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

Piattaforme FaaS per la ottimizzazione di servizi FinTech con modelli MapReduce serverless

Candidato:

Giulio Grasso

Relatore:

Chiar.mo Prof. Antonio Corradi

Correlatori:

Dott. Ing. Luca Foschini
Dott. Andrea Sabbioni
Dott. Lorenzo Patera
Dott. Ing. Filippo Bosi

Anno Accademico 2018-2019

Sessione III

Abstract

Nel panorama della gestione dei servizi business ospitati nel cloud, le soluzioni serverless, che trovano la loro massima espressione nei modelli Function as a Service (FaaS), si pongono come potenziale nuova tecnologia di punta su cui investire, con il potere di ridurre i costi di utilizzo delle risorse mentre si astraggono le difficoltà derivanti dalla configurazione e dalla gestione dei servizi. L'efficienza e l'elasticità promesse dalle FaaS, insieme ai numerosi vantaggi economici che ne derivano, hanno catturato l'interesse dei maggiori cloud provider presenti, portando ad una sempre crescente adozione della tecnologia e ad importanti investimenti nel settore. Risulta dunque utile e necessario analizzare le piattaforme disponibili e verificare l'affidabilità e le prestazioni che queste offrono in ambienti industriali, con una focalizzazione sulle soluzioni open source. In particolare, si vuole riportare un caso d'uso di funzioni serverless nell'ambito FinTech, utilizzandole come strumento per la generazione parallela di documentazione per lo smart banking in una struttura MapReduce. L'obiettivo è quello di implementare tale modello, selezionando una tra le piattaforme open source disponibili, per mostrare i vantaggi di un sistema FaaS parallelo rispetto a un tradizionale servizio sequenziale, a fronte di simulazioni di ambienti di produzione che prevedano un numero consistente di utenti.

Sommario

1 INTRODUZIONE ... 4

2 SISTEMI DISTRIBUITI, CLOUD E SERVERLESS COMPUTING ... 7

2.1 EVOLUZIONE DELLE ARCHITETTURE DISTRIBUITE ... 8
2.2 L'INTRODUZIONE AL CLOUD COMPUTING ... 12
2.2.1 Cloud pubblici, privati e ibridi ... 14
2.2.2 Best practice per servizi cloud: "12-Factor App" ... 16
2.2.3 Modelli di cloud computing ... 17
2.3 SERVERLESS E FAAS ... 20

3 PIATTAFORME DI ORCHESTRAZIONE IN OTTICA SERVERLESS ... 29

3.1 CONTAINER E DOCKER ... 29
3.2 DOCKER SWARM ... 31
3.3 KUBERNETES ... 32
3.3.1 Componenti principali in Kubernetes ... 33
3.3.2 Networking in Kubernetes ... 34
3.3.3 Kubernetes e hybrid cloud ... 37
3.4 CONFRONTO TRA DOCKER SWARM E KUBERNETES ... 38
3.5 MICROVM E FIRECRACKER ... 39

4 PIATTAFORME FUNCTION AS A SERVICE ... 41

4.1 STATO DELL'ARTE DELLE PIATTAFORME FAAS ... 41
4.2 PIATTAFORME FAAS OPEN SOURCE ... 44
4.3 OPENFAAS ... 45
4.3.1 Architettura ... 46
4.3.2 Utilizzo ... 48
4.3.3 Autoscaling ... 49
4.4 OPENWHISK ... 50
4.4.1 Architettura ... 50
4.4.2 Utilizzo ... 52
4.4.3 Autoscaling ... 52
4.5 KNATIVE ... 54
4.5.1 Architettura ... 54
4.5.2 Utilizzo ... 58
4.5.3 Autoscaling ... 59
4.6 FISSION ... 60
4.6.1 Architettura ... 61
4.6.2 Utilizzo ... 63
4.6.3 Autoscaling ... 64

5 COMPARAZIONE DELLE PIATTAFORME **65**

5.1 CONSIDERAZIONI QUALITATIVE 66
5.2 APACHE JMETER 69
5.3 TEST DI SCALABILITÀ 71
5.3.1 Risultati sul cluster *74*
5.3.2 Risultati in locale *77*
5.4 TEST SUL CONSUMO DELLE RISORSE 79
5.4.1 Risultati test matrice 10000x10000 .... *81*
5.4.2 Risultati test matrice 15000x15000 .... *85*
5.4.3 Risultati test con frequenze maggiorate *87*
5.5 CONSIDERAZIONI CONCLUSIVE 88

6 CASO D'USO: MAPREDUCE SERVERLESS IN AMBITO FINTECH **91**

6.1 MAPREDUCE 92
6.1.1 L'esempio del word count *94*
6.2 STATO DELL'ARTE DEL MAPREDUCE SERVERLESS 95
6.3 IMPLEMENTAZIONE 100
6.3.1 Apache Kafka e Zookeeper *104*
6.3.2 MinIO *106*
6.3.3 Restdb *107*
6.3.4 Realizzazione delle funzioni *109*
6.4 ANALISI DEL WORKFLOW 121
6.5 DESCRIZIONE DEI TEST 122
6.6 RISULTATI 124
6.6.1 Risultati test con pagine incrementali *124*
6.6.2 Risultati test con richieste incrementali *127*
6.6.3 Risultati test con frequenze maggiorate *130*
6.6.4 Indagine sui cold start *131*

7 CONCLUSIONI E SVILUPPI FUTURI **133**

BIBLIOGRAFIA **137**

1 Introduzione

Nel corso degli ultimi 40 anni, la mole di dati prodotta ed elaborata sulla rete è cresciuta costantemente, presentando un andamento esponenziale. Per far fronte a una tale espansione, sempre più aziende hanno dovuto, come punto di partenza, riorganizzare la propria infrastruttura, abbandonando il tradizionale modello monolitico, per orientarsi verso scenari distribuiti. Tuttavia, sebbene questo abbia concesso ai sistemi di tali aziende di rimanere saldi senza farsi sopraffare dalla quantità di richieste e operazioni da smaltire, la gestione dell'infrastruttura e dei servizi è diventata presto una limitazione per questioni di spazi, tempi, costi e competenze. La soluzione a questo problema esiste e viene proposta sotto il nome di **cloud computing**.

Oggi, la quasi totalità delle aziende ha affidato la gestione dei propri servizi a provider di servizi cloud, evitando di dover spendere ingenti somme di denaro in server, spesso costosi e difficilmente gestibili, soprattutto senza le adeguate conoscenze. Sebbene le fondamenta e il concetto di cloud siano tutt'altro che recenti, il moderno approccio di affitto di risorse virtualmente illimitate ha subito, nello scorso decennio, una forte spinta da parte di *Amazon AWS*. L'azienda americana, capendo di poter sfruttare le enormi risorse che già aveva a disposizione, ma che non utilizzava completamente, propose un modello di business basato sull'affitto remoto di queste. Da quel momento in poi, ha preso piede una sorta di "competizione" tra le più importanti compagnie di servizi Internet, quali *Google*, *Microsoft*, *IBM*, e molti altri, in cui ognuna cercava di offrire l'ambiente cloud migliore, con le risorse più affidabili e ai prezzi più vantaggiosi. Anni di evoluzione e consolidamento verso l'approccio cloud hanno permesso la proliferazione di modelli atti a garantire un livello sempre maggiore di astrazione dei dettagli non legati alla business logic di un'applicazione.

Nel 2014, a seguito di lunghe ricerche e investimenti ingenti, Amazon presenta l'innovativo progetto *Lambda* [1], un nuovo servizio che promette di rendere accessibile a chiunque il potere computazionale presente nel cloud, fino a quel momento gestibile unicamente da chi possedeva le competenze necessarie. Prima di *Lambda*, infatti, ogni azienda doveva disporre di almeno un esperto in amministrazione di sistemi, poiché erano richieste complete conoscenze nell'ambito dei sistemi distribuiti e della sicurezza per poter gestire i propri servizi in cloud. Al contrario, il servizio proposto da Amazon si basa su un nuovo modello, chiamato **serverless computing**, che consiste in un ulteriore progresso nella gestione dei carichi di lavoro, astraendo ogni aspetto relativo a complessità di gestione e configurazione del sistema distribuito sottostante. L'aspetto cardine che lo differenzia dai precedenti modelli è che l'allocazione delle risorse richieste da un utente viene fatta on-demand, senza che vi sia un insieme di unità computazionali prestabilite e prepagate. Le tariffe presentate poi all'utente sono calcolate in base al tempo di utilizzo di queste risorse o al numero di richieste effettuate. Ciò diventa estremamente

vantaggioso da un punto di vista economico, andando a semplificare, in aggiunta, il processo di pianificazione delle risorse necessarie a svolgere una determinata operazione.

Altri punti a favore sono senz'altro la scalabilità e l'elasticità che un sistema così dinamico e aperto possono offrire, pur non mancando alcuni svantaggi sul piano delle performance e della sicurezza. Il primo di questi due aspetti, le performance, è un problema di trade-off con il vantaggio economico: se da un lato l'attivazione di risorse on-demand dà la possibilità di risparmiare durante i periodi di inutilizzo del sistema, dall'altro lato genera tempi di latenza maggiori rispetto a un ambiente tradizionale perennemente in ascolto, con richieste che possono essere servite in tempi dilatati dovuti alle attese necessarie allo startup delle risorse.
Il secondo aspetto, la sicurezza, è legato principalmente al fatto che, in generale, un sistema composto da più servizi dislocati presenta maggiori vulnerabilità, essendo ogni componente un potenziale punto di accesso per un attaccante. Inoltre, l'elevato livello di astrazione realizzato dal modello serverless impedisce agli utilizzatori di controllare o installare misure di sicurezza personali e ad hoc.

Come si vedrà più avanti nella trattazione, il serverless è un concetto ampio e generico, che racchiude diverse soluzioni, ma nella fattispecie ci si soffermerà sul suo servizio principale, ovvero il meccanismo delle **Function as a Service** (**FaaS**). In estrema sintesi, si potrebbe per il momento introdurre il servizio FaaS come un paradigma tramite il quale è sufficiente scrivere una funzione in un qualsiasi linguaggio e invocarla, delegando alla piattaforma utilizzata (es. *Lambda*) l'onere di gestire le risorse necessarie all'esecuzione, pagando in base alla sola durata dell'operazione. In quanto tema centrale del lavoro qui presentato, sia da un punto di vista teorico che tecnologico, questo nuovo modello di cloud computing sarà ampiamente ricoperto nelle sezioni successive.

Dopo la presentazione di *Lambda*, diversi sono stati i competitor entrati nel mercato serverless, incrementando il numero di soluzioni offerte, e, nel corso degli anni, la relativa qualità di ognuna di queste. Il presente lavoro mira alla valutazione e conseguente utilizzo dei più influenti framework FaaS, concentrandosi in particolar modo sul mondo open source. La trattazione parte da una panoramica sui sistemi distribuiti e sul cloud computing, per spiegare cosa questi hanno rappresentato e continuano a rappresentare negli scenari industriali, analizzando l'evoluzione che il modello ha dovuto subire a causa di motivazioni socio-tecnologiche, fino a portare alla nascita del serverless computing. Segue una descrizione dettagliata delle tecnologie che rendono possibile la realizzazione delle FaaS, per poi passare a un'analisi dei principali framework serverless open source, approfondendo le loro architetture e fornendo valutazioni sia quantitative che qualitative del loro comportamento e performance. Le metriche di valutazione verteranno principalmente su:

- Throughput

- Percentuale errori
- Scalabilità
- Occupazione risorse
- Complessità d'uso

Si è scelto di affacciarsi al mondo open source per avere maggiore controllo e visibilità sul processo di esecuzione, senza la necessità di sottoscrivere alcun piano tariffario con un provider.
Lo scopo iniziale è capire se le piattaforme analizzate siano in grado di sostenere elevati carichi di richieste parallele, sottoponendole a stress test orientati alla scalabilità, con numeri che simulino diversi scenari industriali. Una volta stabilito ciò, i risultati ottenuti dalle varie soluzioni dovranno essere confrontati, per poter eleggere la candidata più promettente sul piano della scalabilità. Successivamente dovrà essere testato l'aspetto su cui si insiste maggiormente nel caso d'uso in esame, come ad esempio l'occupazione della memoria o l'utilizzo della CPU. Le considerazioni che ne derivano potranno così aiutare nella scelta del framework più adatto al progetto **FinTech** in questione, ovvero la generazione di report bancari in **PDF** nell'ambito dell'home banking. L'idea è quella di realizzare una struttura assimilabile a un **MapReduce** per dividere il pesante task di generazione di PDF in più subtask paralleli, implementati come funzioni serverless, con particolare attenzione alla modellazione di un punto di sincronizzazione tra queste, ottenendo, come step finale del workflow, un unico output.

2 Sistemi distribuiti, cloud e serverless computing

L'era contemporanea dei settori IT e della programmazione è caratterizzata da una certa mobilità dei componenti e da meccanismi di comunicazione che si verificano su connessioni remote tra sistemi distribuiti. Il concetto di sistema distribuito può essere riassunto dall'immagine di una collezione di componenti computazionali dislocati che cooperano e si coordinano per apparire come un sistema unico e coerente. Gli approcci distribuiti prevedono la disseminazione di informazioni e dati su nodi più o meno distanti tra loro per ottenere un migliore grado di affidabilità e velocità, andando a formare paradigmi di progettazione che aiutano a costruire un design portabile, e, eventualmente, orientato alla mobilità, con una spinta verso l'utilizzo di piattaforme eterogenee. L'introduzione dei sistemi distribuiti, fino a quella del cloud, è stata, come si vedrà, un'evoluzione necessaria al supporto dei moderni servizi, portando con sé tanta complessità quanta elasticità e disponibilità. È soprattutto quest'ultimo il punto focale di un sistema cloud: la possibilità di usufruire di un sistema replicato che non insiste su colli di bottiglia e *single point of failure*, capace di rispondere prontamente alle richieste, a prescindere da guasti o errori nei componenti. Tuttavia, una disponibilità elevata va in conflitto con altre importanti caratteristiche, alle quali, in un sistema centralizzato, non era necessario rinunciare.

Quando ci si muove da un sistema centralizzato a un sistema distribuito bisogna infatti considerare il cosiddetto **teorema CAP**, enunciato da Eric Brewer, professore emerito dell'università di Berkeley e vicepresidente dell'infrastruttura in Google. Egli afferma e dimostra che in un qualsiasi sistema distribuito si possono garantire, come si evince dalla Figura 1, soltanto due delle seguenti tre proprietà:

1. *Consistency* (consistenza) - tutti i nodi del sistema vedono gli stessi dati nello stesso momento, anche in presenza di aggiornamenti; ad esempio, la lettura di un dato deve restituire a ogni nodo lo stesso risultato, ovvero il valore scritto più recentemente.
2. *Availability* (disponibilità) - tutti i nodi possono sempre trovare delle repliche pronte a rispondere e ogni richiesta riceve sempre una risposta corretta, anche in presenza di guasti in una o più parti del sistema.
3. *Partition tolerance* (tolleranza di partizione) - il sistema distribuito continua a funzionare, anche in presenza di ritardi o perdita di pacchetti nella comunicazione tra i nodi, problemi frequenti e inevitabili quando si effettua un partizionamento della rete.

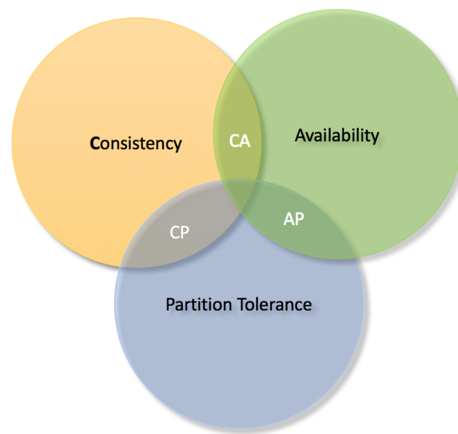

Figura 1 - Diagramma teorema CAP

È proprio da queste tre proprietà che deriva il nome del teorema (C-A-P), la cui spiegazione risulta particolarmente intuitiva. Se si desidera, ad esempio, mantenere un sistema che sia allo stesso tempo consistente e disponibile (C+A) non è possibile avere una rete partizionata (P), in quanto l'avere sempre disponibile un dato aggiornato su tutti i nodi non ammette il verificarsi di perdite o ritardi di pacchetti tra questi: CA è tipicamente l'approccio implementato nei database, dove si fa uso di transazioni tra nodi a stretto contatto. Se invece si vuole avere un sistema partizionato e altamente disponibile (P+A), alla richiesta di un dato, ogni nodo deve sempre restituire la versione più recente che conosce, che però, a causa del partizionamento, potrebbe essere meno aggiornata dell'ultima scrittura effettiva nel sistema, violando il principio di consistenza (C): PA è il modello utilizzato per i servizi DNS, con i quali può capitare che, in diverse parti del mondo, dopo un aggiornamento, per qualche istante si ricevano record DNS diversi. Anche i sistemi cloud vengono comunemente inseriti in questa categoria, adottando un approccio ottimistico più orientato verso la reattività che verso la consistenza, definita per l'appunto "*eventual consistency*", nel senso che, se si dovessero verificare inconsistenze nei dati, queste non verrebbero risolte con priorità massima, accettando dunque la presenza di dati incoerenti per alcuni periodi di tempo.

Viene adesso riportata una panoramica sui modelli che hanno condotto, nel corso degli ultimi decenni, alle moderne architetture distribuite e al cloud, cercando di motivare e giustificare le scelte architetturali fatte, con una rapida analisi dei vantaggi e degli svantaggi di ognuno.

2.1 Evoluzione delle architetture distribuite

Gli elementi costitutivi di ogni applicazione enterprise sono:

- Logica di presentazione
- Logica di business
- Logica di accesso ai dati

La logica di presentazione fornisce l'interfaccia utente dell'applicazione (UI), si trova al livello più alto, visualizza informazioni relative a funzionalità *user-oriented* ed è responsabile di gestire le interazioni tra utente e sistema.
La logica di business, altresì chiamato livello applicativo, controlla le funzionalità centrali di un'applicazione, tipicamente esponendo interfacce verso i servizi disponibili, che possono essere invocati dall'esterno per avviare la computazione.
La logica di accesso ai dati include i meccanismi di persistenza, orientati verso i database e la condivisione di file, fornendo interfacce accessibili dai componenti della logica di business [2].

Il modello centralizzato, in cui i livelli di presentazione, business logic e accesso ai dati sono interdipendenti e fusi insieme in un'applicazione monolitica, è stato utilizzato nelle architetture dei sistemi informatici fino agli inizi degli anni '80. Chiamato anche *single-tier*, consiste in terminali "dumb", connessi direttamente a un mainframe, che si preoccupano soltanto di visualizzare ciò che accade nel server, che invece ospita tutta la computazione. Una tale tassonomia, descritta in Figura 2, si appoggia pesantemente su importanti risorse di rete e su una solida infrastruttura per processamento e storage. Diversi sono i vantaggi di questo approccio, specie se si considera le relativamente ridotte prestazioni delle macchine di quegli anni. In particolare, il centralizzato permette una certa efficienza e semplicità, non essendo presente l'overhead e la complessità legata alle comunicazioni remote.

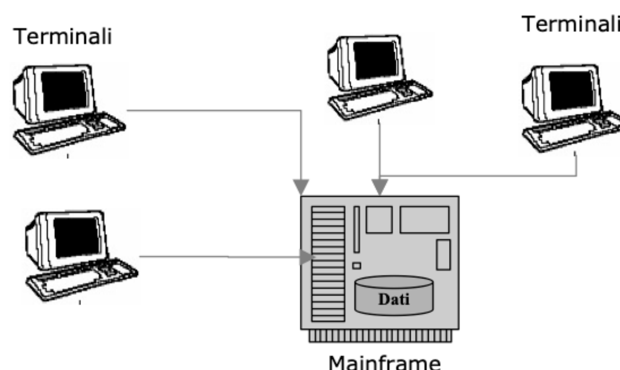

Figura 2 - Modello single-tier

Tuttavia, la crescita della rete, l'evoluzione dell'hardware e una certa necessità da parte delle aziende di una scalabilità orizzontale[1] hanno fatto diventare il modello *single-tier* obsoleto nel decennio successivo, in favore di un'architettura *client-server* (o *two-tier*), come quella presentata in Figura 3. L'idea è quella di un client che richiede dei servizi a un server, il quale dovrà elaborare la richiesta e restituire, eventualmente, un risultato. Parte consistente della logica di business viene fatta girare sul client, cominciando quindi a separare livelli e responsabilità.

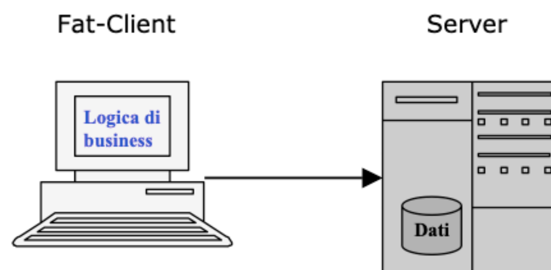

Figura 3 - Modello two-tier

Sebbene questo modello abbia segnato un'importante evoluzione e apertura nei sistemi, risulta ancora molto limitato, specie per quanto riguarda l'eccessivo accoppiamento tra le entità, la sicurezza dei dati e l'ancora non sufficiente livello di scalabilità, dovendo gestire numerose connessioni di più client verso lo stesso server, che diventa inevitabilmente un collo di bottiglia. Un altro disagio è presente in fase di aggiornamento di uno o più dei componenti, generando la necessità di aggiornare l'intero sistema, non essendoci livelli separati. Infine, livelli diversi fusi in un'unica entità comportano l'impossibilità di specializzazione di questi.

Ciò che ha permesso un'ulteriore evoluzione è stato innanzitutto lo sviluppo tecnologico in termini di networking, interessato da un importante ampliamento della banda e da una continua standardizzazione dei protocolli di rete. A questo si aggiunge il bisogno, da parte delle aziende, di distribuire i propri servizi su una scala maggiore, con le problematiche legate alla gestione online di importanti volumi di transazioni, evitando al contempo il *single point of failure* presente nelle architetture esistenti. Si transita così ai sistemi *three-*

[1] Con scalabilità orizzontale si intende l'aumentare la capacità di un sistema collegando più entità hardware o software in modo che funzionino come una singola unità logica. L'approccio contrario, chiamato scalabilità verticale, consiste invece nell'aumentare la capacità andando ad aggiungere più risorse sullo stesso hardware o sostituendo direttamente la macchina presente con una più performante.

tier (o “a layer”), il cui punto di forza sta in una netta separazione delle logiche in livelli, i quali comunicano utilizzando interfacce e protocolli ben definiti che riescono a offrire un ottimo disaccoppiamento. I tre livelli in questione, visualizzabili in Figura 4, sono:

- *Client-tier* - il livello dell’interfaccia utente concretizzato generalmente in un client “leggero”.
- *Business-tier* - il livello in cui risiedono i componenti che implementano la logica di business.
- *Resource-tier* - il livello dei sistemi informativi di backend che si occupano della gestione dei dati e delle funzionalità core.

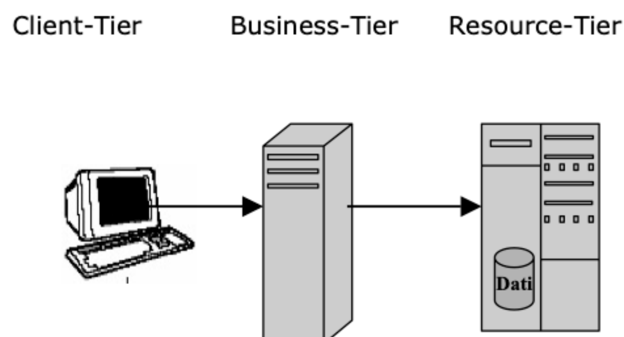

Figura 4 - Modello three-tier o "a layer"

Con questo modello si raggiunge finalmente un’adeguata scalabilità, un buon livello di sicurezza e una separazione delle responsabilità che rende possibile la modifica dei singoli livelli con un impatto minimo sui rimanenti. L’approccio appena descritto ha permesso per anni di gestire elasticamente la maggior parte dei sistemi aziendali presenti. Tuttavia, il livello di informatizzazione raggiunto nella società odierna e l’ormai totale diffusione dei servizi offerti in rete hanno portato a requisiti di scalabilità, affidabilità e bilanciamento del carico tali da saturare, in molti scenari, anche il modello a tre livelli. Inoltre, negli ultimi anni, ha preso piede una forte tendenza e necessità di integrare o interconnettere servizi risiedenti su server diversi e dislocati, che cela complessità di design e di amministrazione incompatibili con il semplice modello a layer.

La parola chiave che sblocca il passaggio alle vere architetture distribuite è “*middleware*”, inteso come lo strato posto nel mezzo che va a coprire le necessità di integrazione e comunicazione delle applicazioni nel distribuito. Questo nuovo layer viene schematizzato nella Figura 5 sottostante.

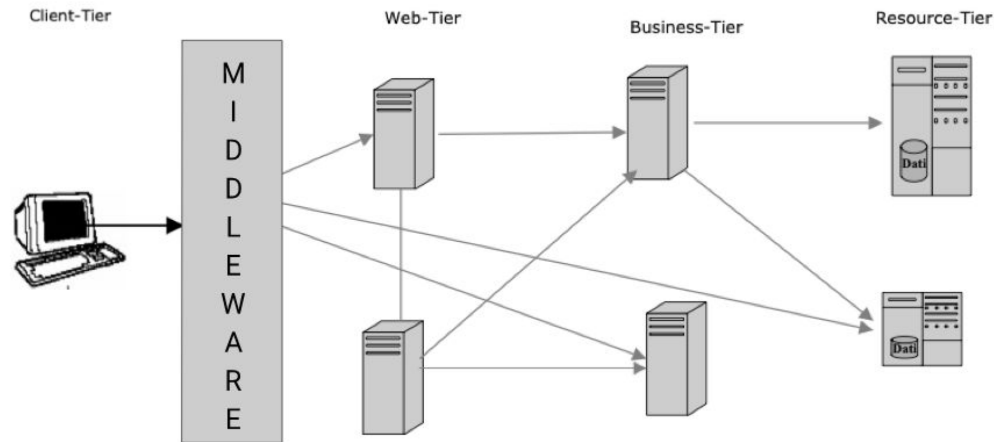

Figura 5 - Modello distribuito con strato middleware

I vantaggi principali del middleware sono il forte disaccoppiamento tra i componenti in gioco, la razionalizzazione delle comunicazioni, l'uniformità tecnologica e applicativa e l'esposizione di interfacce chiaramente definite per l'accesso ai servizi. Esistono varie categorie di middleware a seconda dello scope dell'applicazione su cui si vanno ad applicare, ma tutti condividono i concetti appena citati, improntati all'unificazione di ambienti eterogenei e alla trasparenza della distribuzione.

Avendo chiare le motivazioni che hanno condotto alla scelta e conseguente diffusione dei sistemi distribuiti, la trattazione procede nello spiegare le modalità con cui le aziende hanno cominciato ad adottare questo modello, in un percorso che ha portato fino alla nascita del cloud computing.

2.2 L'introduzione al cloud computing

Il cloud computing consiste nella distribuzione on-demand delle risorse IT tramite Internet, con una tariffazione basata sul consumo. L'introduzione del cloud è giustificata principalmente da tre fattori che si sono fatti strada nel corso degli ultimi anni, ovvero la possibilità, da parte dei computer, di offrire potenza e risorse di calcolo sempre maggiori, l'apertura della quasi totalità delle aziende verso il web e la richiesta, da parte degli utenti, di servizi sempre più reattivi e affidabili. Quando le prime aziende cominciarono a percepire la necessità di affacciarsi al mondo del web e di mantenere un sistema distribuito, cercarono di costruire e mandare avanti un proprio data center interno, preoccupandosi di tutti gli aspetti gestionali, dal problema dell'ospitare fisicamente decine o centinaia di server, contando anche l'onere di doverli installare e manutenere, fino alla configurazione e al mantenimento di tutti gli strumenti software necessari e delle loro operazioni. Risulta evidente come, una soluzione simile, denominata **on-premise**, con la

crescita esponenziale dei servizi offerti, caratterizzati da vincoli di disponibilità e velocità sempre più restrittivi, risultò presto insufficiente e inefficiente. I costi di acquisto e manutenzione erano troppo elevati, i sistemi troppo complessi, e vi erano inoltre seri problemi legati alla scalabilità, poiché si era fisicamente limitati dalla potenza computazionale posseduta, senza la possibilità di distribuire il carico su risorse supplementari.

Il primo passo avanti si identifica nel concetto di **outsourcing**, che fa fronte alle problematiche esposte andando ad affidare, dietro pagamento, la gestione di una parte del sistema distribuito a un provider esterno, seguendo un accordo denominato *SLA* (*Service Level Agreement*). Uno *SLA* è, a tutti gli effetti, uno strumento contrattuale attraverso il quale si definiscono le metriche di servizio, come ad esempio la *QoS* (*quality of service*), il tempo medio di risposta o il tempo massimo di recovery da un guasto, che devono essere rispettate da un fornitore di servizi (provider) nei confronti dei propri clienti. Con questo sistema si è inizialmente riusciti ad arginare il problema, ma l'ovvia e naturale evoluzione dell'outsourcing è stata la successiva migrazione dell'interno sistema su un provider esterno. Da qui nasce l'idea di **cloud**, ovvero un modello in cui ogni tipo di servizio è mantenuto da terzi (cloud provider), è accessibile tramite web, e sottostà a determinati vincoli prestazionali definiti in un contratto. Il tutto avviene letteralmente affittando capacità di calcolo, storage e database, sulla base delle proprie necessità, senza doversi più preoccupare di esaurire le risorse in caso di momenti di picco di richieste o di riservare grandi spazi fisici ai server, con tutti i problemi che derivano dalla configurazione, dal mantenimento, dal raffreddamento e dalla riparazione di questi.

Uno dei primi importanti provider a farsi strada in questa direzione è stato Amazon, il quale, ad un certo punto, era arrivato a possedere una potenza computazionale eccessiva per i propri bisogni, tanto da poterne offrire una buona parte a organizzazioni di ogni tipo, dimensione e settore, per una vastissima gamma di casi d'uso, quali backup dei dati, disaster recovery, e-mail, desktop virtuali, sviluppo e test di software, analisi di Big Data, machine learning e applicazioni Web destinate ai clienti [3].

Volendo elencare le caratteristiche principali del cloud, si hanno:

- Risorse virtualizzate pressoché illimitate, fornite on-demand pagando in base alle capacità desiderate.
- Ambiente affidabile e altamente disponibile, controllato da un contratto di tipo SLA.
- Possibilità di aumentare con elasticità le risorse a disposizione in tempo reale, senza dover allocare in anticipo una quantità maggiore al necessario.
- Elevata scalabilità a fronte di traffico intenso.
- Sicurezza e protezione controllata dei dati garantite dall'utilizzo di strumenti affidabili e consolidati.

- Costi di gestione prossimi allo zero.

2.2.1 Cloud pubblici, privati e ibridi

La distribuzione delle risorse in cloud è disponibile in diverse opzioni che permettono una categorizzazione dipendente dalle esigenze aziendali:

- **Cloud pubblico**
- **Cloud privato**
- **Cloud ibrido**

In un **cloud pubblico** l'infrastruttura e le risorse sono rese disponibili e distribuite alle aziende tramite internet, con pagamento generalmente legato al consumo di queste. Gli utenti possono decidere di aumentare le risorse on-demand e non necessitano di acquistare nessun tipo di hardware per usare il servizio. Le funzionalità computazionali offerte possono variare dal semplice servizio web come un client e-mail, a piattaforme di storage e gestione di applicazioni, fino a interi ambienti di infrastruttura usati per sviluppo e test. In qualità di cloud, tutte le risorse sono sotto controllo del provider in questione, che le gestisce interamente, offrendole come servizio a diverse organizzazioni (sistema *multi-tenant*).
Questo aspetto porta con sé una serie di svantaggi legati innanzitutto alla sicurezza dei dati e alla privacy, nel senso che non si ha il controllo su dove e come i dati di un'azienda vengono memorizzati, né su eventuali utenti non autorizzati che tentano di accedervi. Inoltre, l'avere un pool di risorse dal quale attingono più organizzazioni potrebbe portare, in alcune condizioni di traffico particolarmente critiche, una difficoltà nel rispettare la QoS e i vincoli richiesti per tutti i clienti. Al di là di ciò, il cloud pubblico resta una soluzione ideale per tutte le organizzazioni di piccole-medie dimensioni che hanno un limitato numero di risorse, sia IT che umane, e alle quali conviene, da un punto di vista economico e gestionale, affidarsi a un sistema dai costi relativamente ridotti, dotato di un'ottima scalabilità, senza necessità di manutenzione e ad elevata disponibilità.

