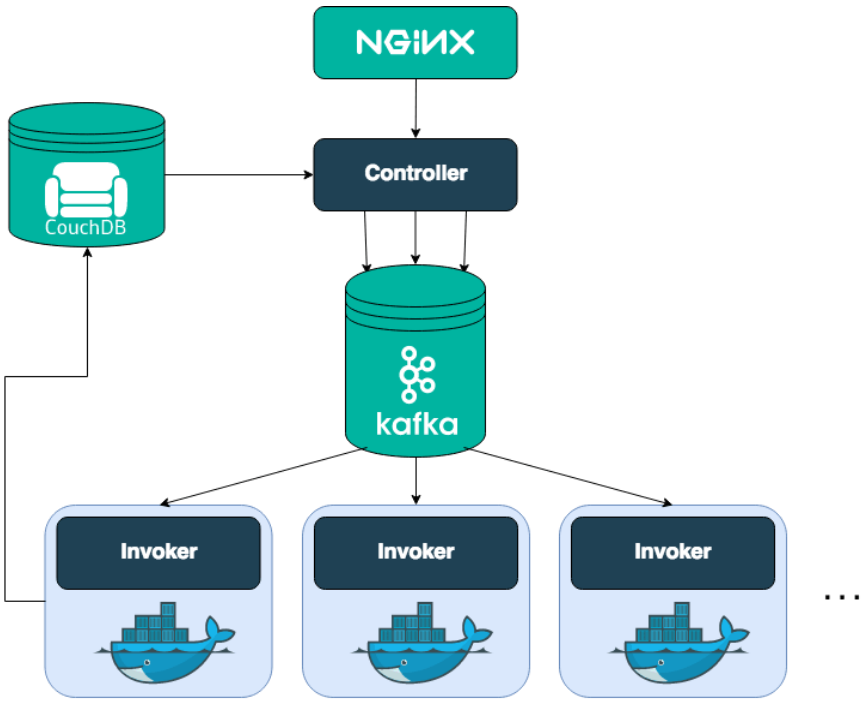

Figura 20 - Architettura di OpenWhisk

L'architettura di OpenWhisk, in Figura 20, si compone principalmente di cinque componenti, necessari per la programmazione ad eventi offerta dal sistema. Partendo dall'alto: NGINX [63] è un HTTP server open source, oltre che reverse proxy, famoso per le ottime performance, la stabilità, le ampie possibilità di configurazione e i costi contenuti. Al contrario di server tradizionali, NGINX non adotta politiche di threading

per gestire le richieste, bensì fonda la propria architettura sugli eventi che garantiscono un sistema estremamente più scalabile, oltre ad essere completamente stateless. In OpenWhisk, NGINX è utilizzato come API gateway per accedere ai servizi della piattaforma FaaS, inoltrando le richieste al Controller.

Il Controller autentica e autorizza tutte le operazioni ricevute dal proxy, permettendo di creare, recuperare, aggiornare ed eliminare (CRUD) tutti i componenti logici del sistema, nonché di invocare le Action. All'interno del Controller è presente un load balancer, inizialmente presente in OpenWhisk come componente separato, il quale controlla lo stato di integrità degli Invoker in modo da affidare la richiesta ricevuta ad uno di questi.

Lo stato dell'intero sistema è invece mantenuto da Apache CouchDB [64], database document-oriented open source NoSQL scritto in Erlang, il cui compito è quello di salvare i metadata, le credenziali, i namespace, le definizioni di action, trigger e rule. Il Controller visiona i dati in CouchDB per verificare l'identità (autenticazione) e i permessi (autorizzazione) di ciascun utente prima di eseguire qualunque operazione. Inoltre, mentre il sistema aggiunge una nuova action, il Controller autorizza l'utente, carica la Action su CouchDB e salva il codice della funzione da eseguire, i parametri formali, i parametri attuali e i limiti imposti sulle risorse allocabili.

Apache Kafka [65] è una piattaforma open source di messaggistica distribuito, sviluppato da LinkedIn per applicazioni in tempo reale e streaming, con elevati criteri di resilienza ai guasti e scalabilità in caso di elevato workload. Il Controller e gli Invoker comunicano esclusivamente attraverso Kafka, realizzando la programmazione ad eventi precedentemente citata, il quale lavora come dispatcher, nonostante il Controller decida a priori l'Invoker a cui è destinata la funzione da eseguire. Infatti, Kafka riceve dal Controller il messaggio contenente l'azione e i relativi parametri, il broker conferma al Controller il corretto inserimento del messaggio nel buffer, e l'utente può ispezionare il messaggio di conferma per conoscere la situazione della action invocata.

Infine, l'Invoker è l'esecutore dell'azione ed utilizza il Docker runtime per eseguire i container in cui vengono isolate ed invocate le action. L'Invoker recupera le informazioni della action direttamente da CouchDB, e, al termine dell'esecuzione di una funzione, il componente accede nuovamente a CouchDB, registra il risulta e distrugge il container. Nonostante questo, è giusto dire che OpenWhisk adotta politiche di riutilizzo dei container, per garantire, quando possibile, warm start delle funzioni e ridurre il tempo di avviamento delle action.

Utilizzo

È possibile interagire con OpenWhisk attraverso un CLI proprietario chiamato wsk. Il CLI permette di creare e di invocare action, recuperare i risultati, isolare Action in differenti namespace, abilitare o disabilitare l'avvio di funzioni allo scatenarsi di un particolare evento. Una volta stabilito l'ambiente più opportuno per la funzione, è possibile creare una Action:

```
wsk action create helloWorld helloWorld.js
```

In questo modo viene generata la Action e, con essa, il sistema è pronto ad invocarla. Una semplice invocazione può essere effettuata direttamente dal CLI:

```
wsk action invoke helloWorld
```

L'invocazione restituisce un codice di attivazione con cui richiedere il risultato, memorizzato in CouchDB. Tramite wsk è possibile accedere direttamente al risultato, passando come argomento il codice attivazione:

```
wsk activation get [ACTIVATION_CODE]
```

Configurazione

Apache OpenWhisk offre numerose opzioni per il deployment della piattaforma, garantendo un alto grado di configurazione per permettere alla piattaforma di adattarsi ai diversi possibili scenari di produzione. Se nessuna opzione viene modificata, OpenWhisk esegue la piattaforma in modalità single-node development, utile soprattutto per testare le funzionalità del framework FaaS. Per permettere alla piattaforma di affrontare le sperimentazioni, è stato necessario lavorare su alcuni parametri di scalabilità offerti da OpenWhisk. Inoltre, su GitHub è presente una guida dettagliata [66] sulle best practice da adottare per scalare il sistema in grandi scenari industriali.

Per permettere al sistema di essere tollerante a guasti ed evitare colli di bottiglia si è deciso di replicare il Controller di due unità:

```
controller:
  replicaCount: 2
```

È possibile migliorare l'affidabilità di OpenWhisk, separando il database CouchDB dal resto del sistema per evitare perdita di dati in caso di guasti della piattaforma:

```
db:
  external: true
```

```
  host: <db hostname or ip addr>
  port: <db port>
  protocol: <"http" or "https">
  auth:
    username: <username>
    password: <password>
```

Il componente Invoker è stato replicato su ogni nodo del cluster, in modo da distribuire equamente il carico computazionale. Inoltre, per ogni nodo è stata riservata la massima quantità di memoria disponibile al componente in questione, per evitare colli di bottiglia dovuti alla mancanza di risorse:

```
whisk:
  containerPool:
    userMemory: “16384m”
```

OpenWhisk offre due possibili factory per la generazione dei container relativi l’esecuzione delle action: il primo è KubernetesContainerFactory, il quale garantisce un ottimo controllo dei container grazie all’adozione dei Pod offerti da Kubernetes, il secondo invece è DockerContainerFactory, il quale lavora a più basso livello, offrendo latenze minori, ma senza i parametri di configurazione offerti dall’orchestratore. Infine, per alleggerire gli Invoker, si è preferito fare offloading del log processing verso soluzioni esterne:

```
invoker:
  containerFactory:
    impl: "kubernetes"
  kubernetes:
    replicaCount: 5
  options:
"-
Dwhisk.spi.LogStoreProvider=org.apache.openwhisk.core.containerp
ool.logging.LogDriverLogStoreProvider"
```

4.2.3 Knative

Eyal Manor, vicepresidente presso Google Cloud Platform, nel luglio 2018 annunciò Knative come un'iniziativa che avrebbe portato agli sviluppatori la possibilità di scrivere applicazioni serverless in Kubernetes.

Knative è un progetto open source il cui obiettivo è quello di aggiungere nuovi componenti all'orchestratore Kubernetes per la realizzazione di applicazioni serverless cloud native, per migliorare la produttività degli sviluppatori ed abbattere i costi operazionali. In Knative, il codice applicativo viene isolato in container, ed la piattaforma si occupa di avviare e fermare ogni istanza invocata nell'orchestratore [67].

> *"Knative is standardizing the primitives needed to simplify the code to production workflows. It promises to be a solid foundation to develop the future of serverless computing" (S. Goasguen – Kubeless founder)*

Knative utilizza le Custom Resource offerte da Kubernetes per definire i componenti della piattaforma. In questo modo, come avviene per Fission, si stabilisce una maggiore integrazione tra l'orchestratore e la piattaforma FaaS, potendo lavorare direttamente con le API di Kubernetes.

Architettura

Knative può essere installato in qualsiasi ambiente – per esempio, cloud pubblici e privati – dove è presente Kubernetes come orchestratore.

Figura 21 - Architettura di Knative

Come mostrato in Figura 21, Knative, al contrario degli altri framework, non offre un API gateway per lo smistamento delle richieste verso il carico serverless ospitato dalla piattaforma. Per questo motivo, Knative necessità di un componente esterno, chiamato service mesh, che vada a gestire le interazioni tra i diversi servizi, e che controlli il traffico esterno ricevuto dall'orchestratore. Data la rilevanza del progetto, Knative consiglia di utilizzare Istio, service mesh attualmente molto diffusa ed utilizzata da importanti piattaforme presenti sul mercato.

Istio è la service mesh raccomandata da Knative, la quale offre funzionalità di load balancing, autenticazione service-to-service, monitoring, e management del traffico. L'architettura di Istio, come mostrato in Figura 22, è composta da diversi componenti indipendenti tra loro, quali:

- Envoy – permette la comunicazione tra servizi, oltre ad offrire un servizio di discovery, ampie possibilità di routing e adozione di protocolli di sicurezza;
- Mixer – colleziona metriche a runtime e mantiene consistente lo stato del cluster;
- Pilot – permette la configurazione dei proxy a runtime;
- Citadel – gestisce la validazione dei certificati di comunicazione.

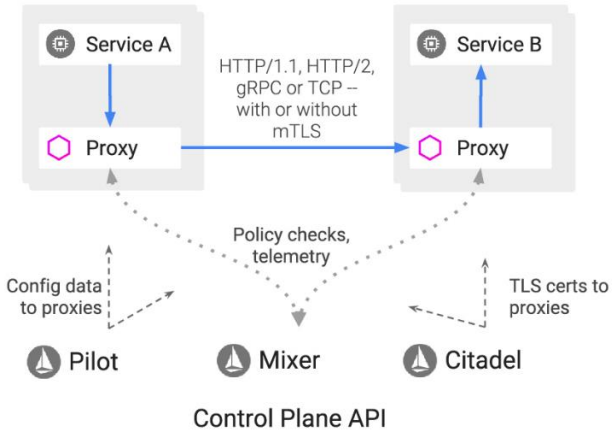

Figura 22 - Architettura di Istio

È possibile adottare altre service mesh, quali Ambassador, soprattutto nel caso di vincoli di risorse particolari, in quanto Istio carica l'orchestratore di numerose funzionalità aggiuntive per il monitoraggio del framework.

Knative è composto principalmente di due componenti: Serving ed Eventing. Il componente Serving permette il deployment di applicazioni serverless su Kubernetes. Questo gestisce la natura serverless dell'applicazione, permettendo di controllare i dettagli relativi allo scaling, al revision tracking, soprattutto in caso di più versioni rilasciate della stessa applicazione, e al routing per accedere ai servizi offerti. Inoltre, il componente offre la funzionalità di staged roll out, la quale consente di rilasciare aggiornamenti graduali e di avere più versioni dello stesso servizio nello stesso istante. Il Serving va a generare quattro Custom Resource, vedasi Figura 23, per definire il comportamento del workload serverless su Kubernetes:

- Service – gestisce il ciclo di vita dei servizi e la loro creazione, garantendo ad ogni servizio una Route, una Configuration e il controllo di diverse Revision;
- Route – mappa un endpoint di rete a uno o più Revision, e gestisce il traffico;
- Configuration – mantiene lo stato desiderato del servizio. La modifica di una Configuration comporta la generazione di una nuova Revision;
- Revision – snapshot immutabile del workload serverless catturata in un determinato istante di tempo. Il meccanismo delle Revision è utilizzato insieme a quello delle Configuration, e permette la separazione tra l'immagine del container e la loro configurazione. In questo modo, dopo ogni cambiamento di configurazione, è sempre possibile effettuare un rollback all'ultima

configurazione funzionante, in quanto viene mantenuta in memoria la storia delle configurazioni adottate.

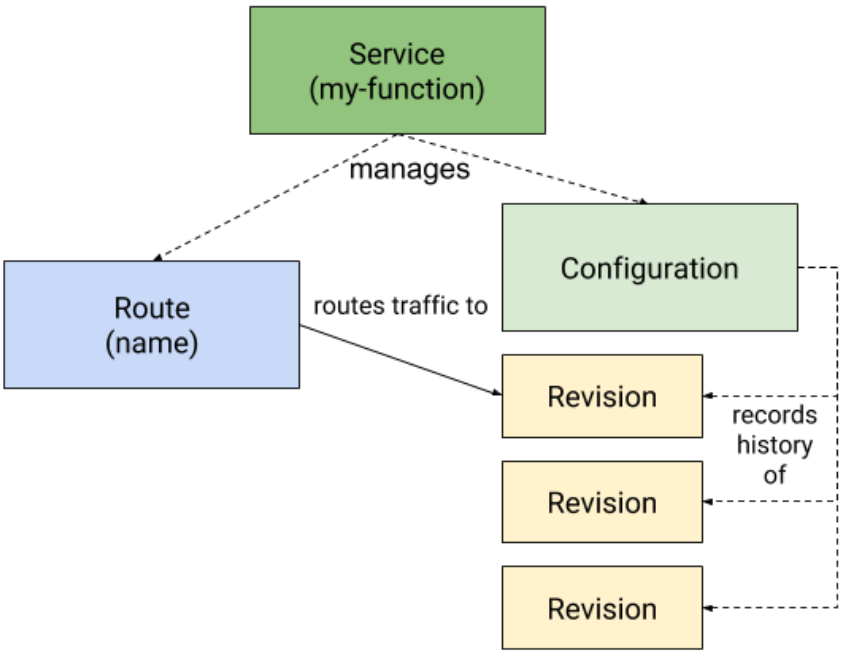

Figura 23 - Architettura di Knative Serving

Il componente Eventing è un sistema nato per permettere l'interconnessione tra componenti secondo i principi dello scambio dei messaggi, offrendo primitive per la comunicazione di componenti cloud native in maniera debolmente accoppiata. L'indipendenza è il goal principale, in quanto i servizi che adottano Eventing possono essere eseguiti su qualsiasi ambiente, senza la necessità di scrivere API specifiche per la comunicazione con gli altri attori in esecuzione. La natura disaccoppiata comporta che l'architettura sia fondata sul principio dei produttori e dei consumatori, i quali comunicano tramite l'interfaccia di un message broker che si fa carico di smistare correttamente i messaggi secondo le politiche più idonee per lo sviluppatore. I produttori ed i consumatori possono essere scritti senza dover pensare all'infrastruttura di comunicazione, completamente gestita dal componente Eventing, riducendo notevolmente gli sforzi e i costi di sviluppo [68].

Utilizzo

L'utilizzo di Knative per il deployment delle applicazioni serverless è completamente trasparente rispetto a Kubernetes. Per caricare una nuova funzione, o più in generale un microservizio serverless, è necessario scrivere un file di configurazione che viene gestito direttamente dall'orchestratore, con la differenza di utilizzare API relative alle Custom Resource introdotte da Knative. In particolare, nel campo relativo alle API da utilizzare per il deployment del servizio, è necessario specificare le API del componente Serving [69].

```
apiVersion: serving.knative.dev/v1 # Current version of Knative
kind: Service
metadata:
  name: helloworld-go # The name of the app
  namespace: default # The namespace the app will use
spec:
  template:
    spec:
      containers:
        - image: gcr.io/knative-samples/helloworld-go # The URL to the image
of the app
          env:
            - name: TARGET # The environment variable printed out by the
sample app
              value: "Go Sample v1"
```

Figura 24 - Risorsa Ksvc per il deployment di un servzio Knative

Una volta scritto il file di configurazione, visibile in Figura 24, è possibile eseguire il deployment su Kubernetes:

```
kubectl apply -f helloworld-go.yml
```

Il servizio viene aggiunto alla lista dei servizi accessibili nel cluster, ma nessun Pod viene generato finché il servizio non viene invocato:

```
curl http://helloworld-go.default.example.com
```

L'invocazione è gestita completamente da Istio, il quale riconosce il servizio a cui si vuole accedere grazie al campo Host presente nell'header del protocollo HTTP. Per questa ragione, se l'invocazione viene eseguita con curl, applicazione terminal-based per effettuare invocazioni HTTP, non ci sono problemi, in quanto basta aggiungere l'opzione Host come parametro prima di invocare la funzione:

```
curl -H "Host: helloworld-go.default.mydomain.com"
http://$ISTIO_INGRESS_IP:$ISTIO_INGRESS_PORT
```

Invece, se l'invocazione viene eseguita tramite browser, non è possibile modificare il campo Host, ed è dunque necessario adottare un DNS pubblico o privato per accedere al servizio. Per esempio, in fase di development e testing, è possibile modificare il file hosts per permettere la traduzione dell'indirizzo IP e l'aggiunta automatica del campo Host all'header della richiesta HTTP:

```
$ISTIO_INGRESS_IP myFun.default.mydomain.com
```

Configurazione

Le configurazioni relative ad ogni servizio eseguito su Knative possono essere ereditate da un file di configurazione offerto da Serving, chiamato config-autoscaler, dove è possibile modificare i parametri relativi ad autoscaling, attivare o disattivare l'opzione scale to zero, definire il grado di concorrenza dei container o stabilire l'intervallo di funzionamento dell'autoscaler. Inoltre, è possibile stabilire il numero di repliche per ogni servizio, grazie all'utilizzo di annotazioni da aggiungere alla risorsa config-autoscaler: [70]

```
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/minScale: "2"
        autoscaling.knative.dev/maxScale: "10"
```

È possibile modificare i parametri di deployment anche a runtime, modificando la risorsa Podautoscaler generata alla creazione di un servizio Serving. Ovviamente, data la completa integrazione con Kubernetes, è possibile adottare l'Horizontal Pod Autoscaler per permettere la replicazione dei Pod in base all'utilizzo di CPU e/o memoria:

```
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/metric: cpu
        autoscaling.knative.dev/target: "70"
        autoscaling.knative.dev/class: hpa.autoscaling.knative.dev
```

4.2.4 OpenFaaS

OpenFaas è una piattaforma FaaS open source basata su licenza MIT, sostenuta da una corposa community in continua crescita. I punti di forza del framework sono la facilità di utilizzo e la portabilità, dato che OpenFaas può essere eseguito su diversi ambienti di deployment, tra cui Kubernetes [71].

Architettura

Il framework risulta accessibile tramite REST API, CLI o interfaccia Web proprietaria, fornite automaticamente in fase di deployment. Tutte le funzioni, quando generate, ricevono un indirizzo del tipo https://<gateway URL>:<port>/function/<function name> con il quale è possibile l'invocazione sincrona con protocollo HTTP. È possibile invocare le funzioni in maniera asincrona utilizzando il sistema a scambio di messaggi offerto da NATS Streaming, invocando le funzioni con l'indirizzo https://<gateway URL>:<port>/async-function/<function name> e ponendo nell'header il campo "X-Callback-Url", necessario al broker per avvertire del completamento della funzione. Oltre la possibilità di invocare le funzioni direttamente con il protocollo HTTP, è possibile agganciare il framework a sorgenti di eventi, quali Apache Kafka, Redis, o soluzioni proprietarie come AWS SQS [72].

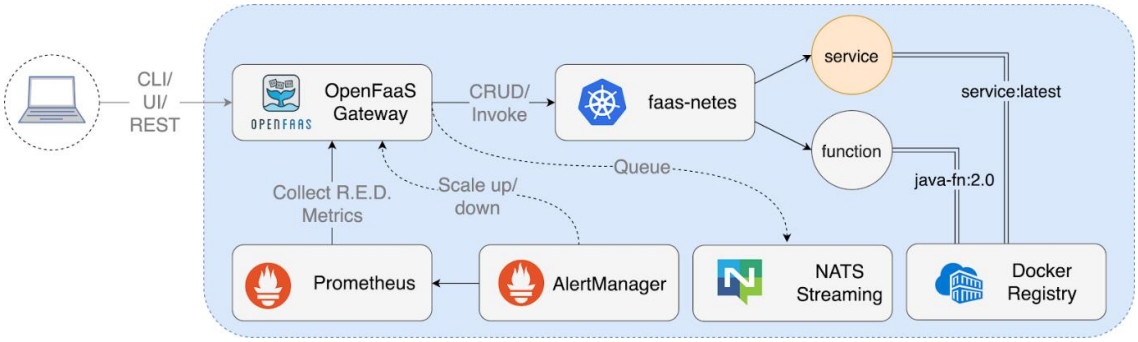

Figura 25 - Architettura di OpenFaaS

In Figura 25 si ha una panoramica dei componenti utilizzati da OpenFaaS. Il Gateway, punto di accesso al framework, fornisce API per la gestione delle funzioni caricate nella piattaforma. Esso comunica direttamente con il componente faas-netes, provider OpenFaaS per Kubernetes, il quale è garante di tutte le operazioni effettuate sulle funzioni. La separazione tra i due componenti citati permette al framework di essere

completamente indipendente rispetto all'ambiente di esecuzione. Per cui, se a Kubernetes si vuole sostituire Docker Swarm, è necessario sostituire unicamente il provider faas-netes con un altro provider relativo all'orchestratore di Docker. Inoltre, il Gateway è responsabile nel mantenere il giusto numero di repliche della funzione, e permette la visualizzazione delle statistiche accumulate dal sistema di monitoring.