Alternativamente esiste la soluzione del **cloud privato**, in cui le risorse non sono più condivise tra più enti, ma dedicate alla singola organizzazione (sistema *single-tenant*). Tali risorse possono essere collocate on-premise o mantenute dai provider e vengono fornite attraverso una rete privata, sicura e personale. Non è dunque pensato per offrire servizi al pubblico, ma per un uso interno all'organizzazione e ai suoi dipendenti. L'idea è quella di disporre della stessa potenza che si avrebbe in un cloud pubblico, senza rinunciare ai lati positivi di un data center personale. I vantaggi principali offerti da questo ambiente isolato stanno ovviamente nella privacy e nella sicurezza dei dati, ed è quindi

pensato per industrie con rigidi vincoli di controllo sui flussi di dati. Allo stesso tempo, un cloud privato consente di avere una maggiore visibilità e controllo sull'infrastruttura ed è quindi possibile modellare e trasformare alcuni aspetti di basso livello per adattarli meglio a ciò che l'organizzazione richiede. Una simile soluzione ha però alcuni svantaggi importanti da tenere in considerazione, come il costo certamente più elevato rispetto a quello di un cloud non dedicato, con conseguente allocazione più limitata delle risorse e quindi una scalabilità meno elastica. Infine, va tenuto in considerazione che un cloud privato si scontra con la diffusione dei dispositivi mobili, nel senso che un utente in movimento potrebbe avere un accesso limitato al proprio cloud privato, non potendo rispettare le misure di sicurezza previste ovunque egli si trovi [4].

Non sorprende dunque la presenza di una terza e ultima soluzione che tenta di integrare i vantaggi di entrambi i modelli sopra menzionati, chiamata **cloud ibrido**, o, più comunemente, **hybrid cloud**. Questo si pone come la composizione di almeno un cloud pubblico e almeno un cloud privato, sulla base di una partnership che si viene a formare tra i rispettivi vendor. Le due parti sono entità logicamente separate, ma allo stesso tempo legate da una tecnologia che abilita la portabilità dei dati e delle applicazioni tra esse. In particolare, il cloud ibrido parte dalla stessa base del cloud privato, con l'obiettivo di sopperire ai problemi di scalabilità e elasticità che lo affliggono. Ad esempio, un'organizzazione che utilizza un cloud privato per i propri workflow interni standard, sui quali è garantita sicurezza e confidenzialità, potrebbe appoggiarsi, in caso di picco di richieste, al provider del cloud pubblico per ottenere risorse extra per un tempo limitato, in modo rapido e flessibile, senza l'overhead di dover istanziare una sovrastruttura di comunicazione sicura e privata. Un altro esempio di utilizzo è quello di un'organizzazione che necessita di un cloud privato per le operazioni interne, ma che espone anche servizi e dati agli utenti (es. CRM) che possono essere tranquillamente ospitati da un cloud pubblico, andando a ottenere un vantaggio sia in termini di costi che di scalabilità, aspetto significativo essendo questi soggetti a un numero potenzialmente più elevato di richieste [5].

L'obiettivo di unire insieme i vantaggi delle due soluzioni, privata e pubblica, sembra dunque raggiungibile grazie alle proposte ibride, tuttavia sono anch'esse affette da alcune limitazioni:

- Sono in assoluto i modelli meno economici, sia se si desidera affidarsi a due provider, in quanto sarà presente un vendor in più da pagare, sia se si decide di ospitare la parte privata on-premise, con tutti i costi relativi a installazione, gestione e manutenzione.
- È richiesta una forte compatibilità e integrazione tra le due infrastrutture cloud, difficile da realizzare soprattutto sulla parte pubblica, in cui l'organizzazione ha uno scarso controllo degli ambienti.

- È necessario introdurre complessità architetturale, aggiungendo strumenti e livelli di astrazione per permettere l'omogeneità tra i sistemi e la portabilità delle applicazioni.

Nel corso della trattazione verranno presentate le modalità con cui è possibile risolvere alcuni dei problemi del cloud ibrido grazie all'utilizzo dei container e delle piattaforme di orchestrazione per container.

2.2.2 Best practice per servizi cloud: "12-Factor App"

Il nome "*12-Factor*" deriva dall'individuazione di dodici linee guida da seguire per sviluppare coerentemente servizi in cloud. Le prime tre di queste riguardano la gestione e il corretto approccio alle fasi di scrittura, build e deployment dell'applicazione:

1) *Codebase* - ad ogni applicazione deve corrispondere un'unica codebase (o repository), gestita da un sistema di controllo di versione (es. Git), e tanti deployment (es. un Git branch per ogni sviluppatore o feature).
2) *Build, release, execute* - ci deve essere una separazione netta tra la fase di build e quella di esecuzione.
3) *Parità tra sviluppo e produzione* - l'ambiente di sviluppo e testing va mantenuto il più simile possibile a quello di produzione (es. digital twin).

Nel secondo blocco si trovano gli aspetti statici relativi alle dipendenze e alle configurazioni del sistema:

4) *Dipendenze* - le dipendenze devono essere dichiarate e isolate.
5) *Configurazione* - le informazioni di configurazione vanno memorizzate come variabili di ambiente.
6) *Backing Service* - i servizi supplementari (es. database) devono essere utilizzati come risorse esterne accessibili tramite API.
7) *Binding delle porte* - i servizi vanno esposti tramite binding delle porte.

I restanti cinque fattori coprono alcuni dettagli dinamici che riguardano la fase di esecuzione:

8) *Processi* - bisogna eseguire l'applicazione come uno o più processi stateless.
9) *Concorrenza* - è necessario poter scalare per far fronte a un aumento delle richieste dirette verso il servizio.
10) *Rilasciabilità* - è da massimizzare la robustezza nei transitori con avvii veloci e terminazioni non brusche.
11) *Log* - bisogna trattare i log come stream di eventi.

12) *Processi di amministrazione* - è buona norma eseguire i task di amministrazione e management come processi una tantum, rispettando il principio di minima intrusione [6].

La *Twelve-factor app* è una metodologia di sviluppo orientata alla costruzione di applicazioni distribuite in cloud, che pertanto devono rispettare alcune caratteristiche fondamentali, una serie di principi e best practice per ottenere un servizio sufficientemente elastico e scalabile:

- Seguire un formato dichiarativo per l'automazione della configurazione, minimizzando tempi e costi di ingresso per ogni sviluppatore che si aggiunge al progetto.
- Interfacciarsi in modo pulito con il sistema operativo sottostante, per offrire la massima portabilità sui vari ambienti di esecuzione.
- Essere conformi e adattarsi alle più recenti cloud platform, ovviando alla necessità di server e amministrazioni di sistema.
- Minimizzare la divergenza tra sviluppo e produzione, permettendo il continuous deployment per una massima espressione della filosofia *agile*.
- Scalare significativamente senza troppi cambiamenti ai tool, all'architettura e al processo di sviluppo.

Su questi principi si basa la modellazione delle varie tipologie di servizi realizzabili in ambiente cloud, esposti nella sezione che segue.

2.2.3 Modelli di cloud computing

Il cloud incarna l'idea, molto generica, di **RaaS** (**Resource as a Service**), dove per "service" si intende l'astrazione di un processo logico, di una risorsa o di un'applicazione, che nasconde i dettagli implementativi. In particolare, questa concezione porta alla definizione di alcuni modelli, o tipologie di cloud computing, che portano il nome rispettivamente di **IaaS** (**Infrastructure as a Service**), **PaaS** (**Platform as a Service**) e **SaaS** (**Software as a Service**). Ognuno di questi modelli va ad agire su un livello diverso dello stack che parte dall'infrastruttura hardware fino ad arrivare al piano applicativo.
Partendo dal livello più basso, si trova il modello IaaS, il quale astrae gli elementi fondamentali di base dell'infrastruttura IT del cloud e consente l'accesso a risorse virtualizzate relative al networking, alla computazione e allo spazio di storage dei dati. Risalendo i livelli si trova poi il servizio PaaS, grazie al quale le organizzazioni non devono più gestire l'infrastruttura di basso livello, come hardware e sistemi operativi.

Questo servizio si occupa della gestione della distribuzione delle applicazioni, fornendo una serie di strumenti utili a sviluppatori e ingegneri del software, i quali potranno concentrarsi su caratteristiche di alto livello, senza doversi dedicare ad attività quali l'approvvigionamento delle risorse, la pianificazione della capacità, l'applicazione di patch o qualsiasi altro tipo di attività onerosa che possa interessare l'esecuzione delle applicazioni.
Infine, nella parte più alta dello stack, il modello SaaS va ad offrire, agli utilizzatori finali, un prodotto completo che viene eseguito e gestito dal provider di servizi. Grazie alla soluzione SaaS non è più necessario preoccuparsi neanche delle modalità di gestione del servizio, oltre che dell'infrastruttura sottostante. L'utente può concentrarsi esclusivamente sull'uso di un software particolare, quale potrebbe essere un client di posta elettronica accessibile via web, senza il bisogno di installare o configurare nulla sulla propria macchina [7]. La Figura 6 permette di visualizzare lo stack appena descritto, evidenziando la netta separazione di responsabilità esistente tra i tre modelli. Questi vengono inoltre messi a confronto con lo stack previsto in uno scenario on-premise, in cui ogni livello è di responsabilità dell'ente proprietario del servizio.

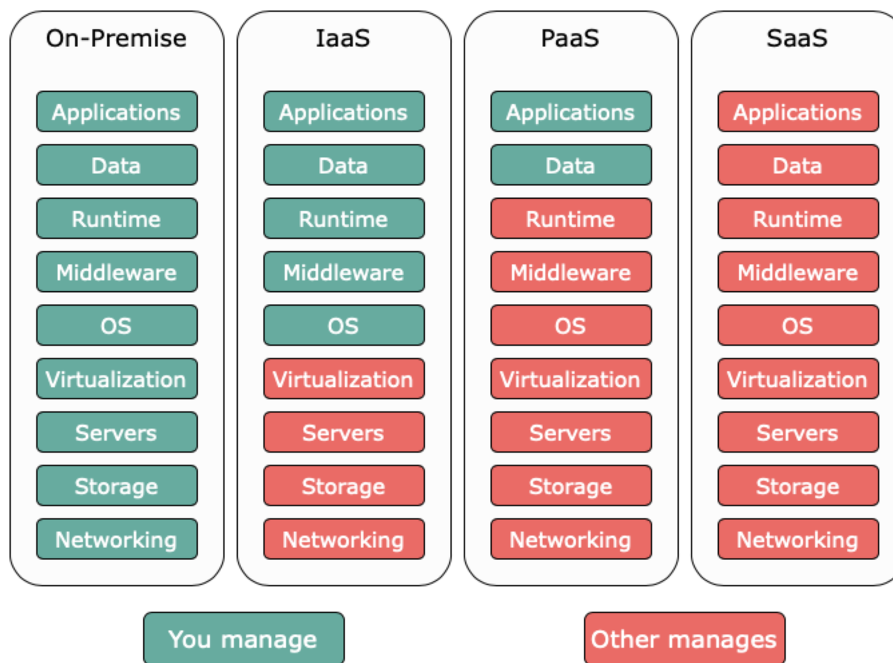

Figura 6 - Separazione delle responsabilità nelle tipologie di cloud computing

Le tre tipologie descritte sono ormai presenti e consolidate da diverso tempo. Molto più recenti sono invece due modelli che vanno a porsi in posizione intermedia tra i tre appena citati. Il primo tra questi è chiamato **CaaS** (**Container as a Service**) e nasce come servizio atto a semplificare il processo di costruzione e deployment di un'applicazione a container, scalabile e sicura, in data center on-premise o in cloud. Viene tipicamente posto a cavallo

tra IaaS e PaaS, in quanto, seppur fornendo allo stesso modo un'infrastruttura facilmente scalabile e ridimensionabile on-demand, differisce dal primo per l'approccio alla virtualizzazione, con un passaggio da macchine virtuali complete a micro-istanze che girano in container isolati. Sostanzialmente nel CaaS si incontrano la flessibilità del modello IaaS e il livello di astrazione del modello PaaS, con l'aggiunta di un livello di controllo delle risorse, e quindi anche di costi più elevati.
Il secondo nuovo modello si pone in un piano intermedio tra PaaS e SaaS, ed è spesso offerto come feature supplementare nelle principali implementazioni PaaS. Questo prende il nome di **FaaS** (**Function as a Service**), viene introdotto con il progetto *Lambda* di Amazon e va ad identificarsi come uno degli elementi cardine di quel moderno approccio che più comunemente è indicato come **serverless computing**. Con l'ingresso delle FaaS si tenta di prendere molti aspetti che già prima erano offerti dalle PaaS, facendo un passo avanti in termini di astrazione, scalabilità e riduzione dei costi. La Figura 7 va quindi ad ampliare lo spettro delle tipologie cloud presenti al momento, ponendo l'accento sul livello di astrazione che, con ognuna di esse, si riesce a raggiungere.

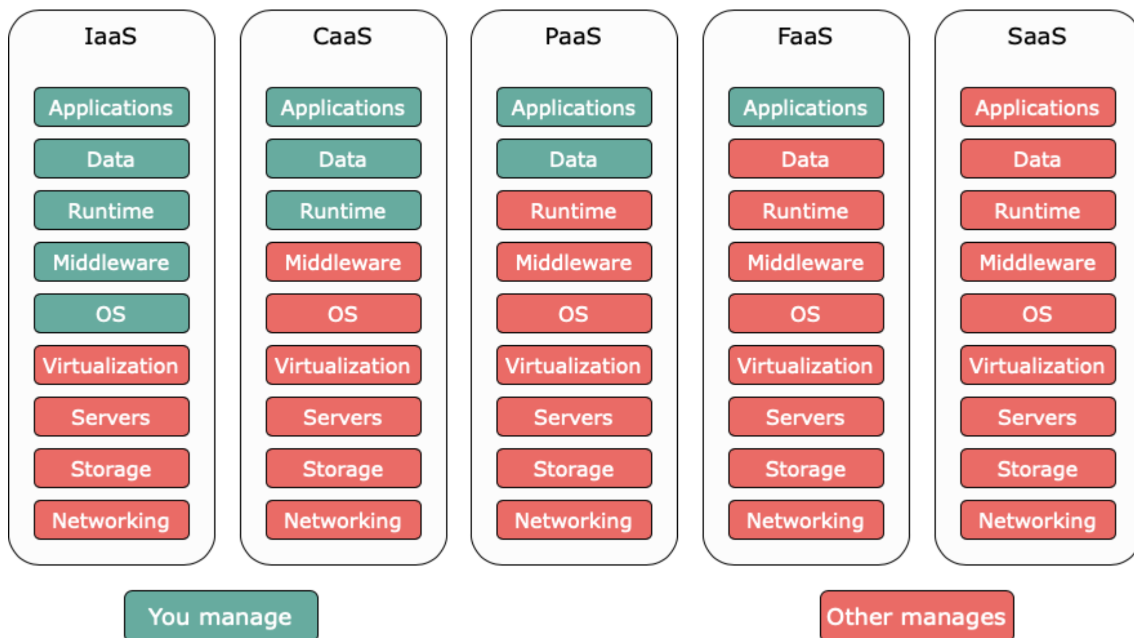

Figura 7 - Separazione delle responsabilità nei nuovi modelli di cloud computing

In realtà all'appello manca un ultimo modello, denominato **BaaS** (**Backend as a Service**), che verrà presentato nelle sezioni successive a confronto con il modello FaaS, essendo i due spesso accomunati.

2.3 Serverless e FaaS

Come è già accaduto per molte nuove "parole d'ordine" prima di questa, quali ad esempio "cloud computing", "big data", "DevOps" o "edge computing", anche l'ingresso del termine "serverless computing" ha generato dibattiti tra gli esperti del settore che hanno dovuto darne una definizione. Si riportano alcune delle definizioni ricorrenti, per tentare di estrapolarne i punti di accordo.

I primi a mettere per iscritto ciò che il serverless offre sono stati i membri del team di AWS [8]:

"Serverless computing allows you to build and run applications and services without thinking about servers. Serverless applications don't require you to provision, scale, and manage any servers. You can build them for nearly any type of application or backend service, and everything required to run and scale your application with high availability is handled for you."

["Il serverless consente di costruire e eseguire applicazioni e servizi senza preoccuparsi dei server. Le applicazioni serverless non richiedono allo sviluppatore di procurarsi, scalare e gestire alcun server. È possibile costruirle per quasi ogni tipologia di applicazione o servizio backend, mentre tutto ciò che serve per eseguire e scalare l'applicazione con alta disponibilità è gestito implicitamente."]

Gli autori del framework *Serverless* [9], un popolare strumento grazie al quale è possibile costruire applicazioni serverless capaci di eseguire su tutti i più famosi cloud provider, condividono il messaggio propinato da AWS:

"Just like wireless internet has wires somewhere, serverless architectures still have servers somewhere. What 'serverless' really means is that, as a developer you don't have to think about those servers. You just focus on code. You don't have to actively manage scaling for your applications. You don't have to provision servers, or pay for resources that go unused."

["Così come l'internet wireless ha dei cavi da qualche parte, le architetture serverless continuano ad avere dei server da qualche parte. Quello che serverless vuol dire davvero è che, come sviluppatore, non bisogna pensare a questi server, focalizzandosi solo sul codice. Non bisogna attivamente gestire lo scaling delle applicazioni, né procurarsi i server, né pagare per le risorse che non vengono utilizzate."]

Tomasz Janczuk, ex vicepresidente di *Auth0* [10], un'azienda che fornisce servizi di autenticazione e autorizzazione per applicazioni web e mobile, propone invece una definizione più concisa e di alto livello:

"The essence of the serverless trend is the absence of the server concept during software development."

["L'essenza del serverless è l'assenza del concetto di server durante lo sviluppo del

software."]

Paul Johnston, che ha militato ai piani alti nel team serverless di AWS Serverless, propone invece un'interessante visione, da un punto di vista economico piuttosto che tecnologico, che fa ben capire una delle motivazioni a cui il serverless deve la sua rapida crescita:

"A Serverless solution is one that costs you nothing to run if nobody is using it."
["Una soluzione serverless è quella che non ha costi se nessuno la utilizza."]

Prosegue sottolineando un errore che spesso viene fatto a livello concettuale:
"Often in people's minds, Serverless means that you're using Functions as a Service (FaaS) which is only one part of a possible Serverless solution."
["Spesso le persone credono che serverless significa che vengono utilizzate le Function as a Service (FaaS), che sono solo una parte di tutte le possibili soluzioni serverless."]

Infine, Peter Sbarski, vicepresidente in *A Cloud Guru* [11], importante piattaforma di servizi cloud enterprise, aggiunge che:
"Serverless is about abstracting users away from servers, infrastructure, and having to deal with low-level configuration or the core operating system. Instead, developers make use of single purpose services and elastic compute platforms (such as AWS Lambda) to execute code."
["Il serverless riguarda l'astrarre gli utenti dal server, dall'infrastruttura e dal doversi interfacciare con configurazioni di basso livello o con il sistema operativo. Gli sviluppatori invece utilizzano servizi single-purpose e piattaforme di computazione elastiche, come AWS Lambda, per eseguire il codice."]

Volendo estrapolare i concetti chiave dalle definizioni riportate, emergono idee ricorrenti riguardo a ciò che il serverless include e esclude. In particolare, il serverless include costi strettamente legati ai livelli di utilizzo e un grande focus sulla business logic di un'applicazione, mentre esclude l'approvvigionamento e la gestione dei server, così come il controllo attivo su scalabilità e disponibilità. In sostanza l'approccio serverless consente agli sviluppatori di dedicarsi, non tanto all'infrastruttura, ma al codice, e lo si chiama in questo modo perché agli sviluppatori non interessano le specifiche del server. Naturalmente, i server restano presenti, ma vengono gestiti in tutto e per tutto dai provider cloud, che allocano le risorse necessarie per conto dell'utente, il quale potrà così distribuire il codice direttamente nell'ambiente di produzione. In un modello IaaS standard, l'utente acquista in anticipo unità di capacità, pagando quindi componenti server sempre attivi necessari per l'esecuzione delle applicazioni. Il modello serverless funziona in modo differente: un evento attiva l'esecuzione del codice applicativo, quindi il provider cloud alloca le risorse per tale codice in modo dinamico e all'utente ne viene addebitato

l'utilizzo fino al termine dell'esecuzione del codice; dunque il pagamento è calcolato in base al solo tempo di esecuzione e non in base alla quantità di risorse utilizzate. Oltre agli ovvi vantaggi in termini di costo ed efficienza, il metodo serverless evita agli sviluppatori le attività di routine ripetitive associate alla scalabilità delle applicazioni e al provisioning del server [12].

Tra le definizioni riportate a inizio sezione, è certamente fondamentale la precisazione fatta da Paul Johnston, che va a porre una certa distanza tra serverless e FaaS, due concetti troppo spesso ed erroneamente intercambiati. Il serverless, da un punto di vista più ampio, è tutto ciò che riguarda logiche server-side, eventualmente con mantenimento di stato, realizzate attraverso più servizi separati, in esecuzione su un'infrastruttura invisibile agli occhi dello sviluppatore. Per quanto concerne il rapporto che intercorre tra serverless e FaaS, è più corretto dire che il primo include il secondo, come una delle due modalità principali in cui realizzare il serverless, insieme al paradigma **BaaS** (**Backend as a Service**).

Il BaaS consente agli sviluppatori di effettuare un outsourcing di gran parte degli aspetti backend di un'applicazione web o mobile, così che questi debbano soltanto scrivere e gestire la business logic. Nello specifico, i vendor di BaaS forniscono servizi solitamente raggiungibili da remoto tramite semplici API e SDK, che vanno a coprire meccanismi quali la gestione di database, i sistemi di registrazione e autenticazione degli utenti, la crittografia, gli aggiornamenti o le notifiche push. Essendo *"plug-and-play"*, l'integrazione di tali servizi nei sistemi di un'azienda è estremamente più semplice e, talvolta, più affidabile, rispetto a svilupparli internamente.
Nelle FaaS, l'approccio è simile per quanto riguarda la possibilità dello sviluppatore di concentrarsi sulla scrittura della logica di business, ma prevede un livello di controllo maggiore rispetto al BaaS, poiché tale logica è scritta interamente dallo sviluppatore, al quale però non è richiesta alcuna conoscenza né di configurazione in reti di calcolatori né di tecniche di deployment, lasciando la gestione dell'esecuzione interamente nelle mani del provider. Un utilizzatore finale di FaaS dovrà pertanto scrivere unicamente il codice della funzione e lanciarne l'esecuzione. Il framework sottostante andrà a generare automaticamente un ambiente effimero in cui eseguire questa funzione, distruggendolo poi quando la funzione termina. Il modello di programmazione FaaS è naturalmente ispirato alla programmazione funzionale, in cui si hanno funzioni trattate come entità di prima classe, ovvero considerate alla stregua degli altri tipi di valore, stateless e modulabili, favorendo meccanismi di innestamento e parallelizzazione.

La frequente sovrapposizione che si fa dei modelli FaaS e BaaS non è pertanto lecita, essendo presenti differenze significative. Innanzitutto, una prima distanza viene presa sul modo in cui le applicazioni vengono costruite: se nel mondo FaaS il paradigma impone di progettare un'applicazione come composizione di più funzioni interagenti ad eventi, dall'altro lato si trovano servizi server-side costruiti nel modo che il provider preferisce

e sviluppatori che non devono neanche conoscerne la struttura. La seconda importante differenza sta nel momento di attivazione di un servizio, nel senso che le FaaS eseguono in risposta a eventi, altrimenti sono "spente", mentre i servizi BaaS non sono event-driven e richiedono pertanto molte più risorse server, solitamente attive e in ascolto. In particolare, con le FaaS viene eliminata la necessità di mantenere un'infrastruttura always-on, tipica delle architetture a VM, in favore di environment effimeri, tipicamente realizzabili per mezzo della tecnologia a container, che vengono agilmente creati, distrutti e sostituiti a runtime, abbattendo i costi e i tempi relativi alle risorse utilizzate. Infine, i servizi BaaS non sono progettati per scalare automaticamente a fronte di carichi elevati, aspetto che è invece reputato cruciale in tutte le piattaforme FaaS [13].

Martin Fowler, figura tra i massimi esperti della programmazione, definisce sinteticamente le **Function as a Service** come servizi per fare deployment di codice senza affrontare le complessità derivate da architetture distribuite [14]. Dopo una decade di cloud computing, il serverless pone un'evoluzione del paradigma per estremizzare ulteriormente l'immagine di uno sviluppatore che non deve focalizzarsi sull'infrastruttura che mantiene la sua applicazione. Tecnicamente si tratta di funzioni stateless di ridotta dimensione e complessità, eseguite alla ricezione di eventi asincroni o richieste sincrone, misurate in base al tempo di esecuzione e non di allocazione. La possibilità di concentrarsi unicamente sulla business logic, tramite l'utilizzo dei linguaggi di programmazione più diffusi, è, in un certo senso, una rivoluzione, perché permette l'adozione di questa tecnologia a figure professionali quali scienziati, ingegneri, economisti o utenti non esperti di IT che diversamente non potrebbero sfruttare completamente le enormi possibilità offerte dal cloud. Inoltre, in un mondo in cui l'IoT e la connettività sono in crescita esponenziale, il serverless si pone come assoluto protagonista, ottimizzando e garantendo la distribuzione di tali servizi in modo efficiente e automatizzato. Ovviamente sarebbe possibile unire, in una stessa applicazione, l'utilizzo delle FaaS e dei BaaS, in modo da avere un ambiente interamente serverless, ma con il rischio di limitare eccessivamente la flessibilità e la personalizzazione dei sistemi. Inoltre, si potrebbe incorrere nel problema del *vendor lock-in*, ovvero un accoppiamento troppo stretto con un provider, che rende complessa e costosa l'eventuale decisione di spostarsi su un altro provider.

Il serverless, data la semplicità intrinseca delle operazioni per le quali è utilizzato, non vuole essere il sostituto di ogni piattaforma di cloud computing, ma una funzionalità aggiuntiva. Volendo brevemente ripercorrere le tappe evolutive che hanno condotto i servizi di cloud computing ad evolversi fino al serverless e alle FaaS, si ricorda che:

- Il cloud computing, in generale, ha dato la possibilità astrarre la rigidità degli ambienti on-premise.

- Le IaaS hanno dato una spinta verso l'idea di data center virtualizzati, astraendo l'infrastruttura e ottenendo ottimi livelli di scalabilità, andando ad allocare risorse virtuali.
- I CaaS hanno permesso l'utilizzo di container isolati sul cloud, cambiando le proprietà infrastrutturali di un'applicazione.
- Le PaaS hanno posto un livello aggiuntivo che astrae la gestione dei servizi direttamente al di sopra dell'infrastruttura virtualizzata.
- I SaaS hanno permesso l'utilizzo di software completi ed efficienti, senza nessuna installazione lato utente.
- Le FaaS, infine, sono arrivate ad astrarre l'ambiente di esecuzione (language runtime) delle applicazioni, permettendo agli utenti di organizzare un'applicazione secondo unità funzionali, mentre il serverless provider gestisce il runtime sottostante [15].

Scendendo maggiormente nel dettaglio, fra le possibili modalità di invocazione diretta di un processo serverless, uno degli approcci più comuni è l'utilizzo di chiamate REST verso il cosiddetto **API Gateway**, elemento tipicamente presente nelle piattaforme FaaS, formato da un server HTTP, in cui ogni endpoint corrisponde all'attivazione di una funzione. Solitamente si effettua un mapping dalla richiesta HTTP a un formato semplificato, ad esempio un oggetto JSON, che possa essere più facilmente utilizzato come parametro di ingresso della funzione. Una volta presa in carico la richiesta, la funzione provvederà ad eseguire la sua logica stateless e a restituire il risultato allo stesso API Gateway, che ritrasformerà questo output in una risposta HTTP da inoltrare al chiamante. Oltre alla semplice funzione di smistatore, il gateway può, all'occorrenza, effettuare operazioni di autenticazione dell'utente e validazione dell'input.
Nella Figura 8 è indicato il workflow completo di un'invocazione serverless, dalla richiesta fino alla restituzione della risposta.

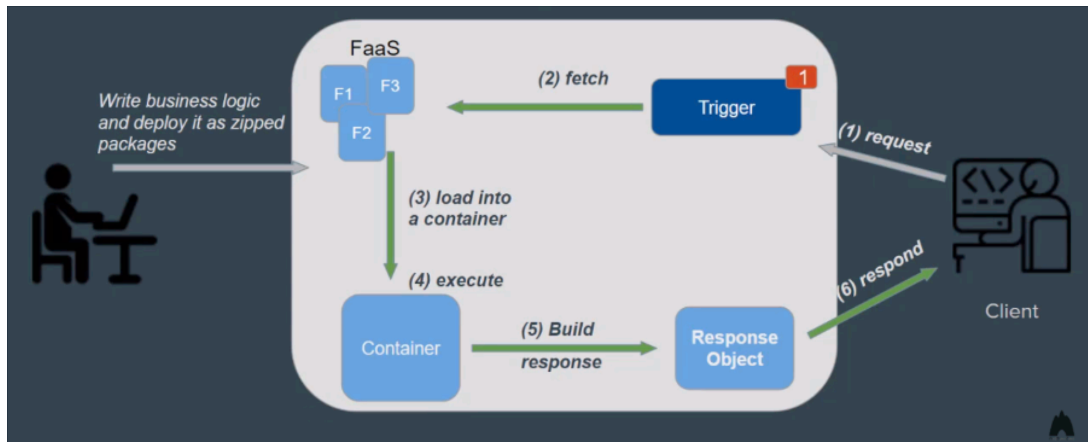

Figura 8 - Serverless workflow

Solitamente, quando si transita da un'applicazione tradizionale a un'applicazione serverless, si scompone la business logic uniforme del componente server in più funzioni dislocate, similmente a come si farebbe per un'applicazione a microservizi. Se in partenza si possiede un server HTTP che risponde a più endpoint, i gestori delle richieste dirette a tali endpoint diventeranno delle FaaS, che prenderanno in carico la richiesta e, una volta servita la risposta, termineranno. Questo meccanismo donerà al sistema una certa flessibilità e apertura alle modifiche, consentendo, nella maggior parte dei casi, l'aggiornamento indipendente dei componenti. Ovviamente il trade-off di questo design sta nella necessità di possedere un sistema con buone capacità di monitoring distribuito e affidabile nelle interazioni, ma anche nel porre una certa attenzione sul fronte della sicurezza [16].

La tematica della sicurezza, infatti, rientra tra le prime preoccupazioni degli sviluppatori che si affacciano al mondo FaaS, poiché diventa necessario reinventare il modo in cui si mettono in sicurezza le applicazioni. La domanda è infatti come si possano aggiungere controlli di sicurezza quando non si ha alcun controllo sul sistema che esegue il codice su un networking astratto. Le misure di sicurezza tradizionali sono improntate sul monitoraggio dei server e delle comunicazioni di rete tra questi, entrambi elementi al di fuori dalla visuale del progettista in questione. Inoltre, il design su cui il serverless si fonda, che insiste molto sul semplificare l'aggiunta di nuovi sorgenti di input, come ad esempio l'inserimento di un trigger in un processo CI/CD, diventa una potenziale minaccia, poiché si facilita l'ingresso di eventuali attaccanti. Fortunatamente, se da un lato il paradigma serverless appare poco gestibile dal punto di vista della sicurezza, bisogna considerare alcune caratteristiche che lo rendono, per certi aspetti, automaticamente più sicuro:

- Il codice è eseguito su immagini di container ben consolidate e standardizzate, dunque le vulnerabilità del sistema operativo e del kernel non sono significativamente rilevanti.
- Il tempo relativamente limitato di esecuzione e l'essere stateless rendono più complesso per un attaccante l'agganciarsi al sistema e l'inserirsi radicalmente in esso.
- Tramite l'API Gateway è sempre possibile definire degli upper bound sulla quantità di richieste che lo attraversano, restringendo, di fatto, lo spazio di manovra disponibile a un attaccante [15].

Nel momento in cui il nuovo modello ha fatto il suo ingresso nel panorama del cloud computing, molti sono stati i competitor entrati in gioco. Non è semplice, in generale, trovare la piattaforma serverless più adatta, in quanto dipende dai bisogni dell'utilizzatore e, spesso, un confronto diretto è difficile da realizzare, poiché ogni piattaforma supporta il concetto di FaaS in maniera diversa. Ad ogni modo, se si desidera avere un orientamento verso una scelta, i principali e più comuni fattori da tenere in considerazione, oltre alle misure di sicurezza evidenziate, sono:

- **Cold start** - i tempi di risposta di una funzione per la quale non esiste ancora nessuna istanza pronta a rispondere, risultano essere dilatati; esso è una delle prime limitazioni delle piattaforme serverless e può variare da pochi millisecondi a diversi secondi.
- **Concorrenza** - quando si sceglie un provider bisogna considerare qual è il numero di richieste e il numero totale di invocazioni processate al secondo; è buona norma controllare quali sono i gradi di concorrenza, i limiti imposti dal provider, e se questi possono essere aumentati su richiesta.
- **Capacità di autoscaling** - non tutte le piattaforme scalano allo stesso modo, in quanto sfruttano meccanismi diversi di scaling automatico; utilizzare servizi di autoscaling poco ottimizzati può rappresentare un importante collo di bottiglia nel momento in cui il cluster riceve un numero significativo di richieste.
- **Runtime** - non tutti i linguaggi sono supportati da tutti i provider, un fattore che potrebbe incidere sulla selezione della piattaforma da utilizzare.
- **Strumenti di debugging** - prima di entrare in produzione è necessaria una lunga fase di sviluppo e testing, per la quale alcuni provider offrono ambienti simulati in locale.
- **Monitoraggio, gestione e risoluzione di problemi** - certi provider forniscono strumenti di monitoring in cloud, mentre altri suggeriscono di utilizzare servizi di monitoraggio di terze parti.
- **Costo** - nelle prime fasi di analisi è necessaria un'attenta considerazione sulle proprie capacità di sostenere i costi della piattaforma, scegliendo accuratamente

tra i piani tariffari messi a disposizione; questo riguarda tipicamente le sole piattaforme proprietarie.

Il primo tra questi fattori, il **cold start**, merita un approfondimento maggiore poiché costituisce un problema, o più precisamente un limite, intrinseco nel design delle FaaS, non presente nelle tradizionali architetture *always-on*. Quando un utente invoca una funzione, questa viene tipicamente messa in esecuzione in un container, che, al termine delle operazioni, verrà mantenuto attivo per un certo periodo di tempo (stato di *warm*). Così facendo, se dovesse arrivare una seconda richiesta per la stessa funzione prima che questo venga distrutto, sarà possibile servirla istantaneamente. Ovviamente non è possibile mantenere il container in stato di *warm* per un tempo eccessivamente lungo, altrimenti si andrebbe contro i principi del modello serverless, e, pertanto, dopo qualche secondo di inattività, bisogna chiuderlo. È proprio da ciò che nasce la problematica dei *cold start*: prendere in carico una richiesta quando non ci sono container in stato di *warm* già pronti a servirla implica l'aggiunta di un certo delay necessario a istanziare un nuovo container. La durata del *cold start*, che può estendersi fino a diversi secondi, varia in base al provider e al linguaggio utilizzato e può influire drammaticamente su certi tipi di applicazione che richiedono latenze minime. Nella situazione attuale, nonostante le FaaS stiano maturando molto velocemente, queste non sono adatte a tutti i workload esistenti, e, se il *cold start* si rivelasse essere un problema troppo limitante in un determinato processo, allora converrebbe cambiare modello. È necessario che gli utenti comprendano che esiste un trade-off intrinseco nel serverless computing tra basso costo e *cold start*, che non consente di superare agilmente il problema. Se il serverless è così vantaggioso da un punto di vista economico rispetto al cloud computing tradizionale, è solo grazie alla possibilità che hanno i provider di terminare le risorse inutilizzate, il che ha come ovvia conseguenza l'introduzione dei *cold start*. Data l'elevata preoccupazione nei confronti di questo tema, in molte piattaforme, specialmente in quelle open source, è consentito, tramite opportune configurazioni, mantenere sempre almeno una replica attiva.

Oltre a quelli citati in precedenza, per le piattaforme open source bisogna valutare alcuni fattori addizionali, tra i quali:

- **Semplicità di installazione** - alcune piattaforme risultano essere più complesse di altre in fase di installazione, solitamente a causa del fatto che queste non sono in grado di funzionare stand-alone, ma necessitano di strumenti ausiliari per abilitare servizi fondamentali quali monitoring, autoscaling o traffic management.
- **Semplicità di utilizzo** - per alcune piattaforme è sufficiente scrivere il codice della funzione e pubblicarlo, mentre altre hanno bisogno di effettuare operazioni intermedie per impacchettare le funzioni nel formato supportato dalla specifica piattaforma, spesso vincolandosi all'ausilio di specifici tool esterni.

- **Utilizzo di risorse** - nel momento in cui si installa una piattaforma open source su un cluster composto da nodi di proprietà dell'utente, diventa una prerogativa anche la capacità di esporre un servizio ad elevata disponibilità e scalabilità senza sovraccaricare o saturare le risorse a disposizione; le differenze architetturali delle varie soluzioni possono influire in modo consistente su questo aspetto.

Quanto detto, sarà considerato più avanti, una volta presentate le principali piattaforme FaaS open source, per identificare, congiuntamente a una serie di valutazioni quantitative, il framework più adatto a seconda degli scenari di utilizzo.

3 Piattaforme di orchestrazione in ottica serverless

In questa sezione vengono esposte le basi teoriche e tecnologiche alla base dei framework per il serverless, così da dare un solido background all'implementazione del caso d'uso preso in esame.
Prima di addentrarsi nella descrizione delle piattaforme FaaS di riferimento, è infatti necessario fare un passo indietro per spiegare le tecnologie che si trovano nei livelli sottostanti e che ne consentono la realizzazione. Come si è detto, le FaaS vengono messe in esecuzione all'interno di ambienti effimeri generati on-demand e vengono tipicamente gestite da orchestratori di risorse, intesi come strumenti atti a ordinare e coordinare le attività da svolgere in un sistema, assegnandole a momenti specifici e per obiettivi specifici. La maggior parte delle soluzioni FaaS, specialmente quelle open source, si appoggia a questo scopo sull'astrazione fornita dai container, che risultano particolarmente adeguati in quanto ambienti leggeri, riproducibili e versionabili, facilitando dunque il processo di creazione e avvio. Per dare un'idea più concreta del loro funzionamento, in questa sezione si è scelto di fare riferimento alle reali piattaforme di containerizzazione e di orchestrazione di container utilizzate come standard de facto, sebbene i concetti base esplicitati valgano a prescindere dall'effettiva implementazione.

Nel panorama attuale, il leader tra le piattaforme di containerizzazione è **Docker** [17], il quale però, in caso di presenza di un elevato numero di container, come accade solitamente negli scenari industriali, necessita del supporto di uno strumento addizionale di orchestrazione che possa gestire l'ingente quantità di richieste e di interazioni. La scelta tra le sue varie implementazioni impone alcune considerazioni supplementari, poiché questa dipende dalla dimensione e dai requisiti dello specifico ambiente da realizzare. In particolare, le due soluzioni di orchestrazione più diffuse attualmente sono **Docker Swarm** [18], proprietario della stessa piattaforma Docker, e **Kubernetes** [19], sviluppato da Google.
A fine sezione verrà inoltre riportato un caso a sé stante di tecnologia alternativa ai container, denominata **microVM**, recentemente introdotta da Amazon nei propri workload serverless.

3.1 Container e Docker

Prima dell'introduzione dei **container**, era già comune la pratica di virtualizzare le risorse necessarie a un dato software. Tuttavia, la crescente complessità dei sistemi odierni ha reso necessario lo sviluppo di nuove tecniche di virtualizzazione, in grado di gestire correttamente gli ingenti carichi di lavoro presenti in produzione. Le consolidate tecnologie di virtualizzazione (*virtual machine* o VM) offrono certamente un buon grado di isolamento, ma, d'altra parte, conducono a un elevato consumo di risorse di calcolo,

poiché ogni VM obbliga l'esecuzione dell'intero sistema operativo e di tutte le librerie necessarie al suo ambiente. Ciò impone un limite al numero di VM che uno stesso host può ospitare, se si vogliono preservare le prestazioni generali del sistema. Inoltre, le VM risultano essere poco portabili e le operazioni di migrazione richiedono spesso interventi manuali non esenti da rischi e errori.

Docker [17] è un progetto open source, affermatosi come standard da diversi anni, nato con lo scopo di automatizzare la distribuzione di applicazioni sotto forma di container leggeri, portabili e autosufficienti, eseguibili sia su cloud che in locale, a sostituzione delle VM [20]. Con "container" va inteso l'insieme dei dati necessari all'esecuzione di un'applicazione, quali librerie, altri file eseguibili, file system, file di configurazione, script, etc. Il processo di distribuzione di un'applicazione si traduce così nella creazione di un'**immagine Docker**, ovvero di un file (*Dockerfile*), contenente tutti i dettagli sopra citati. L'immagine viene utilizzata da Docker per creare un container, ossia un'istanza dell'immagine che eseguirà l'applicazione in essa descritta, in una sorta di sandbox. I container, quindi, risultano innanzitutto autosufficienti, contenendo, già in partenza, tutte le dipendenze dell'applicazione, senza alcuna necessità di configurazione addizionale, e sono inoltre completamente portabili, poiché vengono distribuiti nel formato delle immagini, ormai divenuto standard, leggibile ed eseguibile da qualunque server Docker. Infine, il problema dello spreco di risorse relativo alle tradizionali VM viene tamponato dalla leggerezza intrinseca dei container, in quanto essi sfruttano i servizi offerti dal kernel del sistema operativo ospitante, invece di richiedere l'esecuzione di un kernel virtualizzato, come avviene per le macchine virtuali. Ciò permette di ospitare un considerevole numero di container sulla stessa macchina fisica.

Come anticipato, una piattaforma come Docker, in scenari con elevato numero di interazioni, risulta essere complessa da gestire senza il supporto di un **orchestratore**. Il compito principale di un orchestratore è quello di istanziare i container, provvedendo a crearne di nuovi al fine di rispettare determinati requisiti di scalabilità e allocando tutte le risorse necessarie. Oltre a ciò, esso si occupa di garantire una certa tolleranza ai guasti, monitorando l'esecuzione e la salute dei container, per poter sostituire eventuali entità non funzionanti. In aggiunta, l'orchestratore effettua un importante lavoro di networking per consentire il load balancing delle richieste in ingresso, la discovery dei container e la loro interconnessione [21].

Di seguito vengono analizzati i due competitor principali sul piano degli strumenti di orchestrazione per container, *Docker Swarm* e *Kubernetes*, con un'attenzione particolare al secondo, più complesso del primo, ma sicuramente più completo e adatto a importanti ambienti industriali.

3.2 Docker Swarm

Docker Swarm [18] è l'orchestratore di container proprietario di Docker, e, in quanto tale, si occupa di gestire in modo centralizzato i container appartenenti a un cluster di nodi. Docker Swarm solleva gli utenti da compiti più "meccanici" come controllare quale server ospiterà il container e come quest'ultimo verrà avviato, monitorato o terminato. Esso è inoltre in grado di interagire con più container nello stesso momento e consente di scalare automaticamente in base al carico da gestire, occupandosi al contempo di fornire i servizi di rete, storage (volumi), sicurezza e telemetria necessari ai container. Inoltre, in caso di fallimento dei container, Docker Swarm si attiva per ripristinare una situazione di stabilità, come indicato in Figura 9. Infine, l'orchestratore ha il compito di effettuare gli aggiornamenti software dei container (*rolling updates*).

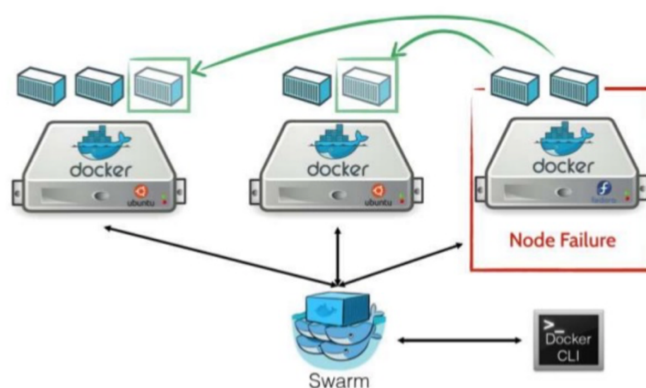

Figura 9 - Orchestrazione con Docker Swarm

Docker Swarm si basa su un'architettura *master-slave*, in cui lo **swarm manager** è responsabile per l'amministrazione del cluster e la delegazione di compiti, mentre gli **swarm worker** eseguono l'effettivo lavoro. L'inizializzazione di un cluster con Swarm è particolarmente semplice e realizzabile attraverso un solo comando, sia sul nodo master che sui vari worker:

```
# docker swarm init    //sul nodo master
# docker swarm join    //sui nodi worker
```

Analogamente, nel caso in cui un nodo voglia abbandonare il cluster, basterà eseguire il comando inverso alla *join*:

```
# docker swarm leave  //sui nodi worker
```

3.3 Kubernetes

Kubernetes [19] è un altro orchestratore per container, il più utilizzato a livello industriale. È un progetto open source di Google, consigliato per la gestione di cluster di medie e grandi dimensioni e di applicazioni complesse. Anche Kubernetes ha una struttura con interazione di tipo *master-slave*, ma aggiunge un livello di indirezione raggruppando i container all'interno dei cosiddetti **pod**. I container, all'interno di un pod, condividono storage e network, dunque si può dire che i pod modellino host logici contenenti uno o più container, in qualche modo correlati tra loro (*tightly coupled*).

Per inizializzare e utilizzare Kubernetes, sono tre i componenti necessari:

- **Kubeadm** - il comando per dare inizio, modificare ed eliminare il cluster.

```
# kubeadm init     //sul nodo master
# kubeadm join     //sui nodi worker
# kubeadm reset    //per eliminare il cluster
```

- **Kubelet** - il demone di Kubernetes presente in tutte le macchine del cluster.
- **Kubectl** - lo strumento per la comunicazione tra utente e cluster.

La struttura base di un nodo appartenente a un cluster Kubernetes è schematizzata nella Figura 10 sottostante.

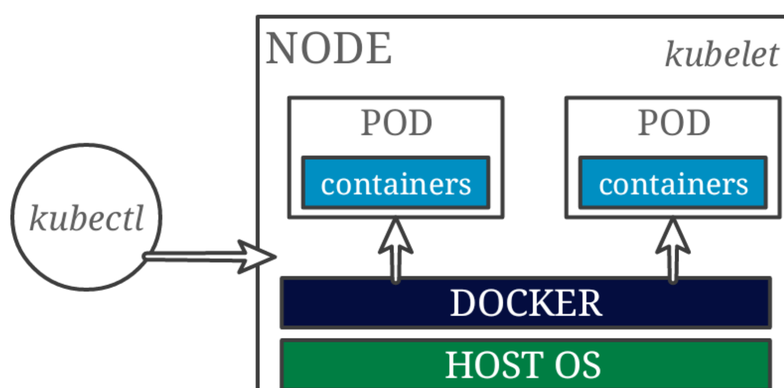

Figura 10 - Struttura di un nodo in Kubernetes

Come anticipato, è necessario in questo caso aggiungere alcuni dettagli relativi ai componenti e al livello di networking in Kubernetes, in quanto la completezza che si ha rispetto al precedente Docker Swarm si paga con una struttura altrettanto completa, ma allo stesso tempo complessa. Per poter utilizzare questo potente strumento è stato

indispensabile comprendere al meglio gli elementi in gioco e le modalità con le quali questi interagiscono, di seguito esposti.

3.3.1 Componenti principali in Kubernetes

Tutti i componenti presenti in Kubernetes possono essere istanziati utilizzando file di configurazione **YAML** [22], dai quali l'orchestratore legge la tipologia del componente, dove reperire l'immagine del container relativa a quel componente, come montare i volumi persistenti, dove memorizzare i log generati dal componente [21]. Viene qui riportato un esempio di file in formato *.yaml* che descrive un pod contenente un'istanza del web server *Nginx* [23], esposto sulla porta 80 e denominato *nginx-web-server*.

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

Il componente chiave di partenza è l'**immagine**, essendo Kubernetes un orchestratore di container: attualmente sono supportate soltanto le immagini di Docker, ognuna in esecuzione in un container all'interno del proprio pod.
Una volta scaricata l'immagine da una repository Docker, è compito dei **job** monitorare la creazione di uno o più pod associati ad essa. Quando il numero di pod specificato nella configurazione è stato avviato con successo, allora il job può considerarsi completato senza errori.
Un **service** è un insieme logico di pod che eseguono un servizio, eventualmente accessibile anche dall'esterno [24], secondo diverse soluzioni che verranno presentate più avanti.
Il ciclo di vita dei pod viene invece gestito dai **deployment**, che rappresentano un insieme di più pod identici. Il loro scopo primario è garantire che l'esatto numero di repliche desiderato per un dato servizio sia rispettato in qualsiasi istante di tempo, creando nuovi pod e rimpiazzando eventuali pod falliti. Oltre a ciò, i deployment vengono utilizzati per

aggiornare i pod in modo graduale o *staged roll out*. In particolare, quando viene aggiornata un'immagine, quella precedente non viene sostituita istantaneamente sui pod in esecuzione, poiché la nuova immagine potrebbe avere dei bug e condurre a errori del sistema. L'update viene quindi fatto inizialmente soltanto su un nuovo pod, e si procede se e solo se la nuova immagine non presenta problemi e supera la fase di creazione. Analogamente è possibile effettuare un rollback a una versione precedente [25].
Per quanto riguarda la persistenza, in Kubernetes si conserva il concetto di **volume** già presente in Docker, assimilabile a una cartella accessibile dai container all'interno di un pod.
Tutte le risorse citate vengono accompagnate da un'indicazione del **namespace** di appartenenza, tramite il quale si instaura un muro che separa logicamente servizi appartenenti ad applicazioni diverse: una funzionalità estremamente utile quando più team utilizzano lo stesso cluster e si vogliono evitare conflitti, generando difatti più cluster virtuali sullo stesso cluster fisico.

3.3.2 Networking in Kubernetes

I container all'interno di uno stesso pod comunicano utilizzando l'interfaccia di rete locale (*localhost*). Inoltre, ad ogni pod viene assegnato un indirizzo IP così che possa comunicare con gli altri pod, anche se questi si trovano su nodi diversi. La regola di base è che i pod devono essere in grado di riconoscersi a vicenda come se l'host sottostante non esistesse, e a sua volta l'host deve essere in grado di comunicare con i pod sopra di esso. Nell'architettura di Kubernetes è anche presente un servizio di risoluzione di nomi, pertanto nelle comunicazioni è possibile utilizzare sia direttamente gli indirizzi IP univoci, sia, preferibilmente, i nomi logici tradotti dal DNS interno.

Kubernetes non fornisce direttamente un'implementazione di rete, ma ne definisce solo il modello, nella forma di **overlay network**, ovvero di una connessione logica a livello applicativo tra entità che risiedono su nodi diversi. La più diffusa implementazione, selezionata nel presente lavoro, è **Flannel** [26], soluzione divenuta particolarmente popolare per la sua semplicità d'utilizzo. Anche in questo caso, non è tanto importante l'implementazione considerata, quanto il concetto e le modalità con cui un'overlay network interconnette il cluster Kubernetes, ma si è scelto ugualmente di descrivere un esempio reale per dare un'idea più concreta del funzionamento. Flannel instaura la propria overlay network sopra le reti dei singoli host, affinché i pod possano comunicare direttamente grazie a un meccanismo di *tunneling* dei pacchetti effettuato dal processo demone di *flanneld*, come si può vedere nella Figura 11.

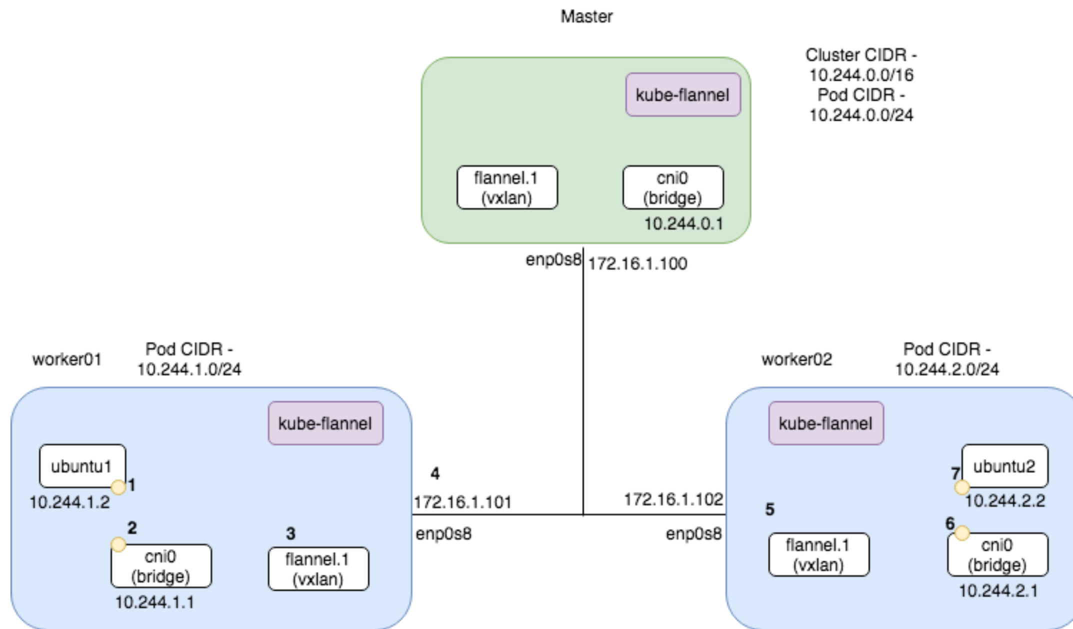

Figura 11 - Overlay network Flannel in Kubernetes

Kubernetes fornisce diverse strategie per la gestione del traffico in ingresso, in base alla complessità e sicurezza richieste, garantendo flessibilità e completezza nella connessione tra i diversi servizi offerti nel cluster. Di default un servizio in esecuzione è di tipo **ClusterIP**, ovvero accessibile da tutte e sole le applicazioni interne al cluster.

Il meccanismo più primitivo per raggiungere un servizio dall'esterno è invece denominato **NodePort** (Figura 12), e consiste nell'aprire una specifica porta su tutti i nodi del cluster, così che ogni richiesta verso l'IP di uno dei nodi, alla porta esposta, venga ridirezionata al servizio in questione.

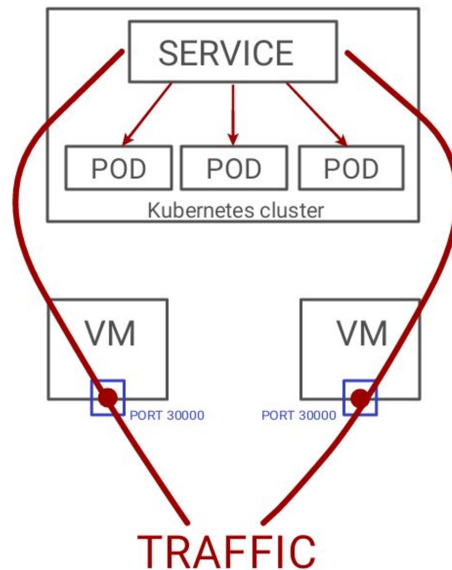

Figura 12 - Flusso del traffico in presenza di NodePort

Adottare la politica NodePort in produzione non è consigliato, in quanto con questa strategia si può mappare un solo servizio per porta, limitando il numero di porte disponibili. Il metodo standard utilizzato per esporre un servizio è piuttosto il **LoadBalancer** (Figura 13), il quale assegna un IP pubblico a un dato servizio, indipendentemente dal nodo in cui si trova, con l'unico svantaggio di diventare oneroso se sono presenti molti servizi, poiché assegna un nuovo IP ad ogni nuovo servizio.

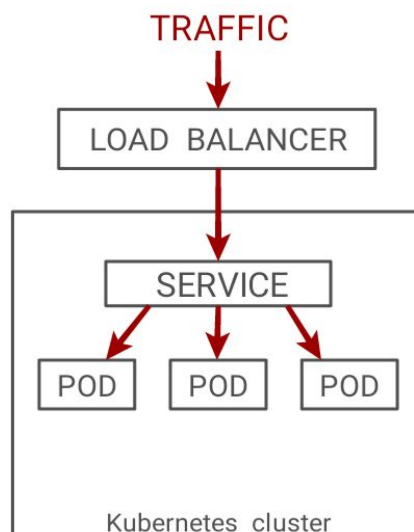

Figura 13 - Flusso del traffico in presenza di Load Balancer

Esiste una quarta e ultima modalità di accesso ai servizi Kubernetes dall'esterno, ovvero l'**Ingress** (Figura 14). Seppure più complicata come configurazione, essa permette di esporre molteplici servizi sotto il medesimo indirizzo IP. A differenza degli altri non si tratta di una tipologia di servizio, ma di un componente aggiuntivo che funge da router [27].

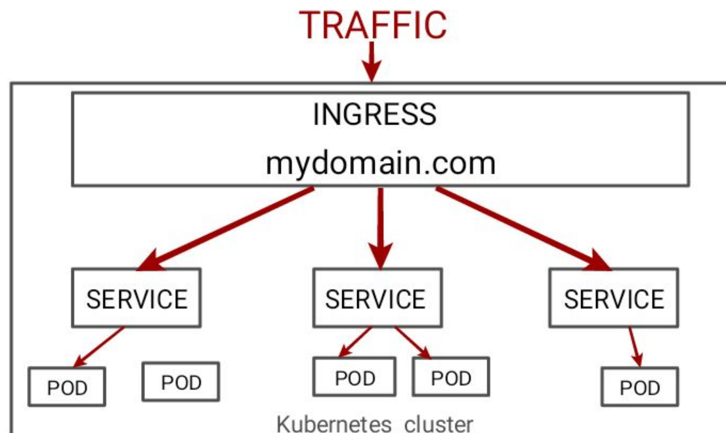

Figura 14 - Flusso del traffico in presenza di Ingress

3.3.3 Kubernetes e hybrid cloud

A partire dal 2015, l'utilizzo di container orchestrati in Kubernetes ha permesso di far evolvere e al contempo di consolidare gli scenari **hybrid cloud**.
La tecnologia dei container è diventata in breve tempo il minimo comune denominatore per eseguire workload in modo trasversale su macchine fisiche, cloud privato e cloud pubblico, come alternativa flessibile e semplificata ai tradizionali *hypervisor*: ad esempio, un'applicazione sviluppata e containerizzata su *MacOS* può essere facilmente fatta girare su un'istanza *Amazon EC2* [28], sulle *Azure VM* [29] o sul *Google Compute Engine* [30], senza nessuna modifica al codice o alla configurazione.

Se le piattaforme a container, prima tra tutte Docker, sono considerate i nuovi hypervisor, allora Kubernetes diventa di conseguenza il sostituto dei *virtual machine manager*, riuscendo a fare accordare l'industria IT su un livello di infrastruttura standard. *Red Hat* [31], *VMware* [32], *Canonical* [33], *Mirantis* [34], *Rancher* [35] e molti altri provider offrono oggigiorno piattaforme basate su Kubernetes che possono eseguire sia su data center on-premise, che su cloud privati e pubblici.
Grazie a questa forte spinta di standardizzazione da parte dei maggiori competitor sul mercato, l'affermazione dei cloud ibridi diventa sempre più vicina. Il 2019 ha infatti visto il proliferare di soluzioni hybrid cloud basate su Kubernetes che consentono di gestire

cluster distribuiti su diversi ambienti, on-premise e in cloud, fino a interconnettere addirittura cloud di provider diversi, il cosiddetto **multi-cloud**, superando i problemi legati al *vendor lock-in*.

Per completezza vengono riportati i nomi di alcune tra le piattaforme più importanti introdotte nel 2019: *IBM Cloud Parks* [36], *Google Anthos* [37], *VMWare Tanzu* [38], *Azure Arc* [39]. Tutte le piattaforme citate hanno Kubernetes come nucleo centrale nella strategia di ibridizzazione che va ad abilitare la portabilità e la possibilità di scalare i workload su ambienti eterogenei [40].

3.4 Confronto tra Docker Swarm e Kubernetes

Pur essendo stata prestata più attenzione al secondo orchestratore, viene di seguito riportata una breve comparativa per meglio evidenziare le differenze principali tra i due orchestratori presentati.

Partendo dalla fase iniziale di installazione, la prima differenza da sottolineare è che il setup di un cluster con Docker Swarm richiede due soli comandi, mentre quello di Kubernetes risulta essere molto più complesso, in quanto va compreso e configurato il livello di networking sottostante. Questa sostanziale differenza è giustificata dal fatto che Docker Swarm è un orchestratore più semplice e snello, adatto alla presenza di poche decine di container, mentre la complessità di Kubernetes viene ripagata quando ci si sposta in un ambiente di produzione con centinaia o migliaia di container. Sul piano della gestione delle richieste, gli scenari ridimensionati a cui è destinato Docker Swarm gli consentono, ad esempio, di offrire possibilità di load balancing automatico, mentre in Kubernetes è richiesta una configurazione manuale, sebbene più minuziosa e completa.

La persistenza è affidata in entrambi casi al concetto di volumi di storage, i quali in Docker Swarm possono essere condivisi da tutti i container su un nodo, mentre in Kubernetes soltanto all'interno dello stesso pod, garantendo un migliore isolamento e controllo. Inoltre, in Docker Swarm, i volumi sono molto simili a delle cartelle su un disco locale, senza alcuna gestione del ciclo di vita, mentre in Kubernetes sono entità più avanzate, con un ciclo di vita esplicitamente legato a quello dei rispettivi pod.

Il *version control* e il *roll-out* degli aggiornamenti sono semplificati, in entrambi i casi, grazie alla possibilità di effettuare *rolling update* progressivi. Il discorso cambia per quanto riguarda il procedimento inverso, con Docker Swarm che non consente di effettuare *rollback* automatico in caso di fallimento, al contrario di Kubernetes che invece offre sofisticati meccanismi di fault tolerance e ripristino.

Tenendo conto dell'esperienza utente e della fruibilità in fase di testing, va detto che Docker Swarm non fornisce alcuna GUI, mentre Kubernetes mette a disposizione una

completa dashboard pronta all'uso, accessibile da browser e facilmente installabile, dove è possibile visualizzare e modificare tutti gli elementi facenti parte del cluster, ma anche monitorare l'impatto del cluster a livello di CPU e RAM, con metriche divise per nodo, pod o namespace. Restando nell'ambito del monitoraggio, va anche detto che nessuno dei due orchestratori fornisce strumenti di logging nativi cluster-level, ovvero tecnologie di logging resilienti alla caduta di container, pod o nodi. Ciononostante, entrambi sono facilmente integrabili con i sistemi di monitoraggio più popolari.

Infine, Kubernetes supporta la quasi totalità di piattaforme serverless, mentre, tra quelle selezionate in questo lavoro, soltanto una, *OpenFaas*, è funzionante anche al di sopra di Docker Swarm. Ciò rende l'orchestratore di Google una scelta praticamente obbligata per chi necessita di esplorare le diverse soluzioni serverless presenti sia sul mercato che nel mondo open source.