In OpenFaaS le repliche vengono gestite nativamente, ovvero la piattaforma non si avvale del sistema di autoscaling offerto da Kubernetes, nonostante esso sia adoperabile come soluzione alternativa. In particolare, Prometheus, popolare sistema di monitoring open source, colleziona metriche relative all'utilizzo del framework e, quando ritenuto necessario, notifica il Gateway che andrà a modificare il numero di repliche della funzione. Per realizzare il suo scopo, Prometheus si avvale del modulo AlertManager, interpretabile come insieme di regole definite dall'utente che, se violate, vanno a scatenare il sistema di monitoraggio. Oltre ai componenti necessario all'invocazione delle funzioni, OpenFaaS offre NATS Streaming, broker di eventi distribuito, ed i relativi queue-worker, i quali permettono l'invocazione asincrona delle funzioni.

Ogni funzione, quando distribuita, oltre al processo responsabile per l'esecuzione del codice, possiede un processo watchdog**,** visibile in Figura 26, aperto alla porta HTTP per accettare connessioni provenienti dal provider di OpenFaaS.

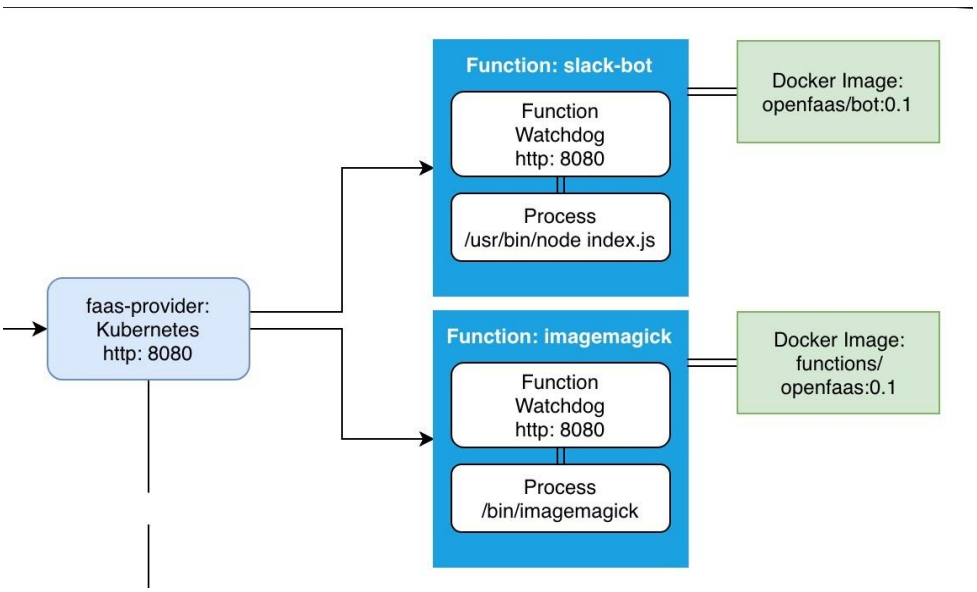

Figura 26 - Architettura della funzione, composta da watchdog e logica da eseguire

Il processo watchdog scatena l'esecuzione della funzione, monitorando le richieste concorrenti e i possibili timeout. Attualmente, OpenFaaS offre una nuova implementazione, chiamata of-watchdog, il quale raggiunge performance migliori, in

quanto il vecchio watchdog andava a generare un nuovo processo per ogni nuova richiesta, mentre la nuova implementazione gestisce un pool di connessioni e ricicla i processi per una gestione più efficiente delle richieste [73].

Utilizzo

In OpenFaaS, le funzioni vengono controllate da un CLI proprietario, chiamato faas-cli, il quale comunica direttamente con il Gateway. Per permettere l'interazione tra i due componenti, è necessario definire la variabile d'ambiente OPENFAAS_URL. Una volta configurato il CLI, si può iniziare a creare la nuova funzione da eseguire nel framework. OpenFaaS offre template per la maggior parte dei linguaggi di programmazione disponibili sul mercato, come per esempio l'ambiente di esecuzione di Node 10:

```
faas-cli template store pull node10-express
```

Una volta salvato il template, è possibile utilizzarlo per creare una funzione:

```
faas-cli new helloWorld –lang node10-express
```

A questo punto, viene generata automaticamente la struttura della funzione, basata sul template passato come parametro, e che presenta i seguenti file:

```
./helloWorld.yml
./helloWorld/
./helloWorld/handler.js
./helloWorld/package.json
```

Nel file handler.js viene scritta la logica della funzione, mentre nel file di configurazione helloWorld.yml è possibile modificare i parametri di deployment. Prima di pubblicare la funzione, è necessario costruire l'immagine Docker della funzione:

```
faas-cli build -f helloWorld.yml
```

Se il cluster è distribuito, è necessario fare l'upload dell'immagine su un registro Docker remoto:

```
faas-cli push -f helloWorld.yml
```

Infine, si può effettuare il deployment della funzione:

```
faas-cli deploy -f helloWorld.yml [74]
```

Configurazione

Di default, OpenFaaS presenta una singola regola in AlertManager, la quale definisce il massimo numero di richieste al secondo e segnala il Gateway in caso di superamento. Oltre a poter scalare in base alle richieste ricevuto al secondo, è possibile scalare le funzioni in base all'utilizzo di CPU e/o memoria utilizzate in un determinato intervallo di tempo. Per fare questo, OpenFaaS si avvale dell'Horizontal Pod Autoscaler (HPA) offerto da Kubernetes, utilizzato anche in altri framework per poter scalare allo stesso modo [75].

Come detto precedentemente, ogni funzione offre un file di configurazione per poter definire il comportamento a runtime della function. È possibile definire il numero minimo e massimo di repliche, per garantire tempi di avviamento più rapidi, ed evitare di appesantire eccessivamente il carico del cluster:

- `com.openfaas.scale.min` – minimo numero di repliche;
- `com.openfaas.scale.max` – massimo numero di repliche.

Kubernetes offre la possibilità di impostare i valori minimi e massimi di CPU e memoria da affidare ai container delle funzioni. In caso di load testing è consigliato impostare i limiti inferiori con valori importanti, e non definire limiti di risorse per evitare strozzamenti alle performance. Se le quantità minime richieste di CPU e memoria non possono essere soddisfatte dal cluster, lo scheduler di Kubernetes ferma i Pod insoddisfatti, lasciandoli in stato di Pending.

4.2.5 Fission

Fission è un framework FaaS open source per Kubernetes, supportato da Platform9, società americana di servizi hybrid cloud. Gli aspetti fondamentali sono la facilità d'uso, le performance, soprattutto in fase di avviamento dei container, e l'importante community, cresciuta negli ultimi mesi. Fission è fondato su tre concetti fondamentali:

- Trigger – permettono di catturare diverse sorgenti di eventi, in modo da invocare le funzioni. Quando il trigger riceve una richiesta, questo invoca la funzione inviando a sua volta una richiesta HTTP dal router alla funzione;
- Environment – offrono processi per compilare ed eseguire funzioni in qualsiasi linguaggio di programmazione. È composta da container che catturano le richieste HTTP ed eseguono le istruzioni della funzione;
- Function – la classica funzione, solitamente composta da interfaccia e relative implementazioni [76].

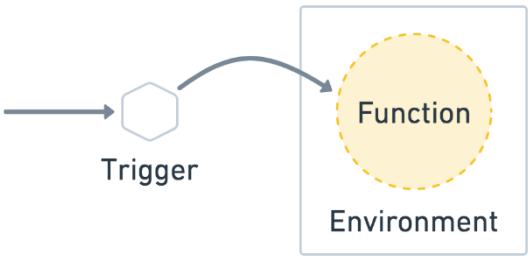

Figura 27 - Componenti principali di Fission

In Figura 27 è possibile notare come la Function sia isolata all'interno dell'Environment, la quale definisce i parametri di esecuzione di tutti i container presenti all'interno.

Architettura

L'architettura di Fission, come mostrato in Figura 28, è fondata completamente su Kubernetes. Per questa ragione, Fission risulta essere meno portabile rispetto ai framework già analizzati, nonostante Kubernetes sia l'orchestratore più popolare sul mercato.

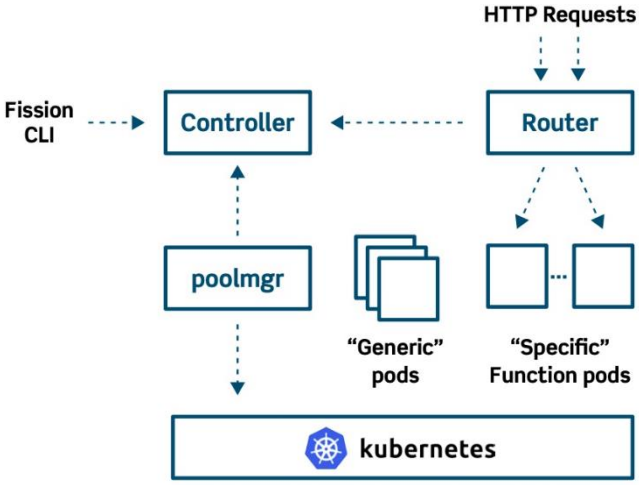

Figura 28 - Architettura di Fission

È possibile iniziare ad interagire con Fission grazie al CLI proprietario offerto dal framework. Il CLI accede al Controller, componente centrale dell'architettura, per invocare, creare, modificare ed eliminare le funzioni presenti nella piattaforma. Le risorse con cui Fission lavora, sono definite come Custom Resource di Kubernetes, e il Controller ha pieni poteri sulle risorse del framework, a patto che sia stato definito un Service Account con permessi di amministrazioni sul cluster [77]. Fission sfrutta le Custom Resource di Kubernetes per definire funzioni, trigger, environment, in modo da integrare il mondo della piattaforma FaaS con l'orchestratore.

Il Controller gestisce le invocazioni dal CLI proprietario, mentre gli eventi generati da richieste HTTP sono gestiti dal Router, componente stateless che si interpone tra le richieste e le specifiche funzioni. Nel momento in cui riceve una richiesta, il Router controlla se esistono indirizzi in cache a cui inoltrare la richiesta, altrimenti fallisce (cache miss) e chiede aiuto ad un altro componente, denominato Executor, il quale fornisce il nuovo indirizzo al Router [78].

L'Executor è il responsabile per la generazione dei Pod in cui vengono eseguite le funzioni. Il Router chiede all'Executor un indirizzo per inoltrare la richiesta (Figura 29), quest'ultimo analizza le informazioni relative alla funzione e produce i Pod per l'elaborazione della richiesta, restituendo l'indirizzo del servizio quando i relativi endpoint sono pronti a ricevere.

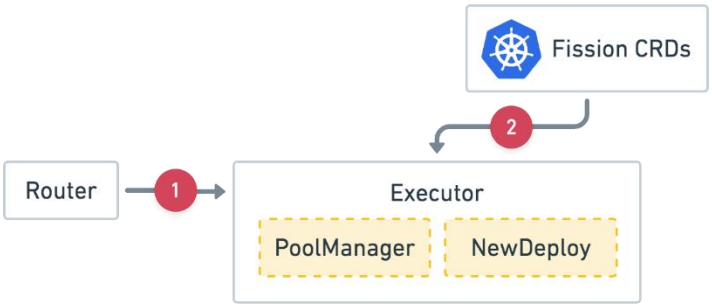

Figura 29 - Workflow di Fission

Fission offre due tipi di Executor, PoolManager e NewDeploy. NewDeploy, mostrato in Figura 30, in fase di deployment di una nuova funzione, genera un Deployment, un Service ed un Horizontal Pod Autoscaler in Kubernetes. Invece, PoolManager genera un insieme di container generici, dove ognuno di questi ha la facoltà di caricare in memoria la funzione quando necessario, specializzandosi al momento opportuno [79].

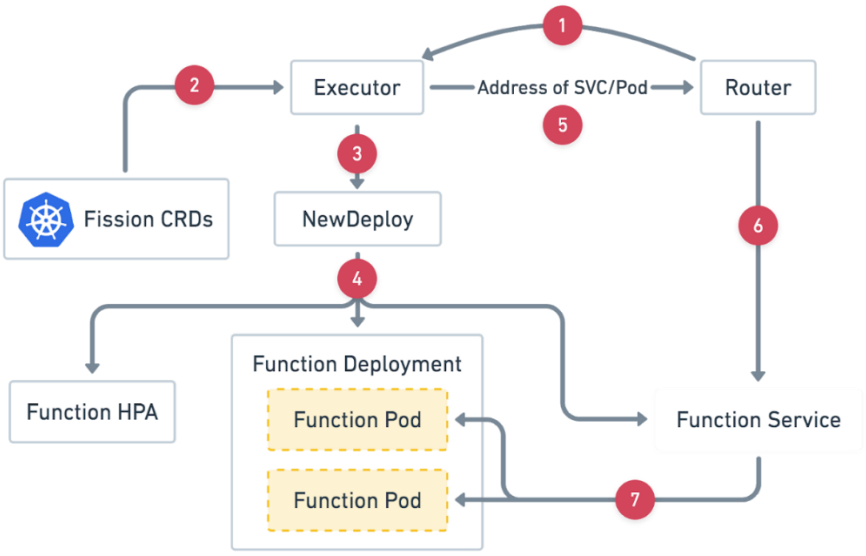

Figura 30 - Architettura di NewDeploy

Il Function Pod, visibile in Figura 31, è la risorsa responsabile per l'esecuzione della funzione e della restituzione del risultato. Il componente è composto da due container, Fetcher ed Environment Container. Il primo cattura l'indirizzo della funzione, e ne scarica l'immagine contenente il codice e le risorse invocando il componente StorageSvc. Invece,

il secondo attende che Fetcher salvi l'immagine in un volume condiviso ai due container, per poi caricare l'immagine e, infine, eseguire la funzione [80].

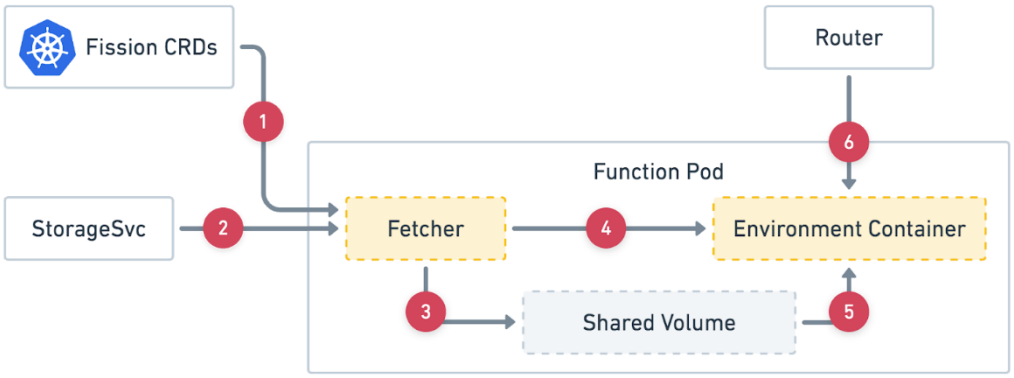

Figura 31 - Architettura di Function Pod

Utilizzo

Per eseguire correttamente il deployment del framework è necessario, per prima cosa, istanziare un nuovo Environment dove eseguire la funzione. Le interazioni verso la piattaforma avvengono grazie al CLI proprietario di Fission, attraverso il quale è possibile creare l'ambiente per la funzione, potendo scegliere i diversi runtime resi disponibili dal framework:

```
fission env create --name nodejs --image fission/node-env
```

Una volta generato l'Environment, si può creare la funzione inserendo come argomenti l'ambiente appena generato, il nome, il file con il codice e il tipo di Executor. Inoltre, è possibile definire il numero massimo e minimo di repliche desiderate:

```
fission fn create --name helloWorld --code helloWorld.js --env
node --minscale 1 --maxscale 5 --executortype newdeploy
```

Infine, va configurato l'indirizzo per invocare la funzione tramite protocollo HTTP:

```
fission route create --function helloWorld --url /helloWorld
```

Il deployment è così completato, ed è possibile invocare la funzione direttamente dal CLI di Fission:

```
fission function test --name helloWorld
```

Configurazione

Come detto precedentemente, Fission è fondato su Kubernetes, e per questo adotta le politiche di autoscaling offerte dall'orchestratore. In particolare, Fission permette di replicare i Pod grazie all'HPA, definendo durante il deployment della funzione i valori della CPU che il carico deve raggiungere per iniziare a scalare le repliche. I valori devono essere passati come parametri durante la creazione, e sono i seguenti:

- `--mincpu` – quantità minima di CPU da assegnare ad un Pod (misurata in millicore);
- `--maxcpu` – quantità massima di CPU da assegnare ad un Pod (misurata in millicore);
- `--minmemory` – quantità minima di memoria da assegnare ad un Pod (misurata in Mb);
- `--maxmemory` – quantità massima di memoria da assegnare ad un Pod (misurata in Mb);
- `--targetcpu` – percentuale di CPU da raggiungere per poter scalare i Pod (di default pari a 80 percento)

Oltre ai parametri relativi alle risorse da allocare per ogni funzione, è possibile definire la quantità minima e massima di repliche, definite anche come parametri in fase di creazione. Questi parametri devono essere inseriti unicamente se il tipo di Executor utilizzato è NewDeploy, ovvero `--executortype newdeploy`, in quanto PoolManager mantiene sempre una certa quantità di container pronti ad accettare le richieste:

- `--minscale` – numero minimo di Pod per funzione;
- `--maxscale` – numero massimo di Pod per funzione.

4.2.6 Confronto qualitativo

Nella Tabella 2 vengono evidenziati gli aspetti tenuti in considerazione durante la comparazione qualitativa delle piattaforme.

	OpenFaaS	OpenWhisk	Knative	Fission
Service mesh necessaria	No	No	Yes (Istio)	No
CLI	Yes	Yes	No	Yes

Tool esterni per autoscaling	No	No	Yes (metrics-server)	Yes (metrics-server)
Upload su registro Docker	Yes	No	Yes	No
Vincolo sul numero di CPU necessarie	No	Yes (2 core)	Yes (6 core)	No
Vincolo sulla disponibilità di memoria	No	Yes (4 GB)	Yes (8 GB)	No
Vincolo sulla disponibilità di memoria di massa	No	No	Yes (30 GB)	NO

Tabella 2 - Sintesi qualitativa delle piattaforme

Primo aspetto analizzato nel confronto qualitativo tra le piattaforme è l'architettura, valutata principalmente per la complessità, per la consistenza e stabilità dei singoli componenti, e per la quantità di interazioni in gioco. Tutte le piattaforme condividono il modo con cui distribuiscono le responsabilità tra i diversi componenti, mostrando come le piattaforme FaaS stiano convergendo verso una certa standardizzazione, come mostrato in Figura 32. In tutte le soluzioni è presente un punto di accesso per interagire con la funzione richiesta, dove in Knative è presente Istio che si fa garante del traffico proveniente dall'esterno.

Poi, ogni piattaforma possiede un controllore che smista le richieste ricevute e provvede ad autenticare l'utente che ne ha richiesto il servizio. Inoltre, il controllore offre un punto di accumulo per log e metriche, utile per monitorare la situazione del middleware. Infatti, spesso uno o più componenti di monitoring sono eseguiti insieme al resto della piattaforma, per migliorare il controllo e la sicurezza nel suo insieme.

Tipicamente, le funzioni vengono containerizzate e salvate come immagini presso registri Docker locali o pubblici, osservando come i container Docker siano ormai uno standard profondamente accettato in ambito industriale. Infine, i worker rappresentano l'effettiva esecuzione delle funzioni, e sono spesso distribuiti equamente tra i nodi del cluster, secondo algoritmi standard o decidibili dall'amministratore del sistema.

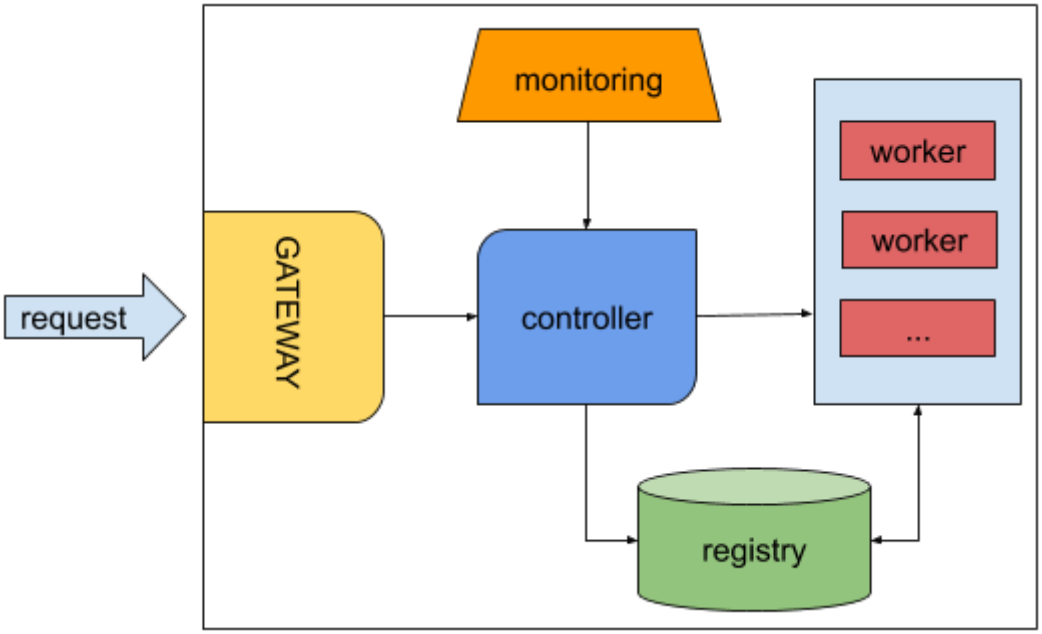

Figura 32 - Generica architettura di una piattaforma FaaS

Tutte le piattaforme stanno convergendo verso l'adozione di Kubernetes come unico ambiente di esecuzione possibile. La scelta deriva dalla possibilità di integrare la piattaforma all'orchestratore grazie all'utilizzo di Custom Resource che rendono l'approccio completamente trasparente, data la possibilità di rendere le risorse della piattaforma FaaS native in Kubernetes. Inoltre, il livello di astrazione offerto da Kubernetes permette di eseguire il deployment in ambienti cloud differenti, diffondendo l'idea di multi-cloud che oggi sta riscuotendo un enorme successo.

Un altro aspetto analizzato riguarda la facilità d'uso offerta dalle piattaforme. Le installazioni sono risultate semplici ed efficaci per tutti i middleware, soprattutto grazie all'utilizzo di Helm, fatta eccezione per Knative, il quale ha richiesto l'installazione a priori di Istio. Inoltre, poiché Istio richiede la presenza di un load balancer, è stato necessario installare MetalLB, soluzione per inserire load balancer su cluster bare-metal. Infatti, mentre provider di cloud pubblici offrono load balancer integrati e pronti all'utilizzo, per le sperimentazioni è stato necessario configurarlo manualmente, nonostante lo sforzo richiesto sia stato minimo. È importante citare OpenWhisk per l'importante quantità di configurazioni disponibili in fase di deployment, permettendo una versatilità ed un grado di personalizzazione che non ha pari rispetto alle altre soluzioni analizzate. Knative, in quanto add-on per workload serverless in Kubernetes, offre la possibilità di eseguire funzioni utilizzando direttamente i file di configurazione dell'orchestratore. Al contrario, gli altri framework richiedono l'utilizzo di un CLI proprietario, preventivamente configurato, per eseguire operazioni sui middleware.

La trattazione si conclude con le possibilità di configurazione e di autoscaling offerte dai framework. OpenWhisk utilizza un meccanismo di autoscaling proprietario e, pertanto, non sono necessari step aggiuntivi per permettere la replicazione dei Pod, nonostante siano consigliati dalla guida ufficiale dei parametri da modificare, in fase di deployment della piattaforma, per raggiungere livelli di scalabilità più adatti a scenari produttivi. OpenFaaS è l'unico che permette l'utilizzo di autoscaling basato sia sulle metriche di Prometheus per replicare in base al numero di richieste ricevute, e sia sull'Horizontal Pod Autoscaler offerto da Kubernetes per replicare in base alla quantità di risorse utilizzate. Infine, Fission e Knative obbligano l'utilizzo dell'HPA come meccanismo di autoscaling, il quale richiede che API aggiuntive all'orchestratore vengano integrate grazie all'installazione del package metrics-server, il quale permette di raccogliere informazioni su CPU e memoria accumulate dagli agenti Kubelet, e di utilizzarli come parametri di replicazione. Il metrics-server ha presentato un po' di difficolta in fase di deployment, data la necessità di interagire con i Kubelet tramite protocolli di cifratura, i quali richiedono un certo livello di conoscenze relative a chiavi e certificazioni.

La sezione qualitativa si conclude con i vincoli relativi alle risorse che ogni nodo deve possedere per permettere l'esecuzione dei framework. Fission ed OpenFaaS possono essere eseguiti su ogni tipo di nodo, senza particolari richieste di CPU o memoria, mentre OpenWhisk richiede minimo una CPU con dual core e 4GB di memoria. Knative, soprattutto a causa dei forti requisiti di Istio, richiede risorse impegnative, con almeno sei core, 8GB di memoria e 30GB di memoria di massa, rendendolo inadatto per single-node development e cluster con risorse limitate. È importante dire che Knative offre service mesh alternative più leggere, ma che non offrono lo stesso livello di documentazione e supporto rispetto ad Istio.

4.3 Comparazione delle piattaforme

Per iniziare a confrontare i framework analizzati precedentemente, è necessario capire quali aspetti misurare per poter generare report significativi, che possano definire quali soluzioni siano più adatte a scenari di produzione industriale. La profonda comprensione del dominio applicativo permette di orientare le sperimentazioni, in modo da definire quali aspetti confrontare per valutare le diverse piattaforme. In sistemi distribuiti, la prima caratteristica da testare è la scalabilità, soprattutto se gli attori in gioco sono numerosi e variabili nel tempo, come in applicazioni IoT, oppure in applicazioni Web dove possono essere richiesti contemporaneamente migliaia di servizi.

Per misurare il grado di scalabilità si adottano processi di load testing che consistono nell'inviare un certo numero di richieste per poi analizzare le risposte ricevute, permettendo di valutare il tempo medio di risposta o la percentuale di errori in un determinato intervallo di tempo. Il load testing permette, dunque, di valutare un sistema in condizioni di normale esecuzione. In particolare, quando il processo stressa la piattaforma con condizioni di carico eccessive, questo viene chiamato stress test, utile per stabilire il livello di tolleranza in condizioni di anormalità. Ogni sistema risponde in maniera diversa alle situazioni di carico imposte durante i load testing, perciò è fondamentale capire quali sperimentazioni effettuare per setacciare eventuali criticità o punti deboli. Per esempio, framework che offrono servizi particolarmente sensibili dovrebbero essere robusti di fronte a situazioni di stress improvvise, mentre servizi comuni e molto utilizzati dovrebbero garantire basse latenze con qualche errore.

Valutare la scalabilità è un processo complesso, e prende in considerazione diversi aspetti durante la sua valutazione:

- Tempo medio di risposta;
- Throughput;
- Percentuale di errori;
- Utilizzo della rete;
- Utilizzo della CPU;
- Utilizzo della memoria;
- Durata della sessione.

La valutazione di questi parametri passa anche dalla credibilità della sperimentazione, dunque per i test è stato utilizzato un ambiente di load testing, come Apache JMeter, capace di simulare comodamente anche migliaia di utenti simultaneamente, con la possibilità di definire tempi di attesa tra una richiesta e la successiva per ogni utente, per rendere più credibile la simulazione.

Apache JMeter [81] è un software open source scritto in Java ed utilizzato per eseguire load testing in scenari industriali. Originariamente, era stato sviluppato per valutare le performance di applicazioni Web, mentre adesso viene adoperato per qualsiasi tipo di applicazione. La scelta è ricaduta naturalmente verso Apache JMeter per la sua popolarità e il supporto garantito dalla community durante il corso degli anni. La piattaforma offre un'interfaccia grafica molto intuitiva, sviluppata per permettere la configurazione ed il debugging, ed un CLI per l'effettiva esecuzione dei test in modo da ridurre l'overhead causato principalmente dalla GUI dell'applicazione stessa. Il framework permette l'interazione con tutti i protocolli più noti, quali ad esempio HTTP, LDAP, FTP, JDBC, SMTP, e consente di simulare un certo numero di utenti, ognuno dei quali può effettuare un numero arbitrario di richieste.

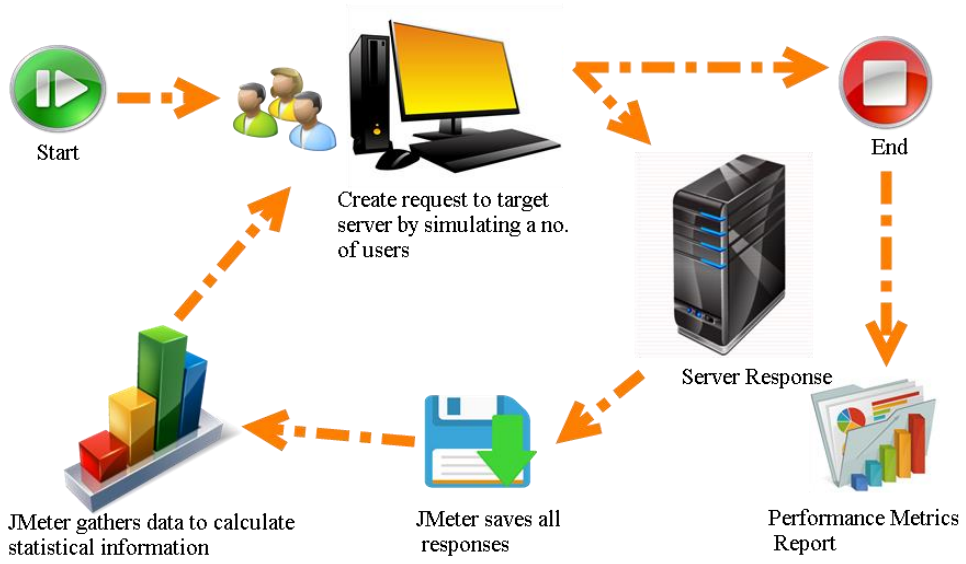

Figura 33 - Workflow di JMeter

Il workflow di JMeter è semplice ed efficiente, come mostrato in Figura 33, ed offre strumenti per il salvataggio e la visualizzazione dei dati da analizzare comodamente out-of-the-box. Una volta avviato il test, sul client viene generato un certo numero di thread, ognuno dei quali corrisponde ad un utente fittizio, in un determinato intervallo di tempo prestabilito. Per esempio, possono essere generati in maniera uniforme cento utenti in cento secondi. La scelta di rendere graduale la creazione degli utenti è necessaria per evitare picchi computazionali eccessivi e per rendere più credibile l'accesso degli utenti a determinati servizi, dato che questi non accederanno contemporaneamente, ma con un piccolo scarto l'uno dall'altro.

Ogni utente inizia ad inviare un certo numero di richieste mentre il server di JMeter resta in ascolto delle risposte ricevute, producendo metriche significative che possono essere analizzati in file di logging salvati nel filesystem. Inoltre, il server di JMeter può agganciarsi a framework esterni tramite connettori, questo nel caso in cui si vogliano salvare i risultati in database o visualizzare i dati in applicazioni più sofisticate.

Apache JMeter offre la possibilità di distribuire gli utenti su più nodi, soluzione utile soprattutto quando il numero di utenti da simulare è troppo alto per essere gestito da un unico nodo. Infatti, la saturazione delle risorse computazionali può compromettere la validità dei risultati ottenuti dai test, in quanto le valutazioni non sono basate unicamente sulla capacità del server, ma anche sulla capacità del client JMeter. L'architettura distribuita di JMeter, come mostrato in Figura 34, presenta una topologia master-slave, con il JMeter Master che invia ad ogni JMeter Slave una copia del descrittore del test, i quali, a loro volta, eseguono il test.

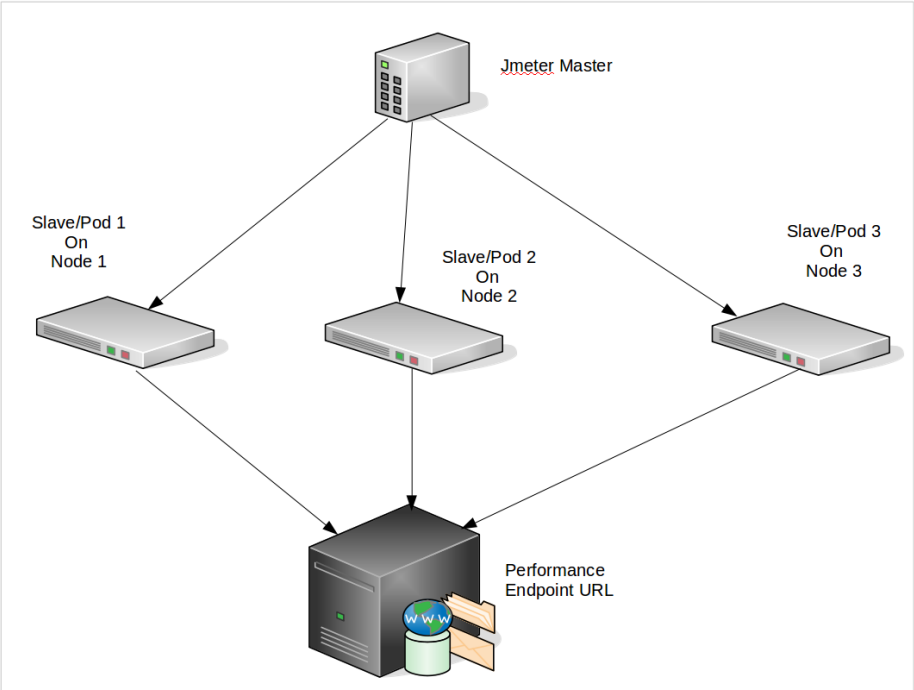

Figura 34 - Architettura di JMeter distribuito

Come descritto precedentemente, i test devono essere realizzati basandosi sulla natura del caso applicativo. Tipicamente, la trasformazione di immagini è un processo che richiede un uso intensivo della CPU, soprattutto per operazioni quali traslazione e rotazione, e in scenari industriali questo tipo di servizio potrebbe essere richiesto contemporaneamente da più macchine. Per queste ragioni, le piattaforme devono essere valutate con processi

I/O-bound, ovvero con processi che effettuano numerose operazioni in ingresso e in uscita, e devono essere valutate anche con processi CPU-bound, ovvero con processi che richiedono un utilizzo intensivo della CPU. Entrambe le sperimentazioni sono state effettuate con Apache JMeter, utilizzato come strumento per generare le richieste, lavorando però con algoritmi differenti e prendendo in considerazione metriche differenti.

Le sperimentazioni sono state eseguite su un cluster di cinque nodi, necessario per simulare un vero ambiente di produzione. In particolare, per ogni nodo:

- OS – Ubuntu Server 18.04 64-bit
- vCPU – 6 core Intel Xeon E5-2603 @ 1.60GHz
- Memoria – 16GiB

4.3.1 Consumo di risorse in idle

Le quattro piattaforme analizzate mostrano consumi differenti di risorse. Al completamento della messa in esecuzione, ogni framework ha mostrato metriche diverse, soprattutto a causa delle diverse architetture. Il primo parametro preso in considerazione è stato l'utilizzo della CPU, che però non ha presentato risultati interessanti. Infatti, ogni piattaforma, in idle, non richiede particolari quantità di CPU, ed è possibile trascurare le minime fluttuazioni rilevate, in quanto irrilevanti al fine di valutare l'efficienza di ogni piattaforma. Al contrario, la memoria si è dimostrata un parametro importante, in quanto ogni piattaforma ne richiede una certa quantità non trascurabile a seconda dei componenti messi in esecuzione dal framework.

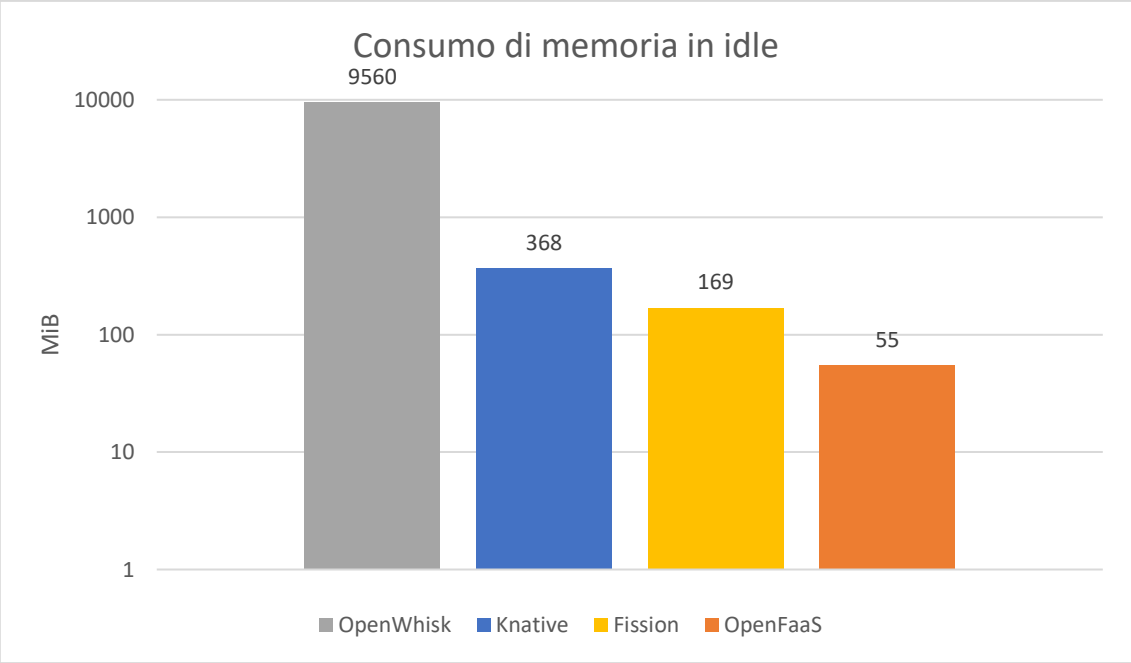

Figura 35 - Quantità di memoria occupata in stato di idle

In Figura 35 è mostrato il consumo di memoria per ogni piattaforma, prendendo in considerazione l'intera piattaforma, più una funzione Hello World configurata e pronta ad essere invocata. OpenWhisk registra il consumo di memoria, arrivando a toccare i 10 GiB di memoria allocata. Le altre piattaforme mostrano risultati di almeno un ordine di grandezza inferiore: Knative occupa 368 MiB, Fission ne occupa 169 e OpenFaaS 55.

4.3.2 I/O-bound

L'approccio alle sperimentazioni è stato quello di aumentare gradualmente il numero di utenti che interagiscono con la piattaforma, per capire il grado di sopportazione del framework FaaS. Ogni utente invia una quantità non limitata di richieste, con un delay tra una richiesta e l'altra di 400 millisecondi, per favorire uno scenario più realistico ed evitare di pesare troppo sulle risorse computazionali del client JMeter. I risultati sono stati catturati da un listener integrato in JMeter, capace di raccogliere informazioni su:

- Numero di richieste inviate (samples);
- Tempo minimo di risposta (min);
- Tempo massimo di risposta (max);
- Tempo medio di risposta (average);
- Variazione rispetto alla media (standard deviation);
- Percentuale di richieste fallite (errors);
- Numero di richieste gestite al secondo dal server (throughput).

I dati più importanti, che verranno riportati nei grafici successivi, sono relativi al numero di richieste inviate, il tempo medio di risposta, la percentuale di richieste fallite e il numero di richieste gestite al secondo dal server. Il numero di utenti è stato scelto in base alle possibilità della strumentazione in possesso, e alla consapevolezza che un numero di utenti troppo alto sarebbe stato inutile ai fini delle sperimentazioni. Sono stati simulati 10, 50, 100, 500 e 1000 utenti concorrenti, mentre la durata dei test è stata scelta in base al tempo necessario per gestire tutti i thread e al tempo per avere informazioni sufficienti per classificare le piattaforme.

JMeter genera un nuovo utente ogni secondo, dunque nel test da 100 impiega circa 1 minuto e 40 secondi per generare tutti gli utenti, mentre impiega circa 17 minuti per generare gli utenti nel test da 1000. Per queste ragioni la durata del test ha richiesto il tempo di generazione dei thread, più il tempo per simulare effettivamente le richieste degli utenti. I primi tre test, quelli da 10, 50, 100, durano circa dieci minuti, mentre i test da 500 e 1000 durano rispettivamente 1000 secondi (circa 15 minuti) e 2000 secondi (circa mezz'ora). In questo modo, si è potuto analizzare correttamente metriche sia in transitorio che a regime, simulando un reale scenario di produzione. Con questi valori temporali crescono anche le richieste effettuate alle piattaforme:

- 10 utenti: circa quindicimila richieste in 10 minuti;
- 50 utenti: circa settantamila richieste in 10 minuti;
- 100 utenti: circa centocinquantamila richieste in 10 minuti;
- 500 utenti: circa seicentocinquantamila richieste in 15 minuti;
- 1000 utenti: circa due milioni cinquecentomila richieste in 30 minuti.

Throughput

Nel grafico a barre, mostrato in Figura 36, è possibile valutare il throughput, ovvero la quantità di richieste elaborate al secondo, raggiunto da ogni piattaforma al variare del numero di utenti che effettuano le richieste. Nel primo test con dieci utenti non è possibile notare differenze tra i diversi framework, in quanto i quattro competitor raggiungono le venti richieste al secondo, senza evidenti forbici da segnalare. Nel test con cinquanta utenti, OpenWhisk mostra prestazioni inferiori, raggiungendo le 79,8 richieste al secondo, mentre le altre piattaforme superano le cento richieste al secondo. Con cento utenti, è possibile evidenziare un vantaggio di Knative rispetto a OpenFaaS e Fission, in quanto questo raggiunge le 221,9 richieste al secondo, e supera di circa venti punti gli altri concorrenti.

Nel primo test importante con cinquecento utenti è evidente la difficoltà di OpenWhisk, che non supera le duecento richieste al secondo, e il vantaggio di Knative, che sfiora gli ottocento punti, mentre OpenFaaS e Fission si mantengono rispettivamente sui 554,2 punti e sui 535,9 punti. Nell'ultimo test con mille utenti, OpenWhisk aumenta di poco il punteggio ottenuto nel test precedente, raggiungendo le 223,4 richieste al secondo, mentre Knative sfiora le mille richieste al secondo, superando di circa quattrocento punti sia OpenFaaS che Fission.