Considerando i vantaggi e svantaggi di ognuno, insieme al supporto che i due ricevono dalla community e dalle aziende del settore, risulta evidente che Docker Swarm è adatto per sviluppo e testing in ridotti ambienti a container, permettendo di astrarre le complesse configurazioni di networking richieste da Kubernetes. Al contrario, in tutti gli scenari industriali di produzione, in particolar modo in ambito serverless, è necessario affidarsi alla completezza offerta da Kubernetes, e, per questo motivo, nel seguito della trattazione ci si riferirà sempre a cluster orchestrati da Kubernetes.

3.5 MicroVM e Firecracker

Per completezza, si aggiunge all'elenco delle tecnologie alla base del serverless il modello alternativo e controcorrente utilizzato nella piattaforma **Firecracker** [41], sviluppata da *Amazon AWS*.
La tecnologia in questione si pone come soluzione intermedia nella disputa tra container e virtual machine, e prende il nome di *microVM* (o *micro virtual machine*). I container, in generale, offrono come feature di punta tempi di avvio ridotti e la possibilità di creare densi raggruppamenti di istanze concorrenti sullo stesso nodo, grazie ai requisiti sulle risorse poco restrittivi, mentre le virtual machine si presentano con ottimi livelli di sicurezza *hardware-based* e di isolamento dei workload. La promessa delle microVM è quella di unire i vantaggi di entrambe le soluzioni menzionate in macchine virtuali leggere che possano garantire al contempo sicurezza, isolamento, velocità di avvio e ottimizzazione delle risorse.

In questo scenario, Firecracker si pone come VMM (virtual machine monitor) che utilizza la **KVM**[2] per creare e gestire microVM (Figura 15). A tale scopo, Firecracker presenta un design minimalista, escludendo tutti i dispositivi e le funzionalità non necessarie, riducendo l'impatto sulla memoria, i tempi di startup e la superficie esposta ad attacchi dall'esterno. Il tutto consente di risparmiare sulle risorse necessarie a mantenere l'infrastruttura, con conseguente possibilità di istanziare un numero elevato di microVM sullo stesso nodo, proprio come si farebbe per dei container.

Sul fronte della sicurezza e dell'isolamento, grazie alla virtualizzazione basata su KVM, è garantita la possibilità di eseguire sullo stesso nodo workload appartenenti a utenti diversi, senza possibilità di interferenza reciproca.

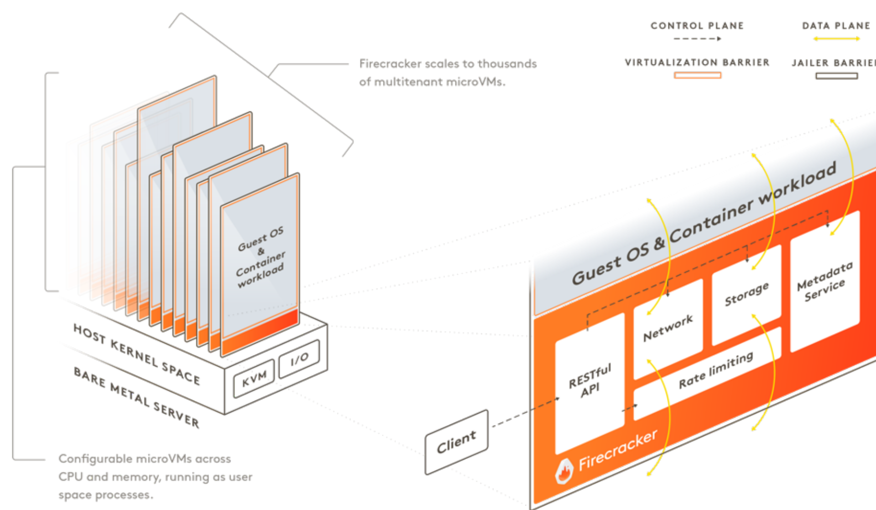

Figura 15 - Struttura di Firecracker

Da questo punto di vista Firecracker sembra ideale per il supporto all'esecuzione di workload serverless, tanto che è stato recentemente integrato nei processi di *AWS Lambda*. In merito a ciò, il team di *AWS*, dopo l'annuncio della piattaforma, ha eseguito una dimostrazione dal vivo in cui sono state istanziate 4000 microVM contemporaneamente, con una media di tempo di avvio pari a *125ms*. Il risultato è piuttosto incoraggiante, tuttavia, sia per la sua ancora scarsa maturità, sia per il fatto che il suo utilizzo necessita una certa gestione di basso livello dell'infrastruttura, attualmente non viene annoverata come effettivo competitor delle altre piattaforme serverless, ma resta comunque un'interessante alternativa di virtualizzazione, che probabilmente penetrerà sempre di più nei moderni scenari di orchestrazione.

[2] La KVM (Linux Kernel-based Virtual Machine) è una soluzione di virtualizzazione completa del kernel Linux, particolarmente ottimizzata, che garantisce l'isolamento tra le varie VM istanziate.

4 Piattaforme Function as a Service

A partire dalla sua introduzione nel panorama dei sistemi cloud, il serverless computing è andato incontro, nel corso degli anni, a diverse declinazioni, che si sono concretizzate in un numero in continua crescita di piattaforme che aiutano gli sviluppatori ad abbracciare questa nuova tecnologia, seguendo interpretazioni e approcci eterogenei. Dopo il progetto *AWS Lambda*, i maggiori vendor nel panorama serverless e FaaS hanno rilasciato, e continuano a rilasciare, nuovi strumenti e feature che consentono alle rispettive piattaforme di offrire un'esperienza sempre più completa e user-friendly.

La scelta tra le diverse piattaforme esistenti non è affatto banale e va orientata in base alle necessità dei casi d'uso che si interessa approfondire. Avendo già affrontato gli aspetti prettamente tecnologici che sono alla base delle principali piattaforme, lo scopo di questa sezione è quello di presentare le soluzioni FaaS disponibili, partendo da quelle proprietarie per motivare poi la focalizzazione sulle implementazioni open source.

4.1 Stato dell'arte delle piattaforme FaaS

Viene ora elencata e descritta la letteratura di riferimento per questo lavoro, presentando gli avanzamenti della ricerca sul panorama serverless e sulle piattaforme più utilizzate. La conoscenza e l'esperienza relative alle soluzioni FaaS sono tutt'altro che sature, sebbene negli ultimi anni siano state numerose le pubblicazioni sull'argomento, con investimenti sulla tecnologia sempre più consistenti. Le considerazioni sul tema portate avanti da un team dell'università di Berkeley [42], spunto iniziale di questa tesi, tentano di trovare i limiti principali di questa nuova tecnologia. Il loro approfondimento si focalizza sulla piattaforma *AWS Lambda* ed evidenzia come punti deboli delle funzioni serverless:

- Il tempo di esecuzione massimo, poiché dopo 15 minuti le funzioni vengono spente dall'infrastruttura di *Lambda*.
- La bandwidth utilizzabile, con test che hanno mostrato una media di banda disponibile pari a circa 538Mbps, e quindi circa un ordine di grandezza inferiore a quella di una moderna SSD; in più, questo limite di bandwidth è riferito a una VM, e, considerando che le funzioni lanciate da un singolo utente sono tipicamente schedulate sulla stessa macchina, si otterrebbe, con 20 funzioni concorrenti, una banda limitata a soli 28.7Mbps (2.5 ordini di grandezza inferiori a quella di una SSD).
- La difficoltà di accedere ad hardware specializzato, nel senso che AWS permette di accedere soltanto a risorse di CPU e RAM, senza fornire un'API o un meccanismo per accedere ad altro hardware (es. GPU).

- L’aspetto intrinsecamente stateless, infatti la conseguenza di essere stateless impone, nel caso in cui si voglia mantenere lo stato su più richieste consecutive dello stesso utente, di scrivere e leggere ogni volta lo stato su un sistema esterno (es. *Amazon S3* [43], non particolarmente veloce se comparato a un meccanismo locale).

Infine, la trattazione si conclude con la proposta di sopperire alle limitazioni citate, avvicinando le moderne piattaforme serverless alle feature dei più tradizionali sistemi cloud, ad esempio osservandole in azione su cluster con orchestrazione di container, come del resto si vedrà più avanti in questa trattazione.

Altri lavori hanno individuato alcuni design pattern per una buona ingegnerizzazione di un sistema serverless [44], specie dal punto di vista della sicurezza. In particolare, si sono focalizzati sulla piattaforma *AWS Lambda* e sulle best practice per quanto riguarda la gestione degli eventi, le invocazioni, la trasformazione e la trasmissione dei dati, e la gestione dello stato. Anche in questo caso, nonostante alcune limitazioni rispetto a una tradizionale architettura, viene messo l’accento sull’enorme potenzialità dei sistemi serverless, che potrebbero diventare dominanti di qui a poco, se adeguatamente supportati fino alla loro fase di maturità.

Ulteriori pubblicazioni si concentrano sul mettere a confronto soluzioni serverless proprietarie, quali *AWS Lambda*, *Google Cloud Functions* [45] e *Microsoft Azure Functions* [46], tra cui, solitamente, la piattaforma di Amazon viene eletta come vincitrice, per la sua efficienza e ridotta latenza, oltre all’esperienza maturata grazie al fatto di essere stata la prima introdotta sul mercato. La valutazione delle piattaforme proprietarie risulta, però, essere limitata a causa dell’impossibilità di accedere ad informazioni significative sull’infrastruttura sottostante.
I test eseguiti da una collaborazione tra le università del Wisconsin-Madison, dell’Ohio e della Cornell Tech di New York [47] tentano di risalire al numero di VM e alle risorse utilizzate dai vari framework, osservando le informazioni collezionate a runtime dal proc filesystem di Linux (*procfs*). La loro sperimentazione si basa sul lanciare l’esecuzione di 50000 funzioni sulle diverse piattaforme e osservare poi il loro comportamento. Il primo aspetto considerato riguarda l’isolamento degli utenti, con risultati ottenuti che mostrano come in AWS i set di VM istanziate siano disgiunti per diversi utenti, mentre in Azure è possibile che questi siano condivisi tra i clienti, dando a quest’ultima piattaforma uno svantaggio in termini di sicurezza. Google invece oscura questo tipo di dettagli, non rendendo possibile un’analisi del livello di isolamento, e risultando essere, in generale, la più rigida nel tenere nascoste le informazioni relative al numero e alla tipologia di VM utilizzate. La ragione è che tali informazioni sul runtime potrebbero rappresentare un’apertura verso un eventuale attaccante.
Anche per quanto riguarda la scalabilità, la piattaforma di Amazon ha mostrato una marcia in più, riuscendo a generare fino a 200 istanze della stessa funzione, a fronte della

ricezione di un numero elevato di richieste, mentre con Azure sono state create al massimo 10 istanze. Le cloud functions di Google, invece, hanno fallito il test di scalabilità, accodando le richieste piuttosto che generare più istanze concorrenti.
Si passa poi al confronto sulle latenze nei *cold start*, ovvero negli avvii di funzioni quando non esiste ancora alcuna istanza pronta a rispondere. I risultati mostrano valori mediani di 265.21ms per AWS, 493.04ms per Google e 3640.02ms per Azure, con evidenti problemi di design sulla piattaforma di Microsoft, che però, ad oggi, sembrerebbero essere stati perlopiù risolti.
L'allocazione dei tempi della CPU è, infine, in tutti e tre i competitor una funzione della memoria messa a disposizione delle istanze: quelle con più memoria ottengono di default più cicli di CPU. Amazon ha registrato valori che vanno dal 7.7% al 92.3% di utilizzo all'aumentare della memoria, mentre Google e Azure hanno riportato range di valori rispettivamente pari a 11.1%-100% e 14.1%-90% di utilizzo di CPU.
In generale, si ribadisce che questi risultati derivano da un'osservazione indiretta degli ambienti e che non tutte le piattaforme espongono i dettagli necessari a una valutazione oggettiva e a una comparazione precisa. Ciò ha spinto a concentrarsi, in questa tesi, sulle piattaforme open source per avere un'idea più concreta di ciò che accade durante l'esecuzione di una funzione serverless.

Un'altra indagine ha invece individuato le caratteristiche che dovrebbe avere una piattaforma serverless ideale per affermarsi come standard, ipotizzando un modello minimo comune denominatore che possa portare, nello scenario FaaS, la stessa unificazione portata dall'orchestratore di container Kubernetes nello scenario PaaS [48].
In particolare, vengono presentati i seguenti aspetti come conformi a un ipotetico modello standard:

- Packaging delle applicazioni secondo la specifica *OCI*, uno standard per le immagini di container [49].
- Utilizzo di eventi conformi alla specifica *CloudEvents* [50], un formato comune per consentire interoperabilità tra i servizi.
- Supporto nativo ai modelli sincroni e asincroni, senza lasciare allo sviluppatore il compito di gestire i due modelli differentemente.
- Supporto nativo a una service mesh, i cui vantaggi vengono descritti nelle sezioni successive di questo documento.
- Autoscaling e concorrenza con modello pull-based, in modo che sia la piattaforma a sapere come e quando è necessario scalare o servire una risposta.
- Avere un meccanismo built-in per la gestione concorrente di versioni diverse dell'applicazione.

Nell'indagine in questione, è stato preso *Knative* [51] come framework di riferimento, in quanto, più di tutte, rispecchia o si avvicina ai punti appena elencati.

Bisogna citare, infine, il lavoro svolto da un team all'interno del Trinity College di Dublino [52], per alcuni aspetti simile all'indagine effettuata in questo documento. La loro analisi ha messo a confronto le medesime piattaforme serverless selezionate nella prima fase di questo progetto, *Apache OpenWhisk* [53], *OpenFaas* [54] e *Knative*, con l'aggiunta di *Kubeless* [55], con applicazione sul mondo dell'IoT e dell'edge computing. Questo particolare scope ha portato al design di test con carichi computazionali relativamente inferiori a quelli presentati più in avanti in questo documento. Il massimo grado di concorrenza a cui sono state sottoposte le piattaforme è infatti di sole 20 richieste ricevute simultaneamente dai framework. Sarà dunque interessante confrontare i risultati da loro ottenuti, derivanti da uno scenario più ristretto, con quelli mostrati in questa tesi, per comprendere se, all'aumentare delle interazioni, le varie soluzioni mantengono una certa proporzionalità o se si verificano colli di bottiglia meno lineari.
Ad ogni modo, osservando i valori riportati dai test nel caso di maggior carico generato, si ha *Kubeless* in testa alla classifica, con un throughput di circa 60 transazioni al secondo, una percentuale di successo delle richieste del 100% e un tempo di attesa medio di circa 100ms. A seguire, *OpenFaas* e *Knative* hanno raggiunto valori molto simili, con throughput di circa 50 transazioni al secondo, percentuale di successo del 100% e tempo di attesa medio rispettivamente di circa 100ms e 130ms. Il meno efficiente si è rivelato essere *OpenWhisk*, con throughput leggermente al di sotto alle 40 transazioni al secondo, tempo di risposta medio circa equivalente a quello di *Knative*, ma percentuale di successo del solo 10%.

4.2 Piattaforme FaaS open source

Come evidenziato da alcune pubblicazioni prodotte [47], nello scenario delle soluzioni proprietarie per il serverless, i tre maggiori competitor sono *Amazon Lambda* [1], *Google Cloud Functions* [45] e *Microsoft Azure Functions* [46]. Il loro sviluppo tecnologico è veloce, promettente e ha raggiunto di recente ottimi livelli di performance ed efficienza, specie per quanto riguarda il framework di Amazon. Tuttavia, questi nascondono dettagli importanti su diversi aspetti relativi al loro funzionamento e al modo in cui gestiscono le risorse, al fine di offrire, agli utenti, sistemi consistenti e pronti all'uso.
Se l'intenzione non è però quella di porsi come semplice utente, ma piuttosto di analizzare le fondamenta delle soluzioni serverless con un'introspezione sull'architettura e sulle interazioni tra i vari componenti, allora risulta necessario affacciarsi al mondo open source. In questo modo si può avere un controllo sul sistema notevolmente maggiore ed effettuare test e valutazioni concrete, senza dover ipotizzare o intuire cosa accade sotto il cofano.

Recentemente, diversi sono stati i framework open source proposti per realizzare il modello FaaS, e i nomi più ricorrenti sono ***Apache OpenWhisk*** [53], ***OpenFaas*** [54],

Knative [51], ***Fission*** [56], *Kubeless* [55], *Nuclio* [57] e *OpenLambda* [58]. Un criterio frequentemente utilizzato per selezionare un progetto open source è la solidità della community che lo supporta. In questo lavoro il principio di selezione si è basato sul numero di *contributor* e *follower* delle diverse repository su *GitHub* [59] associate ai rispettivi framework. La Tabella 1 evidenzia il numero di contributor come parametro primario, e, in caso di pareggio, il numero di stelle riconosciute, dove per “stella” si intende un meccanismo di GitHub col quale gli utenti possono salvare una repository come preferita e/o segnalare apprezzamento nei confronti di questa.

Framework	Contributors	Stars
Apache OpenWhisk	173	4586
Knative	164	2737
OpenFaas	137	17029
Fission	91	4970
Kubeless	82	5489
Nuclio	51	3194
OpenLambda	19	667

Tabella 1 - Contributors e stelle GitHub (aggiornata al 02/03/2020)

I primi quattro framework, evidenziati nella tabella, sono i candidati più promettenti. Nelle sezioni successive viene riportata una panoramica sulle quattro soluzioni selezionate, dove, per ognuna, è presentata una breve descrizione iniziale, seguita da un’analisi dell’architettura e delle modalità di interazione tra i componenti, per continuare poi con un’esposizione dei meccanismi di utilizzo basilari per generare e invocare funzioni serverless. Infine, si termina l’osservazione di ogni piattaforma con una rapida indagine su alcuni aspetti legati alla scalabilità automatica (*autoscaling*), punto di forza e principio fondamentale delle FaaS.

4.3 OpenFaas

OpenFaas [54] è un framework serverless gratuito ed open source basato su licenza MIT, sostenuto da una sempre maggiore community di sviluppatori. La sua popolarità costantemente in espansione e la curva di apprendimento non eccessivamente ripida che lo caratterizza lo hanno reso la scelta ideale per iniziare ad approcciarsi al mondo delle FaaS da un punto di vista teorico e applicativo. I punti cruciali sono appunto la facilità di utilizzo e la portabilità, essendo una delle poche piattaforme capaci di funzionare anche

su orchestratori alternativi a Kubernetes, come descritto nella sezione seguente [60]. Questo aspetto è fondamentale, in quanto la complessità e le risorse necessarie per istanziare Kubernetes possono, in alcuni casi, rappresentare un forte limite, specie se ci si trova in un ambiente sandbox locale con la sola intenzione di esplorare, approfondire e testare il modello serverless. In questi casi, scegliendo OpenFaas, è possibile utilizzare orchestratori alternativi, come ad esempio Docker Swarm, ben più semplici e ridotti rispetto a Kubernetes.

4.3.1 Architettura

La Figura 16 presenta i vari componenti dell'architettura e le loro interazioni.

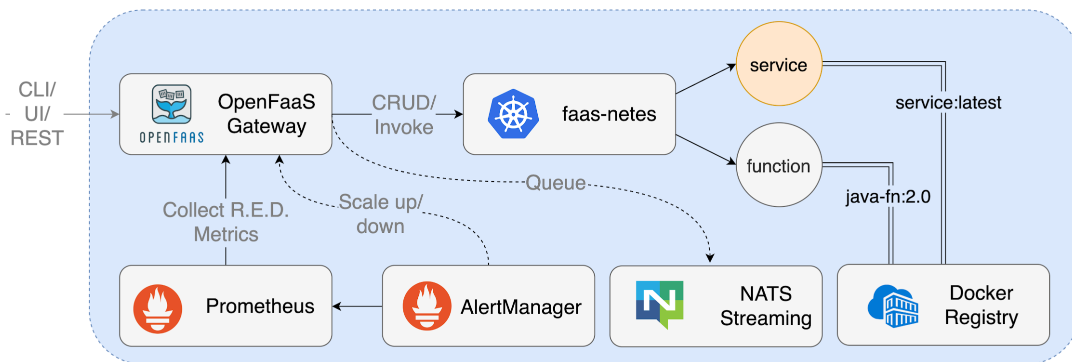

Figura 16 - Architettura di OpenFaas

Il framework è accessibile tramite REST API, CLI o interfaccia web proprietaria fornita nel package. Tutte le richieste vengono indirizzate verso il **Gateway**, primo componente che si incontra nel workflow di OpenFaas. Il Gateway, punto di accesso al framework, fornisce una collezione di API per gestire le funzioni caricate nel framework e comunica direttamente con il componente **faas-netes,** provider OpenFaas per Kubernetes, e unico controllore delle operazioni relative alle funzioni. Una volta ricevuta la richiesta inoltrata dal Gateway, faas-netes provvede a mettere in esecuzione la funzione richiesta, appoggiandosi a un registro di immagini Docker per avviare il relativo container. È proprio questa separazione tra il Gateway e i provider che permette di rendere il framework *infrastructure-independent*. Per cui, se a Kubernetes si volesse sostituire Docker Swarm, basterebbe sostituire unicamente faas-netes con un'altra implementazione. Oltre che dell'attivazione delle funzioni, il Gateway è infine responsabile del numero di repliche della funzione e permette la visualizzazione delle statistiche accumulate dal sistema di monitoring.

In OpenFaas le repliche vengono gestite nativamente, senza avvalersi del meccanismo di scaling automatico offerto da Kubernetes, nonostante esso sia selezionabile come soluzione alternativa. In particolare, il componente **Prometheus** [61]**,** popolare sistema di monitoring open source, colleziona metriche relative all'utilizzo delle risorse da parte del framework e notifica, se necessario, il Gateway, il quale andrà a modificare il numero di repliche disponibili nel cluster. Per realizzare il suo scopo, Prometheus sfrutta il componente **AlertManager**, visualizzabile come insieme di regole definite dall'utente che, se violate, vanno a far scattare il sistema di autoscaling.

L'invocazione di una funzione in OpenFaas può essere sia sincrona che asincrona, tuttavia, poiché le funzioni qui utilizzate in fase di test sono stateless e logicamente semplici, si è preferito concentrarsi sull'approccio sincrono, ma va comunque sottolineata la possibilità di cambiare facilmente paradigma [62]. Il componente che consente di realizzare l'asincronicità delle funzioni è **NATS Streaming** [63], broker di eventi open source e distribuito, che, con i relativi *queue-workers*, garantisce un disaccoppiamento delle interazioni nello spazio e nel tempo.

Ogni funzione, quando distribuita, oltre al processo responsabile per l'esecuzione del codice, possiede un processo **watchdog**, esposto sulla porta HTTP 8080, per accettare connessioni provenienti dal provider di OpenFaas.

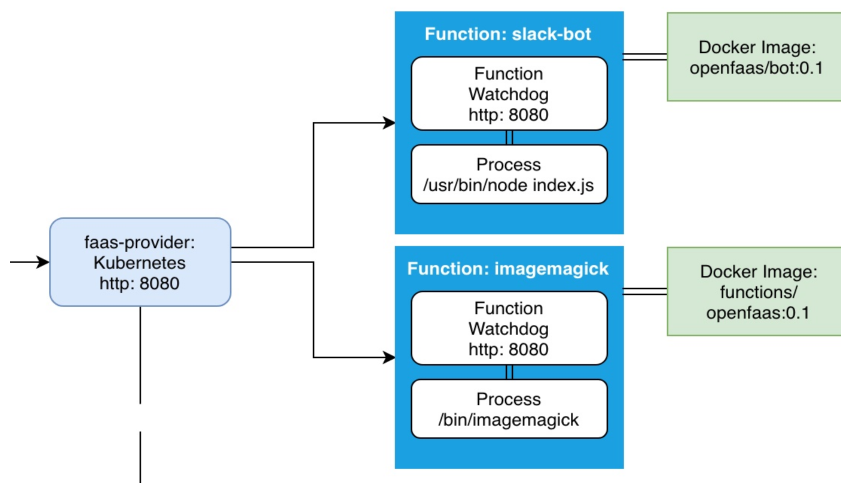

Figura 17 - Watchdog in OpenFaas

Il processo watchdog, descritto in Figura 17, scatena l'esecuzione della funzione, monitorando le richieste concorrenti e i possibili timeout. Attualmente OpenFaas offre una nuova implementazione, chiamata **of-watchdog**, con la quale si raggiungono performance migliori, gestendo pool di connessioni riciclabili invece di generare un nuovo processo per ogni richiesta, come faceva la vecchia versione [64].

Infine, il componente **faas-idler** permette lo *zero-to-scale*, ovvero il meccanismo grazie al quale nessuna replica della funzione viene fatta eseguire finché non vi sono richieste da elaborare. Ciò migliora la gestione e l'utilizzo delle risorse, ma aumenta la latenza a freddo (*cold start*). Nelle sperimentazioni effettuate sulla piattaforma, questa funzionalità è stata disattivata, per cui il componente non presenta alcun interesse [65].

4.3.2 Utilizzo

In OpenFaas, le funzioni vengono gestite da un CLI proprietario, **faas-cli,** il quale comunica direttamente con il Gateway, ma, per rendere questo possibile, è necessario definire la variabile d'ambiente *OPENFAAS_URL*. In alternativa, risulta essere estremamente comoda anche la dashboard integrata e accessibile da browser, grande punto a favore della piattaforma, essendo l'unica a fornire uno strumento del genere.

Il framework offre un **template** per diversi ambienti di esecuzione, permettendo di scrivere codice nei linguaggi più popolari. Nell'esempio qui riportato, così come nei test eseguiti in questo lavoro, viene utilizzato il template per framework *NodeJS*. La prima cosa da fare è scaricare e salvare il template necessario per il deployment della funzione desiderata:

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

A questo punto è possibile utilizzare il template per creare una funzione per NodeJS:

```
$ faas-cli new myFun --lang node10-express
```

Viene automaticamente generata la struttura della funzione, basata sul template passato come argomento, che presenta i seguenti file:

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

Nel file *handler.js* va inserita la logica di business, mentre nel file *.yml* è presente la configurazione per il deployment, con indicazione del numero di repliche desiderate dei limiti imposti sull'utilizzo delle risorse.

Prima di pubblicare la funzione bisogna costruire l'immagine Docker della funzione, salvandola nel registro Docker locale:

```
$ faas-cli build -f myFun.yml
```

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

```
$ faas-cli push -f myFun.yml
```

Infine, si può effettuare il deployment della funzione:

```
$ faas-cli deploy -f myFun.yml
```

Per invocare la funzione è possibile utilizzare il CLI, la UI accessibile via browser o effettuare richieste HTTP, raggiungendo l'indirizzo del Gateway:

```
$ OPENFAAS_URL:8080/function/myFun
```

4.3.3 Autoscaling

Di default, OpenFaas presenta una singola regola in AlertManager, che definisce il massimo numero di richieste al secondo che un singolo pod può ricevere prima di dover scalare e che avverte il Gateway in caso di superamento. Sono importanti i parametri di configurazione con cui decidere i limiti di autoscaling di ogni funzione, in particolare:

- `com.openfaas.scale.min` - numero minimo di repliche, impostato a 1 durante i test.
- `com.openfaas.scale.max` - numero massimo di repliche; durante i test che si vedranno più avanti, i valori sono stati scelti a seconda dell'ambiente di esecuzione; sul cluster il valore è rimasto a 20 (default), mentre in locale il valore è stato abbassato a 10 per evitare di saturare le risorse a disposizione.

In generale, quando si fissano le soglie per l'autoscaler, bisogna fare attenzione e considerare che non è possibile istanziare arbitrariamente le repliche. Lo **scheduler**, componente di Kubernetes, promette ad ogni container una certa quantità di risorse, ma se le risorse richieste da una determinata applicazione superano quelle effettivamente disponibili, la configurazione desiderata non può essere soddisfatta e Kubernetes non schedula le repliche, lasciandole in uno stato di *pending*.

Oltre al meccanismo proprietario di OpenFaas, è possibile, come già anticipato, anche scalare le repliche in base all'utilizzo delle risorse, come CPU e memoria. Per farlo, è necessario disattivare AlertManager ed utilizzare l'autoscaler nativo di Kubernetes,

Horizontal Pod Autoscaler (**HPA**), il cui funzionamento viene descritto nelle sezioni successive. OpenFaas è stato testato con entrambi gli autoscaler per poterlo confrontare meglio con gli altri framework, che invece utilizzano quasi esclusivamente HPA.

4.4 OpenWhisk

Apache OpenWhisk [53] è una piattaforma serverless open source, basata su programmazione ad eventi ed adottata da *IBM Cloud Functions* [66]. Le funzioni, chiamate qui **action**, possono essere scritte in qualsiasi linguaggio di programmazione, secondo i principi del serverless, e possono essere invocate sia a seguito di eventi (*trigger* e *rule*) emessi da sorgenti esterne (*feed*), sia direttamente da richieste HTTP. L'architettura orientata ai microservizi open source e le interazioni disaccoppiate ad eventi la pongono come una delle principali soluzioni serverless per sistemi ad alta scalabilità. Di seguito è riportata una panoramica sugli aspetti fondamentali della piattaforma.

4.4.1 Architettura

L'architettura di OpenWhisk viene presentata nella Figura 18 come composta da un insieme di popolari componenti del panorama open source.

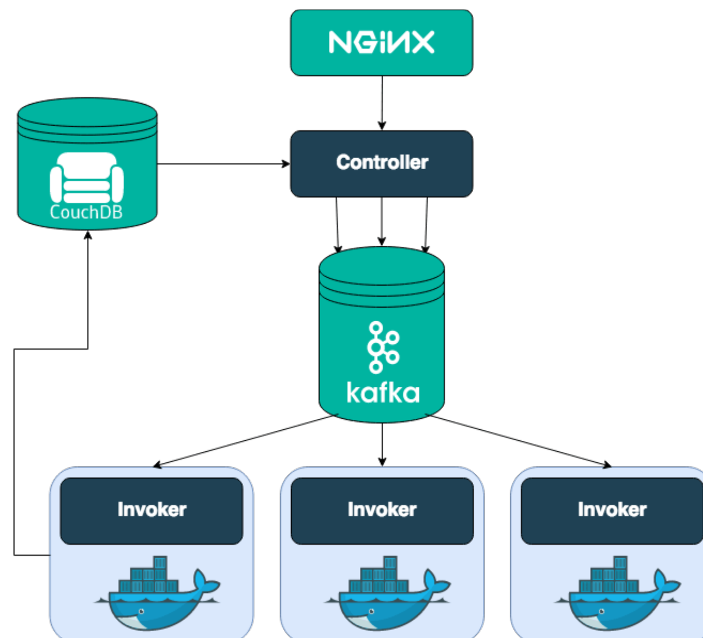

Figura 18 - Architettura di OpenWhisk

Nginx [23] è un HTTP server open source, oltre che *reverse proxy*, famoso per performance, stabilità, ampie possibilità di configurazione e costi contenuti. Al contrario di server tradizionali, Nginx non utilizza thread per gestire le richieste, bensì eventi che garantiscono un'architettura estremamente più scalabile, oltre ad essere completamente stateless.

In OpenWhisk, Nginx è utilizzato come reverse proxy per le API del servizio, inoltrando le richieste al **Controller**. Il Controller autentica ed autorizza tutte le operazioni ricevute dal proxy, permettendo di creare, recuperare, aggiornare ed eliminare (CRUD) entità interne al sistema, ma soprattutto di invocare le funzioni. All'interno del Controller è integrato un **load balancer**, inizialmente presente come componente esterno, che ha una visione globale degli esecutori disponibili nel sistema, chiamati Invoker, dei quali controlla continuamente lo stato di integrità. Il Controller, che conosce quali sono gli Invoker disponibili, delega questi ultimi per l'esecuzione effettiva delle funzioni.