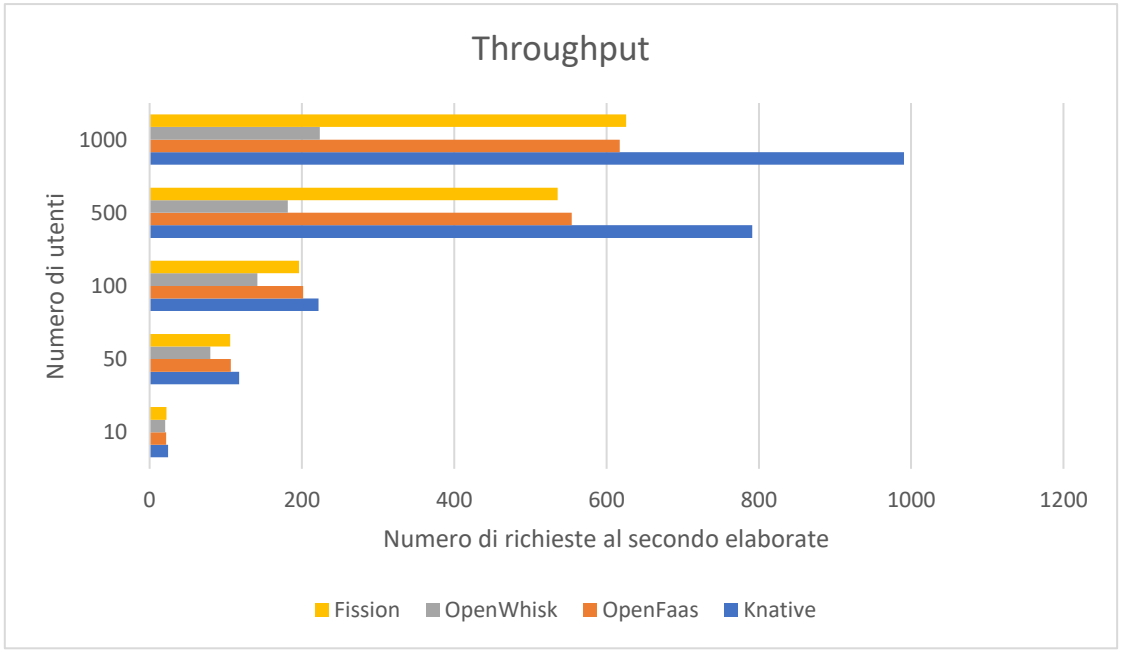

Figura 36 - Throughput in cluster di produzione

Errori

Il grafico, presentato in Figura 37, mostra la quantità di errori rilevati durante l'esecuzione dei test, rappresentati come percentuale rispetto alla quantità totale di risposte ricevute. Le prime tre sperimentazioni con dieci, cinquanta e cento utenti non hanno mostrato nessun errore per nessuna piattaforma. Con cinquecento utenti, l'unica piattaforma ha descrivere un punteggio non nullo è OpenFaaS, con una percentuale di errori dello 0,01%. Il test con mille utenti mostra Knative come la piattaforma con più errori rilevati, raggiungendo lo 0,25% di errori, mentre OpenWhisk è la piattaforma con il minor numero di errori, mostrando lo 0,06%. Infine, OpenFaaS raggiunge lo 0,17%, mentre Fission arriva allo 0,08%.

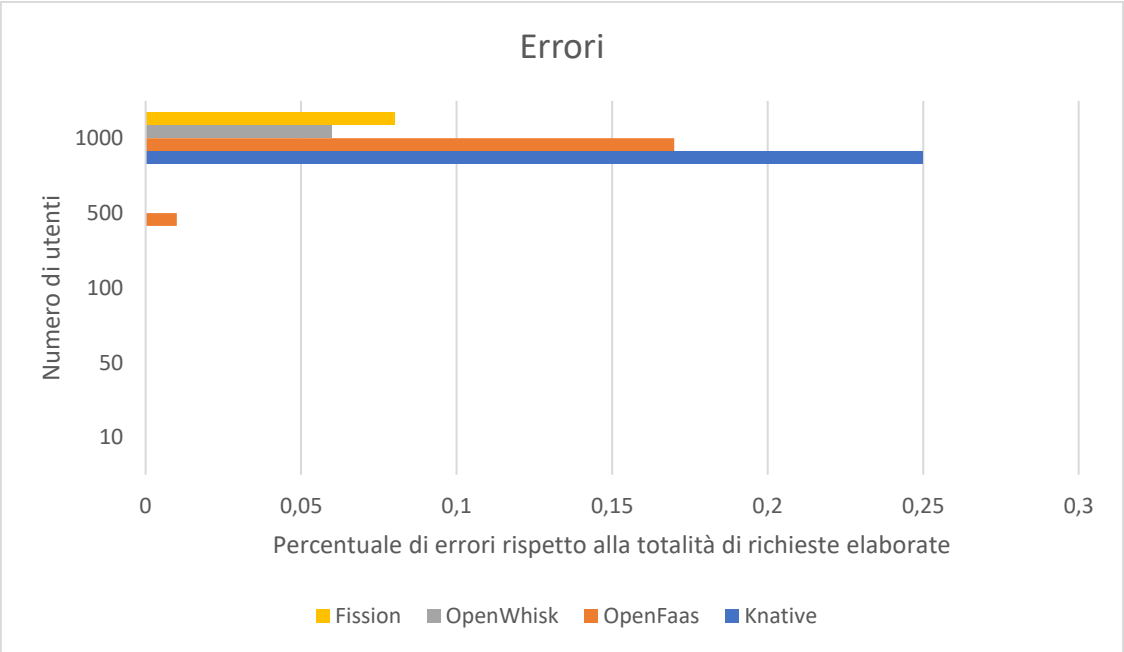

Figura 37 - Errori rilevati rappresentati in percentuale (Errori 5xx)

Tempo di risposta medio

Il grafico, visibile in Figura 38, descrive il tempo di risposta medio in millisecondi per l'elaborazione di ogni richiesta ricevuta. Il test con dieci utenti mostra Knative con un tempo di risposta di 9 millisecondi, circa quattro volte inferiore rispetto al punteggio raggiunto da OpenFaaS e Fission, e circa otto volte inferiore rispetto a quello raggiunto da OpenWhisk. Con cinquanta utenti, i punteggi di Knative, OpenFaaS e Fission restano

invariati, mentre OpenWhisk raggiunge i 197 millisecondi, più del doppio rispetto al test precedente. Nel test con cento utenti la situazione è analoga al test precedente, con OpenWhisk che allarga la forbice e raggiunge i 245 millisecondi.

Con cinquecento utenti è possibile valutare i primi cambiamenti, con Knative che raggiunge i 73 millisecondi, OpenFaaS che arriva a 232 millisecondi, Fission di poco sotto con 280 millisecondi, ed infine OpenWhisk che supera abbondantemente il secondo, arrivando a 1643 millisecondi. Infine, con mille utenti Fission e OpenFaaS offrono gli stessi punteggi, raggiungendo circa i settecentocinquanta punti. Knative non supera i cinquecento millisecondi, mentre OpenWhisk supera i tre secondi di media, raggiungendo i 3256 millisecondi.

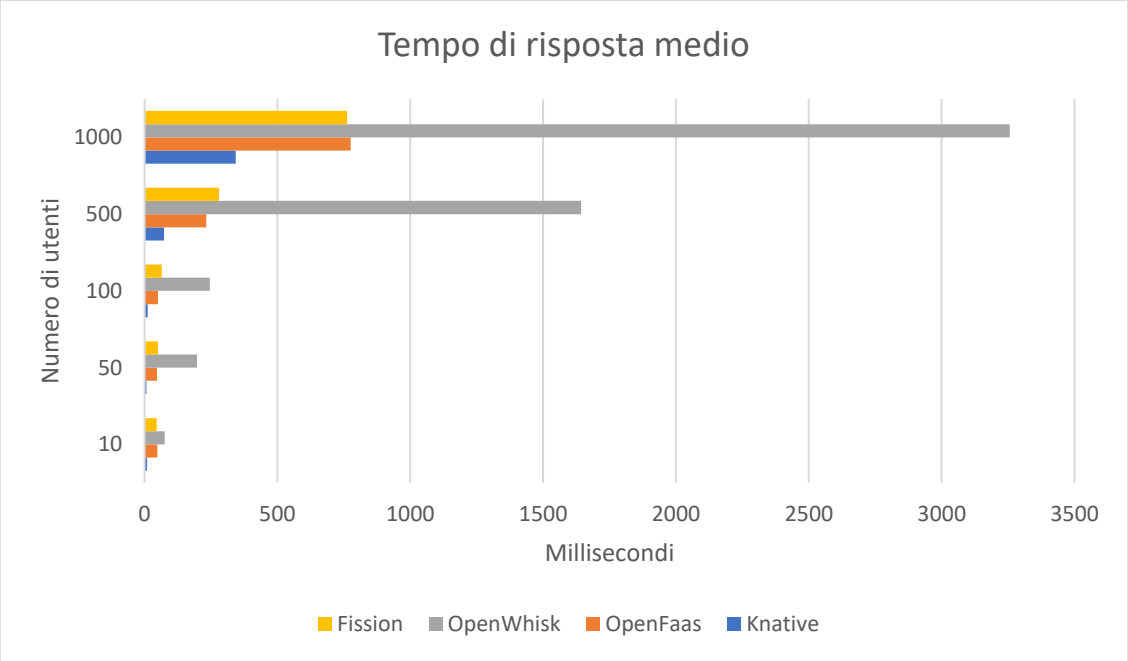

Figura 38 - Tempo di risposta medio in millisecondi

4.3.3 CPU-bound

Le seguenti sperimentazioni sono state progettate per poter analizzare l'utilizzo di risorse da parte delle piattaforme di fronte a workload computazionalmente onerosi. La scelta di questo tipo di test è da ricondursi al caso applicativo che seguirà alla trattazione, in quanto le operazioni di image processing richiedono tipicamente un importante utilizzo della CPU. Per queste ragioni, si è deciso di realizzare una funzione mock, la quale richiede una gran quantità di CPU, in modo da focalizzare l'attenzione sul consumo di questa risorsa. La scelta è ricaduta sul prodotto fattoriale di grandi numeri, visibile nello Snippet 1, il quale è risultato essere un algoritmo molto semplice da implementare ma anche molto efficace per la valutazione della gestione CPU da parte delle piattaforme.

```java
package it.unibo;

import java.math.BigInteger;

public class Main {

    public static void main(String[] args) {
        int n = 100000;

        String fact = factorial(n);
    }

    public static String factorial(int n) {
        BigInteger fact = new BigInteger("1");

        for (int i = 1; i <= n; i++) {
            fact = fact.multiply(new BigInteger(i + ""));
        }

        return fact.toString();
    }
}
```

Snippet 1 – codice java per eseguire il fattoriale di 100000

Data la necessità di offrire questo tipo di servizio a più utenti contemporaneamente, si è optato per dividere i test in base al numero di utenti che effettuano la richiesta, diminuendo però i numeri rispetto alle sperimentazioni precedenti. Le sperimentazioni sono divise per ambienti con 10, 20, 50 e 100 utenti, ed in ognuno di questi test viene

generato un nuovo utente al secondo, ed ognuno di questi invia una sola richiesta HTTP e rimane in attesa della risposta. Gli aspetti che si vogliono catturare sono:

- Tempo totale di esecuzione (inclusa creazione e terminazione dei Pod);
- Utilizzo medio della CPU;
- Utilizzo massimo della CPU.

Le seguenti sperimentazioni sono state eseguite con Apache JMeter, come per i test precedenti, e la durata dei singoli test non è stata stabilita a priori, bensì il test termina quando viene restituita l'ultima richiesta.

Tempo di esecuzione

Il grafico, visibile in Figura 39, mostra il tempo totale di esecuzione per l'elaborazione delle richieste, considerando anche il tempo di avviamento e terminazione dei pod dedicati. Il test con dieci utenti mostra Knative raggiungere il punteggio migliore, impiegando poco più di due minuti per portare a termine le operazioni, e vede OpenFaaS come il peggiore, in quanto il framework richiede 3 minuti e 52 secondi per completare i task assegnati, Fission e OpenWhisk seguono rispettivamente con 263 secondi e 232 secondi. Con venti utenti OpenWhisk consegue il risultato peggiore, impiegando 381 secondi per terminare con successo le richieste, segue OpenFaas con 288 secondi e Fission con 277 secondi, e, infine, ancora primo Knative con 193 secondi.

Nel terzo test con cinquanta utenti Knative mantiene all'incirca lo stesso punteggio del test precedente, mentre Fission e OpenFaaS raggiungono rispettivamente i 342 secondi e i 372 secondi. Sempre ultimo OpenWhisk, che impiega circa nove minuti per completare il test. Infine, con cento utenti Knative conferma la prima posizione, impiegando 306 secondi per terminare tutte le operazioni, segue OpenFaaS con 393 secondi, Fission con 441 secondi, ed infine OpenWhisk con 681 secondi.

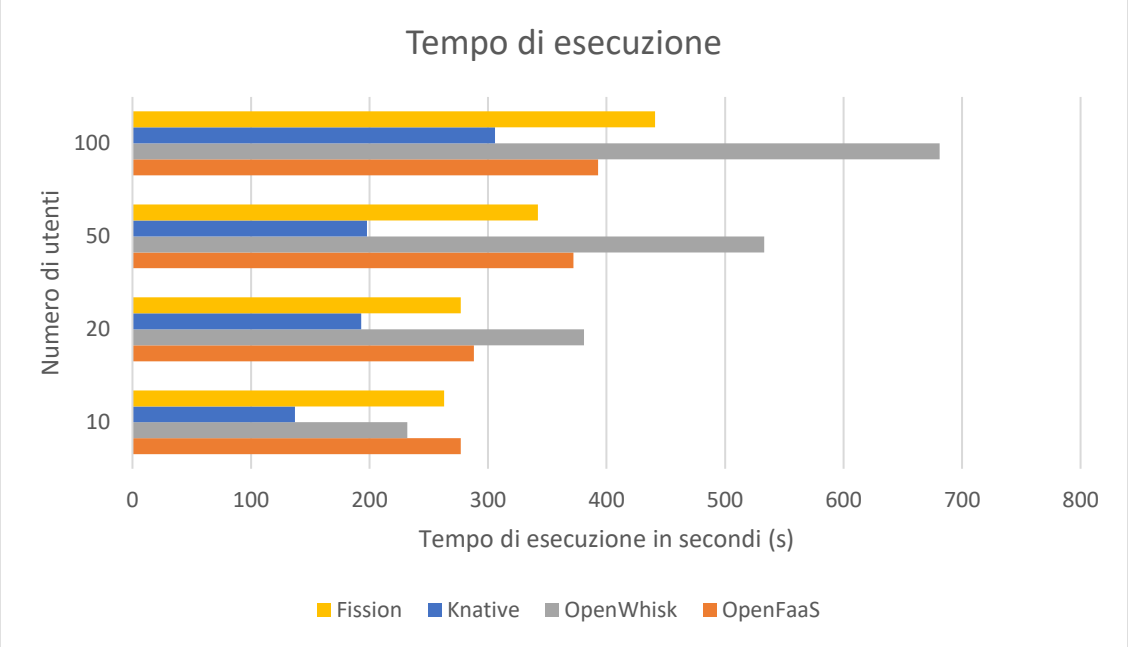

Figura 39 - Tempo di esecuzione in secondi

Utilizzo medio della CPU

Il grafico, descritto in Figura 40, mostra l'utilizzo medio della CPU durante lo svolgimento delle sperimentazioni. In particolare, le CPU sono descritte come vCPU, o virtual CPU, dove l'astrazione in vCPU permette ai software di virtualizzazione di generare un numero superiore di CPU virtuali rispetto a quelle fisiche effettivamente disponibili. La conversione tra utilizzo della CPU fisica ed utilizzo della CPU virtuale viene effettuata prendendo in considerazione il tempo dedicato ad una specifica macchina virtuale. Per esempio, quattro CPU possono essere mappate su un processore single-core virtuale, che potrà arrivare al massimo ad una vCPU di utilizzo.

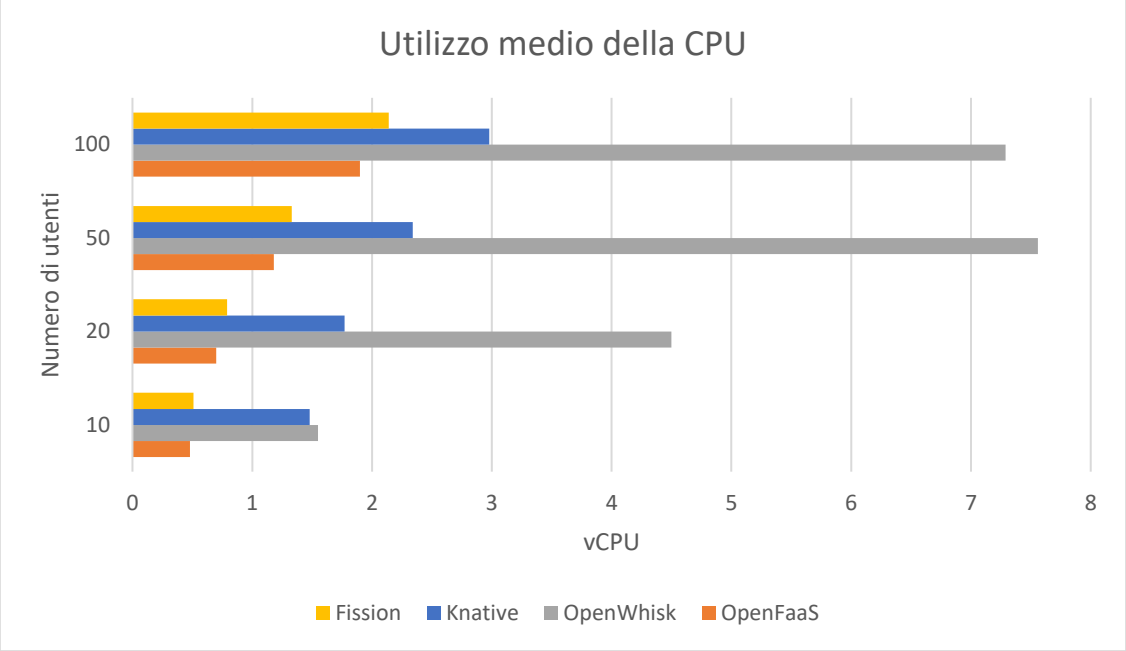

Figura 40 - Utilizzo medio della CPU misurato in vCPU

Il primo test da dieci utenti mostra OpenFaaS e Fission come i framework con la minore necessità computazionale, raggiungendo rispettivamente circa 500 millicore, o 0,5 CPU virtuali, utilizzate in media. Seguono Knative e OpenWhisk, con una CPU e mezza virtuale consumata per l'esecuzione del test. Con venti utenti, OpenFaaS e Fission mantengono basso il consumo di CPU, arrivando a circa 0,7 vCPU utilizzate, mentre Knative ne richiede 1,77 CPU virtuali. OpenWhisk triplica la richiesta di CPU, arrivando a 4,5 vCPU consumate.

Il test da cinquanta utenti vede OpenFaaS richiedere la minor quantità di CPU con 1,18 vCPU, seguito da Fission con 1,33 vCPU e Knative con 2,34 vCPU. OpenWhisk mostra

il risultato peggiore, arrivando a superare le sette CPU virtuali. Infine, con 100 utenti, OpenWhisk rimane ultimo con 7,29 CPU virtuali, mentre Knative, Fission e OpenFaaS raggiungono rispettivamente 2,98 vCPU, 2,14 vCPU e 1,9 vCPU utilizzate.

Utilizzo massimo della CPU

Il grafico, illustrato in Figura 41, mostra il massimo consumo di CPU virtuali registrato durante le sperimentazioni. Il test con dieci utenti mostra OpenFaaS e Fission con il picco di CPU più basso, rispettivamente 0,55 vCPU e 0,61 vCPU, seguono 1,71 vCPU raggiunti con OpenWhisk e 1,8 vCPU con Knative. Con venti utenti, OpenFaaS rimane il framework con il punteggio migliore grazie alle 0,82 vCPU utilizzate, segue vicino Fission con 0,96 vCPU consumate. Knative raggiunge le 1,99 vCPU utilizzate, mentre OpenWhisk supera le sei CPU virtuali utilizzate durante il test.

Nella sperimentazione con cinquanta utenti, OpenWhisk raggiunge le 13,4 vCPU. Seguono Knative con 3,32 vCPU, Fission con 1,87 vCPU e, infine, OpenFaaS che mostra ancora il punteggio migliore con 1,72 vCPU. Il test finale con cento utenti vede OpenFaaS e Fission superare le tre CPU virtuali registrate per il picco massimo raggiunto, Knative segue con 5,04 vCPU, mentre OpenWhisk raggiunge le 15,31 CPU virtuali.

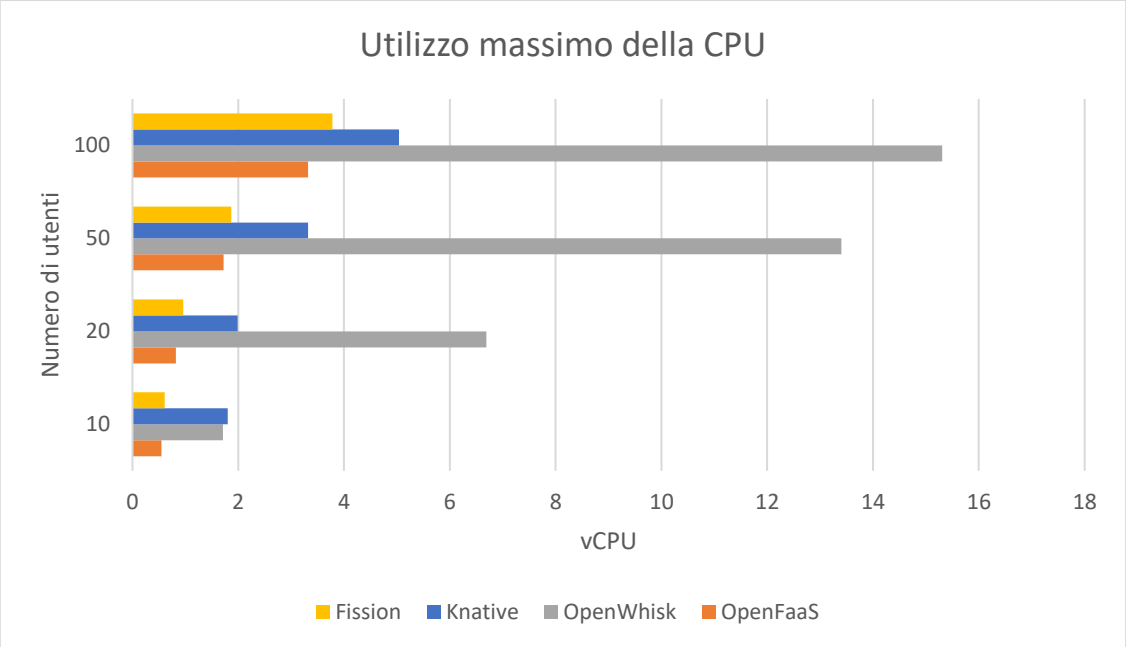

Figura 41 - Utilizzo massimo di CPU misurato in vCPU

4.3.4 Confronto quantitativo

Il confronto quantitativo tra le piattaforme ha portato alla luce limitazioni e punti di forza, fondamentali per la scelta della soluzione da adottare per il caso d'uso. Le sperimentazioni I/O intensive e CPU intensive hanno mostrato risultati convincenti, e nessuna piattaforma ha avuto risultati discordanti tra le due tipologie di test, risultando in una coerenza richiesta ed attesa negli scenari presi in esame.