Lo stato del sistema è mantenuto e gestito da **Apache CouchDB** [67], database document-oriented open source NoSQL, che salva i propri dati in JSON ed esegue query in MapReduce. In CouchDB vengono salvati metadata, credenziali, namespace, definizioni di *action*, *trigger* e *rule*. Il Controller legge i dati presenti in CouchDB per verificare identità (autenticazione) e permessi (autorizzazione) di ciascun utente prima di eseguire qualunque operazione. Durante l'aggiunta di una nuova action, il Controller autorizza l'utente, carica l'azione su CouchDB e salva il codice da eseguire, i parametri formali, i parametri attuali e i limiti imposti sulle risorse allocabili.

Apache Kafka [68] è un sistema open source di messaggistica distribuita con architettura pub/sub stateless, sviluppato da *LinkedIn*, per applicazioni in tempo reale e streaming con comunicazioni asincrone, garantendo resilienza in caso di guasti e scalabilità in caso di elevato workload. Il Controller e gli Invoker comunicano esclusivamente attraverso messaggi mantenuti in un buffer da Kafka. Per invocare un'azione, il Controller pubblica un messaggio su Kafka, contenente l'azione da chiamare e i relativi parametri. Una volta che Kafka ha confermato di aver ricevuto il messaggio, la richiesta HTTP riceve una risposta con ID di attivazione. L'utente può utilizzarlo in seguito per ottenere l'accesso ai risultati di quella specifica chiamata.

Apache ZooKeeper [69] è uno strumento di supplemento che gestisce un cluster Kafka, con lo scopo primario di mantenere lo stato dei nodi presenti nel cluster e di tenere traccia dei topic, dei messaggi e delle risorse in uso.

L'**Invoker** è l'esecutore dell'azione ed utilizza Docker per garantire sicurezza e isolamento. Per ogni chiamata di azione viene generato un container dove si esegue il codice corrispondente, preventivamente recuperato da con CouchDB. Quando l'esecuzione termina, viene salvato il risultato, di nuovo su CouchDB, e distrutto il container.

4.4.2 Utilizzo

L'intera architettura viene facilmente pilotata attraverso il CLI proprietario chiamato **wsk**. Il CLI permette di creare e invocare le action, recuperarne il risultato, gestire gruppi di action correlate all'interno dei cosiddetti *package*, e abilitare o disabilitare l'esecuzione di action alla ricezione di un particolare evento. Una volta selezionato l'environment desiderato (es. NodeJS), è possibile creare una action:

```
$ wsk action create myFun function.js
```

L'azione viene creata e, con essa, vengono generati uno o più pod pronti a ricevere richieste di esecuzione dell'azione stessa. L'invocazione può essere testata lanciandola tramite CLI:

```
$ wsk action invoke myFun
```

L'invocazione restituirà un codice di attivazione, identificativo del risultato memorizzato in CouchDB. Tramite wsk si potrà accedere al database, passando il codice di attivazione, e visualizzare il risultato.

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

Risulta già evidente quanto sia più semplice e immediato, rispetto a OpenFaas, il processo che porta dalla scrittura del codice all'esecuzione della funzione: nessuna necessità di scaricare un template, ma soprattutto nessuna necessità di doversi interfacciare con un registro di immagini.

4.4.3 Autoscaling

OpenWhisk utilizza un meccanismo di autoscaling proprietario, senza affidarsi all'HPA offerto da Kubernetes. Il comportamento dell'autoscaler risulta essere principalmente proattivo, con la tendenza di generare un certo numero di pod supplementari non appena il sistema si accorge che il traffico in entrata è in aumento. La configurazione manuale delle metriche per l'autoscaling non è menzionata nella documentazione, tuttavia viene fornito un elenco di best practice per ottenere un sistema con una scalabilità migliore.

Di default viene istanziato un solo Controller centralizzato per l'intera piattaforma, configurazione che, in buona parte dei casi, può condurre a evidenti problemi di scalabilità, colli di bottiglia e *single point of failure*. Sia la documentazione che le sperimentazioni effettuate hanno dimostrato che è sufficiente aggiungere un secondo

Controller per mantenere una buona coesione del sistema, sopperendo al contempo ai difetti sopra citati. Sfortunatamente questo approccio non è attualmente applicabile a tutti i componenti, pertanto, nelle sperimentazioni, si è deciso di replicare soltanto il Controller e gli Invoker, mettendo un Invoker su ogni nodo:

```
controller:
  replicaCount: 2
invoker:
  replicaCount: 5
```

Un'altra best practice in favore di scalabilità e affidabilità, specie in scenari di produzione, è il disaccoppiamento di OpenWhisk da CouchDB, utilizzando quest'ultimo come servizio esterno alla piattaforma, per garantire una maggiore flessibilità e consistenza dei dati in caso di malfunzionamento del cluster:

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

Come anticipato, il componente Invoker può essere replicato su più nodi del cluster. Tipicamente la scelta ottimale consiste nel porre un Invoker su ogni worker node del cluster e di dedicare, alla *container factory* degli Invoker, una quantità di memoria pari alla somma della memoria disponibile sui vari worker diviso il numero dei worker stessi. Questa configurazione è possibile soltanto se ci si affida alla *Kubernetes Container Factory*, grazie alla flessibilità offerta dall'orchestratore. L'altra possibilità, la *Docker Container Factory* implementata direttamente da Docker, garantisce una latenza minore, trovandosi a un livello architetturale più basso e vicino a quello dei container, ma perde le feature offerte da Kubernetes, quali un management delle risorse avanzato e alcuni settaggi per la sicurezza. Durante la fase di testing, sono state valutate le performance di OpenWhisk utilizzando entrambi gli approcci, ottenendo risultati migliori in presenza della factory di Kubernetes, recentemente divenuta la configurazione di default.
Infine, poiché il log processing degli Invoker può diventare un collo di bottiglia per la Kubernetes Container Factory, è possibile delegare il log processing dall'Invoker presso altri provider esterni al framework.

```
invoker:
  containerFactory:
```

```
    impl: "kubernetes"
  options: /
"-
Dwhisk.spi.LogStoreProvider=org.apache.openwhisk.core.containerpool
.logging.LogDriverLogStoreProvider"
```

4.5 Knative

Knative [51] estende Kubernetes fornendo componenti middleware essenziali alla costruzione di applicazioni container-based eseguibili in qualunque ambiente cloud. Sviluppato da *Google*, e supportato da *RedHat*, *IBM* e *T-Mobile*, viene definito come il *"serverless add-on"* per Kubernetes, nel senso che Knative è il *building-block* per eseguire workload serverless su Kubernetes. Dunque, questa piattaforma appare piuttosto diversa da tutte le altre, in quanto non nasce con lo scopo di fornire direttamente le FaaS, ma di estendere Kubernetes con una serie di feature e API utilizzabili per costruire, tra le altre cose, delle funzioni serverless. Sebastian Goasguen, fondatore di *Kubeless*, ritiene che sarà proprio questo il fondamento del futuro del serverless:

"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." [70]
["Knative sta rendendo standard le primitive necessarie a semplificare il codice dei workflow di produzione. Promette di diventare un fondamento solido per sviluppare il futuro del serverless computing."]

Il funzionamento di Knative si basa sulle **Custom Resource Definition** o **CRD** [71], ovvero un meccanismo offerto nativamente da Kubernetes per aggiungere componenti personalizzati all'orchestratore come se fossero oggetti built-in. È proprio questo il motivo per il quale Knative viene considerato il meccanismo più naturale per avvicinare Kubernetes al mondo serverless, grazie a un'integrazione completa con l'orchestratore.

4.5.1 Architettura

Come detto, questa piattaforma è intrinsecamente diversa dalle altre, e il primo divario lo si vede chiaramente nella Figura 19 sottostante, per quanto riguarda le modalità con cui si possono raggiungere i servizi da essa esposti.

Figura 19 - Architettura di Knative

A differenza delle altre piattaforme, che forniscono un API gateway *out-of-the-box*, Knative necessita della presenza di un **ingress gateway** con cui gestire il traffico esterno verso il framework. Diverse sono le soluzioni supportate da Knative, ma la principale e maggiormente utilizzata risulta essere **Istio** [72], descritta per completezza nella sezione successiva.

Knative è composto da due blocchi logici indipendenti tra loro che vengono aggiunti direttamente al di sopra di Kubernetes, *Serving* ed *Eventing*.
Il componente **Serving** gestisce le interazioni sincrone con il framework, oltre che i dettagli di networking, l'autoscaling (anche fino a zero pod) e il revision tracking. Il componente offre infatti l'interessante funzionalità dello *staged roll out*, la quale consente di rilasciare aggiornamenti in modo graduale e di mantenere contemporaneamente più versioni (o revision) dello stesso servizio. Il Serving introduce quattro CRD per definire il comportamento del workload serverless su Kubernetes. Le interazioni tra questi sono elencate di seguito e raffigurate nella Figura 20:

- **Service** - gestisce il ciclo di vita dei servizi e la loro creazione, garantendo ad ogni servizio una Route, una Configuration e una Revision.
- **Route** - mappa un endpoint di rete a una o più Revision e gestisce il traffico.
- **Configuration** - mantiene lo stato desiderato del servizio e segue la metodologia delle *12-Factor App* [6]; la modifica di una Configuration genera una nuova Revision.

- **Revision** - snapshot immutabile del workload serverless (es. di un servizio) catturata in un determinato istante di tempo, ed è pertanto un oggetto immutabile; il meccanismo delle Revision è utilizzato in combinazione a quello delle Configuration, e permette la separazione tra codice (immagini dei container) e loro configurazione, così che, dopo ogni cambiamento di configurazione, sia sempre possibile effettuare un rollback all'ultima configurazione funzionante registrata; per realizzare ciò, è necessario che Knative crei una Revision per ogni nuova configurazione.

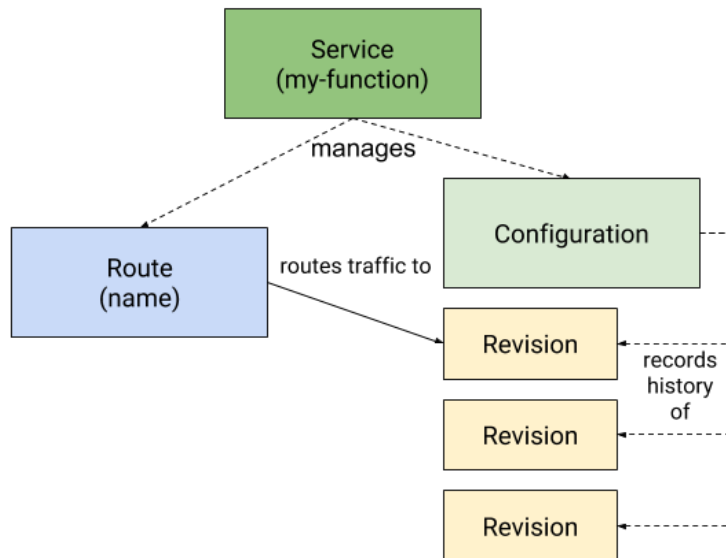

Figura 20 - Knative serving CRD

Il componente **Eventing**, che non può esistere senza la presenza preliminare del componente Serving, permette l'interazione tra servizi con la programmazione ad eventi, disaccoppiando consumer e producer, e favorendo lo sviluppo di servizi asincroni con elevati gradi di scalabilità.

Infine, Knative offre comodi servizi di **monitoring** per la gestione di logging e di metriche utili per la valutazione in tempo reale del sistema in esame. Tali metriche, accumulate da **Prometheus** [61], possono essere poi riportate graficamente tramite il tool **Grafana** [73], popolare soluzione open source che consente di agganciarsi a una sorgente dati e di visualizzare i valori letti da questa attraverso contatori e grafici.

Ingress gateway e service mesh

Nell'analisi architetturale di questa piattaforma, bisogna aggiungere, rispetto alle altre, una breve appendice riguardo alle **service mesh**. È stato già sottolineato che Knative necessita di un gateway esterno, che deve già essere presente prima che il framework

venga installato. Esistono diverse implementazioni di ingress gateway supportate da Knative, come *Ambassador* [74], *Gloo* [75], *Contour* [76]. La scelta più completa proposta per Knative è però quella di una service mesh integrale, che inglobi anche la funzione di API gateway.

Con "*service mesh*" si intende un'infrastruttura di comunicazione dedicata, per far fronte alla crescita di microservizi ed interazioni in sistemi a container. La separazione tra computazione ed interazione ottimizza l'architettura del cluster, permettendo di offrire un maggior numero di servizi senza complicare ulteriormente il controllo del sistema. La service mesh smista le richieste da un servizio all'altro, migliorando l'isolamento tra le applicazioni e garantendo un nuovo livello di flessibilità nelle fasi di sviluppo e test.
La comunicazione tra servizi è resa possibile costruendo una rete di **sidecar proxy**, ovvero proxy che permettono l'invio e la ricezione delle richieste lavorando al fianco dei servizi. L'insieme di proxy e le relative interazioni formano una rete *mesh*, da cui il nome.

Senza la service mesh, ogni microservizio dovrebbe implementare logiche di comunicazione service-to-service, con conseguente aumento di tempo e costi di progettazione. Inoltre, possibili soluzioni proprietarie porterebbero costi maggiori di debugging poiché le logiche service-to-service sarebbero diverse ed inaccessibili per la maggior parte delle applicazioni.
La service mesh colleziona infine metriche per management e recovery dei servizi, migliorando la scalabilità dell'intero sistema: maggiore facilità di debugging, maggiore resilienza ai guasti ed ottimizzazione a runtime dell'ambiente di produzione.

Istio [72] è la service mesh open source adottata da Knative, capace di offrire funzionalità di load balancing, autenticazione service-to-service, monitoring, e management del traffico. L'architettura di Istio è composta da diversi componenti, indipendenti tra loro, riportati in Figura 21:

- **Envoy** - permette la comunicazione tra servizi, oltre ad offrire discovery dei servizi, 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.

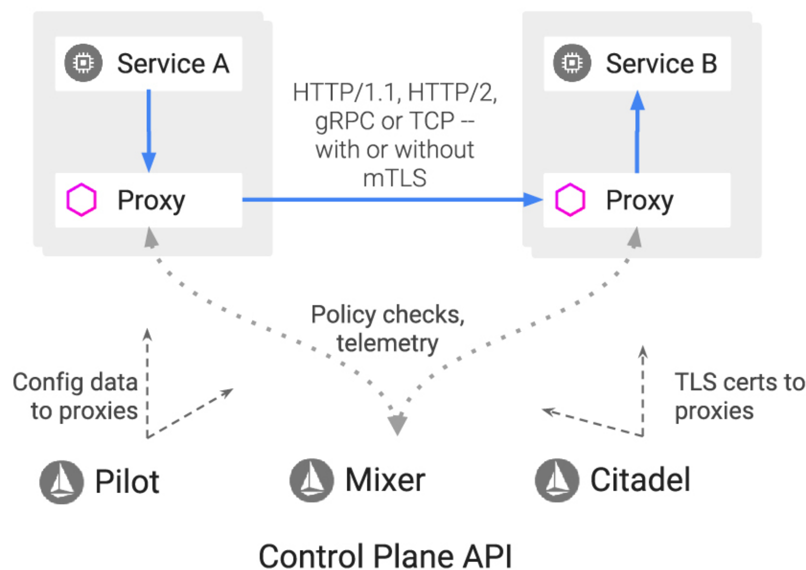

Figura 21 - Architettura di Istio

Istio è stata la soluzione selezionata in fase di testing, giovando così delle feature fornite da una service mesh completa, open source, ben documentata, ma soprattutto affermata in tutti principali utilizzi di Knative.

4.5.2 Utilizzo

In questo caso l'approccio al serverless risulta essere semplificato, in quanto è possibile lavorare direttamente con le funzionalità offerte dalle API di Kubernetes tramite il client *kubectl*, senza alcuno strumento o CLI esterno. In Knative, per caricare una funzione, è necessario descriverla come *custom resource*, applicando il relativo file *.yaml* al cluster, come si farebbe per qualsiasi altra risorsa nativa in Kubernetes:

```
# kubectl apply --filename myFun.yaml
```

Il servizio viene aggiunto alla lista dei servizi di Kubernetes, ma nessun pod viene generato finché il servizio non viene invocato:

```
$ curl http://myFun.default.mydomain.com
```

Va però posto un accorgimento nel caso in cui si voglia invocare il servizio dall'esterno. Infatti, bisogna considerare il vincolo di dover passare attraverso Istio per entrare nel cluster e invocare il servizio. In assenza di un servizio DNS, è necessario indicare

esplicitamente l'host nell'header della richiesta, altrimenti l'ingress gateway non è capace di gestirla:

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

Durante i test, per evitare di definire manualmente l'host desiderato, si è preferito aggiungere una entry nel file */etc/hosts* indicando il mapping tra nome dell'host e indirizzo del gateway:

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

In uno scenario di produzione, il mapping tra nome dell'host e indirizzo verrebbe chiaramente risolto da server DNS esterni, raggiungibili dal cluster.

4.5.3 Autoscaling

Senza sorprese, Knative utilizza l'autoscaler HPA direttamente offerto da Kubernetes, tuttavia lo espande con alcune funzionalità interessanti. Prima tra tutte, lo **scale-to-zero**, ovvero la capacità di terminare tutti i pod corrispondenti a un determinato servizio, dopo un certo intervallo di tempo di inattività (a default, 30 secondi):

```
enable-scale-to-zero: "true"
scale-to-zero-grace-period: "30s"
```

Questo aspetto conduce a un trade-off tra minore consumo di risorse e maggiore tempo di risposta alla prima richiesta. I *cold start*, difatti, affliggono il sistema di Knative, il quale deve inizializzare ogni volta un nuovo pod nel caso in cui sia trascorso un certo periodo di inattività tra due richieste consecutive. Per far fronte a questo problema, è stata aggiunta la possibilità di definire il minimo e massimo numero di repliche desiderate, disattivando dunque lo scale-to-zero:

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

Le metriche utilizzate dall'autoscaler sono:

- Numero di richieste che un container può gestire contemporaneamente.
  `container-concurrency-target-default: "100"`

- Numero di richieste al secondo ricevute.
```
requests-per-second-target-default: "200"
```

- Consumo della cpu (metrica standard di Kubernetes per gestire l'HPA).
```
annotations:
  autoscaling.knative.dev/metric: cpu
  autoscaling.knative.dev/target: 70
  autoscaling.knative.dev/class:
  hpa.autoscaling.knative.dev
```

4.6 Fission

Fission [56] è un framework serverless open source per Kubernetes supportato da *Platform9* [77], società americana di servizi *hybrid cloud*. Gli aspetti più interessanti sono la facilità d'utilizzo, le performance e l'importante community, cresciuta soprattutto negli ultimi mesi.

Alla base di Fission ci sono tre concetti fondamentali (Figura 22):

- **Trigger** - permette di catturare diversi tipi di eventi per invocare funzioni; quando il trigger riceve una richiesta, invoca la funzione inviando a sua volta una richiesta HTTP al router alla funzione.
- **Environment** - offre processi per compilare ed eseguire funzioni in un dato linguaggio; è composto da container che catturano richieste HTTP ed eseguono le istruzioni della funzione; Fission offre numerosi environment, coprendo i linguaggi di programmazione più popolari.
- **Function** - rappresenta la classica funzione, solitamente composta da interfaccia e relative implementazioni [78].

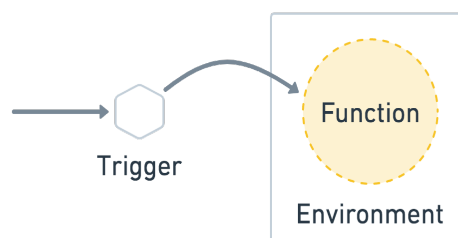

Figura 22 - Componenti principali di Fission

4.6.1 Architettura

Il meccanismo di interazione tra i componenti dell'architettura di Fission è mostrato nella seguente Figura 23.

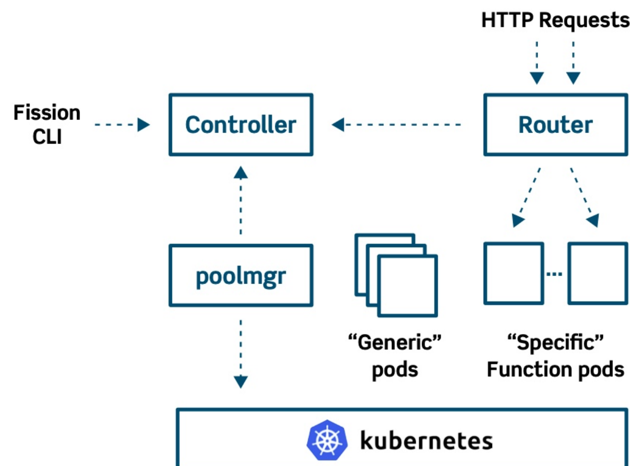

Figura 23 - Architettura di Fission

È possibile accedere alle operazioni di Fission tramite CLI, il quale raggiunge gli endpoint esposti dal componente **Controller** per invocare, creare, modificare ed eliminare le funzioni. Allo stesso modo di Knative, tutte le risorse di Fission sono definite come *custom resource*, e il Controller ha pieni poteri sulle risorse di tutti i namespace, ma è necessario definire un **Service Account**[3] con tutti i diritti sul cluster [79]. Fission utilizza le CRD per definire funzioni, trigger ed environment nativamente in Kubernetes, permettendo una maggiore integrazione con l'orchestratore.

Il Controller gestisce le invocazioni da CLI, ma non gli eventi generati da richieste HTTP. Il *bridge* tra eventi e funzioni è infatti il **Router**, componente stateless, facilmente scalabile in momenti di carico, che inoltra le richieste ai pod. In particolare, il Router controlla se esistono indirizzi in cache a cui inoltrare la richiesta di invocazione (*cache hit*), altrimenti (*cache miss*) chiede aiuto ad un altro componente, denominato Executor, il quale gli fornirà il nuovo indirizzo [80].

---

[3] Quando un utente accede a un cluster (es. tramite kubectl), questo viene autenticato tramite API Server usando un particolare User Account. Anche i processi interni ai pod contattano l'API Server autenticati con un particolare Service Account [101].

L'**Executor** è il responsabile per la generazione dei pod in cui vengono eseguite le funzioni. Quando il Router chiede all'Executor un indirizzo per inoltrare la richiesta, quest'ultimo analizza le informazioni della funzione e genera i relativi pod, restituendo l'indirizzo del servizio quando gli endpoint sono pronti a ricevere.

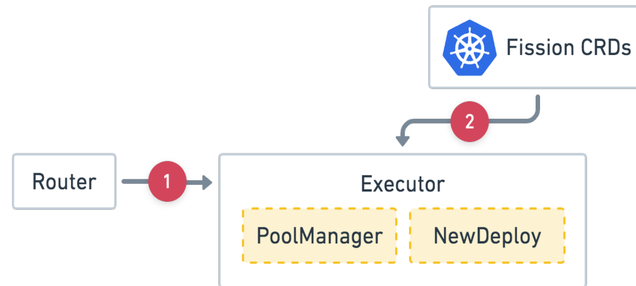

Figura 24 - Workflow di Fission

Fission offre due tipi di Executor, **PoolManager** e **NewDeploy** (Figura 24), e, date le specifiche di questo lavoro, è stata presa in considerazione la seconda tipologia, in quanto più adatta a gestire situazioni di carico. NewDeploy crea un Deployment, un Service e un HPA per ciascuna funzione: il Service realizza il disaccoppiamento rispetto ai relativi endpoint, permettendo di scalare fino ad un numero indefinito di pod, il Deployment gestisce le repliche per la tolleranza ai guasti ed assegna le risorse computazionali e HPA, infine, definisce le regole di autoscaling [81]. Tale struttura è ben visibile nella Figura 25.

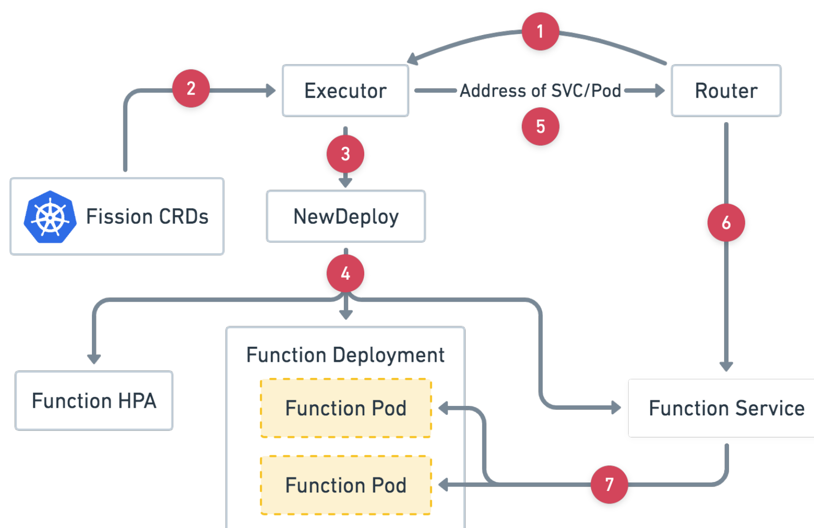

Figura 25 - Architettura di NewDeploy, Executor di Fission

Il **Function Pod** è la risorsa responsabile per l'esecuzione della business logic e della restituzione del risultato della funzione ed è composto da due container, **Fetcher** e **Environment Container** (Figura 26). Il primo cattura l'indirizzo della funzione e scarica l'archivio contenente codice e risorse della funzione dal componente **StorageSvc**; l'archivio viene poi copiato in un volume condiviso tra i due container (**Shared Volume**). Il Fetcher invoca l'Environment, il quale carica l'archivio dal volume condiviso e, infine, esegue la funzione [82].

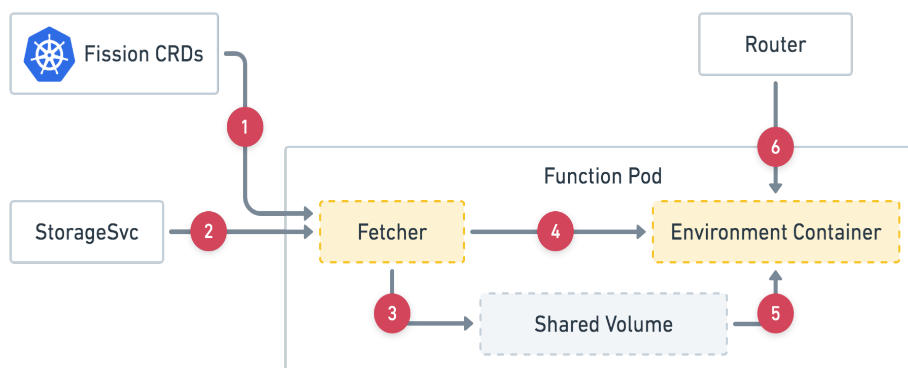

Figura 26 - Architettura di Function Pod

4.6.2 Utilizzo

Per il deployment è necessario, come prima cosa, istanziare un nuovo *environment* dove eseguire ed isolare la funzione. La gestione di Fission è completamente affidata al CLI **fission**, attraverso il quale è possibile creare l'ambiente scegliendo tra i diversi runtime resi disponibili dal framework, come ad esempio NodeJS:

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

Una volta generato l'ambiente, si può creare la funzione inserendo come argomenti l'ambiente appena generato, il nome, il file col codice (es. un file in formato *.js*) e il tipo di Executor (es. NewDeploy). Si possono inoltre definire il numero minimo e massimo di repliche desiderate:

```
$ fission fn create --name myFun --code myFun.js --env nodejs --
minscale 1 --maxscale 5 --executortype newdeploy
```

Infine, va configurato l’indirizzo per invocare la funzione tramite protocollo HTTP. Fission permette di definire un *path* per ogni funzione:

```
$ fission route create --function myFun --url /myFun
```

Il deployment è così completato ed è possibile invocare la funzione, tipicamente tramite richiesta HTTP, ma anche attraverso un semplice comando del CLI:

```
$ fission fn test --name myFun
```

4.6.3 Autoscaling

Come specificato precedentemente, al momento della creazione della funzione è possibile definire il numero minimo e massimo di repliche desiderate, sovrascrivendo i valori di default. Ovviamente, impostando il valore minimo a 1, si elimina il problema dei *cold start*, ma si rinuncia alla possibilità dello *scale-to-zero*. Fission si affida, come Knative, a Kubernetes per realizzare l’autoscaling tramite HPA, il quale sfrutta come parametro decisionale il consumo di RAM e/o la percentuale di utilizzo della CPU. Questo aspetto è definibile in fase di dichiarazione della funzione, utilizzando la sintassi di seguito riportata:

- `mincpu` - quantità minima di CPU da assegnare a un pod (misurata in *millicore* e con valore minimo pari a 1).
- `maxcpu` - quantità massima di CPU da assegnare a un pod (misurata in *millicore* e con valore minimo pari a 1).
- `minmemory` - quantità minima di memoria RAM da assegnare a un pod (misurata in Mb).
- `maxmemory` - quantità massima di memoria RAM da assegnare a un pod (misurata in Mb).
- `targetcpu` - la percentuale di CPU da raggiungere per poter scalare (a default pari a 80%).

Di conseguenza, durante la creazione della funzione devono essere aggiunti i seguenti parametri:

```
$ fission fn create --name hello --env node --code hello.js --
executortype newdeploy --mincpu 100 --maxcpu 200 --minmemory 64 --
maxmemory 128 --minscale 1 --maxscale 6 --targetcpu 50
```

Ogniqualvolta la percentuale di CPU utilizzata sfora il *target* sopra indicato (50%), viene creato un nuovo pod per meglio gestire l’ingente carico di richieste in entrata.

5 Comparazione delle piattaforme

In questa sezione si mira ad orientare la scelta di una tra le piattaforme analizzate, tenendo conto di tutti gli aspetti che le caratterizzano, non soltanto quelli quantitativi. Per questo motivo, una prima comparazione viene effettuata da un punto di vista prettamente qualitativo, basandosi su ciò che si è potuto osservare nella sezione precedente e sulla generale esperienza di uso di ciascuna delle piattaforme.

Alle considerazioni qualitative seguiranno le fasi di effettivo testing che invece dovranno valutare le performance a runtime dei framework. Per poter testare le soluzioni open source riportate, è però necessario capire quali aspetti misurare, in modo da generare report significativi che vadano a definire quali framework possano risultare adatti per usi e criticità industriali.
Per la valutazione delle piattaforme serverless bisogna affidarsi in prima battuta alla pratica del **load testing**. Il load testing è il processo tramite il quale si effettuano richieste a un servizio e se ne misurano le risposte. Esso viene utilizzato per determinare il comportamento di un sistema sottoposto a condizioni di carico normali o di picco. Tale processo aiuta a identificare la capacità operativa massima di un'applicazione, ma anche i suoi colli di bottiglia e criticità, in modo da poter identificare quali aspetti del sistema causano una maggiore degradazione delle performance e della disponibilità.
Quando il carico a cui è sottoposto un sistema viene aumentato oltre il normale utilizzo del servizio si parla invece di **stress testing**. Lo scopo di uno stress test è capire come si comporta il sistema in caso di situazioni critiche impreviste. Il carico generato in questi casi è così elevato che l'errore del sistema è un risultato atteso e cercato, per poi definire i limiti massimi dell'ambiente in questione.
Nei sistemi distribuiti, uno degli obiettivi primari di uno stress test è verificare la **scalabilità**, ovvero la dimensione che più interessa gli scenari di produzione, poiché essa è indispensabile per garantire stabilità dei servizi, elevata disponibilità, tolleranza ai guasti, e capacità di espansione aziendale futura.

Una volta sperimentato il comportamento delle piattaforme da un punto di vista della scalabilità, sarà infine necessario scendere più nel dettaglio del progetto che si vuole realizzare.
Per il momento, allo scopo di motivare le considerazioni fatte nei test presenti in questa sezione, si anticipa soltanto che l'obiettivo del progetto è la realizzazione di una struttura MapReduce composta da componenti serverless, per la generazione di documenti PDF in ambito FinTech. L'idea, come verrà ampiamente esposto più avanti, è quella di suddividere l'elaborazione delle singole pagine che compongono un documento tra più subtask paralleli, per poi concatenarle in un task finale. Bisognerà dunque tenere conto dei requisiti imposti sia dal modello MapReduce che dalla gestione di file PDF per capire

quali risorse esso va ad influenzare maggiormente, così da poter effettuare un ciclo di test progettato ad hoc per stressare quelle stesse risorse.