Partendo dalla quantità di memoria allocata in idle, OpenWhisk raggiunge circa dieci GiB rispetto ai 300 MiB allocati da Knative. La motivazione risiede nell'architettura offerta dalla piattaforma, la quale istanzia per ogni nodo un Invoker che permette di eseguire i container, oltre che ad utilizzare Apache CouchDB come database ed altri componenti che aumentano drasticamente l'utilizzo della memoria. Un'analisi più profonda ha mostrato come il database di Apache consumi oltre quattrocento MiB, ma soprattutto ha portato alla luce come ogni Invoker tenga pronto un certo pool di container per garantire tempi di avviamento inferiori. Nel cluster preso in esame, ogni nodo presenta un Invoker, che ha sua volta mantiene in memoria due Pod con i relativi ambienti di esecuzione, pronti ad accettare le richieste dal message broker. L'immagine offerta da OpenWhisk per eseguire l'ambiente di esecuzione Node 12 consuma oltre 1GiB, probabilmente a causa dell'implementazione del sistema operativo Debian che occupa oltre 700MiB. Altre piattaforme riducono l'impatto utilizzando runtime di Node basati sul sistema operativo Alpine, il quale permette di ridurre i consumi fino a dieci volte, considerato che l'immagine Docker di questa piattaforma consuma circa 50MiB. Considerati i cinque nodi del cluster, e preso in considerazione che ognuno di questi mantiene in memoria due Pod con i relativi ambienti di esecuzione, è facile notare la presenza di dieci Pod con un consumo complessivo di circa 10GiB, senza considerare il costo derivato dagli altri componenti. La necessità di OpenWhisk di eseguire a priori gli Invoker, destinati all'esecuzione delle action, è significativa, perché a fronte di maggiori garanzie e controlli sul singolo nodo si contrappone un costo importante, da prendere in considerazione durante la scelta della piattaforma.

Al contrario, Knative offre un ambiente di esecuzione più leggero rispetto a quello offerto da OpenWhisk, nonostante paghi il prezzo di due componenti importanti: il load balancer e la service mesh. Infatti, per poter utilizzare Knative come piattaforma serverless, è stato necessario eseguire la service mesh Istio per il routing delle richieste ricevute, la quale offre numerose funzionalità a costo di una certa quantità di memoria allocata. Inoltre, Istio richiede un load balancer per il corretto assegnamento di un indirizzo IP all'ingresso della service mesh. In fase di testing non è stato possibile adottare il servizio NodePort offerto da Kubernetes, in quanto Istio utilizza il campo Host presente negli header HTTP per smistare le richieste, e non è possibile assegnare indirizzi logici ad una porta. Per

queste ragioni, il carico combinato di load balancer, service mesh e servizi core di Knative Serving hanno portato a circa 300MiB di memoria occupata.

Fission ed OpenFaaS raggiungono risultati interessanti, in quanto entrambe le soluzioni consumano una quantità trascurabile di memoria occupata, ideale soprattutto in scenari con quantità limitate di risorse disponibili. I componenti core delle due piattaforme gestiscono operazioni di routing, di logging e garantiscono la creazione e la terminazione dei pod generati in caso di richieste, ottimizzando i singoli componenti per offrire buone performance ed un basso utilizzo di risorse.

Per quanto riguarda i test I/O intensive, i risultati raggiunti nella prima parte delle sperimentazioni offrono spunti interessanti in vista della scelta della piattaforma. Knative ha mostrato metriche di efficienza e latenza irraggiungibili, sintomo di una piattaforma matura e pronta all'esecuzione in scenari industriali. Le motivazioni dei punteggi riscontrati sono da ritrovare nelle performance della service mesh Istio, la quale riesce a garantire il corretto smistamento delle richieste in circa 6 millisecondi per il novantesimo percentile in presenza di settantamila richieste al secondo [82]. Sono risultati importanti, motivo per il quale Istio è spesso utilizzato come service mesh per grandi aziende che arrivano a controllare anche centinaia di migliaia di servizi. Un'altra motivazione è da riscontare nel continuo supporto che molte organizzazioni stanno offrendo alla piattaforma, complice le promesse di un supporto universale per workload serverless in Kubernetes. Il tempo di risposta medio, insieme al throughput raggiunto lo rende una scelta obbligata in presenza di numerose richieste, vincolo presente in industrie dove macchine, sensori e attuatori cooperano e comunicano continuamente con i servizi offerti dal cluster di produzione. Il numero di errori è trascurabile, e può essere ridotto lavorando sulle configurazioni offerte, senza contare le migliorie che ogni giorno vengono applicate sul framework.

Fission, nonostante i risultati soddisfacenti, ha presentato comportamenti anomali, soprattutto nella gestione del ciclo di vita dei Pod, in quanto questi richiedono più tempo del previsto per terminare, occupando inutilmente la memoria nel cluster. Anche OpenFaaS ha mostrato, come Fission, risultati interessanti, dimostrandosi però una piattaforma ben più matura della precedente, replicando e terminando correttamente i Pod, oltre che a rilasciare correttamente la memoria al termine delle operazioni effettuate. Il supporto offerto dalla community ad OpenFaaS è maggiore rispetto a quello di Fission, e questo è possibile riscontarlo nella semplicità e nell'utilizzo della piattaforma di Alex Ellis, che nel corso di questi primi test ha lavorato senza mostrare alcun problema. In ogni caso, i risultati ottenuti da entrambe le piattaforme sono molto simili, e dunque la scelta del vincitore tra queste due soluzioni dovrebbe basarsi sull'affidabilità offerta, dove i punteggi relativi agli errori rilevati mettono OpenFaaS in difficoltà e Fission in vantaggio, nonostante i bug riscontrati nella piattaforma supportata da Platform9.

Discorso a parte per OpenWhisk, che ha sofferto l'evidente mancanza di risorse per sostenere i test effettuati. Il collo di bottiglia, evidente nel framework di IBM, deriva da Apache Kafka, componente fondamentale per lo smistamento delle richieste, in quanto i requisiti minimi richiesti sono molto più alti di quelli forniti dall'ambiente di esecuzione. In situazioni comuni, Apache Kafka può lavorare tranquillamente con 6GB di RAM, ma in situazioni di stress load la quantità necessaria può salire fino a 32GB, ben oltre le capacità offerte dal cluster. Inoltre, Kafka coordina e mantiene il proprio lavoro su disco, dunque per garantire elevate performance è necessario offrire al sistema installazioni in RAID, magari coadiuvando il cluster di dischi a stato solido [83].

I risultati registrati nelle sperimentazioni CPU intensive mostrano una forte correlazione con i test I/O intensive riportati precedentemente. OpenWhisk, nei test I/O intensive, ha mostrato forti limitazioni dovute alla mancanza di risorse richieste dal message broker Apache Kafka, che si è rivelato essere il collo di bottiglia per il throughput e il tempo medio di risposta offerto dalla piattaforma. Le sperimentazioni CPU intensive non vertono sulle capacità di smistamento degli eventi, quindi le scarse performance sono da ricondursi ad una mancata ottimizzazione degli Invoker, componenti addetti alla gestione dei container per l'esecuzione della funzione richiesta. Durante i test, OpenWhisk ha generato un numero importante di Pod, numero molto più alto rispetto a quelli riportati dalle altre piattaforme, con conseguente overhead dovuto alla creazione ed alla terminazione dei container generati. Inoltre, è possibile definire il numero di Invoker da istanziare nell'ambiente di esecuzione, ma non è possibile decidere la quantità di Pod che questi andranno a generare, perdendo la possibilità di ottimizzare l'intero cluster.

Le altre piattaforme raggiungono ottimi risultati, rimanendo stabili durante l'intera esecuzione dei test e registrando utilizzi delle risorse accettabili. Knative mostra tempi di esecuzioni inferiori pagando con un maggior consumo di risorse, mentre Fission e OpenFaaS impiegano più tempo per l'intera esecuzione dei test, ma registrano consumi inferiori. In ogni caso, la forbice tra Knative e le altre due piattaforme diminuisce all'aumentare degli utenti, questo perché Knative paga soprattutto la creazione dei container, meno ottimizzati rispetto alle funzioni proprietarie scritte per le singole soluzioni FaaS. Infatti, in Knative lo sviluppatore scrive il microservizio e il file per il deployment del container Docker, perdendo gran parte dell'astrazione offerta dai middleware FaaS.

5 Caso di studio: elaborazione distribuita di immagini per l'industria 4.0

Il caso applicativo preso in esame riguarda principalmente l'incontro delle tecnologie applicate all'industria, in particolare quelle operanti nell'ambito manufatturiero, con i vantaggi offerti da applicazioni serverless. Gli scenari industriali sono numerosi, e la diffusione di questo tipo di architettura è ancora limitato, complice anche la difficoltà di far corrispondere la dimensione IT con la dimensione OT. In questo caso specifico, si vuole proporre un'architettura serverless per l'elaborazione di immagini da imprimere su piastrelle di ceramica, rispettando importanti vincoli di qualità e di latenza. La necessità di elaborazioni così precise deriva dalla scelta diffusa di marketing di offrire prodotti personalizzati per ogni utente, secondo le pratiche dell'hyper personalizzazione. Le stampe prodotte possono raggiungere anche i 15 gigabyte, e devono essere realizzate secondo il regime di produzione dell'azienda, spesso con livelli di throughput molto elevati. In questo contesto, è fondamentale una elevata potenza computazionale da applicare ad una elevata mole di dati, per evitare rallentamenti indesiderati all'intera catena di produzione. L'ambiente di esecuzione prevede delle macchine adibite alla stampa delle piastrelle, ognuna fornita di un PC per il monitoraggio ed il controllo dei dati prodotti. Si vogliono applicare gli algoritmi di image processing direttamente sui PC montati sulle macchine, per evitare l'acquisto di hardware delicato e costoso. Le operazioni di image processing tipicamente effettuate sono le trasformazioni geometriche, la compensazione tonale, e le applicazioni di matrici di convoluzione. Inoltre, le applicazioni industriali richiedono un buon livello di availability, ed è quindi necessario che ogni macchina mantenga le proprie caratteristiche di determinismo in modo da svolgere al meglio il compito per la quale è stata progettata. Il cluster non deve influenzare le prestazioni del plant.

5.1 Image processing

La pratica dell'image processing riguarda la manipolazione di immagini tramite l'utilizzo di computer, o, più genericamente, di dispositivi digitali. Le tecniche di manipolazione di immagini sono cresciute esponenzialmente negli ultimi anni, offrendo servizi importanti nell'ambito dell'intrattenimento, della medicina, e della geolocalizzazione. La disciplina dell'image processing è molto vasta, ma principalmente può essere ricondotta all'applicazione di funzioni a due dimensioni $f(x, y)$ con variabili continue. Un'immagine, per essere processata, deve essere divisa in tanti piccoli campioni, chiamati sample, e trasformata in una matrice di numeri. Inoltre, dato che un computer rappresenta

i numeri secondo una certa precisione, è necessario quantizzare questi numeri per poterli rappresentare [84].

La rotazione di un'immagine è un processo molto comune, utilizzato soprattutto per algoritmi di matching o di allineamento con altre immagini. L'input necessario per la corretta rotazione di un'immagine è l'angolo di rotazione θ, ed il punto dell'immagine in cui effettuare la rotazione.

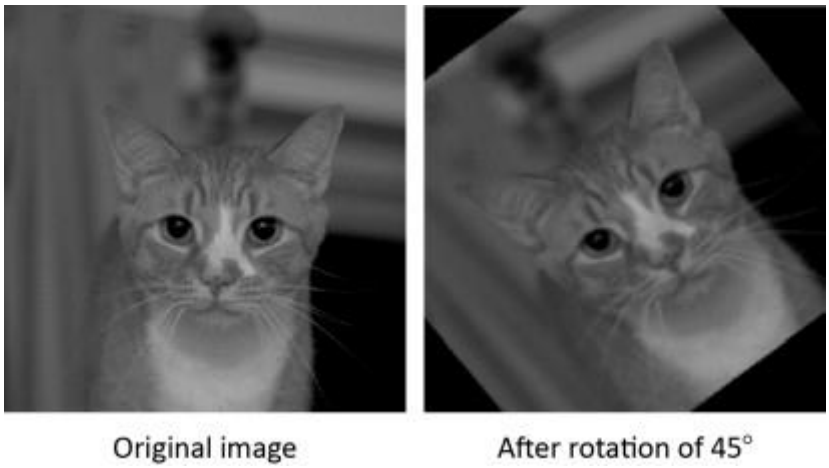

Figura 42 - Un'immagine ruotata di 45° gradi. L'output possiede la stessa dimensione dell'input, e le parti dell'output fuori dal bordo vengono scartate

Le coordinate di un punto qualsiasi (x_1, y_1), quando ruotate di un angolo θ rispetto ad un punto (x_0, y_0) diventano (x_2, y_2), come mostrato nelle seguenti equazioni:

$$x_2 = \cos(\theta) * (x_1 - x_0) + \sin(\theta) * (y_1 - y_0)$$

$$y_2 = -\sin(\theta) * (x_1 - x_0) + \cos(\theta) * (y_1 - y_0)$$

Il calcolo di un nuovo punto (x_2, y_2) può essere effettuato indipendentemente, ovvero non è necessaria la conoscenza del valore delle altre coordinate durante l'esecuzione dell'algoritmo di rotazione. Per questa ragione, è possibile dividere la rotazione complessiva dell'immagine in tanti piccoli task, ognuno dei quali andrà a calcolare la rotazione di un singolo pixel. La rotazione dell'immagine è un ottimo esempio di decomposizione dell'input, dove più worker possono lavorare contemporaneamente per la risoluzione del problema. Problemi di questo tipo vengono definiti algoritmi massicciamente paralleli, e verranno discussi nel paragrafo successivo [85].

5.2 Algoritmi massicciamente paralleli

Una computazione che può essere divisa in parti completamente indipendenti tra loro, ognuna delle quali può essere eseguita in processori separati, viene chiamata massicciamente parallela. Una computazione massicciamente parallela richiede un minimo livello di comunicazione tra le diverse parti, anche se spesso non è richiesto nessun tipo di interazione tra i diversi processori. Esistono molti tipi di computazioni massicciamente paralleli, quali la simulazione di Monte Carlo, algoritmi di ricerca, attacchi di forza brutta in crittografia, algoritmi genetici, e algoritmi di image processing, come in questo caso. Per l'esecuzione di questo tipo di computazioni è necessario definire il task da eseguire, e bisogna assegnare un task ad ogni singolo worker. Nel caso di assegnamento statico, ogni worker conosce a priori il task da computare, mentre in assegnamenti dinamici è necessario avvalersi di una collezione di task da cui i worker possono attingere a runtime [86].

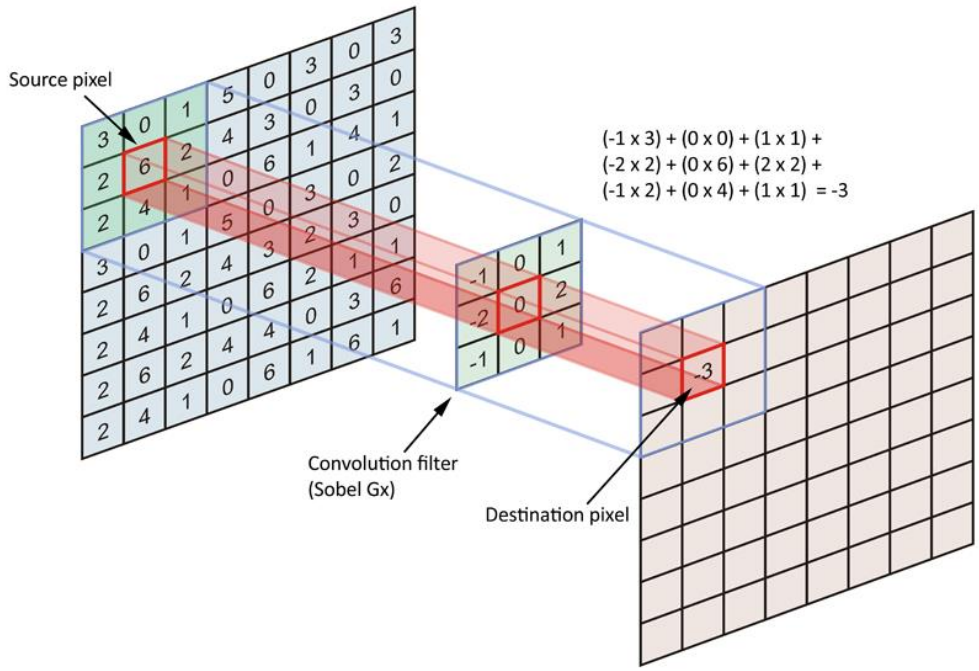

Figura 43 - Esempio di algoritmo massicciamente parallelo: calcolo della matrice di convoluzione, molto utilizzata nel deep learing

L'applicazione di questo tipo di algoritmi è altamente scalabile orizzontalmente, in quanto permette l'assegnazione di task computazionali ad una quantità indefinita di worker, ed è per questo motivo che spesso la computazione viene eseguita in ambienti cloud, dove la quantità di risorse è praticamente infinita. Data l'estrema permeabilità che

il cloud sta avendo in questi ultimi anni in tutti i settori lavorativi, è interessante soffermarsi sul rapporto che è nato tra il cloud computing e l'industria manufatturiera.

5.3 Il cloud e l'Industria 4.0

Negli ultimi anni, il settore secondario ha vissuto una completa trasformazione grazie alla digitalizzazione delle macchine che operano nell'industria manufatturiera. Questa trasformazione ha connotati così importanti da essere stata definita la quarta rivoluzione industriale, o Industria 4.0, e sostanzialmente prevede di interconnettere la totalità dei sistemi operanti per offrire un'ambiente smart, capace di ottimizzare costi e tempo grazie all'elaborazione di dati in tempo reale. Anni prima, con la terza rivoluzione industriale (Figura 44), i computer avevano già arricchito l'arsenale industriale, ma questi lavoravano autonomamente, senza interagire con gli altri calcolatori locali. L'Industria 4.0 è il collegamento tra questi sistemi: computer, sensori, attuatori, dispositivi embedded e access point, tutti connessi sotto la semantica dell'IoT (Internet of Things), ovvero ogni macchina ha la possibilità di trasmettere dati verso qualsiasi rete [87].

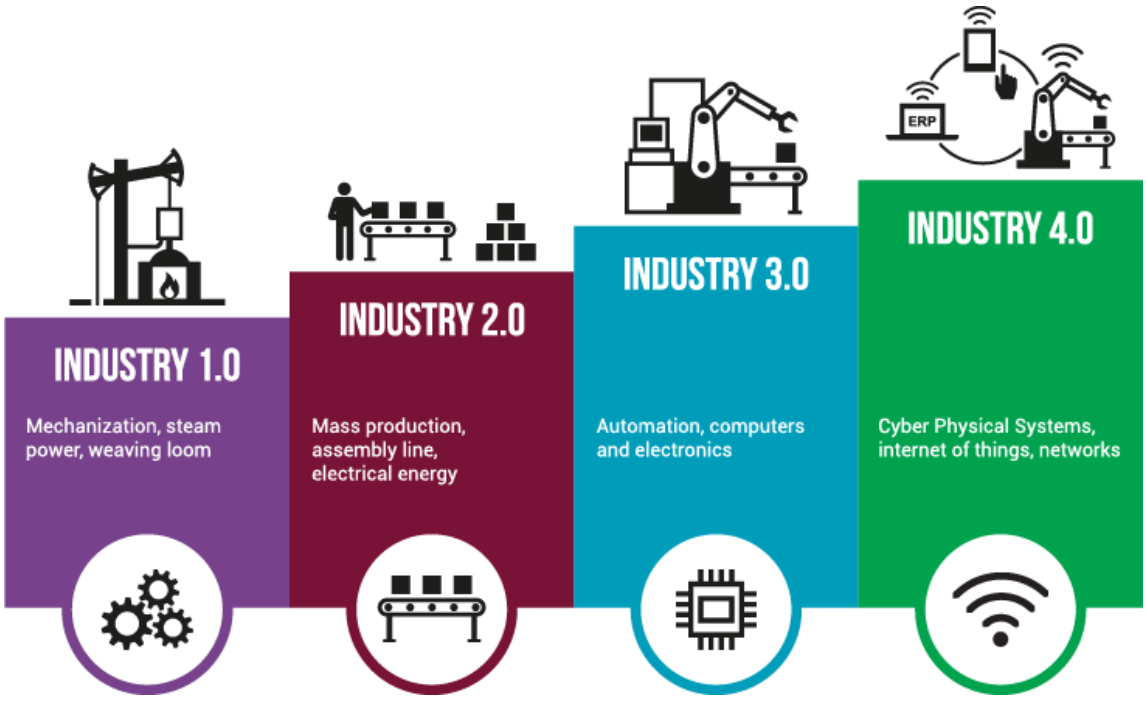

Figura 44 - Evoluzione tecnologica dell'industria

L'Industria 4.0 identifica nuove opportunità di mercato, grazie all'elaborazione di dati con tecniche di data processing, capaci di processare centinaia di gigabyte al secondo. La

profondità delle informazioni elaborate permette ottimizzazioni in tempo reale sulla produzione, su eventuali modifiche da applicare per migliore la supply chain ed ottimizzare le operazioni logistiche, complici anche l'introduzione di nuovi algoritmi di intelligenza artificiale e machine learning che garantiscono completa autonomia alle macchine in produzione, con la capacità di auto migliorarsi.

Sistemi IT on-premise o servizi cloud privati non riescono a gestire una tale quantità di informazioni, se non in maniera inefficiente e poco scalabile. Per queste ragioni, il processamento di una tale quantità di dati viene garantito dai servizi cloud pubblici, capaci di scalare facilmente in qualsiasi scenario di produzione industriale, offrendo nuove prospettive di business e accelerando l'ingresso di nuove idee nel mercato [88] [87].

In ogni caso, i vincoli imposti dal caso applicativo rendono la latenza derivante dalla distribuzione del carico computazionali in nodi distanti dalla produzione troppo elevata e non adatta a sostenere l'importante throughput richiesto dal plant. In scenari industriali, queste situazioni sono sempre più comuni e nuove soluzioni stanno affiorando per far fronte all'eccessivo tempo di latenza derivato dalla distanza tra centro di produzione e data center.

5.4 Edge computing e l'Industria 4.0

Il cloud rappresenta una quantità illimitata di risorse, spesso distribuita in enormi data center sparsi per il mondo. *AWS*, per esempio, distribuisce il carico in Region, ovvero aree geografiche distinte, ed ogni Region possiede una o più Availability Zone, ovvero data center in cui vengono computate le risorse. Ogni Region è completamente indipendente dall'altra, e le Availability Zone presenti nella Region sono interconnesse con interfacce a bassissima latenza [89].

I data center, data la loro importanza strutturale, sono pochi e spesso lontani dall'azienda che ne richiede i benefici. Inoltre, spesso i dati da elaborare non richiedono ingenti risorse, oppure sono troppo sensibili per computarli fuori dal perimetro aziendale. Per queste ragioni, è nata una nuova filosofia, chiamata edge computing, che consiste nell'operare localmente alla sorgente dei dati, piuttosto che inviarli in data center distanti. Inoltre, quando per certe operazioni la latenza risulta essere un parametro fondamentale rispetto alla velocità di computazione, l'edge computing è l'unica soluzione percorribile, data la vicinanza rispetto alla sorgente di dati da elaborare.

L'edge non sostituisce il cloud, bensì i due ambienti cooperano in base al tipo di operazione da eseguire. In scenari applicativi, dove interagiscono numerosi componenti

in tempo reale, quali giochi in multiplayer, veicoli a guida autonoma, o realtà aumentata, l'edge è la scelta consigliata, data la latenza infinitesima offerta. Per elaborazioni più pesanti, dove è necessario un potere computazionale impossibile da ottenere in locale, il cloud resta la soluzione migliore, soprattutto quando le elaborazioni non sono richieste in real time. Nonostante questo, sempre più domini applicativi sfruttano i vantaggi offerti dall'edge computing, data la necessità di realizzare servizi sempre più reattivi e dinamici, offrendo al mercato tecnologie sempre più innovative e vicine al consumatore [90].

Per edge computing si intende la distribuzione del carico verso i "bordi" della rete, ovvero verso i nodi vicini alle risorse che producono i dati da elaborare. In industria, queste risorse possono essere macchine automatizzate, robot, controllori, sensori di rilevazione, e tutti questi dispositivi insieme producono centinaia di migliaia di interazioni. Elaborare i dati vicino a questi dispositivi, quando possibile, risulta essere estremamente più rapido ed efficiente.

Molti tipi di macchine industriali possono beneficiare di latenze così basse, soprattutto quelle che producono numerose quantità di dati, come alcune macchine che generano dati relativi allo stato interno, come la quantità di vibrazione, la temperatura, i giri al minuto, o i consumi energetici [91].

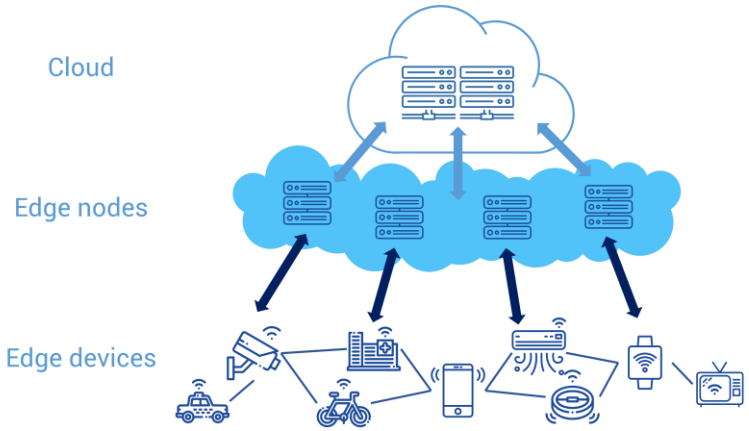

Figura 45 - Edge computing come interfaccia tra IoT e il cloud

I dispositivi edge possono essere numerosi e molto diversi tra loro (Figura 45). Solitamente, sono dispositivi che consumano minime quantità di energia, sono estremamente efficienti, e permettono il monitoraggio di costose macchine industriali. L'applicazione di pratiche di edge computing nell'industria manufatturiera permette di rendere interi impianti di produzione completamente autonomi, senza il costo derivante dall'intervento dell'uomo. Sensori possono controllare le condizioni delle macchine, accelerando o rallentando il ritmo di produzione in caso di possibili ottimizzazioni.

Inoltre, la completa decentralizzazione derivante dall'edge computing può portare numerosi benefici nell'introduzione di nuovi meccanismi IT, velocizzando i tempi di sviluppo e di esecuzione dei servizi. Le macchine possono lavorare in autonomia senza l'ausilio di data center, portando maggiore fiducia sulla sicurezza dei dati elaborati, ed abbattendo costi di mantenimento a lungo termine.

Nel caso di image processing per l'elaborazione di grandi immagini, la necessità di garantire un elevato throughput nonostante l'importanza dei dati da analizzare ha reso la decisione di eseguire il servizio in ambiente edge una scelta obbligata. La possibilità di lavorare con latenze praticamente nulle garantisce prestazioni più soddisfacenti, riuscendo a soddisfare il determinismo richiesto dal plant, e garantendo al contempo sicurezza e sovranità delle informazioni elaborate.

5.5 Architettura serverless del caso applicativo

Il mercato delle applicazioni serverless in ambito industriale è in forte crescita [92]. La causa è soprattutto nei benefici derivanti dall'adozione di questo tipo di architettura, dati i vantaggi economici derivati, e dall'estremo disaccoppiamento favorito dalla scelta del serverless. L'elevato grado di granularità offerto, dato dalla divisione del sistema in tanti piccoli componenti interagenti, permette di scalare ogni singolo servizio autonomamente, offrendo un livello di ottimizzazione delle risorse difficilmente raggiungibile dalle altre architetture. Queste ragioni permettono al serverless di candidarsi come soluzione principale ad ambienti edge, dove la necessità di razionalizzare le risorse a disposizione è elevata, anche a causa del vincolo di richiesta obbligata di risorse da parte del plant industriale.

Le architetture serverless sono composte da due tipi di servizi: le Function-as-a-Service e i Backend-as-a-Service. Framework di questo sono sviluppati secondo la metodologia serverless, e sono necessari per implementare un'applicazione che segua gli standard serverless richiesti. Inoltre, data la disponibilità di un PC per ogni macchina, è possibile eseguire il deployment di questi servizi su orchestratori a container, i quali offrono un elevato grado di scalabilità alle applicazioni. L'architettura serverless fonda i propri principi sul completo disaccoppiamento tra i diversi componenti in gioco, dunque è necessaria la presenza di servizi third-party che vadano a coordinare l'esecuzione dei worker. Inoltre, data la natura serverless delle Function-as-a-Service, è necessario distribuire lo stato di ogni singolo worker in servizi che mantengano in persistenza i progressi effettuati durante l'esecuzione dell'algoritmo parallelo.

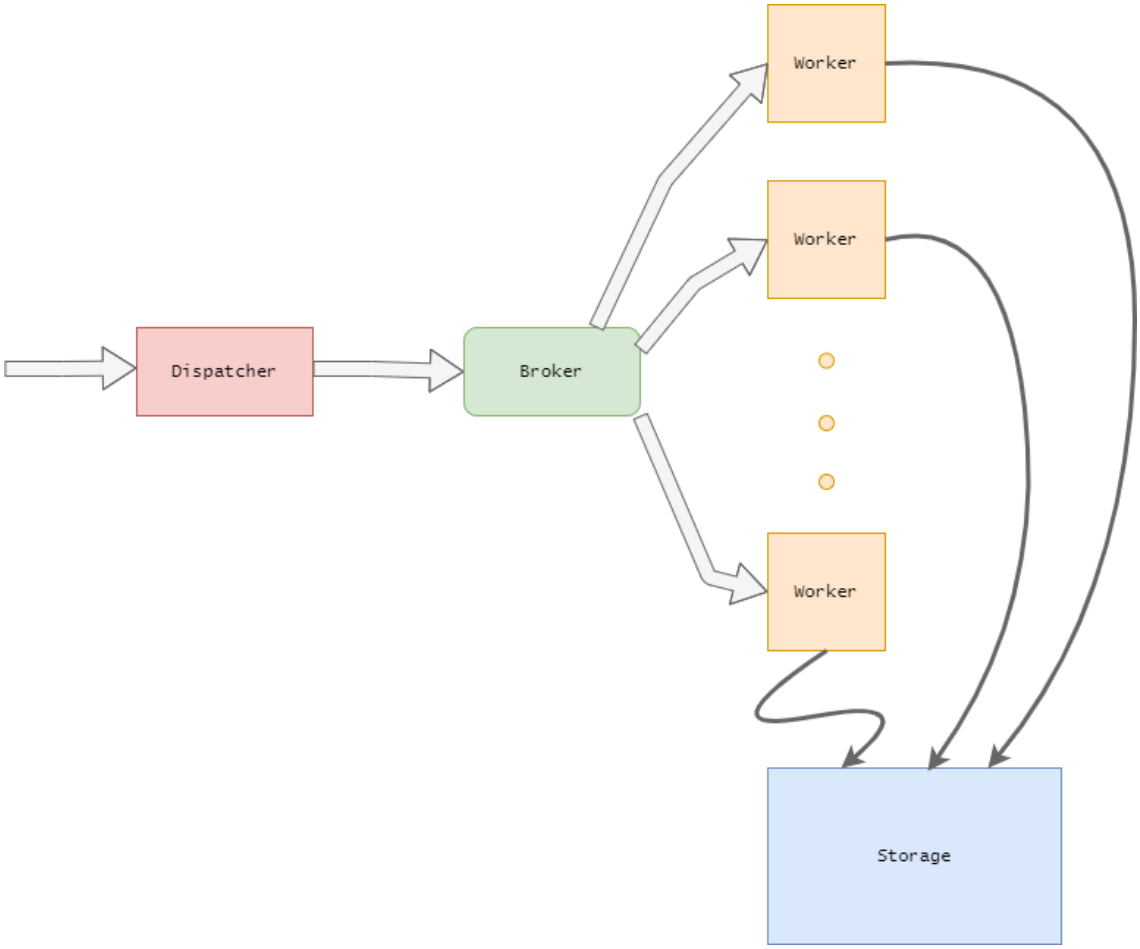

Figura 46 - Architettura serverless del caso applicativo

La Figura 46 rappresenta l'architettura serverless del caso applicativo, la quale è composta da diversi componenti che interagiscono in maniera completamente disaccoppiata. L'immagine da elaborare viene passata come input al Dispatcher, componente che ha il compito di analizzare l'immagine, decidere in quanti task elaborare l'input, e passare i task assegnati al componente successivo: il Broker. Il Broker è implementato come sistema per lo scambio dei messaggi, per permettere il disaccoppiamento tra i componenti, in quanto mantiene in memoria i task assegnati e cerca di recapitarli ai Worker designati. I Worker rappresentano l'effettiva unità computazionale, il cui compito è l'esecuzione della rotazione dell'immagine assegnata dal Dispatcher. Questi, a causa della loro natura stateless, vanno a salvare l'output direttamente in un servizio di Storage esterno. Infine, i componenti scelti devono essere implementati con prodotti che rispettino la natura serverless dell'architettura, in modo da garantire il massimo dei benefici e il minor consumo di risorse possibile.

5.6 Implementazione

I confronti qualitativi e quantitativi delle piattaforme hanno permesso di scegliere il framework più adatto per poter realizzare il caso applicativo secondo i vincoli imposti dall'ambiente industriale, oltre agli importanti requisiti di throughput e consumo delle risorse richiesti. La scelta è ricaduta su Knative, in quanto ha ottenuto i risultati più convincenti, soprattutto in relazione alla velocità di elaborazione delle richieste e all'ottima capacità di gestione delle risorse, soprattutto in situazioni CPU intensive.

Il progetto Knative, come già descritto, offre due componenti, fondamentali per la corretta realizzazione del caso applicativo: Knative Serving e Knative Eventing. Il primo componente ha permesso di trasformare i servizi eseguiti in Kubernetes in workload serverless, il che significa gestione automatica del numero di repliche generate, scale-to-zero per evitare qualsiasi consumo di risorse durante l'inattività del servizio, e routing gestito automaticamente da Istio, evitando di dover eseguire API Gateway o qualsiasi altro componente necessario allo smistamento delle richieste.

Invece, Knative Eventing è stato il ponte tra i diversi servizi di cui è composto l'architettura in esame, offrendo meccanismi di load balancing con sistema a scambio di messaggi, necessario per invocare i worker ed assegnare i task necessari. Inoltre, il componente Eventing ha permesso l'astrazione rispetto al message broker implementato, soluzione offerta unicamente da Knative che va a disaccoppiare ulteriormente i servizi in esecuzione sulla piattaforma, grazie all'adozione di standard per la formattazione degli eventi chiamata CloudEvents.

CloudEvents [93] è un'iniziativa proposta dal Cloud Native Computing Foundation, nata con l'obiettivo di standardizzare come i produttori descrivono i propri eventi. Un CloudEvent consiste in una serie di attributi, alcuni obbligatori ed altri facoltativi, come l'ID dell'evento ed il tipo dell'evento.

Attribute Name	Type	Note
`id`	String	Required. The ID of the event. A CloudEvent is uniquely identified with its `source` and `id`.
`source`	String (URI-reference)	Required. The source of the event.

`specversion`	String	Required. The version of CloudEvents Specification the Cloud Event uses.
`type`	String	Required. The type of the event.
`datacontentencoding`	String (RFC 2045 Section 6.1)	Optional. The encoding of `data` (if the field stores binary data).
`datacontenttype`	String (RFC 2046)	Optional. The content type of `data`.
`schemaurl`	String	Optional. The schema of data.
`subject`	String	Optional. The subject of the event.
`time`	String (Timestamp)	Optional. The timestamp when the event happens.
`data`	N/A	Optional. The payload of the event.

Tabella 3 - Attributi obbligatori e facoltativi richiesti da un evento CloudEvent

In Tabella 3 sono descritti gli attributi offerti dallo standard CloudEvents. Inoltre, è possibile aggiungere set di attributi personalizzati tramite estensioni. La costruzione di eventi CloudEvent può avvenire definendo gli Header presenti nel pacchetto HTTP, oppure è possibile utilizzare gli SDK offerti dal progetto open source di CloudEvents. In definitiva, è necessario definire i campi `id`, `source`, `specversion` e `type` per poter inviare eventi CloudEvent, oltre al campo `data` dove inserire il corpo dell'evento [94].

Knative Eventing permette la comunicazione tra i diversi servizi grazie al componente Broker, ovvero un dispatcher di eventi che smista i messaggi ai servizi iscritti ad un determinato topic. Il Broker viene generato automaticamente da Knative se richiesto esplicitamente, inserendo un label al namespace in cui si vuole istanziare il message broker. È possibile confermare la corretta esecuzione del Broker in quanto Custom Resource di Kubernetes, utilizzando kubectl e visualizzando i Pod in stato di running all'interno del namespace di riferimento.

Una volta eseguito il Broker, è necessario iscrivere il servizio, per permettere la corretta consegna degli eventi, e per farlo bisogna istanziare risorse chiamate Trigger, che fanno da ponte tra il Broker e il servizio. In Figura 47 è possibile vedere la risorsa Trigger che

connette Broker e servizio, ed è anche possibile filtrare gli eventi smistati dal Broker, specificando il tipo di evento desiderato.

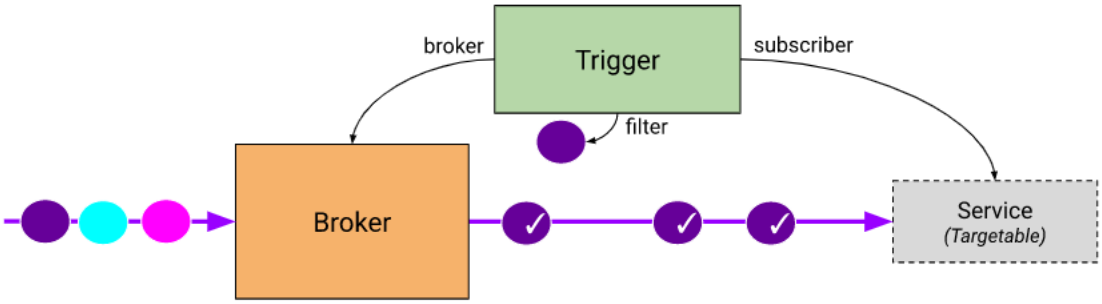

Figura 47 - Architettura semplificata di Knative Eventing

L'infrastruttura offerta da Knative Eventing permette l'interazione tra i due servizi sviluppati per l'applicazione in esame. Il primo è il servizio coordinatore, ovvero colui che parallelizza i task da eseguire invia eventi al Broker, il quale li inoltra al secondo servizio addetto all'effettiva rotazione dell'immagine. Ad ogni evento corrisponde una nuova piccola rotazione, al contrario di un'unica grande rotazione eseguita da un unico servizio.

Data la natura stateless delle funzioni, è stato necessario adottare una soluzione di storage per permettere all'applicazione di recuperare le parti di immagine da elaborare e per salvare le elaborazioni effettuate. Le immagini sono informazioni non strutturate, di dimensione variabile e che possono raggiungere anche diversi gigabyte di spazio occupato. Per queste ragioni, è stato implementato un object storage come soluzione per il caso d'uso applicativo.

L'object storage è la soluzione de facto per dati strutturati e non strutturati in applicazioni cloud native. Gli oggetti sono dati finiti, strutturati in ambienti flat, senza cartelle, directory o complesse gerarchie, ed ogni oggetto contiene i dati, i metadati con informazioni sull'oggetto, e un ID identificativo unico. Gli oggetti possono essere aggregati in storage pool e possono essere distribuiti indefinitamente, garantendo elevati gradi di scalabilità e tolleranza ai guasti. Inoltre, la mancanza di gerarchie promuove tempi di latenza più bassi, in quanto elimina la complessità strutturale. I dati, chiamati oggetti, possono essere acceduti tramite API - è possibile avvalersi anche di un API Gateway – spesso di tipo HTTP. Le query permettono di localizzare gli oggetti grazie ai metadati, ed è possibile effettuare il downstream, ma anche modificare ed eliminare gli oggetti.

È possibile trarre benefici da soluzioni object storage se si lavora con importanti moli di dati non strutturati. In particolare, questo tipo di dati, come video o immagini, sono spesso pesanti e statici ma possono essere richiesti sempre e da qualsiasi posizione, dunque è necessario adottare architetture che premiano la ridondanza e la semplicità per offrire prestazioni migliori. L'aspetto fondamentale dell'object storage è la scalabilità, praticamente infinita, che lo rende il candidato migliore in scenari cloud nativi, dove è necessario aumentare il numero di nodi per garantire prestazioni soddisfacenti ad un numero indefinito di utenti. La distribuzione di più nodi addetti all'object storage offre al cliente sicurezza, in quanto i dati possono essere replicati, oltre ad offrire performance migliori [95].

Il primo servizio presente nel workflow dell'applicazione è Dispatcher (Snippet 2), componente necessario per lo smistamento degli eventi. Il servizio Dispatcher invia un evento per ogni parte di immagine da elaborare, invocando una nuova funzione per ogni rotazione da effettuare. Il componente lavora come coordinatore, inviando eventi al Broker presente nel namespace dell'applicazione, il quale smista le richieste in maniera uniforme al servizio di rotazione delle immagini.

```
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "os"
    "strconv"
    "time"
)

type Event struct {
    BucketName string `json:"bucketName"`
    ObjectName string `json:"objectName"`
}

func handler(w http.ResponseWriter, r *http.Request) {
    log.Printf("received a request\n")

    start := time.Now()

    client := &http.Client{}
    url := "http://default-broker.image-processing.svc.cluster.local"
```

```
    bucketName := "rotimage-go"
    for i := 0; i < 30; i++ {
        partName := "_part_" + strconv.Itoa(i) + "_.jpg"

        event := Event{bucketName, partName}

        b, err := json.Marshal(event)
        if err != nil {
            log.Printf("json.Marshal: %s", err)
        }

        req, err := http.NewRequest("POST", url, bytes.NewReader(b))
        req.Header.Set("Ce-Id", "dispatcher-go")
        req.Header.Set("Ce-Specversion", "0.3")
        req.Header.Set("Ce-Type", "dispatcher-go.rotate")
        req.Header.Set("Ce-Source", "dispatcher-go")
        req.Header.Set("Content-Type", "application/json")

        resp, err := client.Do(req)
        if err != nil {
            log.Printf("client.Do: %s", err)
        }
        defer resp.Body.Close()
    }

    elapsed := time.Since(start)
    log.Printf("request successfully completed\n")
    log.Printf(" - dispatched in %d\n", elapsed)
}

func main() {
    log.Printf("starting server...\n")

    http.HandleFunc("/", handler)

    port := os.Getenv("PORT")
    if port == "" {
        port = "8080"
    }

    log.Printf("listening on port %s", port)
    log.Fatal(http.ListenAndServe(fmt.Sprintf(":%s", port), nil))
}
```

Snippet 2 – codice di Dispatcher, scritto in go 1.13

Il secondo componente progettato per il caso applicativo è il servizio Rotimage (Snippet 3), il quale è responsabile per la rotazione dell'immagine. L'applicazione viene invocata direttamente dal Broker, e gli eventi che riceve contengono informazioni sulla parte da elaborare. Una volta avviato il servizio, Rotimage scarica l'immagine interessata dall'object storage, attende il completamento dell'operazione ed esegue la rotazione. Al termine della rotazione, l'immagine ruotata viene salvata nuovamente nell'object storage, pronta ad essere utilizzata da qualsiasi altro servizio necessiti di quel dato, come ad esempio un aggregatore di parti per ricomporre l'intera immagine ruotata.

```
package main

import (
    "context"
    "log"
    "time"

    cloudevents "github.com/cloudevents/sdk-go"
    "github.com/disintegration/imaging"
    "github.com/google/uuid"
    "github.com/knative/eventing-sources/pkg/kncloudevents"
    "github.com/minio/minio-go/v6"
)

type Event struct {
    BucketName string `json:"bucketName"`
    ObjectName string `json:"objectName"`
}

func receive(event cloudevents.Event) {
    start := time.Now()

    endpoint := "minio-service.default.svc.cluster.local:9000"
    accessKeyID := "minio"
    secretAccessKey := "minio123"
    useSSL := false

    minioClient, err := minio.New(endpoint, accessKeyID, secretAccessKey,
 useSSL)
    if err != nil {
        log.Fatalln(err)
    }

    data := &Event{}
```

```
    if err := event.DataAs(data); err != nil {
        log.Printf("error while extracting cloudevent Data: %s\n", err.Er
ror())
    }

    uuid := uuid.New().String()
    log.Printf("uuid: %s\n", uuid)

    fileName := uuid + ".jpg"
    if err = minioClient.FGetObject(data.BucketName, data.ObjectName, fil
eName, minio.GetObjectOptions{}); err != nil {
        log.Fatalln(err)
    }

    // Retrieve input from file
    img, _ := imaging.Open(fileName)

    // Hard work...
    rotated := imaging.Rotate180(img)

    outputName := "_r_" + fileName
    // Store output as file
    if err = imaging.Save(rotated, outputName); err != nil {
        log.Fatalln(err)
    }

    // Store output in object storage
    if _, err = minioClient.FPutObject("rotimage-
go", outputName, outputName, minio.PutObjectOptions{ContentType: "image/j
peg"}); err != nil {
        log.Fatalln(err)
    }

    elapsed := time.Since(start)
    log.Printf("request successfully completed\n")
    log.Printf(" - rotated in %d\n", elapsed)
}

func main() {
    c, err := kncloudevents.NewDefaultClient()
    if err != nil {
        log.Fatalf("failed to create client, %v", err)
    }
    log.Fatal(c.StartReceiver(context.Background(), receive))
}
```

Snippet 3 – codice di rotimage, scritto in go 1.13

L’architettura serverless presentata garantisce il parallelismo descritto precedentemente, permettendo di implementare qualsiasi algoritmo massicciamente parallelo richiesto. La possibilità di dividere in maniera uniforme l’immagine in tante parti aumenta il livello di scalabilità raggiunto da questo tipo di applicazioni, in quanto si evita il rischio di immagini troppo pesanti che andrebbero a sforzare eccessivamente la CPU, compromettendo il determinismo del plant. Per esempio, sia che l’immagine occupi dieci megabyte e sia che questa ne occupi cento, è possibile assegnare ad ogni funzione una parte dell’immagine dal peso di un megabyte, permettendo la distribuzione del carico su tutti i nodi ed evitando eccessivi carichi sui singoli nodi.