5.1 Considerazioni qualitative

Prima di procedere con l'osservazione dei test e dei risultati quantitativi, è doveroso aggiungere alcuni ragionamenti, più qualitativi, relativi alle singole piattaforme serverless. Nella Tabella 2 vengono evidenziati gli aspetti tenuti in considerazione durante la comparazione qualitativa dei middleware, indicando in colore verde i fattori prettamente positivi e in rosso quelli negativi.

	OpenFaaS	**OpenWhisk**	**Knative**	**Fission**
Necessità di service mesh	NO	NO	YES	NO
CLI	YES	YES	NO	YES
Autoscaling stand-alone	YES	YES	NO	NO
Upload su registro Docker	YES	NO	YES	NO
Vincolo su CPU	NO	YES	YES	NO
Vincolo su RAM	NO	YES	YES	NO
Vincolo su storage	NO	NO	YES	NO

Tabella 2 - Sintesi qualitativa delle piattaforme

Nella trattazione delle varie piattaforme, il primo aspetto considerato è stato l'architettura, valutata principalmente in base alla complessità, alla consistenza dei singoli componenti, al numero e al tipo di interazioni necessarie. Tutte le piattaforme dividono similmente le responsabilità tra i diversi componenti, portando alla luce un quadro significativo delle soluzioni serverless moderne che va verso una certa standardizzazione. In tutte le soluzioni, infatti, si trova un punto di accesso con cui interagire dall'esterno, un controllore che prende in carico le richieste ricevute e le smista ad altri componenti presenti, una serie di worker che eseguono le vere e proprie funzioni e un registro come livello di persistenza su cui salvare le funzioni. Inoltre, tutte le soluzioni offrono la

possibilità di essere agganciate a servizi di monitoraggio ausiliari, utilizzati per avvertire il controllore del sistema in caso di picchi di carico. Infine, in tutti i casi, per ottenere una semantica asincrona, è necessario e reso possibile affidarsi a broker di messaggi. Alla luce di ciò, per avere una visuale più astratta e di alto livello che possa facilitare la comprensione, viene riportata in Figura 27 una rappresentazione di una generica architettura FaaS.

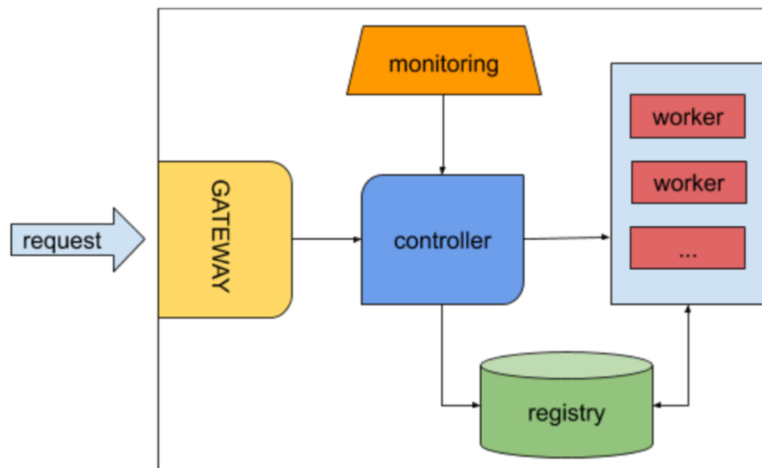

Figura 27 - Architettura di una generica piattaforma FaaS

Volendo valutare come queste architetture si integrano con il livello Kubernetes sottostante, bisogna ricordare che Knative e Fission implementano già la nuova semantica che definisce le risorse come risorse personalizzate di Kubernetes (CDR), permettendo quindi una migliore omogeneità e trasparenza delle piattaforme. OpenFaas si sta avvicinando al supporto di questa possibilità tramite l'introduzione di un nuovo tipo di controller chiamato *openfaas-operator*, mentre OpenWhisk non propone al momento soluzioni alternative.

Il secondo aspetto analizzato riguarda l'utilizzo delle piattaforme. Si sottolinea innanzitutto che tutte le installazioni sono risultate semplici grazie all'ausilio di **Helm** [84], package manager per Kubernetes che impacchetta le applicazioni nei cosiddetti "*chart*", fatta eccezione per Knative, in cui è stato necessario un passaggio addizionale per installare e configurare correttamente Istio. Inoltre, poiché Istio richiede la presenza di un load balancer, si è dovuto installarne uno per ambienti *bare metal*[4], ovvero **MetalLB** [85], che fornisce un indirizzo IP pubblico all'ingress gateway.
È doveroso menzionare che OpenWhisk, in fase di installazione, permette numerose

[4] Con "bare metal" si intende una situazione in cui i server non si trovano nel data center di un provider in condivisione con altri utenti, ma sono privati, single-tenant, e gestiti direttamente dall'utilizzatore.

possibilità di configurazione che portano la piattaforma ad essere la più versatile e personalizzabile, mentre negli altri framework le opzioni modificabili a disposizione sono molto più limitate.
Per quanto concerne la semplicità di gestione delle funzioni, Knative offre un meccanismo efficace, ponendosi come add-on ufficiale per il serverless su Kubernetes. Di conseguenza non richiede alcun tool esterno per creare, gestire e invocare le funzioni. Anche OpenFaas fornisce un'ottima esperienza utente, complice anche l'interfaccia web con cui è possibile controllare le funzioni (sostituibile al CLI **faas-cli**). Ciononostante, prima di poter esporre una funzione, entrambi i framework appena citati necessitano di più passaggi, dovendo interagire con una repository di immagini docker (operazioni di *build*, *push* e *deploy*).
OpenWhisk ha invece un meccanismo molto più semplice e immediato per il deployment di un'azione, che consiste in un unico file, contenente la sola funzione scritta nel linguaggio desiderato. Bisogna però obbligatoriamente utilizzare il CLI **wsk** per poterle creare e invocare.
Fission, infine, richiede passaggi addizionali alla creazione della funzione, dovuti alla necessità di fornire un environment e una route per questa. Anche in questo caso bisogna utilizzare il CLI proprietario **fission**.
Si aggiunge inoltre che, per tutte le piattaforme, le funzioni che si desiderano creare devono rispettare alcune restrizioni su signature e parametri di ingresso, necessarie a implementare le rispettive interfacce utilizzate dai framework.

La trattazione di ogni piattaforma si conclude con una considerazione sulle modalità con cui ogni framework riesce a scalare. OpenWhisk utilizza un autoscaler proprietario abilitato automaticamente e, pertanto, non necessita di operazioni aggiuntive, a meno di qualche aggiustamento nel file di configurazione per seguire le best practice indicate nella documentazione stessa.
OpenFaas è l'unico che offre sia un meccanismo proprietario, quello di default e utilizzato nei test riportati, sia la possibilità di affidarsi all'autoscaling HPA nativo di Kubernetes. Nel caso in cui si voglia utilizzare la seconda opzione, è necessario tuttavia istanziare un componente denominato **metrics-server**, un aggregatore di dati relativi al consumo delle risorse (CPU e RAM) da parte di ogni nodo e pod [86]. L'HPA di Kubernetes, infatti, non può, in generale, prescindere da questo componente che gli segnali quando è il momento di scalare. Una volta creato il metrics-server bisognerà impostare manualmente le richieste di CPU e/o RAM dei singoli container; dopo qualche minuto di attesa il metrics-server comincerà a raccogliere le informazioni necessarie, e, in caso di superamento dei valori soglia impostati, farà scaturire la creazione di nuovi pod.
Questo discorso vale anche per Fission, dove però l'HPA è l'unica opzione disponibile, facendolo risultare quindi come la piattaforma meno user-friendly per quanto riguarda la scalabilità.
Anche Knative, infine, si affida senza sorprese a Kubernetes per realizzare l'autoscaling.

Tuttavia, in questo caso la configurazione è maggiormente agevolata: risulta ancora indispensabile la presenza del metrics-server, ma questo riesce ad agganciarsi automaticamente ai servizi esposti, senza dover attendere che l'amministratore del sistema imposti manualmente le soglie.

Si conclude questa sezione citando i vincoli imposti ai nodi dalle piattaforme per poter essere installate correttamente. In questo caso Fission e OpenFaas non impongono una particolare disponibilità di risorse e si adattano quindi facilmente ad ambienti eterogenei, anche locali.
OpenWhisk richiede che i nodi siano equipaggiati con almeno due core e 4Gb di RAM e risulta quindi leggermente più restrittivo dei primi due, ma con vincoli ancora accettabili. Knative è, in assoluto, il più limitante, in quanto richiede nodi con almeno 6 core, 8Gb di RAM e 30Gb di memoria di massa, rendendolo inadatto alla maggior parte degli ambienti locali, come già espresso nelle sezioni precedenti.

5.2 Apache JMeter

Apache JMeter [87] è un software open source, scritto in Java, utilizzato per effettuare load test e stress test su una certa applicazione. Originariamente era stato pensato per valutare le performance di applicazioni web, ma attualmente può essere idealmente sfruttato per qualsiasi tipo di applicazione. Esso è stato selezionato tra i vari competitor in quanto risulta essere il framework di testing maggiormente usato e supportato negli ultimi anni.
Lo strumento fornisce un'interfaccia grafica molto intuitiva per configurare test di performance su risorse statiche o dinamiche, in modo da lanciare carichi di richieste su un server, su un gruppo di server, su una rete o su un generico oggetto accessibile dall'esterno. JMeter utilizza tutti i più noti protocolli, quali ad esempio HTTP, LDAP, FTP, JDBC, SMTP, e consente di creare molti thread in parallelo per simulare la presenza di un certo numero di clienti che, concorrentemente, effettuano richieste o altre operazioni. Il completo workflow messo in moto da JMeter viene descritto nella Figura 28 e presenta una serie di poche ma efficaci operazioni.

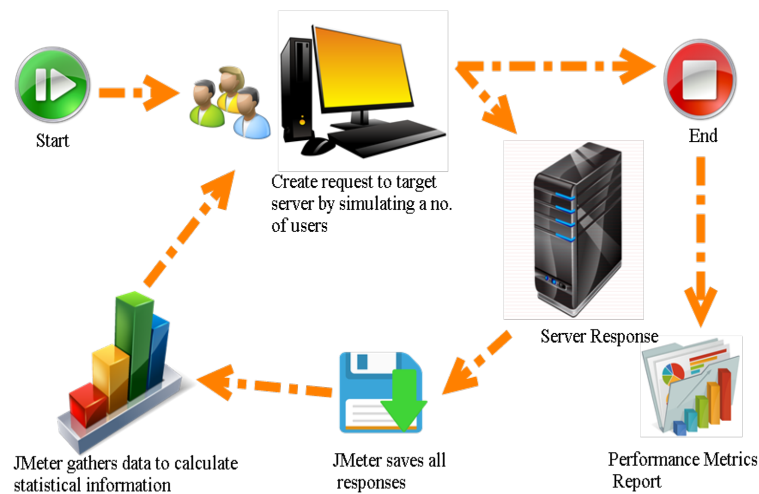

Figura 28 - Il workflow di JMeter

Una volta avviato il test, sul lato cliente vengono generati un numero di utenti fittizi pari alla quantità specificata in fase di configurazione. Ognuno di questi corrisponde a un thread indipendente che a sua volta genera e invia richieste al server da testare. JMeter salva le risposte del server, per poi calcolare e collezionare le informazioni statistiche relative. Al completamento del test, le osservazioni raccolte vengono trasposte in un report finale visualizzabile in formato testuale, numerico o attraverso grafici cartesiani.

Un'altra interessante feature riguarda la capacità di aggiungere degli script funzionali per manipolare dinamicamente i dati che vengono considerati durante i test. È inoltre possibile agganciare dei listener, così da poter avere in tempo reale dati statistici sui test in esecuzione, senza dover attendere la terminazione della simulazione per capire come il sistema si sta comportando.
È inoltre importante comprendere che JMeter non funziona come un browser automatizzato, ma lavora direttamente a livello di protocollo, e, di conseguenza, non è in grado di eseguire del codice Javascript trovato all'interno della pagina HTTP richiesta, né di renderizzare la pagina stessa.

Come già detto, JMeter fornisce una pratica GUI per comporre, avviare, monitorare e stoppare i test. Tuttavia, il meccanismo di default costringe a delegare su un unico nodo, quello su cui viene aperta l'interfaccia grafica, tutto il carico generato dalla creazione dei thread JMeter e dalla raccolta delle informazioni. Questo potrebbe causare un degradamento nelle performance di tale nodo e risultati falsati del test, in quanto questi andrebbero a dipendere non solo dalla capacità del server nel rispondere, ma anche dalla capacità del client JMeter di effettuare un numero elevato di richieste. Fortunatamente JMeter consente di far partire un test anche in modo distribuito, scaricando il carico su più nodi client gestiti da un unico nodo master. La Figura 29 mostra il concept di un load test distribuito su un cluster Kubernetes. L'idea è quella di avere un nodo centrale

(**JMeter master**) che invia script di test ad altri nodi (**JMeter slave**), i quali, a loro volta, eseguiranno i test. Con questa architettura distribuita è possibile realizzare un test di carico intenso, simulando centinaia o migliaia di utenti che effettuano richieste simultaneamente, senza preoccuparsi del carico a cui sono sottoposti i nodi client, che, in questo caso, risulterà essere sufficientemente ridimensionato e bilanciato.

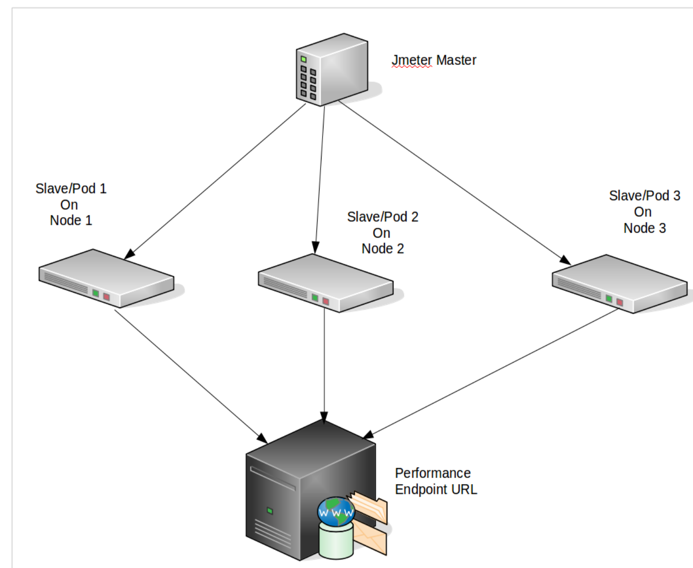

Figura 29 - Architettura di un test JMeter distribuito

5.3 Test di scalabilità

Per essere il più equi possibile, ognuna delle piattaforme da confrontare espone un servizio serverless che consiste in un'unica funzione NodeJS, che restituisce una semplice stringa testuale a ogni invocazione. Lo scopo è infatti capire il comportamento dei vari framework a fronte di un elevato carico di richieste, senza rendere l'esecuzione stessa delle funzioni un collo di bottiglia. Le richieste vengono inviate tramite protocollo HTTP utilizzando le funzionalità offerte da JMeter.

Il test è composto come segue:

- **Thread Group** - specifica quanti thread (*users*) creare e in quanto tempo (*ramp up period*), quante richieste ognuno di questi dovrà inviare (*loop count*) e dopo quanto tempo dovrà terminare il test (*lifetime duration*).
- **Loop Controller** - permette di definire quante volte ripetere l'esecuzione degli elementi al suo interno; è simile al loop count intrinseco del thread group, ma è preferibile ad esso in quanto conduce a un utilizzo più efficiente delle risorse di rete, limitando in particolare l'apertura di un eccessivo numero di porte.

- **Constant Timer** - deve la sua introduzione al fatto che ogni thread invia richieste in cascata una dopo l'altra e, teoricamente, potrebbe farlo il più velocemente possibile, facendo partire una nuova richiesta non appena finisce di inviare quella precedente; tuttavia, questo comporterebbe un carico eccessivo, sia per il client che per il server, la generazione di uno scenario poco realistico e quindi un'inutile degradazione delle performance; a tale scopo viene utilizzato un temporizzatore, il *constant timer* appunto, che specifica quanto tempo deve trascorrere tra due richieste successive di uno stesso thread.
- **HTTP Request** - concretizza i dettagli della richiesta HTTP da inviare a ogni ciclo e per ogni thread; in particolare si usa per indicare il metodo HTTP da utilizzare (es. GET o POST), l'indirizzo IP, la porta e il path ai quali si trova la funzione di destinazione da invocare; è necessario in questo caso specificare l'utilizzo dell'header di *keep-alive* per evitare di instaurare troppe connessioni e occupare tutte le porte disponibili.

Il primo di questi componenti è quello che maggiormente è stato variato durante le varie sessioni di testing. In particolare, l'approccio è stato quello di aumentare gradualmente il numero di utenti nel thread group per capire il tipo di carico che una specifica piattaforma è in grado di sostenere. Il loop controller è stato impostato su "infinito", così da far effettuare ai thread un numero illimitato di richieste fino allo scadere dei test, lasciando che il sistema serverless venisse messo sotto condizioni di vero stress. Per quanto riguarda l'assegnamento del delay tra la generazione di due richieste successive, il constant timer è stato impostato a *400ms*, avendo osservato, con tale valore, una buona stabilità generale, senza sovraccarichi lato cliente.

La visualizzazione dei risultati del test è affidata a un listener agganciabile al test e chiamato **summary report**. Esso si presenta come un'interfaccia grafica tabellare, contenente informazioni relative a:

- *Samples* - numero di richieste inviate.
- *Min* - tempo minimo di risposta.
- *Max* - tempo massimo di risposta.
- *Average* - tempo medio di risposta.
- *Standard deviation* - variazione rispetto alla media.
- *Errors* - percentuale di richieste fallite.
- *Throughput* - numero di richieste gestite al secondo dal server.

Tra queste verrà posta particolare attenzione al tempo medio di risposta, al throughput e alla percentuale di errori, in quanto i tempi minimi e massimi di risposta, così come la deviazione standard, sono eccessivamente influenzati da fluttuazioni e fenomeni aleatori, restituendo una valutazione del comportamento del sistema meno oggettiva e attendibile rispetto al tempo medio.

Il numero di utenti scelto è stato ponderato in base alle caratteristiche di scalabilità di ogni framework. In particolare, sono stati simulati rispettivamente 10, 50, 100, 500 e 1000 utenti concorrenti, di cui ognuno effettua una nuova richiesta ogni 400 millisecondi per l'intera durata del test. Tale durata è stata scelta in base al tempo necessario per generare tutti i thread. Nei primi tre casi, poiché il test risultava essere troppo breve per una corretta valutazione, la durata della simulazione è stata impostata a 600 secondi (10 minuti). Il test con 500 utenti è stato impostato per durare 1000 secondi (circa 15 minuti), mentre il test con 1000 utenti è stato fatto terminare dopo 2000 secondi (circa mezz'ora). Così facendo, si è avuta una visione delle performance sia in transitorio che a regime, dando tempo al framework di scalare correttamente i pod e simulando effettivamente uno scenario di produzione. Con i valori selezionati si arriva alla generazione di un numero di richieste con incremento sempre maggiore all'aumentare degli utenti. In particolare:

- Test con 10 utenti - invio di circa 15000 richieste in 10 minuti.
- Test con 50 utenti - invio di circa 70000 richieste in 10 minuti.
- Test con 100 utenti - invio di circa 150000 richieste in 10 minuti.
- Test con 500 utenti - invio di circa 650000 richieste in 15 minuti.
- Test con 1000 utenti - invio di circa 2500000 richieste in 30 minuti.

Di fronte alla possibilità di aumentare ulteriormente il numero di richieste da effettuare ci si è resi conto che i risultati non sarebbero stati altrettanto indicativi e interessanti. Difatti si è notato che, rimanendo nello stesso ordine di grandezza degli utenti da generare, le differenze nei risultati dei test sarebbero state marginali. Differenze significative sarebbero piuttosto emerse facendo partire almeno 10000 utenti, provocando l'invio di circa 250 milioni di richieste, valore reputato eccessivo sia per le risorse a disposizione che per lo scope della ricerca. I comportamenti delle diverse piattaforme sono infatti già ben delineabili mantenendosi sull'ordine di 10^3.

Il cluster su cui sono stati effettuati i test è composto da cinque VM, una per il nodo master e quattro per i nodi worker. Una sesta macchina, identica alle prime, è stata utilizzata come JMeter client, per la generazione delle richieste. I test sul cluster sono stati affiancati da una serie di test effettuati in un ambiente locale, su un'unica macchina, per comparare le performance nei due scenari. Nella Tabella 3 sono indicate le specifiche di entrambi gli ambienti.

	OS	CPU	RAM
cluster	Ubuntu Server 18.04	5x 8 core	5x 16Gb
locale	Docker Desktop for Windows	1x 2 core	1x 4Gb

Tabella 3 - Schede tecniche degli ambienti

5.3.1 Risultati sul cluster

I grafici riportati di seguito (Figura 30-34) mostrano i risultati ottenuti nei vari test effettuati sul cluster, considerando i valori di **throughput** (alto è meglio), **tempo medio di risposta** (basso è meglio) e **percentuale di errori** (bassa è meglio).

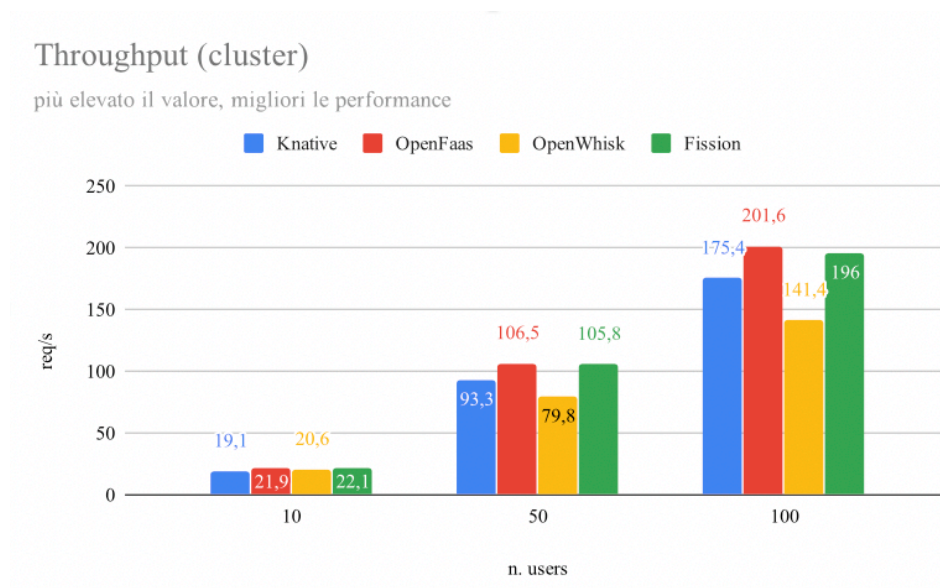

Figura 30 - Throughput, test sul cluster con 10, 50 e 100 utenti

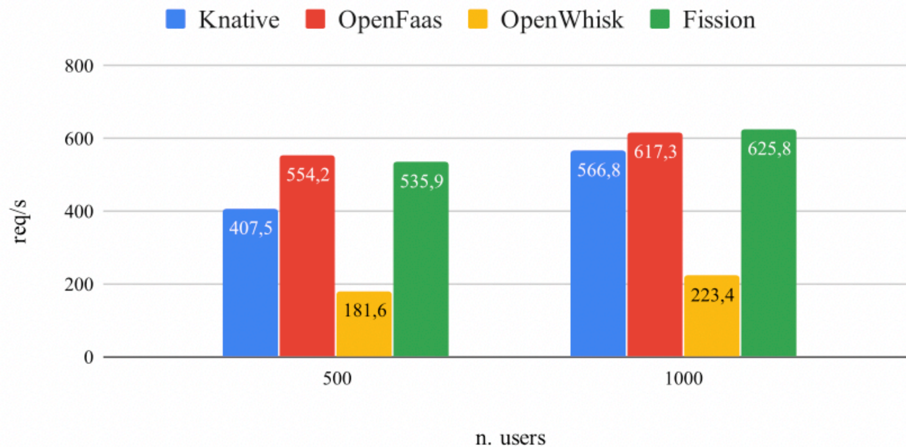

Figura 31 - Throughput, test sul cluster con 500 e 1000 utenti

Per quanto riguarda il throughput, le differenze si evidenziano man mano che si aumenta il numero di utenti, in particolar modo per OpenWhisk, che presenta evidenti problemi di gestione di elevati carichi. Il massimo throughput raggiunto da quest'ultimo è infatti quasi un terzo di quello dei suoi competitor. OpenFaas e Fission si pongono invece come i framework più efficienti nello smaltire le richieste in ingresso, con performance quasi alla pari. Knative occupa una posizione intermedia, ottenendo risultati accettabili in tutte le simulazioni. L'ottima scalabilità ed elasticità dei tre framework migliori permette di raggiungere circa 600 richieste gestite al secondo, nel caso di carico maggiore.

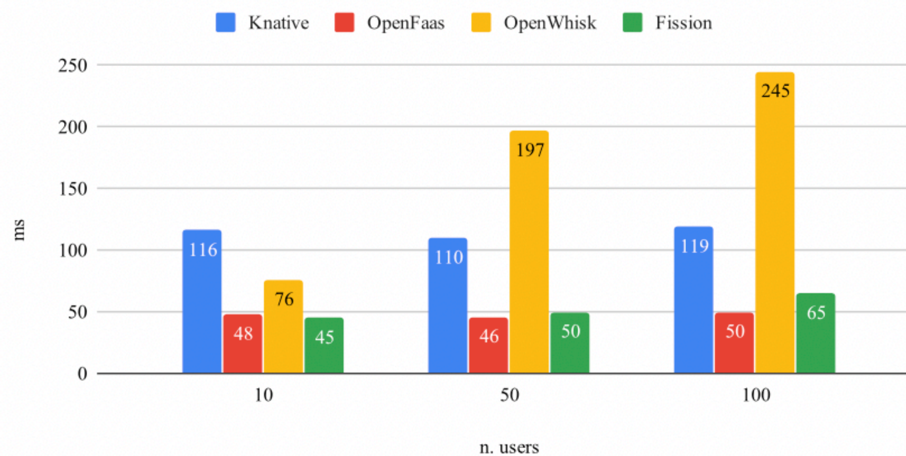

Figura 32 - Tempo medio di risposta, test sul cluster con 10, 50 e 100 utenti

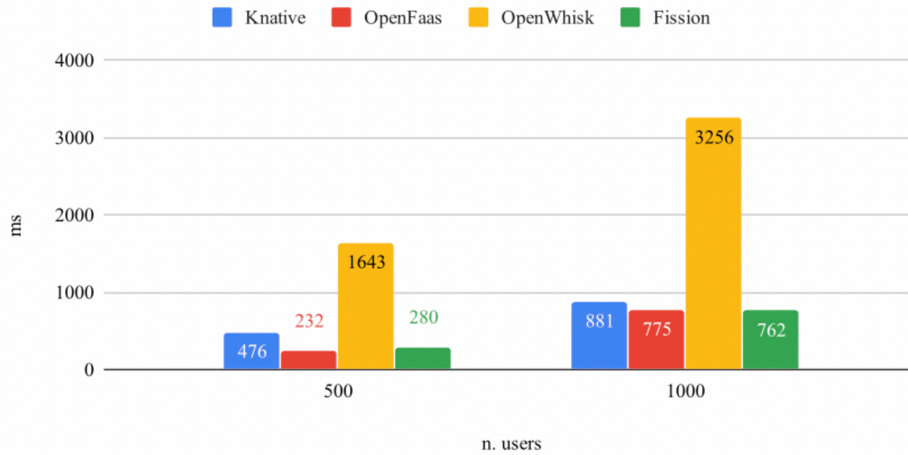

Figura 33 - Tempo medio di risposta, test sul cluster con 500 e 1000 utenti

Anche nell'ambito del tempo medio registrato nel servire una richiesta, OpenWhisk si pone come candidata peggiore. In questo caso la sua inferiorità è ben visibile già nel terzo test da 125000 richieste (100 utenti), fino a toccare i 3 secondi di attesa media nel caso più complesso con 1000 utenti concorrenti. Va comunque sottolineato che il meccanismo a scambio di messaggi, insito in OpenWhisk, dona al framework una forte asincronicità, che va a compensare questi valori. Differenze marginali tra i restanti framework, che mostrano ottime performance e tempi di risposta proporzionati e ragionevoli, che si attestano sempre al di sotto del secondo.

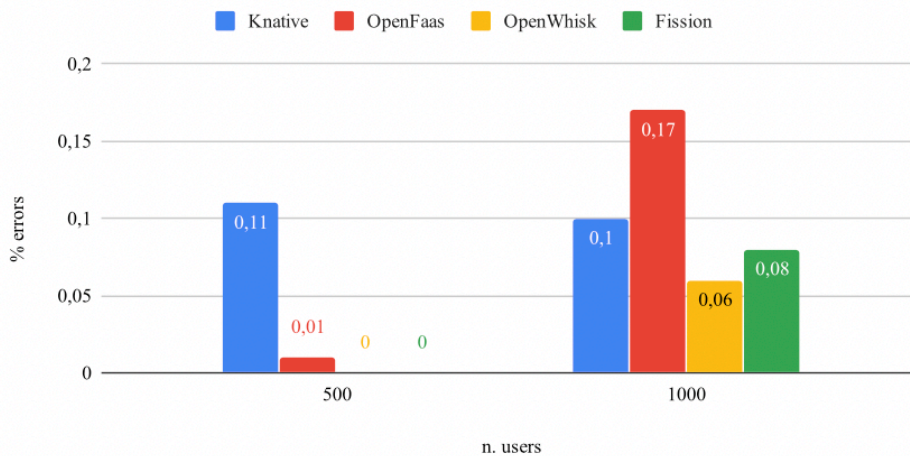

Figura 34 - Percentuale di errori, test sul cluster con 500 e 1000 utenti

In questo caso sono state riportate soltanto le percentuali di errore relative ai test rispettivamente con 500 e 1000 utenti, in quanto, nei test con un numero di utenti inferiore, tutte le piattaforme hanno terminato l'esecuzione con lo 0% di errori e quindi servendo correttamente la totalità delle richieste. Su questo piano, OpenWhisk risulta essere il più affidabile, con percentuali di errori nulle o tendenti allo zero. Knative è l'unico a presentare un andamento non proporzionale, riuscendo ad ottenere una percentuale di errori minore all'aumentare del numero di utenti e superando perfino l'ottima affidabilità offerta da OpenFaas. Infine, anche Fission, insieme a OpenWhisk, è stato in grado di terminare senza alcun errore il test con 500 utenti, ottenendo un buon risultato anche nell'ultimo test. In generale tutti gli errori riportati sono stati causati da un timeout della richiesta HTTP, tipicamente dovuto all'impossibilità di rispondere in modo reattivo durante la fase transitoria di creazione di nuovi pod per far fronte alle richieste in arrivo.

5.3.2 Risultati in locale

In questa sezione sono presentati i risultati dei test effettuati in ambiente locale (Figura 35-36). Bisogna precisare che non è stato possibile valutare la piattaforma di Knative in quanto essa non è installabile in un ambiente così limitato. Essa necessita infatti di un vero e proprio cluster di più nodi e disponibilità di CPU e RAM ben al di sopra di quelle fornite da una singola macchina locale. Inoltre, data l'esigua disponibilità di risorse, non è stato possibile completare gli ultimi due test, rispettivamente da 500 e 1000 utenti, poiché causavano una saturazione del sistema e una quantità di errori tale da rendere insignificanti i risultati.

Alla luce di ciò, i grafici a seguire mostrano soltanto i risultati dei framework OpenWhisk, OpenFaas e Fission, a fronte dei test con 10, 50 e 100 utenti concorrenti. Anche in questo caso saranno omessi i risultati in termini di percentuale di errori, in quanto nessun errore è stato rilevato fintanto che si è rimasti entro la soglia dei 100 utenti.