5.7 Descrizione dei test

Per comprendere completamente le potenzialità di questo tipo di architettura, è necessario avvalersi di risultati che mostrino in maniera evidenti i vantaggi conseguenti all’adozione del serverless. In particolare, per poter valutare correttamente i vantaggi offerti è necessario affiancare la soluzione serverless ad una più classica soluzione cloud native che non adotti questo approccio. Per questa ragione, è stato eseguito Rotimage come microservizio (Snippet 4), riscrivendo l’applicazione in modo da accettare richieste HTTP e non più CloudEvents, ed eseguendolo direttamente su Kubernetes senza alcuna piattaforma serverless di supporto. Semplicemente, il microservizio Rotimage HTTP riceve in input un JSON con le informazioni per reperire l’intera immagine dall’object storage, in modo da eseguire la rotazione.

```
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "os"
    "time"

    "github.com/disintegration/imaging"
    "github.com/google/uuid"
    "github.com/minio/minio-go/v6"
)
```

```
type Event struct {
    BucketName string `json:"bucketName"`
    ObjectName string `json:"objectName"`
}

func handler(w http.ResponseWriter, r *http.Request) {
    start := time.Now()

    endpoint := "minio.minio.svc.cluster.local:9000"
    accessKeyID := "AKIAIOSFODNN7EXAMPLE"
    secretAccessKey := "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
    useSSL := false

    minioClient, err := minio.New(endpoint, accessKeyID, secretAccessKey,
 useSSL)
    if err != nil {
        log.Fatalln(err)
    }

    var req Event
    err = json.NewDecoder(r.Body).Decode(&req)
    if err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
    }

    uuid := uuid.New().String()
    log.Printf("uuid: %s\n", uuid)

    fileName := uuid + ".jpg"
    if err = minioClient.FGetObject(req.BucketName, req.ObjectName, fileN
ame, minio.GetObjectOptions{}); err != nil {
        log.Printf("minioClient.FGetObject: %s", err)
    }

    // Retrieve input from file
    img, _ := imaging.Open(fileName)

    // Hard work...
    rotated := imaging.Rotate180(img)

    outputName := "_r_" + fileName
    // Store output as file
    if err = imaging.Save(rotated, outputName); err != nil {
        log.Fatalln(err)
    }

    // Store output in object storage
```

```
    if _, err = minioClient.FPutObject("rotimage-
go", outputName, outputName, minio.PutObjectOptions{ContentType: "image/j
peg"}); err != nil {
        log.Fatalln(err)
    }

    elapsed := time.Since(start)
    log.Printf("request successfully completed\n")
    log.Printf(" - rotated in %d\n", elapsed)
}

func main() {
    log.Printf("starting server...\n")

    http.HandleFunc("/", handler)

    port := os.Getenv("PORT")
    if port == "" {
        port = "8080"
    }

    log.Printf("listening on port %s\n", port)
    log.Fatal(http.ListenAndServe(fmt.Sprintf(":%s", port), nil))
}
```

Snippet 4 – Rotimage HTTP, scritto in go 1.13

Il codice è stato eseguito come risorsa Deployment su Kubernetes, in modo da lasciar gestire all'orchestratore il ciclo di vita delle replice dell'applicazione. Inoltre, è stata eseguita una risorsa Service di tipologia NodePort per offrire un punto di accesso esterno al cluster per l'invocazione del microservizio.

La scelta dei test del caso applicativo ha richiesto un certo livello di analisi per poter rendere evidenti le differenze e le situazioni in cui una soluzione lavora meglio dell'altra. Dato il vincolo del throughput richiesto dall'impianto di produzione, il primo parametro che si è voluto prendere in considerazione è il tempo totale di esecuzione, ovvero in quanto tempo le macchine riescono ad elaborare la rotazione di un'immagine richiesta da un certo numero di utenti. Il secondo parametro, data la necessità di conservare le risorse per mantenere il determinismo del plant, è stato l'utilizzo delle risorse durante le elaborazioni, prendendo in considerazione soprattutto la CPU data la natura computazionale delle operazioni.

Per garantire risultati apprezzabili è stato necessario lavorare con immagini di grandi dimensioni, per poter impiegare la CPU in modo significativo e valutare le sperimentazioni per un certo intervallo di tempo. Dati i vincoli dati dalle risorse offerte

dal cluster, è stato necessario testare le applicazioni con immagini di dimensione differente, in modo da trovare il compromesso ideale tra saturazione delle risorse del cluster ed impegno significativo della CPU. Alla fine, è stata generata un'immagine da 30 megabyte per i test sul microservizio, mentre sono state generate trenta immagini da un megabyte per l'applicazione serverless.

È importante sottolineare un'assunzione sullo scenario relativo al caso applicativo, ovvero la presenza di parti di immagine già pronte da elaborare per l'applicazione serverless, in quanto l'operazione di divisione dell'immagini in parti uguali comporta un'overhead importante all'intera architettura. Dare la responsabilità di dividere le immagini direttamente all'applicazione serverless vorrebbe dire valutare un'applicazione con due operazioni computazionalmente onerose, ovvero divisione e rotazione delle immagini, rispetto ad un'applicazione con un'unica operazione onerosa, ovvero rotazione di un'unica grande immagine. Per queste ragioni, si è scelto di generare a priori le parti di immagini da ruotare.

I test sono stati eseguiti generando un certo numero di utenti in un determinato intervallo di tempo, ed ognuno di questi richiedeva la rotazione delle immagini messe a disposizione per le sperimentazioni. Ad ogni test il numero di utenti viene incrementato di cinque unità, ed il tempo di generazione di questi corrisponde al numero di utenti stesso. Per esempio, il primo test da dieci utenti consiste nella generazione di dieci richieste nell'arco di dieci secondi, mentre il test da cinquanta utenti consiste in cinquanta richieste in cinquanta secondi. L'intervallo tra le richieste, come già descritto nel confronto tra le piattaforme, è necessario per evitare picchi di carico e saturazione delle risorse.

I test descrivono le metriche di elaborazione a partire da cinque utenti fino ad arrivare a cinquanta utenti, analizzando nello specifico il tempo di esecuzione per il completamento dell'operazione, la quantità media di CPU utilizzata e il picco massimo di CPU rilevato. Come per i test precedenti, le richieste sono state eseguite con Apache JMeter, e le metriche sono state rilevate grazie all'utilizzo di tool di monitoring, quali Prometheus e Grafana, i quali permettono una visione chiara e dettagliata del cluster durante le sperimentazioni.

5.8 Risultati

Di seguito sono riportati i risultati ottenuti durante le sperimentazioni. I risultati consistono in un confronto tra le due architetture descritte precedentemente, e mettono in risalto i benefici e gli svantaggi di entrambe le soluzioni. Il primo test valuta il tempo di esecuzione al variare degli utenti, il secondo mostra l'utilizzo medio della CPU ed il terzo rileva i picchi massimi di CPU ottenuti durante le sperimentazioni.

Tempo di esecuzione

Il grafico, visibile in Figura 48, mostra il tempo totale di esecuzione, necessario per effettuare la rotazione dell'immagine. Le due architetture impiegano circa lo stesso tempo per smistare venti utenti, impiegando 38 secondi, ed impiegano lo stesso tempo per eseguire i test con un numero di utenti inferiori. Lo stesso non vale per i test con un numero maggiore di utenti, dove l'architettura sequenziale impiega 107 secondi contro i 51 secondi dell'architettura serverless nel test con 35 utenti. Infine, il test con 50 utenti mostra il divario maggiore tra le due soluzioni, dove l'architettura serverless, con i 69 secondi registrati, risparmia oltre due minuti rispetto all'architettura sequenziale.

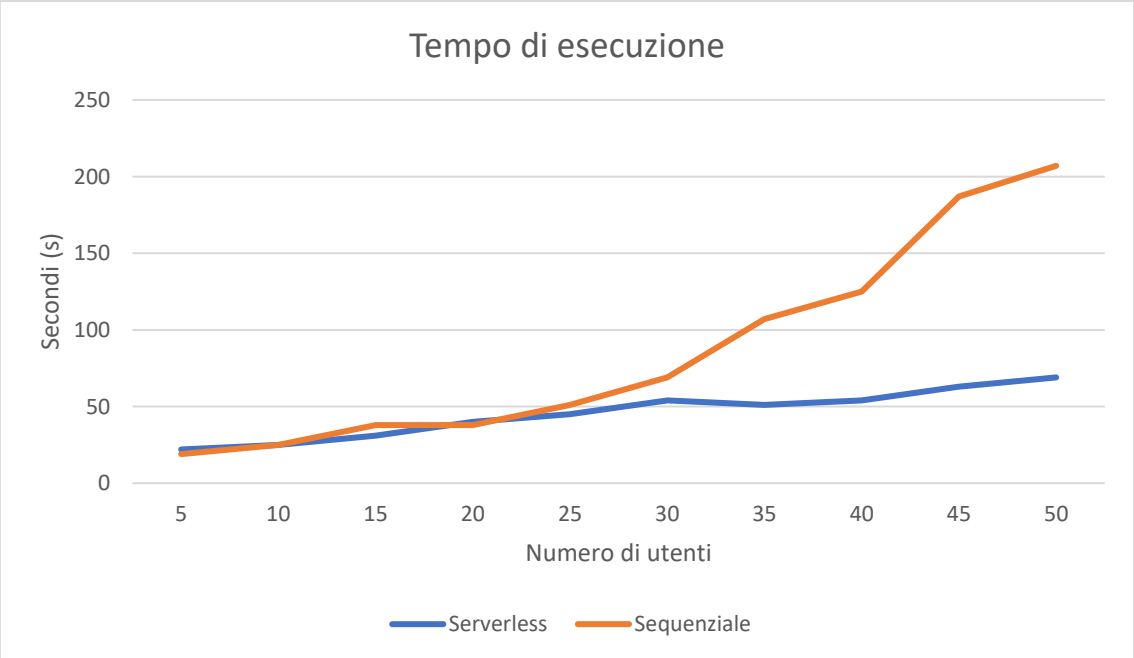

Figura 48 - Tempo di esecuzione in secondi

Consumo medio della CPU

Il grafico, presentato in Figura 49, mostra l'utilizzo medio della CPU durante le operazioni di rotazione delle immagini, misurando il consumo in CPU virtuali, come già successo precedentemente nei test di confronto tra le piattaforme. Il consumo medio della CPU mostra l'architettura serverless con punteggi migliori rispetto all'architettura sequenziale per qualsiasi numero di utenti. Con 15 utenti la soluzione sequenziale arriva a consumare una CPU virtuale mentre la soluzione serverless richiede meno della metà della richiesta computazionale. Con 45 utenti, la richiesta di CPU del sequenziale tripla quella del serverless, richiedendo 3,5 vCPU invece delle 1,3 vCPU richieste da quest'ultima.

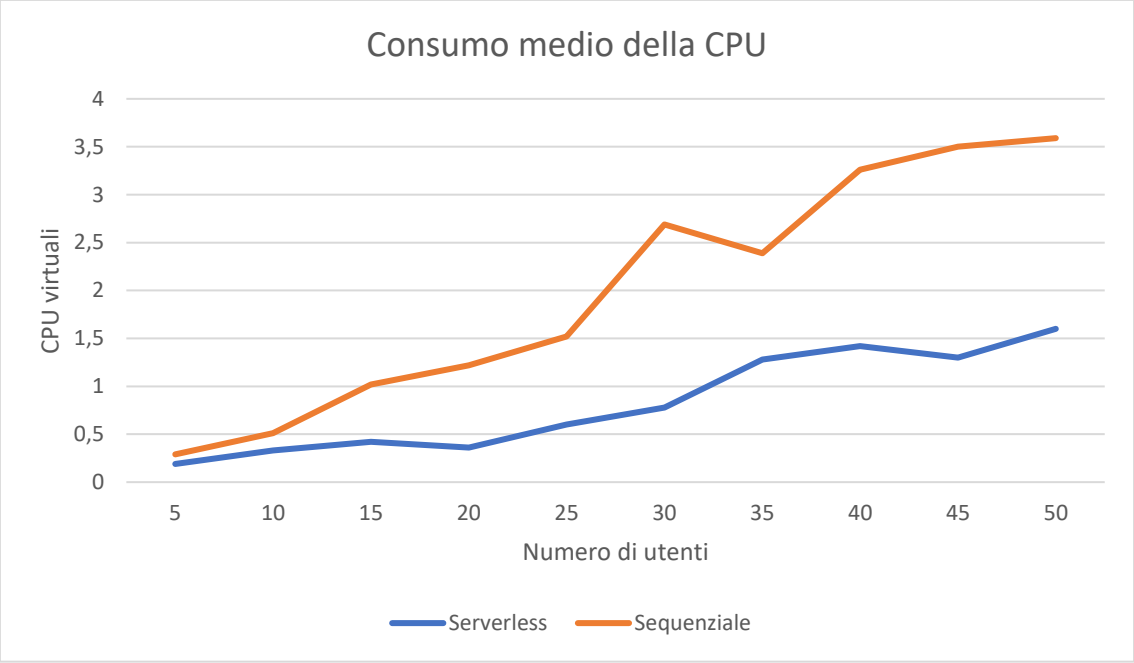

Figura 49 - Consumo medio di CPU misurato in vCPU

Consumo massimo di CPU

Il grafico, visibile in Figura 50, mostra il picco massimo di CPU rilevato durante l'elaborazione delle immagini. I test da 5 e 10 utenti mostrano gli stessi punteggi, rispettivamente con 0,29 vCPU e 0,58 vCPU, mentre con 15 utenti si apre una forbice di circa mezza CPU virtuale tra la soluzione sequenziale e la soluzione serverless. Con 30 utenti, il serverless raggiunge le 1,37 vCPU, mentre il sequenziale raggiunge le 3,47 vCPU, con uno scarto di oltre due CPU virtuali. Infine, il test da 40 utenti rileva la

maggiore forbice tra sequenziale e serverless, dove il primo raggiunge le 5,51 vCPU mentre il secondo si ferma a 2,55 vCPU consumate.

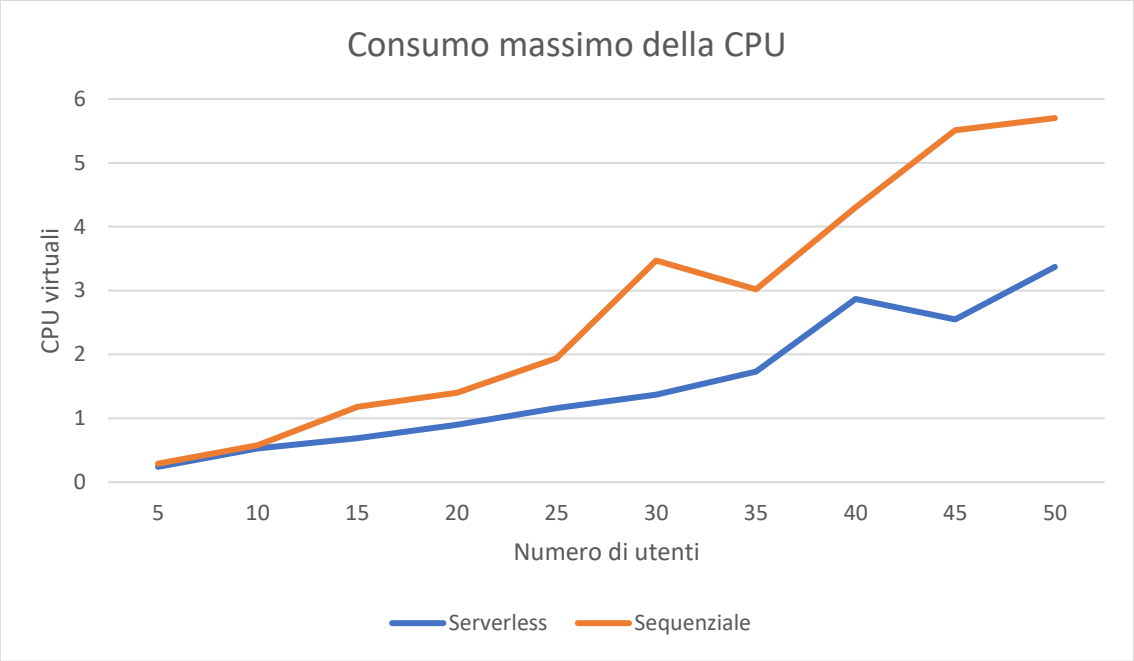

Figura 50 - Consumo massimo di CPU misurato in vCPU

5.9 Considerazioni

I risultati riportati mostrano metriche significative per la valutazione dell'architettura più adatta rispetto al caso applicativo preso in esame. I punteggi hanno mostrato l'architettura serverless come soluzione più adatta alla gestione di algoritmi massicciamente paralleli, grazie alla gestione automatica delle repliche, che permettono ottimizzazioni nei consumi delle risorse.

I risultati relativi ai tempi di esecuzione mostrano come l'architettura serverless sia più adatta per grandi scenari produttivi, dato l'andamento quasi costante delle metriche all'aumentare degli utenti. Al contrario, la soluzione sequenziale ha mostrato una continua crescita dei tempi di esecuzione all'aumentare degli utenti, limitando la soluzione in caso di numerose richieste, come in scenari industriali di produzione.

Il guadagno in termini di consumi derivanti dall'adozione di architetture serverless garantisce meno probabilità di saturazione del cluster, parametro fondamentale per evitare di perdere il determinismo richiesto dai vincoli specificati precedentemente. Inoltre, è necessario avvertire che la soluzione sequenziale ha mostrato errori oltre i

cinquanta utenti, saturando le risorse e restituendo errori a JMeter, mentre la soluzione serverless ha garantito alte prestazioni anche in presenza di cento utenti. Per queste ragioni, data l'impossibilità di un confronto legittimo, si è evitato di mostrare metriche oltre i cinquanta utenti.

Le oscillazioni visibili sui grafici precedentemente riportati sono dovute ai warm start dei container, i quali hanno ridotto i tempi di esecuzione e il consumo di risorse. In alcuni test, Knative ha mantenuto i Pod pronti per le invocazioni, evitando di generare overhead per la creazione di nuove istanze. In ogni caso, le variazioni dovute ai warm start non sono state prese in considerazione durante le sperimentazioni, in quanto trascurabili rispetto al tempo complessivo di esecuzione di ogni invocazione.

6 Conclusioni e sviluppi futuri

Le sperimentazioni presentate in questa tesi hanno mostrato ampiamenti i benefici derivanti dall'utilizzo di architetture serverless. Infatti, dai risultati ottenuti si evincono chiaramente consumi e tempi di esecuzione più convincenti nei framework serverless, grazie alla loro capacità di allocare autonomamente le risorse in maniera più efficiente ed ottimizzata rispetto al classico approccio a microservizi. Le soluzioni serverless si sono dimostrate vincenti in scenari industriali, soprattutto grazie all'esecuzione del cluster in ambiente edge, che ha permesso la replicazione di istanze secondo politiche di località e frequenza delle richieste, offrendo prestazioni complessivamente migliori.

La scelta di Knative ha portato vantaggi in termini di prestazioni, mostrandosi nelle sperimentazioni come la soluzione più efficiente e performante. Per queste ragioni, si è deciso di lavorare con un framework serverless anziché con framework FaaS, perdendo gran parte delle astrazioni offerte da questo tipo di piattaforme. Infatti, l'utilizzo di Knative non ha permesso di scrivere un qualsiasi codice sorgente ed attendere la messa in esecuzione di quest'ultimo, bensì è stato necessario containerizzare il codice manualmente, pratica che ha richiesto tempo per la comprensione e l'implementazione di questa tecnologia. Inoltre, il deployment dell'API gateway, necessario per l'accesso verso le applicazioni, è stato effettuato manualmente installando Istio sul cluster, mentre una soluzione FaaS avrebbe garantito la presenza di questo componente out-of-the-box.

L'adozione di Knative non favorisce l'esecuzione immediata di funzioni con il minimo sforzo, bensì offre un potente plug-in a Kubernetes per realizzare workload serverless. L'astrazione offerta da Knative non è maggiore rispetto a quella offerta da Kubernetes per l'esecuzione di una risorsa Deployment, ma garantisce la gestione automatica di ogni Pod presente nel cluster, sollevando il team DevOps da qualsiasi operazione a runtime, e permettendo al contempo di lavorare con la consapevolezza di raggiungere parametri di scalabilità idealmente infiniti. Per queste ragioni, le organizzazioni che scelgono di adottare soluzioni serverless vengono ripagate con risparmi significativi, offrendo a quest'ultime la possibilità di rilasciare più servizi in minor tempo.

Dati i vantaggi descritti precedentemente, non è difficile immaginare perché il serverless sia uno dei servizi con il maggior tasso di crescita tra i modelli offerti in cloud, stimato al 75% [96]. Stephen Bergstein, Chief Software Architect presso Micro Focus, scrive:

> *"We will see further adoption of serverless, especially in public clouds. The large cloud providers are going to invest into maturity, programming languages coverage and improved IDE integrations. These aspects will drive adoption." – [97] Stefan Bergstein*

Nei prossimi anni la standardizzazione del serverless sarà un argomento principale, soprattutto per evitare alle organizzazioni di rimanere legate ad un unico provider cloud. Knative sarà l'ago della bilancia, in quanto permetterà la trasformazione di workload serverless a tutti coloro che possiedono conoscenze relative all'esecuzione di container in Kubernetes. A tal proposito, Joe Fernandes, Vice President for Cloud Platform Business, scrive:

> *"In the same way that Kubernetes and containers enabled a hybrid abstraction for applications that can span a hybrid cloud environment, we expect Knative could do the same for serverless." – Joe Fernandes*

Inoltre, l'utilizzo di CloudEvents come formato standard di eventi permette alle applicazioni di non essere vincolati ad un unico message broker, sponsorizzando l'adozione di applicazioni multi-cloud, dove gli eventi sono generati da provider cloud differenti. Con l'adozione di soluzioni serverless ibride, dunque, le organizzazioni possono replicare i propri servizi trasparentemente tra data center e cloud privati, favorendo la crescita dell'hybrid cloud in tutti i settori economici, tra cui l'industria manufatturiera presa in esame [98].

La crescita del serverless porterà le organizzazioni ad investire sulla messa in sicurezza di questa tecnologia. Infatti, una società israeliana di cyber-sicurezza, chiamata PureSec, ha rilevato in mille applicazioni serverless oltre duecento vulnerabilità [99], portando alla luce importanti difetti di questa tecnologia. Gadi Naor, CTO presso la società di cyber sicurezza Alcide, scrive:

> *"We expect security vendors to deliver solutions that uncover blind spots and control serverless. We also expect companies to adopt new distributed policies for compliance and control requirements." – Gadi Naor*

Il report di DigitalOcean Currents identifica come sfida più importante del serverless il monitoring ed il debugging di applicazioni [100]. La motivazione risiede nelle astrazioni presenti nella tecnologia serverless, che rendono difficile l'analisi di centinaia di applicazioni a runtime. La crescente complessità dei sistemi distribuiti porterà alla semplificazione di queste operazioni. A tal proposito, Stefan Bergstein scrive:

> *"Both serverless platforms and application management tools are going to improve and simplify the monitoring and observability of serverless application service and code." – Stefan Bergstein*

Come ogni tecnologia emergente, il serverless sta vivendo una fase di crescita e di importante sviluppo, portando al contempo confusione ed indecisione sulla scelta nell'adozione di questo tipo di architettura. In ogni caso, i prossimi anni vedranno l'arrivo di standardizzazioni e nuove astrazioni, catapultando il serverless a tecnologia dominante in tutti i settori economici [98].

7 Bibliografia

[1] J. Swartz, «How Amazon created AWS and changed technology forever,» MarketWatch, 7 Dicembre 2019. [Online]. Available: https://www.marketwatch.com/story/how-amazon-created-aws-and-changed-technology-forever-2019-12-03. [Consultato il giorno 2 Marzo 2020].

[2] A. S. Tanenbaum e M. Van Steen, Distributed Systems: Principles and Paradigms, Amsterdam: Pearson Prentice Hall, 2007.

[3] «What is Middleware? - Definition and Example,» Microsoft Azure, [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-middleware/. [Consultato il giorno 2 Marzo 2020].

[4] «Getting Started Using Java RMI,» Oracle, [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html. [Consultato il giorno 5 Marzo 2020].

[5] «rpc,» The Go Programming Language, [Online]. Available: https://golang.org/pkg/net/rpc/. [Consultato il giorno 5 Marzo 2020].

[6] «RPyC - Transparent, Symmetric Distributed Computing,» [Online]. Available: https://rpyc.readthedocs.io/en/latest/. [Consultato il giorno 5 Marzo 2020].

[7] «The CORBA Architecture,» Sun Microsystem, 2005. [Online]. Available: https://www.math.uni-hamburg.de/doc/java/tutorial/idl/intro/corba.html. [Consultato il giorno 5 Marzo 2020].

[8] «AMQP 0-9-1 Model Explained,» RabbitMQ, [Online]. Available: https://www.rabbitmq.com/tutorials/amqp-concepts.html. [Consultato il giorno 5 Marzo 2020].

[9] «Getting Started with MQTT,» HiveMQ, 14 Luglio 2019. [Online]. Available: https://www.hivemq.com/blog/how-to-get-started-with-mqtt/. [Consultato il giorno 5 Marzo 2020].

[10] «Message-Oriented Middleware (MOM),» Sun Java, 2010. [Online]. Available: https://docs.oracle.com/cd/E19340-01/820-6424/aeraq/index.html. [Consultato il giorno 4 Marzo 2020].

[11] R. S. Chowhan, Evolution and Paradigm Shift in Distributed System Architecture, 2018.

[12] J. Lewis e M. Fowler, «Microservices,» 25 March 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Consultato il giorno 25 February 2020].

[13] C. Richardson, «Building Microservices Using an API Gateway | NGINX,» NGINX, 15 June 2015. [Online]. Available: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/. [Consultato il giorno 25 February 2020].

[14] E. Curry, «Message-Oriented Middleware,» National University of Ireland, Galway, 2004.

[15] P. Mell e T. Grance, «The NIST Definition of Cloud Computing,» U.S. Department of Commerce, 2011.

[16] K. Costello, «Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.5 Percent in 2019,» Gartner, 2 April 2019. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g. [Consultato il giorno 23 January 2020].

[17] Y. Izrailevsky, «Completing the Netflix Cloud Migration,» Netflix, 11 February 2016. [Online]. Available: https://media.netflix.com/en/company-blog/completing-the-netflix-cloud-migration. [Consultato il giorno 23 January 2020].

[18] M. Fowler, «CloudComputing,» 11 July 2013. [Online]. Available: https://martinfowler.com/bliki/CloudComputing.html. [Consultato il giorno 23 January 2020].

[19] «Cloud Service and Deployment Models,» IEEE.

[20] M. Moravcik, P. Segec e M. Kontsek, «Overview of Cloud Computing standards,» ICETA, Zilina, 2018.

[21] «What are Containers and their benefits | Google Cloud,» Google Cloud, [Online]. Available: https://cloud.google.com/containers. [Consultato il giorno 12 February 2020].

[22] «Docker Containers vs. VMs: Pros and Cons of Containers and Virtual Machines,» BackBlaze, 28 June 2018. [Online]. Available:

https://www.backblaze.com/blog/vm-vs-containers/. [Consultato il giorno 12 February 2020].

[23] I. C. Education, «FaaS (Function-as-a-Service),» IBM, 30 July 2019. [Online]. Available: https://www.ibm.com/cloud/learn/faas. [Consultato il giorno 26 January 2020].

[24] «AWS Lambda,» AWS, [Online]. Available: https://aws.amazon.com/lambda/. [Consultato il giorno 26 January 2020].

[25] «IBM Cloud Functions - Pricing,» IBM, [Online]. Available: https://cloud.ibm.com/functions/learn/pricing. [Consultato il giorno 26 January 2020].

[26] M. M. Islam, S. Morshed e P. Goswami, «Cloud Computing: A Survey on its limitations and Potential Solutions,» Sydney, Dhaka, 2013.

[27] M. Roberts, «Serverless,» Martin Fowler, 22 May 2018. [Online]. Available: https://martinfowler.com/articles/serverless.html. [Consultato il giorno 24 January 2020].

[28] P. Johnston, «A simple definition of "Serverless",» 8 Settembre 2017. [Online]. Available: https://medium.com/@PaulDJohnston/a-simple-definition-of-serverless-8492adfb175a. [Consultato il giorno 3 Marzo 2020].

[29] S. Wardley, «Twitter,» 14 Dicembre 2018. [Online]. Available: https://twitter.com/swardley/status/1073379461473730560. [Consultato il giorno 3 Marzo 2020].

[30] R. Stephens, «Serverless: More Than Just Functions,» RedMonk, 14 Dicembre 2018. [Online]. Available: https://redmonk.com/rstephens/2018/12/14/serverless-more-than-just-functions/. [Consultato il giorno 3 Marzo 2020].

[31] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. M. Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica e D. A. Patterson, «Cloud Programming Simplified: A Berkeley View on Serverless Computing,» Berkeley, 2019.

[32] «What is Serverless?,» Pivotal, [Online]. Available: https://pivotal.io/serverless. [Consultato il giorno 3 Marzo 2020].

[33] N. Malishev, «AWS Lambda Cold Start Language Comparison, 2019 edition,» 4 Settembre 2019. [Online]. Available: https://levelup.gitconnected.com/aws-

lambda-cold-start-language-comparisons-2019-edition-%EF%B8%8F-1946d32a0244. [Consultato il giorno 3 Marzo 2020].

[34] M. Shilkov, «Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP,» 26 September 2019. [Online]. Available: https://mikhail.io/serverless/coldstarts/big3/. [Consultato il giorno 27 January 2020].

[35] J. Manner, M. Endreß, T. Heckel e G. Wirtz, «Cold Start Influencing Factors in Function as a Service,» UCC Companion, Bamberg, 2018.

[36] «Serverless,» AWS, [Online]. Available: https://aws.amazon.com/serverless/. [Consultato il giorno 24 January 2020].

[37] A. Pash, «Serving 39 Million Requests for $370/Month,» Postlight, 21 June 2017. [Online]. Available: https://postlight.com/trackchanges/serving-39-million-requests-for-370-month-or-how-we-reduced-our-hosting-costs-by-two-orders-of. [Consultato il giorno 24 January 2020].

[38] «Why use Serverless Computing? | Pros and Cons of Serverless,» CloudFlare, [Online]. Available: https://www.cloudflare.com/learning/serverless/why-use-serverless/. [Consultato il giorno 25 February 2020].

[39] «Function as a service Market is Projected to Attain Substantial Absolute $ Opportunity in Terms of Value Through 2023,» MarketWatch, 12 September 2019. [Online]. Available: https://www.marketwatch.com/press-release/function-as-a-service-market-is-projected-to-attain-substantial-absolute-opportunity-in-terms-of-value-through-2023-2019-09-12. [Consultato il giorno 27 January 2020].

[40] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov e C. Wu, «Serverless Computing: One Step Forward, Two Steps Back,» UC Berkley, Asilomar, 2019.

[41] «AWS Lambda - Elaborazione Serverless,» AWS, [Online]. Available: https://aws.amazon.com/it/lambda/. [Consultato il giorno 5 Marzo 2020].

[42] «Elaborazione serverless di Funzioni di Azure,» Microsoft Azure, [Online]. Available: https://azure.microsoft.com/it-it/services/functions/. [Consultato il giorno 5 Marzo 2020].

[43] «Cloud Functions,» Google Cloud, [Online]. Available: https://cloud.google.com/functions. [Consultato il giorno 5 Marzo 2020].

[44] L. Wang, M. Li, Y. Zhang, T. Ristenpart e M. Swift, «Peeking Behind the Curtains of Serverless Platforms,» UW-Madison, Ohio State University, Cornell Tech, 2018.

[45] «Github,» Github, [Online]. Available: https://github.com/. [Consultato il giorno 5 Marzo 2020].

[46] A. Palade, A. Kazmi e S. Clarke, «An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge,» Trinity College, Dublin, 2019.

[47] «Apache OpenWhisk is a serverless, open source cloud platform,» Apache OpenWhisk, [Online]. Available: https://openwhisk.apache.org/. [Consultato il giorno 5 Marzo 2020].

[48] «Home,» OpenFaaS, [Online]. Available: https://www.openfaas.com/. [Consultato il giorno 5 Marzo 2020].

[49] «Knative,» Knative, [Online]. Available: https://knative.dev/. [Consultato il giorno 5 Marzo 2020].

[50] «Serverless Functions for Kubernetes,» Fission, [Online]. Available: https://fission.io/. [Consultato il giorno 5 Marzo 2020].

[51] «Kubeless,» Kubeless, [Online]. Available: https://kubeless.io/. [Consultato il giorno 5 Marzo 2020].

[52] «Nuclio: Serverless Platform for Automated Data Science,» Nuclio, [Online]. Available: https://nuclio.io/. [Consultato il giorno 5 Marzo 2020].

[53] «An open source serverless computing platform,» OpenLambda, [Online]. Available: https://github.com/open-lambda/open-lambda. [Consultato il giorno 5 Marzo 2020].

[54] «Swarm mode overview | Docker Documentation,» Docker, [Online]. Available: https://docs.docker.com/engine/swarm/. [Consultato il giorno 12 February 2020].

[55] S. Caceres, «How does it work? Docker! Part 1: Swarm general architecture,» Octo, 8 August 2017. [Online]. Available: https://blog.octo.com/en/how-does-it-work-docker-part-1-swarm-general-architecture/. [Consultato il giorno 12 February 2020].

[56] «Production-Grade Container Orchestration,» Kubernetes, [Online]. Available: https://kubernetes.io/. [Consultato il giorno 5 Marzo 2020].

[57] «Helm Docs,» Helm, [Online]. Available: https://helm.sh/. [Consultato il giorno 5 Marzo 2020].

[58] «Service - Kubernetes,» Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/. [Consultato il giorno 19 February 2020].

[59] «Ingress - Kubernetes,» Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress/. [Consultato il giorno 19 February 2020].

[60] «Cluster Networking - Kubernetes,» Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/networking/. [Consultato il giorno 12 February 2020].

[61] «flannel is a network fabric for containers, designed for Kubernetes,» CoreOS, [Online]. Available: https://github.com/coreos/flannel. [Consultato il giorno 5 Marzo 2020].

[62] «Creating a single control-plane cluster with kubeadm - Kubernetes,» Kubernetes, [Online]. Available: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/. [Consultato il giorno 12 February 2020].

[63] «High Performance Load Balancer, Web Server, & Reverse Proxy,» NGINX, [Online]. Available: https://www.nginx.com/. [Consultato il giorno 5 Marzo 2020].

[64] «Apache CouchDB,» Apache CouchDB, [Online]. Available: https://couchdb.apache.org/. [Consultato il giorno 5 Marzo 2020].

[65] «Apache Kafka,» Apache Kafka, [Online]. Available: https://kafka.apache.org/. [Consultato il giorno 5 Marzo 2020].

[66] «Scaling-up OpenWhisk Deployment on custom-built-kubernetes cluster,» Apache OpenWhisk, 5 Dicembre 2019. [Online]. Available: https://github.com/apache/openwhisk-deploy-kube/blob/master/docs/k8s-custom-build-cluster-scaleup.md. [Consultato il giorno 5 Marzo 2020].

[67] «What is Knative?,» RedHat, [Online]. Available: https://www.redhat.com/en/topics/microservices/what-is-knative. [Consultato il giorno 12 February 2020].

[68] «Knative Eventing | Knative,» Knative, [Online]. Available: https://knative.dev/docs/eventing/. [Consultato il giorno 12 February 2020].

[69] «Getting started with App Deployment | Knative,» Knative, [Online]. Available: https://knative.dev/docs/serving/getting-started-knative-app/. [Consultato il giorno 12 February 2020].

[70] «Configuring autoscaling | Knative,» Knative, [Online]. Available: https://knative.dev/docs/serving/configuring-autoscaling/. [Consultato il giorno 12 February 2020].

[71] «Introduction - OpenFaaS,» OpenFaaS, [Online]. Available: https://docs.openfaas.com/. [Consultato il giorno 10 February 2020].

[72] «Triggers,» OpenFaas, [Online]. Available: https://docs.openfaas.com/reference/triggers/. [Consultato il giorno 10 February 2020].

[73] «Watchdog - OpenFaaS,» [Online]. Available: https://docs.openfaas.com/architecture/watchdog/. [Consultato il giorno 11 February 2020].

[74] «Create functions - OpenFaaS,» OpenFaaS, [Online]. Available: https://docs.openfaas.com/cli/templates/. [Consultato il giorno 11 February 2020].

[75] «Autoscaling - OpenFaaS,» OpenFaaS, [Online]. Available: https://docs.openfaas.com/architecture/autoscaling/. [Consultato il giorno 11 February 2020].

[76] «Concepts | Fission,» Fission, [Online]. Available: https://docs.fission.io/docs/concepts/. [Consultato il giorno 12 February 2020].

[77] «Controller | Fission,» Fission, [Online]. Available: https://docs.fission.io/docs/concepts/components/core/controller/. [Consultato il giorno 12 February 2020].

[78] «Router | Fission,» Fission, [Online]. Available: https://docs.fission.io/docs/concepts/components/core/router/. [Consultato il giorno 12 February 2020].

[79] «Executor | Fission,» Fission, [Online]. Available: https://docs.fission.io/docs/concepts/components/core/executor/. [Consultato il giorno 12 February 2020].

[80] «Function Pod | Fission,» Fission, [Online]. Available: https://docs.fission.io/docs/concepts/components/core/function-pod/. [Consultato il giorno 12 February 2020].

[81] «Apache JMeter,» Apache JMeter, [Online]. Available: https://jmeter.apache.org/. [Consultato il giorno 5 Marzo 2020].

[82] «Istio / Performance and Scalability,» Istio, [Online]. Available: https://istio.io/docs/ops/deployment/performance-and-scalability/. [Consultato il giorno 23 February 2020].

[83] «Apache Kafka: Ten Best Practices to Optimize Your Deployment,» InfoQ, 19 November 2018. [Online]. Available: https://www.infoq.com/articles/apache-kafka-best-practices-to-optimize-your-deployment/. [Consultato il giorno 10 February 2020].

[84] E. A. da Silva e G. V. Mendonca, «Digital Image Processing,» in *The Electrical Engineering Handbook*, 2005.

[85] R. B. e. a. Gaster, «Image Rotation Example,» in *Heterogeneous Computing with OpenCL (Second Edition)*, 2013.

[86] J.-C. Régin, M. Rezgui e A. Malapert, «Embarrassingly parallel search,» Antipolis, 2013.

[87] B. Marr, «What is Industry 4.0? Here's A Super Easy Explanation For Anyone,» Forbes, 2 September 2018. [Online]. Available: https://www.forbes.com/sites/bernardmarr/2018/09/02/what-is-industry-4-0-heres-a-super-easy-explanation-for-anyone/. [Consultato il giorno 27 January 2020].

[88] «Cloud Computing in the Manufacturing Industry: Don’t Get Left Behind,» IndustryWeek, 13 September 2017. [Online]. Available: https://www.industryweek.com/cloud-computing/article/22024167/cloud-computing-in-the-manufacturing-industry-dont-get-left-behind. [Consultato il giorno 27 January 2020].

[89] «Regions, Availability Zones, and Local Zones,» AWS, [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html. [Consultato il giorno 3 February 2020].

[90] F. T. C. Expert Panel, «12 Business Use Cases For Edge Computing,» Forbes, 14 August 2019. [Online]. Available: https://www.forbes.com/sites/forbestechcouncil/2019/08/14/12-business-use-cases-for-edge-computing/#7bf8db6662b4. [Consultato il giorno 3 February 2020].

[91] J. Klaess, «Edge Computing for Manufacturers,» Tulip, 14 October 2019. [Online]. Available: https://tulip.co/blog/connected-factory/edge-computing-for-manufacturers/. [Consultato il giorno 3 February 2020].

[92] «Serverless Architecture Market Size & Share | Industry Analysis, 2025,» Allied Market Research, May 2019. [Online]. Available: https://www.alliedmarketresearch.com/serverless-architecture-market. [Consultato il giorno 13 February 2020].

[93] «A specification for describing event data in a common way,» CloudEvents, [Online]. Available: https://cloudevents.io/. [Consultato il giorno 5 Marzo 2020].

[94] R. Y., «Using CloudEvents and CloudEvents Generator - Google Cloud,» Google Cloud, 6 September 2019. [Online]. Available: https://medium.com/google-cloud/using-cloud-events-and-cloud-events-generator-4b71b8a90277. [Consultato il giorno 24 February 2020].

[95] «Object Storage: An Introduction,» IBM, [Online]. Available: https://www.ibm.com/cloud/learn/object-storage. [Consultato il giorno 17 February 2020].

[96] J. MSV, «10 Key Takeaways From RightScale 2019 State Of The Cloud Report From Flexera,» Forbes, 3 Marzo 2019. [Online]. Available: https://www.forbes.com/sites/janakirammsv/2019/03/03/10-key-takeaways-from-rightscale-2019-state-of-the-cloud-report-from-flexera/. [Consultato il giorno 5 Marzo 2020].

[97] «The Future Of Cloud,» Walvis, 10 Gennaio 2020. [Online]. Available: https://www.walvis.ca/blogs/The_Future_Of_Cloud.php. [Consultato il giorno 5 Marzo 2020].

[98] C. Null, «The state of serverless: 6 trends to watch,» TechBeacon, [Online]. Available: https://techbeacon.com/enterprise-it/state-serverless-6-trends-watch. [Consultato il giorno 5 Marzo 2020].

[99] T. Seals, «One-Fifth of Open-Source Serverless Apps Have Critical Vulnerabilities,» InfoSecurity, 6 Aprile 2018. [Online]. Available: https://www.infosecurity-magazine.com/news/onefifth-of-serverless-apps/. [Consultato il giorno 5 Marzo 2020].

[100] «Currents Research on DigitalOcean,» DigitalOcean, Giugno 2018. [Online]. Available: https://www.digitalocean.com/currents/june-2018/. [Consultato il giorno 5 Marzo 2020].