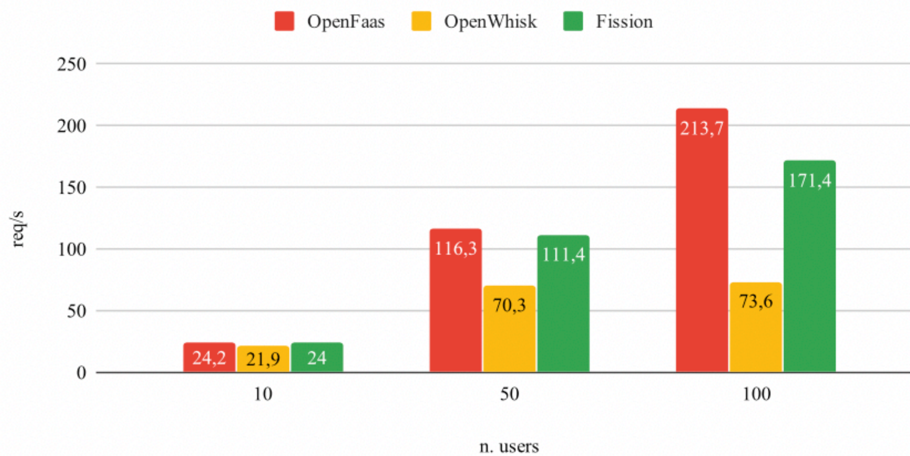

Figura 35 - Throughput, test in locale con 10, 50 e 100 utenti

Per valori ridotti di utenti, le performance in locale sono comparabili a quelle nel cluster distribuito. Ciò è dovuto al fatto che, per un numero limitato di richieste, non è necessaria un'elevata scalabilità e dunque un singolo nodo risulta sufficiente a gestire adeguatamente il carico. È interessante notare come, anche in questo scenario ridimensionato, OpenWhisk faccia fatica a stare al passo con i suoi competitor, riportando valori di throughput inferiori in tutti e tre i casi, seppure non particolarmente critici. OpenFaas mostra invece la sua potenza anche in questo contesto, con Fission immediatamente al seguito.

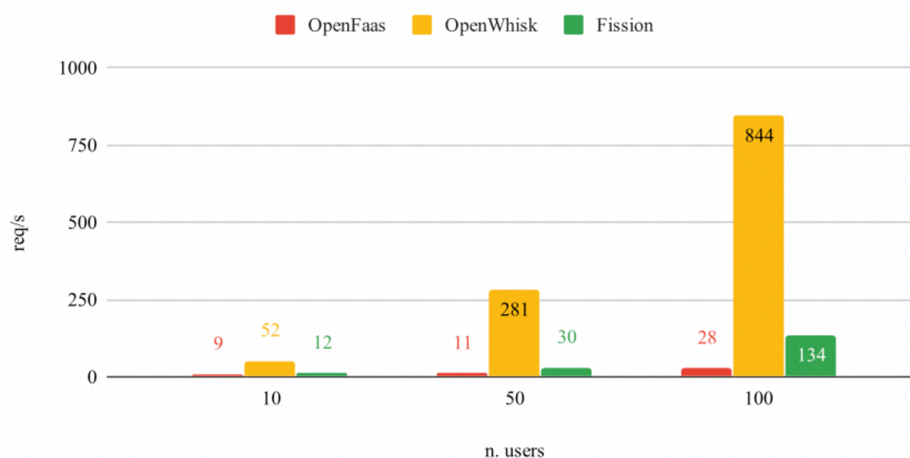

Figura 36 - Tempo medio di risposta, test in locale con 10, 50 e 100 utenti

L’assenza del networking esterno permette invece ai tempi di risposta di essere chiaramente inferiori in locale rispetto a quelli registrati sul cluster. Solita eccezione fatta per OpenWhisk, che, in questo caso, non potendo contare sul supporto di altri nodi su cui scalare, trova una notevole difficoltà nel servire rapidamente le richieste. Si riporta infatti un tempo di attesa medio di poco meno di un secondo, nonostante ci si trovi di fronte a soli 100 utenti concorrenti.

In merito ai risultati tendenzialmente negativi osservati con OpenWhisk, sia nel cluster che in locale, va tuttavia considerato che, a causa dei limiti imposti dalle risorse a disposizione per sviluppo e test, non è stato possibile seguire l’indicazione riportata nella documentazione di *Apache Kafka*, che suggerisce di istanziare il servizio di eventi su un nodo dotato di almeno 32Gb di RAM, se si vogliono sfruttare a pieno le sue potenzialità. È quindi plausibile che la piattaforma in questione sia stata negativamente influenzata da questo dettaglio.

5.4 Test sul consumo delle risorse

Il secondo ciclo di test ha avuto come obiettivo l’osservazione delle piattaforme sottoposte a operazioni che insistono sulla stessa tipologia di risorse messa sotto stress nel caso d’uso. Per prima cosa è stato necessario dunque valutare quale parte del sistema subisce il maggiore impatto durante l’elaborazione di documenti PDF.
Si è partiti da un’analisi del problema, in cui sono state analizzate le istruzioni da eseguire, ipotizzando se si trattasse di istruzioni *cpu-bound* o *memory-bound*. Con “cpu-bound” si intende quell’insieme di funzioni per le quali il tempo necessario ad eseguirle dipende principalmente dalla velocità del processore, come ad esempio il calcolo del fattoriale di un numero di grandi dimensioni. Queste si contrappongono alle funzioni “memory-bound” in cui il collo di bottiglia è invece la memoria necessaria a mantenere i dati utilizzati, come ad esempio le operazioni su una matrice con un numero particolarmente elevato di righe e/o colonne.

Nel caso in esame di compilazione e concatenazione di documenti PDF non sono presenti operazioni algebriche computazionalmente pesanti, essendo i dati da utilizzare già pronti e forniti in partenza, mentre senz’altro bisogna considerare che lavorare su dei file significa aprire e modellare stream di byte, di lunghezze anche significative, che vanno mantenuti in memoria fino al termine delle operazioni. Sebbene con le infrastrutture hardware a disposizione oggi la generazione di un PDF non risulta essere particolarmente onerosa, se si pensa a uno scenario industriale, in cui centinaia o migliaia di utenti effettuano contemporaneamente la relativa richiesta, la reazione del sistema in termini di risorse diventa un punto su cui è giusto focalizzarsi. Per dare un razionale non empirico a queste considerazioni, si è monitorato l’utilizzo delle risorse in un semplice ambiente locale durante l’elaborazione di 200 pagine PDF in parallelo ($\cong$ 1Mb per pagina). I

risultati hanno indicato un aumento della memoria RAM utilizzata di circa 1Gb, a fronte di un innalzamento di utilizzo della CPU del solo 4%. È dunque evidente come l'operazione in questione rientri nella categoria **memory-bound** e che bisogna quindi verificare le performance delle piattaforme dal punto di vista dell'utilizzo della memoria.

Per simulare questo stesso tipo di carico sulla memoria, è stata utilizzata una funzione "mock" in Java che si occupa di effettuare la **generazione di matrici** di grandi dimensioni, un problema rinomatamente memory-bound. In particolare, alcuni esperimenti atti a calibrare al meglio l'impatto che questa provocava sul sistema, hanno portato alla definizione di una funzione che genera matrici quadrate con 10000x10000 valori numerici in formato *double* (64Bit). Utilizzando questi valori si è raggiunta un'occupazione di memoria molto vicina a quella allocata per i 200 documenti PDF sopra menzionati e utilizzati come metrica esemplificativa, così da considerare il Gb come unità di misura.

I passaggi successivi sono consistiti nel deployment di questa funzione su ciascuna delle piattaforme FaaS e nel monitoraggio della loro esecuzione, in diverse tipologie di test, riassumibili come segue:

1) Consumo di RAM a fronte di una singola invocazione (≅ 200 pagine PDF) per matrice 10000x10000.
2) Consumo di RAM a fronte di rispettivamente 10, 20, 30, 40, 50 e 100 invocazioni concorrenti, analoghe alla precedente, caratterizzate da una frequenza di una richiesta inviata al secondo.
3) Consumo di RAM con test analoghi al caso 2, ma con la richiesta di generare una matrice di dimensioni maggiori (15000x15000, corrispondente a un'occupazione di circa 2Gb di RAM).
4) Consumo di RAM a fronte di 50 invocazioni, andando però a modificare il numero di richieste inviate al secondo.

In questo caso non è stato effettuato il test in ambiente locale, data l'inadeguatezza del sistema nel gestire l'ingente quantità di memoria che sarebbe stata occupata, pertanto i risultati che seguono sono quelli conseguiti sul solo cluster, configurato analogamente al test precedente (Tabella 3). Le funzioni sono state replicate su tutti e cinque i nodi del cluster di sviluppo a disposizione, per poter usufruire della parallelizzazione, mentre le richieste sono state effettuate ancora una volta avvalendosi di *Apache JMeter*.

5.4.1 Risultati test matrice 10000x10000

Per quanto riguarda il primo test, quello da una sola invocazione, i risultati vengono qui omessi in quanto omogenei tra di loro e perfettamente in linea con quanto osservato in ambiente locale, con un'occupazione di circa 1Gb di RAM in tutti i casi. Si omettono inoltre i valori relativi alla CPU, poiché la percentuale di utilizzo di questa è rimasta, come ci si aspettava, sostanzialmente bassa durante tutta la durata dei test, con piccoli e non rilevanti picchi soltanto in fase di inizializzazione delle varie funzioni.

Il primo confronto viene fatto presentando, per ogni piattaforma, i risultati ottenuti nei casi con richieste di generazione della matrice 10000x10000, riportando una rappresentazione grafica che riassume l'intero spettro dei test eseguiti in questa prima fase, per mostrare la variazione del consumo di memoria all'aumentare graduale del numero delle richieste, da 10 a 100 (Figura 37-40). Ciascuno dei grafici che seguono cattura un'osservazione che parte dall'inizio del test e che prosegue per i 10 minuti successivi, tempo sufficiente a capire come una piattaforma gestisce l'ammontare di memoria allocata. I valori sono stati campionati utilizzando lo strumento *Prometheus*, estrapolando la quantità di memoria richiesta da tutti e soli i componenti di ogni framework, senza considerare dunque eventuali carichi supplementari presenti sui nodi del cluster a causa di componenti esterni alle FaaS. La frequenza di campionamento è stata fissata a 15 secondi, mentre nei grafici, sull'asse delle ascisse, l'unità di misura del tempo è di 30 secondi, per una migliore leggibilità. Sull'asse delle ordinate si trova invece la quantità di memoria RAM allocata, misurata in Gb.

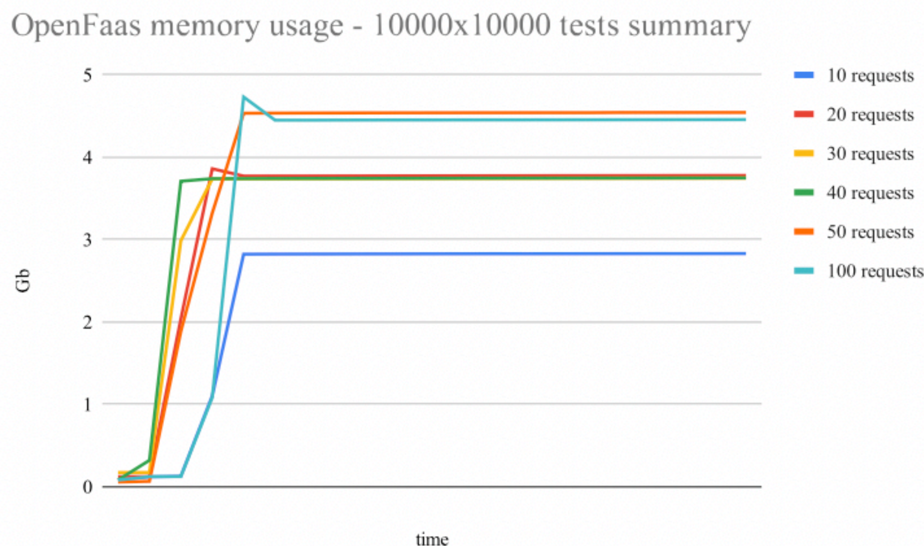

Figura 37 - Risultati memory-bound OpenFaas con matrice 10000x10000

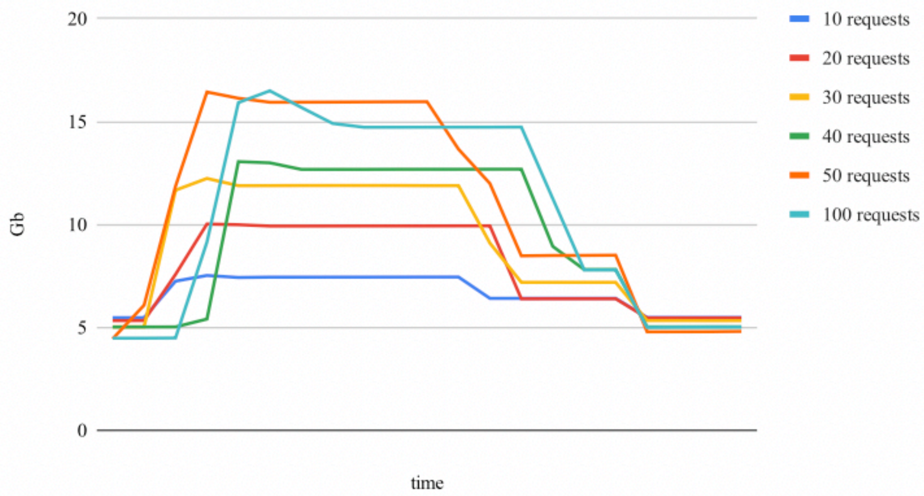

Figura 38 - Risultati memory-bound OpenWhisk con matrice 10000x10000

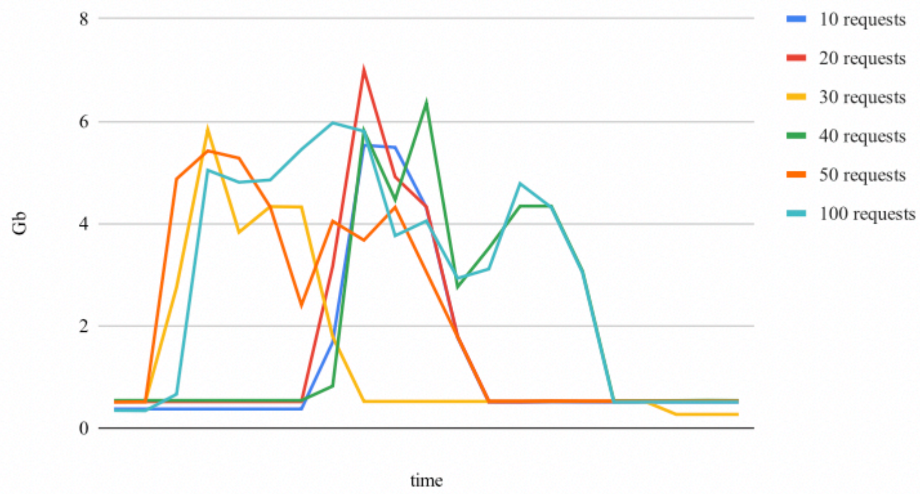

Figura 39 - Risultati memory-bound Knative con matrice 10000x10000

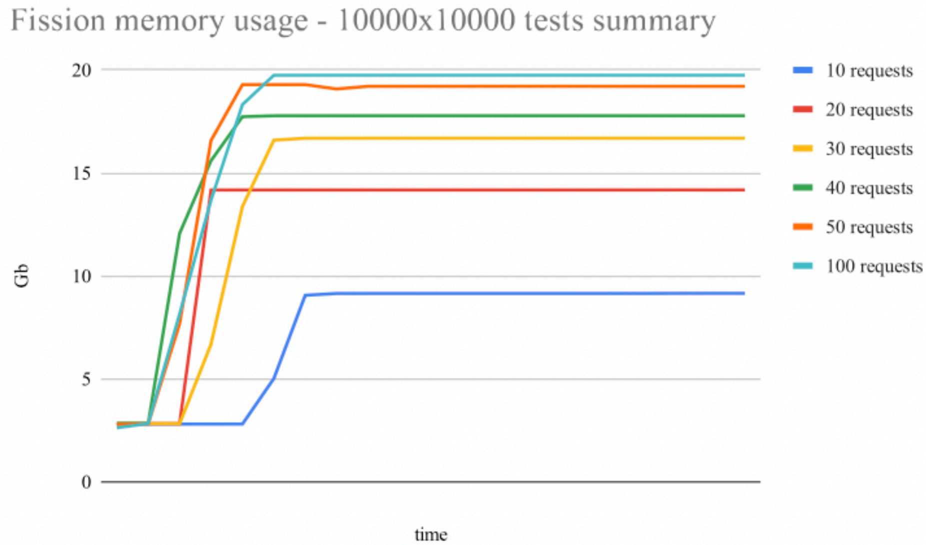

Figura 40 - Risultati memory-bound Fission con matrice 10000x10000

Dividendo idealmente i grafici presentati in tre aree, è possibile effettuarne un'analisi più approfondita, così da osservare gli andamenti registrati nelle tre fasi salienti, ovvero prima che cominci il test (fase **idle**), al raggiungimento del picco massimo durante l'esecuzione a regime delle funzioni (fase **peek**) e nei minuti immediatamente successivi alla terminazione del test (fase **release**).

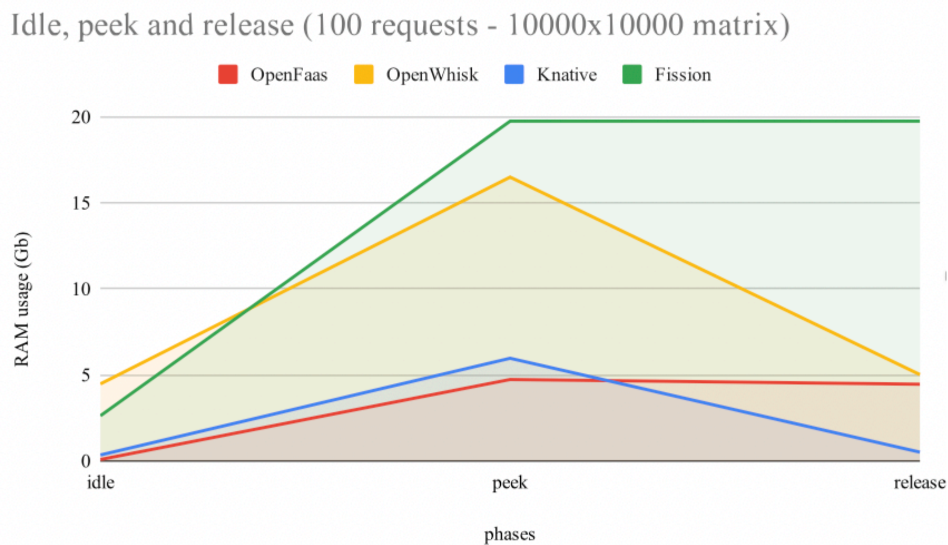

Figura 41 - Confronto fasi di esecuzione con 100 richieste per matrice 10000x10000

Osservando il grafico relativo alle 100 richieste in Figura 41, partendo da sinistra, la quantità di memoria occupata da ogni framework in assenza di esecuzioni di funzioni è, in generale, relativamente bassa, seppure OpenWhisk si mantenga stabile intorno ai 5Gb per permettere il funzionamento degli altri componenti della sua architettura, quali ad esempio Kafka, Zookeeper e CouchDB, che evidentemente necessitano più risorse rispetto ai competitor. A seguire si trova Fission, con un consumo medio in idle di circa 3Gb, mentre si equivalgono gli andamenti di OpenFaas e Knative, capaci di rimanere al di sotto della soglia del Gb, quando non utilizzati (*scale-to-zero*).
L'aspetto più interessante riguarda invece il picco massimo raggiunto, nel momento di massimo grado di concorrenza delle richieste. OpenFaas e Knative riescono ad ottenere i valori migliori, intorno ai 5Gb, grazie alla loro capacità di rispondere in breve tempo a ognuna delle richieste in arrivo, cosa che gli evita di gestire contemporaneamente un numero di richieste troppo elevato: si ricorda infatti che in questa prima fase di test le richieste vengono inviate con una frequenza di una richiesta al secondo, per poi fare un confronto con i risultati ottenuti successivamente, all'aumentare di tale frequenza. OpenWhisk e Fission seguono con valori significativamente maggiori, che si aggirano rispettivamente sui 16Gb e 20Gb. Tuttavia, se non si considera il consumo a riposo di 5Gb della piattaforma OpenWhisk, l'aumento di memoria generato dall'effettiva esecuzione delle funzioni risulta essere più proporzionato a quello dei competitor, con un incremento massimo che si aggira intorno agli 11Gb, nel caso di 100 richieste.
Per quanto riguarda la fase finale, in cui le risorse potrebbero effettivamente essere rilasciate, Knative e OpenWhisk sono gli unici a effettuare una deallocazione, seguendo difatti il paradigma FaaS secondo il quale le risorse vadano utilizzate soltanto per la durata di tempo necessaria. Le restanti due piattaforme, invece, seguono un approccio molto più conservativo, che consente certamente una risposta più pronta in caso di richieste successive, ma che, in caso contrario, porta a un'occupazione eccessiva delle risorse, senza che queste vengano effettivamente sfruttate. È da segnalare un particolare comportamento anomalo nelle piattaforme Fission e OpenFaas, per le quali è spesso necessario eliminare manualmente le istanze messe in esecuzione pur di rilasciare le risorse.
Come ultima considerazione sul punto di rilascio, è doveroso aggiungere che è stato portato avanti un secondo ciclo di test col fine di comprendere se le risorse mantenute a termine esecuzione da OpenFaas e Fission venissero poi riutilizzate in caso di una seconda serie di richieste concorrenti, immediatamente successiva alla prima. I risultati hanno evidenziato che un riciclo delle risorse viene effettuato, ma soltanto parziale.
Nell'esempio riportato in Figura 42, OpenFaas, rimasto fisso tra i 4Gb e i 5Gb di memoria occupata dalla fine del test precedente, è stato sottoposto a una seconda ondata di richieste identiche, senza attendere che questo rilasciasse le risorse, per vedere se fosse stato in grado di riutilizzare tutte e sole le risorse precedentemente allocate. Come si può notare dal grafico, una parte della memoria viene reimpiegata, tuttavia il framework va ad

allocare ulteriore spazio fino a superare i 7Gb, per far fronte alle stesse richieste che un attimo prima aveva gestito con soli 4Gb. Comportamenti analoghi si sono verificati in Fission. Una tale situazione fa chiaramente allontanare i due framework dall'ipotesi di selezionarli per un eventuale uso applicativo.

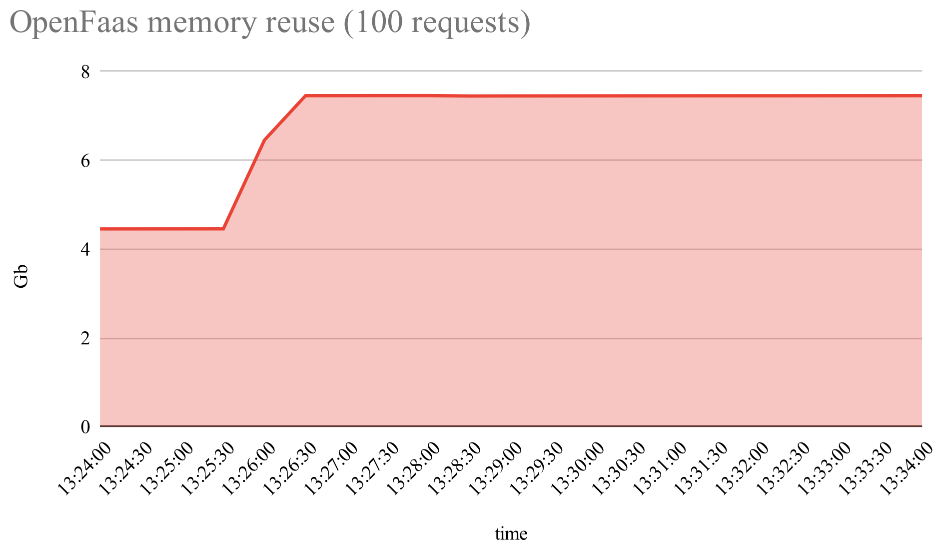

Figura 42 - Test sul riutilizzo della memoria in OpenFaas

5.4.2 Risultati test matrice 15000x15000

La seconda parte di questa fase di test ha previsto l'aumento nel numero di righe e colonne della matrice da generare, passando da 10000x10000 a 15000x15000 per vedere se le piattaforme avrebbero mantenuto un comportamento lineare o meno. In proporzione, la generazione di questa matrice va ad occupare intorno ai 2Gb di RAM, corrispondente all'elaborazione di circa 400 pagine PDF nel caso d'uso in esame. Il risultato viene riassunto nel grafico in Figura 44, prendendo come riferimento il test mediano eseguito con 50 richieste. Anche in questo caso, per avere una visione più chiara degli andamenti, viene presentata una grafica che divide il test nei tre momenti di riposo, picco massimo e rilascio. I risultati vengono inoltre preceduti da quelli ottenuti con il medesimo numero di richieste, ma nel caso con la matrice di dimensioni minori (Figura 43) per consentire un confronto più agevole.

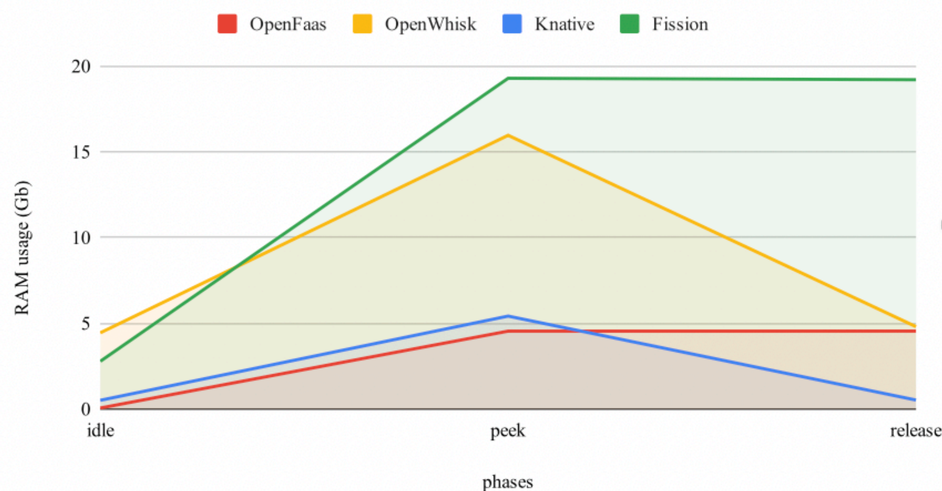

Figura 43 - Confronto fasi di esecuzione con 50 richieste per matrice 10000x10000

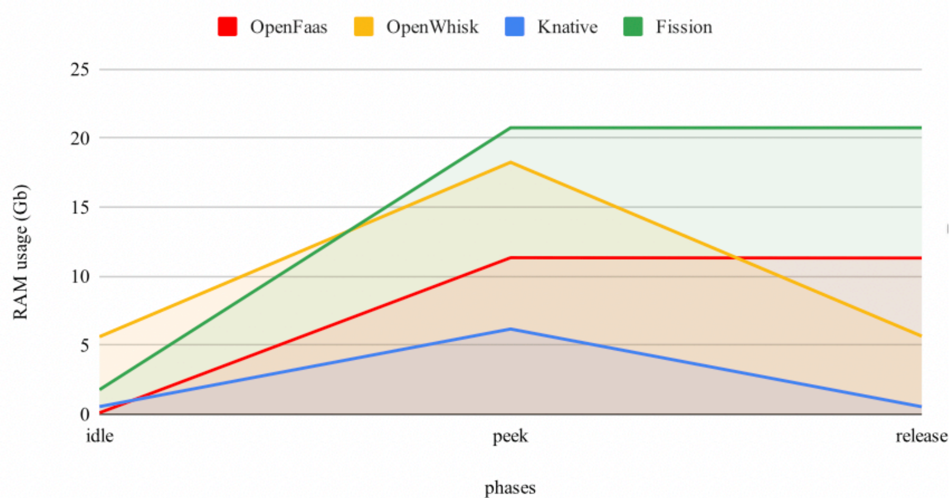

Figura 44 - Confronto fasi di esecuzione con 50 richieste per matrice 15000x15000

Affiancando i due grafici si può osservare come le differenze siano state soltanto marginali, fatta eccezione per OpenFaas che ha subito l'incremento maggiore, da meno di 5Gb a più di 10Gb. Degno di nota, infine, il fatto che Fission non sia riuscito a sostenere il nuovo tipo di carico richiesto, concludendo il test con una percentuale di richieste fallite pari al 48%.

Analizzando ancora una volta le tre fasi salienti, i comportamenti in fase di idle, naturalmente, non differiscono rispetto al precedente test, così come quelli nella fase di rilascio. Nonostante la maggiore difficoltà, rispetto a OpenFaas e Knative, nel gestire il

momento di concorrenza più intenso, di nuovo OpenWhisk è il framework che si comporta meglio nella fase finale, rilasciando la totalità delle risorse occupate nelle fasi precedenti.

5.4.3 Risultati test con frequenze maggiorate

Le differenze poco significative rispetto al test con la matrice di dimensioni minori dipendono dal fatto che le piattaforme riescono a smaltire in tempo le singole richieste, mantenendo un grado di concorrenza sufficientemente basso per consentire al sistema di rispondere in modo adeguato al carico. Partendo da questa consapevolezza, si è deciso di effettuare un ultimo ciclo di test andando ad insistere non tanto sul carico operativo della singola richiesta, ma piuttosto sul numero di richieste inviate al secondo. Le prove sono state effettuate con frequenze rispettivamente di 2, 3 e 5 richieste al secondo, utilizzando sempre la funzione che genera la matrice 15000x15000, che si ricorda essere un'operazione con un'occupazione media di 2Gb, con lo scopo di individuare un'eventuale vulnerabilità in uno o più framework.
Dai risultati ottenuti è evidente come la vera differenza l'abbia fatta appunto la modifica della frequenza nell'invio delle richieste, capace di mettere in difficoltà anche la piattaforma di OpenFaas, che aveva finora risposto con valori molto più bassi rispetto alla media, facendola terminare, in questo caso, con un significativo aumento della RAM occupata e con una percentuale di errori sempre maggiore man mano che si procedeva nei test. In Figura 45 viene riportato il solo grafico corrispondente al caso con frequenza di *2 req/s*, in quanto, l'eccessivo numero di errori riportati nei test successivi, ha reso gli andamenti non attendibili; per questo stesso motivo, nei valori che seguono viene escluso Fission, il quale aveva già riscontrato errori con l'invio di una sola richiesta al secondo.

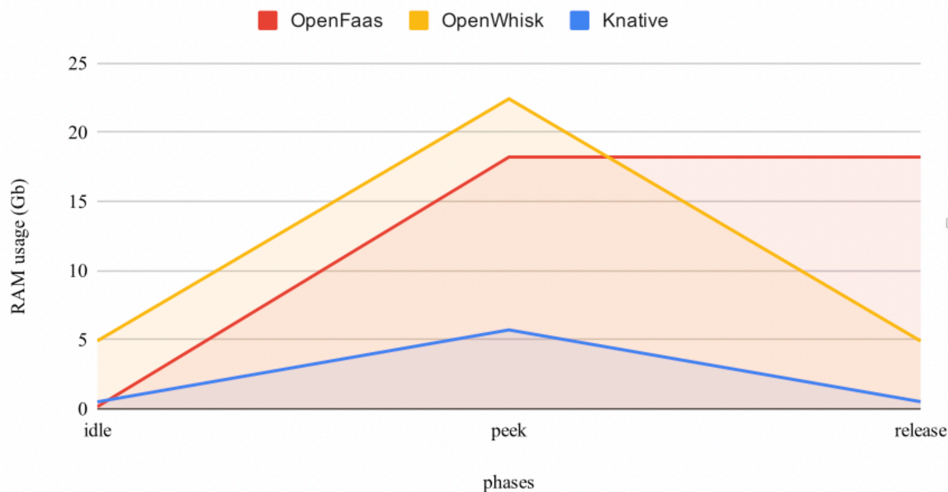

Figura 45 - Confronto fasi di esecuzione con matrice 15000x15000 a frequenza maggiorata

Knative e OpenWhisk sono gli unici framework che hanno sostenuto il nuovo tipo di carico senza grosse differenze e con una percentuale di errori dello 0%. Il motivo per il quale Knative riesce a mantenere un valore sempre così basso deriva dal fatto che, in presenza di un carico troppo elevato, non essendo presente all'interno della sua semplice architettura un componente controllore che gestisca il traffico e smisti le richieste, esso tende a sequenzializzare parte di queste, impiegando un tempo necessario a eseguire la totalità delle operazioni anche 5 volte maggiore rispetto ai competitor, che sono invece sempre stati in grado di terminare il test entro il minuto e con differenze reciproche non rilevanti.

5.5 Considerazioni conclusive

Le sperimentazioni che si sono susseguite negli ultimi mesi hanno visto OpenFaas come la piattaforma dotata della gestione più ottimizzata. Il framework è risultato il più performante, leggero e versatile, garantendo una facile installazione su tutti gli ambienti e con punteggi che hanno quasi sempre superato quelli raggiunti dai competitor, specialmente nei test sulla scalabilità. Le motivazioni si trovano nell'ampio e continuo sviluppo che giorno dopo giorno viene apportato al framework, rendendolo, al momento, il più popolare sulla rete. Inoltre, oculate scelte di analisi hanno concesso alla piattaforma il beneficio di un sistema modulare e reattivo, avvalendosi di componenti open source ben strutturati e qualitativamente superiori. Ciononostante, nell'ultimo banco di test, questo non si è comportato all'altezza delle aspettative, riportando percentuali di errori che aumentano proporzionalmente al carico in ingresso e un'allocazione delle risorse

troppo conservativa. Infine, bisogna aggiungere che OpenFaas manca, data la sua intrinseca semplicità, di alcune importanti caratteristiche di sicurezza e controllo delle risorse. Infatti, sebbene garantisca un ambiente pronto all'uso, permettendo l'adozione del serverless in pochi semplici passi, offre limitate possibilità di configurazione che non lo rendono idoneo per scenari di utilizzo più specifici.

Anche Fission è stato convincente in tutte le casistiche testate sul fronte della scalabilità, raggiungendo punteggi e valori molto simili a quelli di OpenFaas. La piattaforma ha concesso una piacevole esperienza d'uso, con risultati adatti a scenari di produzione industriali, il tutto legato a un'ottima e ben strutturata documentazione alle spalle che lo ha reso uno dei framework più trasparenti in termini di fruizione. Dall'altro lato si sono però presentati comportamenti inadeguati per quanto concerne lo stress test sulle risorse. Con questo si fa riferimento, in particolare, all'incapacità di sostenere un'elevata occupazione della memoria e a un bug che affligge le proprietà di autoscaling: al termine di ogni test, la piattaforma non è stata in grado di chiudere correttamente le repliche create, andando ad occupare inutilmente le risorse e obbligando a forzare manualmente l'interruzione del servizio. Si tratta di problemi e imprecisioni, probabilmente in parte dovuti a una certa immaturità, che verranno presumibilmente risolti nei prossimi aggiornamenti, vista la popolarità crescente della piattaforma.

OpenWhisk, posto inizialmente come framework più supportato e maturo nel panorama serverless, ha conseguito i risultati di scalabilità peggiori. La scelta di una soluzione *general-purpose* quale quella di Kafka, porta, nell'intero sistema, un overhead significativo, poiché ogni interazione viene da questo schedulata e smistata sotto forma di eventi asincroni. Ciò garantisce senza dubbio il miglior disaccoppiamento possibile tra i componenti in gioco, perdendo però l'efficienza e l'ottimizzazione che le altre piattaforme trovano in comunicazioni dirette e sincrone tra componenti *specific-purpose*. Tuttavia, l'andamento che ha presentato nella gestione della memoria, molto rilevante per il caso in esame, è stato il più soddisfacente, con esecuzioni affidabili e un sistema di deallocazione particolarmente elastico e adatto alle esigenze del progetto qui presentato, sebbene questi pregi siano stati controbilanciati da un consumo di RAM più elevato rispetto agli altri framework, per lo più dovuto alla presenza di un'architettura consistente e già di per sé più pesante da mantenere. Si ricorda infine l'impossibilità di soddisfare, con il cluster a disposizione, i requisiti di memoria suggeriti per Apache Kafka, con una conseguente degradazione delle performance che ha, già in partenza, svantaggiato la piattaforma OpenWhisk.

Il nuovo progetto di Google, Knative, nonostante si trovi ancora alla prematura versione *0.12*, ha ottenuto ottimi risultati, seppure non piazzandosi mai al primo posto nei test di scalabilità. L'unicità del progetto merita qualche considerazione che vada oltre gli effettivi punteggi mostrati dai test. Innanzitutto, Knative si pone come blocco aggiuntivo di Kubernetes, con la promessa di dotare nativamente l'orchestratore di funzionalità

serverless, rendendo trasparente l’approccio e lo sviluppo delle FaaS nella dimensione dei container. Inoltre, l’interazione tra servizi gestita da service mesh, porta numerosi benefici, come ad esempio la possibilità di monitoring *out-of-the-box*, di comunicazioni cifrate tra i servizi e di load balancing avanzato, che ben ripagano l’effort iniziale di aggiungere un livello di comunicazione (Istio) sopra l’orchestratore. Data l’importanza delle sue dimensioni, requisiti e dipendenze, esso risulta essere una soluzione realmente orientata verso grandi scenari industriali, e non per piccole realtà. Knative offre la più completa esperienza serverless disponibile adesso sulla scena, consentendo di raggiungere gradi di libertà e di controllo della piattaforma molto vicini alle possibilità offerte da anni dai già consolidati servizi di cloud.

6 Caso d’uso: MapReduce serverless in ambito FinTech

Tra i più diffusi impieghi delle piattaforme serverless, si trovano i servizi finanziari e, più nello specifico, le applicazioni di home banking e trading dove spesso sono presenti portali e documenti dedicati al cliente che vanno popolati in tempo reale con i dati di quest’ultimo. Le funzioni serverless vengono inserite nel workflow di operazioni quali il processamento di pagamenti, la valutazione dei rischi, i controlli di conformità e la gestione dei dati dell’utente. Essendo le funzioni stateless, solitamente queste vengono esposte sul lato frontend e fatte interagire con database lato backend. Non è però detto che tali funzioni entrino nel processo come punto di partenza, ovvero lanciate da un comando dell’utente. Difatti, è possibile anche far scattare la loro esecuzione a seguito di un evento nel sistema, come ad esempio la presenza di nuovi dati in un database, così che possano restituire risultati di interesse per l’utente come step finale dell’operazione [88].

Le FaaS vengono qui sfruttate all’interno di un processo **FinTech**, inserendole nel workflow di una struttura **MapReduce** per la generazione di un **report bancario** in formato **PDF**. L’idea è quella di più funzioni serverless che, lavorando in parallelo, leggono dati da sorgenti diverse e generano ognuna una porzione del file finale (**map**), per poi ricongiungersi con un punto di sincronizzazione (**reduce**), fino a comporre il PDF completo in output.

La motivazione che ha scaturito l’orientarsi verso questo approccio parallelo risiede nel collo di bottiglia che questa operazione solitamente crea in un modello più tradizionale e sequenziale, in quanto i dati da mantenere in memoria sono significativi, oltre al problema di dover reperire in successione tutti template da elaborare. La generazione di un PDF, infatti, consiste nel partire da uno o più template, tipicamente uno per pagina e ognuno composto da sezioni compilabili (“*fields*”), per imprimervi, attraverso opportune librerie, collezioni di dati nei rispettivi campi (Figura 46). Scomporre l’operazione in più funzioni parallele dividerebbe il carico a cui sono sottoposti i nodi dove queste eseguono, con un risparmio in termini di tempo e un bilanciamento più ottimizzato delle risorse.

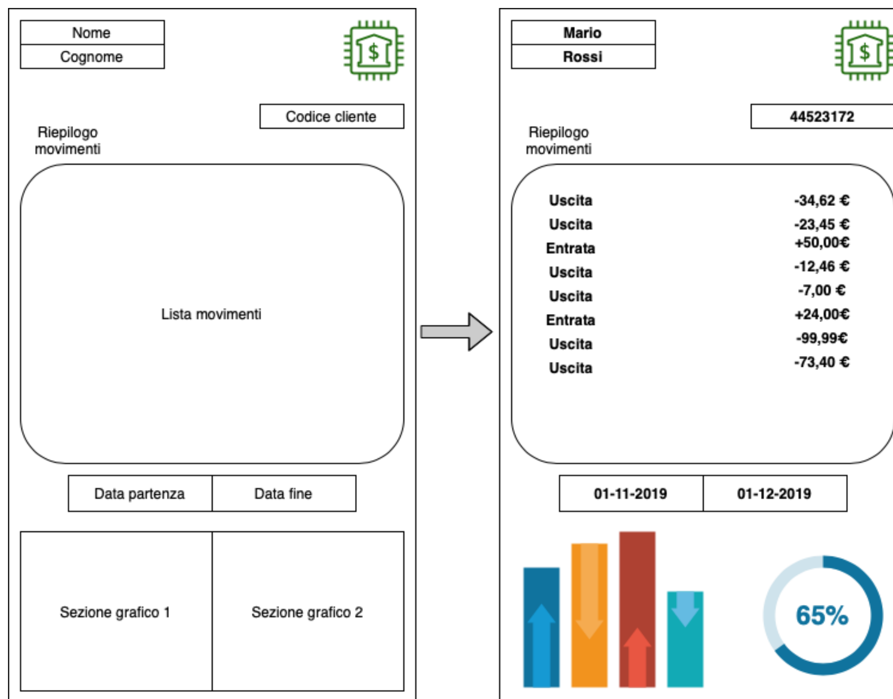

Figura 46 - Esempio compilazione template PDF

L'analisi va dunque condotta innanzitutto sul piano della fattibilità della soluzione, per poi proseguire andando ad indagare sull'efficienza di esecuzione guadagnata (o eventualmente persa), considerando anche come e quanto varia il modello di programmazione, la leggibilità e la semplicità del codice.

Prima di addentrarsi verso le scelte implementative, verrà fatta una descrizione del meccanismo e delle motivazioni del MapReduce, riportando anche un classico esempio di utilizzo su cui applicare i concetti basilari, ovvero il conteggio delle parole parallelo e distribuito, effettuato su file di grandi dimensioni.

6.1 MapReduce

Il **MapReduce** appartiene a quella classe di operazioni denominata *batching*, ovvero elaborazioni effettuate su quantità di dati significative. Tipicamente queste operazioni non hanno vincoli di tempo e vengono eseguite *out-of-band*, come ad esempio durante le ore notturne, quando il sistema è scarico di lavoro. Un esempio che viene comunemente riportato per spiegare un utilizzo del *batching* è il processamento notturno di tutti i dati raccolti dai social network durante il giorno, fatto allo scopo di proporre a un utente, nei giorni successivi, persone, prodotti o eventi di suo potenziale interesse. Il batching è reso possibile dalla scalabilità orizzontale, grazie alla quale viene realizzata la parallelizzazione delle unità di elaborazione.

Il MapReduce, in particolare, nasce come modello di programmazione, ideato e implementato in un framework da Google, per definire task paralleli al di sopra del *Google File System*. A differenza della programmazione *multithreading*, in cui i thread condividono i dati oggetto delle elaborazioni, presentando così una certa complessità proprio nel coordinare l'accesso alle risorse condivise, qui si elimina la condivisione dei dati, che vengono invece passati direttamente ai task come parametri di ingresso o valori di ritorno.
Il nome e il funzionamento del framework si ispirano a due funzioni cardine della programmazione funzionale: **map** e **reduce**. La prima è una funzione che, partendo da un insieme o una lista di elementi, applica a ognuno di questi la medesima trasformazione, restituendo poi la lista dei risultati, come nell'esempio di seguito riportato:

```
map quadrato [1,2,3,4,5] → [1,4,9,16,25]
```

La seconda funzione, invece, elabora una serie di dati per restituire uno o più risultati. Nell'ipotesi di una funzione reduce che, prendendo in ingresso l'output della map sopra riportata, ne restituisce la somma, si avrebbe dunque:

```
reduce somma [1,4,9,16,25] → 1+4+9+16+25 → 55
```

Il MapReduce si presenta quindi come la composizione di questi due step, uno detto di "esplorazione parallela" e uno di "riduzione a un risultato", lavorando secondo il principio del *divide et impera*: una suddivisione dell'operazione di calcolo in diverse parti processate in modo autonomo, seguita, una volta calcolata ciascuna parte del problema, da una ricomposizione dei vari risultati parziali in un unico risultato finale. È il framework stesso che si occupa dell'esecuzione dei vari task di calcolo, del loro monitoraggio e di eventuali riavvii in caso si verifichino problemi. Ciò è possibile grazie all'architettura di tipo *master-slave* che lo caratterizza, in cui vi è un componente, il master appunto, incaricato di lanciare e controllare l'avanzamento delle operazioni eseguite dai worker (slave).

Tra gli scenari di utilizzo più frequenti si trovano:

- Creazione di liste di parole da documenti di testo, indicizzazione e ricerca; appartengono a questa categoria esempi applicativi relativi a conteggi, somme, estrazione di liste univoche di valori (ad esempio analisi dei log dei Web server) e applicazioni di filtraggio dei dati.
- Analisi di strutture dati complesse, come grafi (ad esempio per applicazioni di social network analysis).
- Data mining e machine learning.

- Esecuzione di task distribuiti per calcoli matematici complessi e analisi numeriche.
- Correlazioni, operazioni di unione, intersezione, aggregazione e join (ad esempio analisi di mercato, analisi predittive e previsione dei trend) [89].

6.1.1 L'esempio del word count

Quando si parla di MapReduce, il conteggio delle occorrenze delle parole in un testo (o *word count*) è l'esempio più popolare, sia per la sua semplicità e sia perché riflette un caso d'uso familiare e diffuso. L'esempio viene dunque qui riportato per dare più concretezza ai meccanismi di *map* e *reduce*.

Si parte dall'ipotesi di avere un database distribuito contenente file di testo di grandi dimensioni, partizionati su diversi nodi, e, per velocizzare le operazioni, si decide di far lavorare ogni nodo su una partizione locale ad esso. In particolare, su ogni nodo viene lanciato un task di *map* che ha il compito, parallelamente agli altri, di leggere la porzione di file ad esso assegnata (es. una riga) e, di generare, per ogni parola incontrata, una coppia chiave-valore del tipo `<parola,1>` (Figura 47).

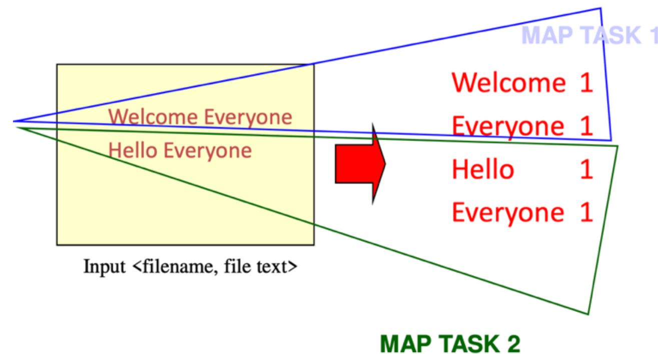

Figura 47 - Fase di map nel word count

Il framework prevede anche la possibilità di trasferire parte delle operazioni da un nodo troppo lento a nodi che hanno già completato il proprio lavoro.

In ogni caso, al termine della fase di *map,* una volta memorizzati gli output prodotti dai vari *mapper*, vengono lanciati dei task paralleli per la fase di *reduce*, e, come prima, le operazioni vengono effettuate in locale per una maggiore velocità. Bisogna evidenziare che il dover attendere la fine della prima fase dà la possibilità di riutilizzare gli stessi nodi di prima, andando a risparmiare le risorse necessarie a compiere il lavoro. La funzione di

reduce, in questo caso, si occupa di aggregare i risultati parziali, le coppie chiave-valore, andando a sommare tra loro i valori corrispondenti alla medesima chiave (Figura 48).

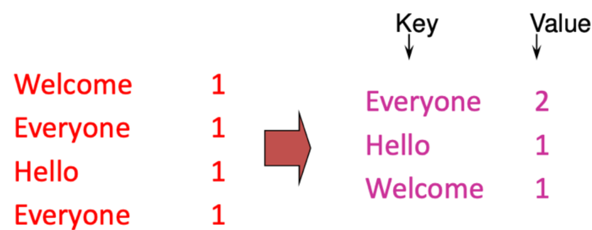

Figura 48 - Fase di reduce nel word count

Per consentire ai *reducer* di lavorare localmente, tra le due fasi tipicamente si prevede la presenza di uno step intermedio denominato **shuffling**, osservabile nella Figura 49, con il quale vengono raggruppate sullo stesso nodo di *reduce* le coppie aventi la stessa chiave.

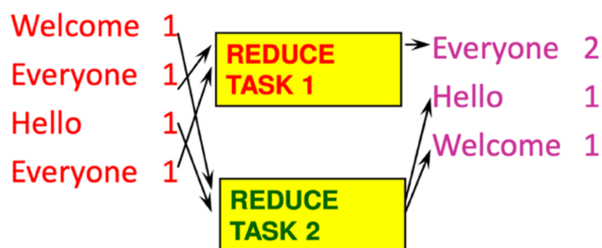

Figura 49 - Fase di reduce con shuffling nel word count

In questo passaggio il meccanismo più popolare per effettuare la ripartizione è detto *hash partitioning* e consiste nell'assegnare una chiave al nodo indicato dalla seguente formula:

```
reducer # = hash(key) % number_of_reducers
```

6.2 Stato dell'arte del MapReduce serverless

Il MapReduce sul serverless è un terreno ancora poco esplorato, sebbene non manchino in letteratura alcuni esempi che potrebbero dare uno spunto da cui partire.

Il team di *AWS* riporta una prima possibile soluzione nel suo blog: un'architettura serverless per eseguire *job* di MapReduce utilizzando *Lambda* e *S3* [90]. Il modello ideato

viene posto innanzitutto a confronto con *Hadoop*, il framework più popolare e diffuso nell'ambito del processamento di big data, evidenziando come la nuova soluzione abbia una curva di apprendimento meno ripida, sia meno complessa a livello architetturale e con un ridotto numero di interazioni in gioco. In aggiunta, poiché le applicazioni serverless vengono pagate solo in base al tempo di esecuzione, l'approccio risulta essere anche più economico. Gli obiettivi del progetto sono:

- Astrarre dall'infrastruttura sottostante.
- Ottenere un tempo di setup vicino allo "0".
- Fornire un modello di esecuzione *pay-per-execution*.
- Essere più economica delle altre soluzioni ad-hoc di data processing.
- Abilitare esecuzioni multiple sullo stesso dataset.

L'architettura, descritta in Figura 50, è composta da tre funzioni principali:

- Mapper
- Reducer
- Coordinator

Il coordinatore si appoggia a S3 per le informazioni di stato dei vari job. Una volta terminati i mapper, il coordinatore genera ricorsivamente i reducer fino alla restituzione del risultato finale.

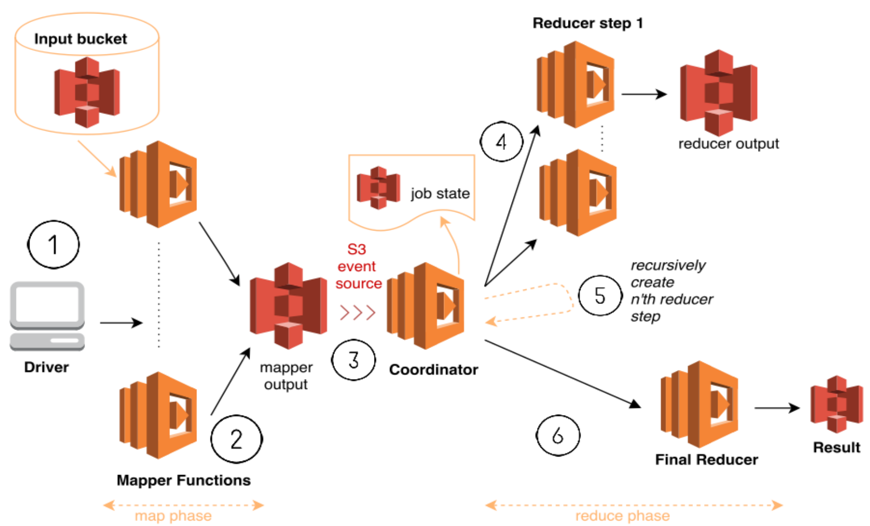

Figura 50 - Architettura MapReduce Serverless su AWS Lambda

Ovviamente questa soluzione non prescinde dall'utilizzo della piattaforma di *Amazon*, ma è utile per avere un'idea di come si potrebbe organizzare il modello delle interazioni

e i componenti in gioco.

Sempre basato sulla piattaforma *Lambda*, nel 2018 è stato sviluppato il progetto ***Corral*** [91], un lavoro nato con l'intento di semplificare il processo di deployment di un'applicazione MapReduce in *Hadoop* e *Spark*, ritenuto dall'autore macchinoso e permeato di codice "*boilerplate*". Inoltre, grazie all'approccio serverless di *Corral*, si andrebbero ad abbattere i costi e si eviterebbe la necessità di avere una base di conoscenza riguardo alle infrastrutture di *Hadoop* e *Spark*. L'idea di base è quella di utilizzare *Lambda* come ambiente di esecuzione, similmente a come fa *Hadoop* con *YARN*, e *S3* come storage.
Il maggiore ostacolo, a detta del progettista, sembra essere stato il tempo di esecuzione delle funzioni (massimo 5 minuti in *AWS Lambda*) che non ha reso possibile il trasferimento diretto dei dati tra mapper e reducer, in quanto avrebbe potuto portare a un time-out dei worker. In particolare, questa limitazione ha luogo a causa della fase centrale di shuffling che va a dilatare i tempi di esecuzione. Per risolvere il problema, in *Corral* si è deciso di utilizzare *S3* per effettuare uno shuffling stateless. Il modello realizzato è stato riportato in Figura 51.

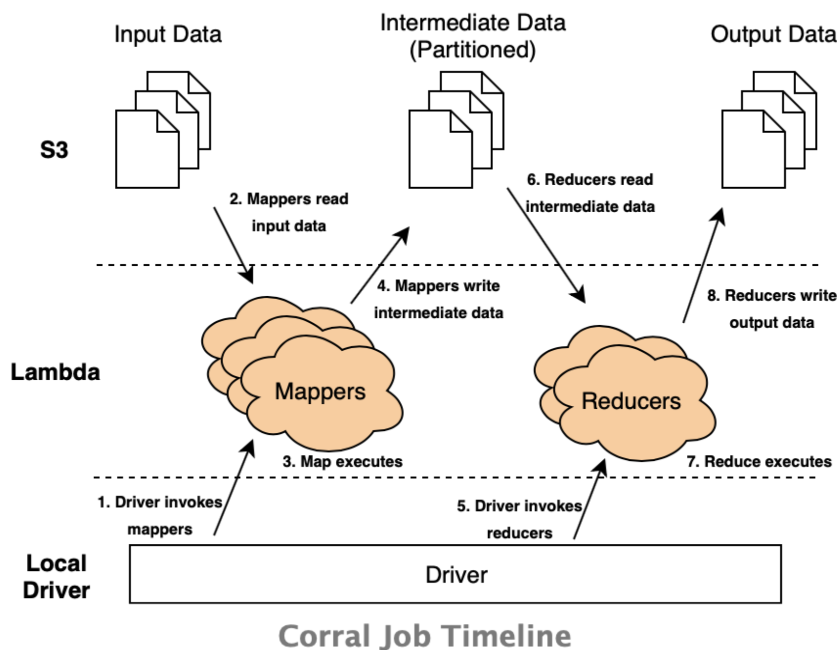

Figura 51 - Modello di Corral

Lo sviluppatore afferma che *Corral* è per lo più "agnostico" e quindi, in buona parte, non è vincolato alla presenza di *Lambda* e *S3*. Questo aspetto, legato al suo essere open source, fa del progetto un ottimo punto di riferimento da tenere in considerazione.

Anche *Microsoft* propone una sua soluzione per il MapReduce, introducendo il concetto

di ***Durable Function*** [92], viste come un’estensione delle normali funzioni di *Azure*, che consentono di scrivere una funzione più duratura e stateful in un ambiente serverless. Questa estensione abilita un nuovo tipo di funzione chiamata *orchestrator function*, con le seguenti caratteristiche:

- Può mantenere lo stato dell’esecuzione (stateful).
- Può invocare altre funzioni in modo sincrono o asincrono.
- Può salvare il proprio output in variabili locali.
- Può effettuare checkpoint automatici del loro progresso, così da non perdere lo stato in caso di problemi.

L’utilizzo primario per cui sono state pensate le *Durable Function* è il pattern *fan-out/fan-in*, descritto nella Figura 52.

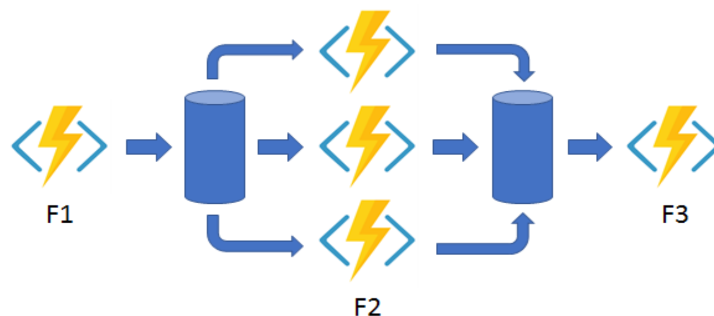

Figura 52 - Struttura del pattern fan-out/fan-in

Con le normali funzioni, il *fan-out* può essere facilmente realizzato facendo inviare a una funzione (F1) una serie di messaggi in una coda. Tuttavia, la fase complicata sarebbe il ricongiungimento (*fan-in*), in quanto bisognerebbe scrivere un codice che tenga traccia di quando le funzioni (F2) messe in esecuzione terminano. Le funzioni Durable riescono invece a gestire il pattern in modo semplice ed efficace:

```
public static async Task Run(DurableOrchestrationContext ctx){
    var parallelTasks = new List<Task<int>>();
    // get a list of N work items to process
    object[] workBatch = await
ctx.CallActivityAsync<object[]>("F1");
    for (int i = 0; i < workBatch.Length; i++)
    {
        Task<int> task =
ctx.CallActivityAsync<int>("F2", workBatch[i]);
        parallelTasks.Add(task);
    }
```

```
    await Task.WhenAll(parallelTasks);
    // aggregate all N outputs and send result to F3
    int sum = parallelTasks.Sum(t => t.Result);
    await ctx.CallActivityAsync("F3", sum);
}
```

Il *fan-out/fan-in* pattern viene quindi applicato al caso più specifico del MapReduce. Si può notare come, nell'esempio riportato dal team di Azure, viene utilizzato un solo Reducer, presentando dunque ha un approccio molto più vicino allo scope del presente lavoro, rispetto alle soluzioni riportate precedentemente. Al di là del vendor lock-in dovuto al fatto che si è vincolati all'utilizzo della piattaforma di Microsoft, in questo caso il team specifica anche che le macchine su cui eseguono le funzioni Durable hanno delle caratteristiche limitate e che quindi questo meccanismo non risulta essere adatto in caso di carichi molto elevati.

Una versione del pattern *fan-out/fan-in* è stata pensata anche per *AWS Lambda*, integrando l'utilizzo di ***Amazon SNS*** (*Simple Notification System*), ovvero un web service che coordina e gestisce la consegna e l'invio di messaggi a endpoint o client che si sono sottoscritti a un determinato topic [93]. Per quanto riguarda la fase di *fan-out*, c'è una funzione che pubblica un messaggio su un topic SNS, al quale sono registrati i vari worker, in modo da scaturire l'invocazione parallela di questi, per suddividere un task pesante in più subtask leggeri. Nella fase di *fan-in* bisognerà collezionare i risultati dai worker. Il meccanismo utilizzato è molto semplice e consiste nel lasciare ai worker stessi il compito di scrivere i propri risultati su uno storage comune. Vengono proposte come alternative ***DynamoDB*** e ***S3***, in base alla mole di dati da registrare: *DynamoDB* ha un limite massimo di 400Kb per elemento, mentre *S3* accetta elementi fino a 5Tb. Viene anche proposta un'idea su come realizzare il trigger di un eventuale funzione reducer. È possibile infatti, all'inizio del *fan-out*, scrivere sul database quanti worker sono stati lanciati. A questo punto, ogni worker che termina può decrementare di una unità questo valore e, se si accorge di essere l'ultimo rimasto in esecuzione (*workersLeft == 0*), può segnalare al reducer il completamento del mapping (Figura 53).

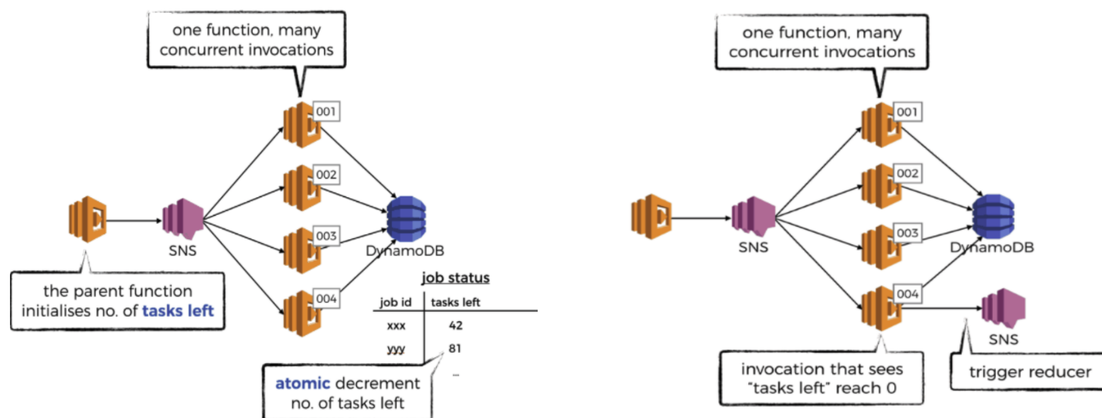

Figura 53 - Proposta di fan-out/fan-in in AWS Lambda

È stata proprio quest'ultima soluzione ad aver aiutato maggiormente nella formalizzazione di un punto di sincronizzazione nel progetto di seguito esposto.

6.3 Implementazione

Sulla base delle conoscenze apprese in questo campo, ciò che resta è la realizzazione della struttura MapReduce descritta, sfruttando però soluzioni open source, non essendo ancora presente nessun lavoro analogo in letteratura.

Le considerazioni effettuate sullo scenario serverless open source indicherebbero di affidarsi a Knative come piattaforma di riferimento al di sopra di un cluster orchestrato da Kubernetes, sia per gli ottimi risultati raggiunti sul piano della scalabilità, sia per l'occupazione di memoria relativamente basso. Ciononostante, Knative potrebbe rivelarsi non adatto allo sviluppo del progetto MapReduce. La questione è che le potenzialità di Knative sono ottime, soprattutto dal un punto di vista del processo di standardizzazione del serverless che la piattaforma sta portando avanti. La standardizzazione di un sistema software è infatti spesso più importante degli aspetti quantitativi di una tecnologia, specie se sommata a un livello di performance, quello di Knative, che è, già di per sé, più che soddisfacente. Tuttavia, ciò di cui soffre questo framework rispetto ai competitor è un importante "*abstraction gap*" ancora da colmare. Knative in sé offre la possibilità di gestire workload serverless su Kubernetes, ma da solo non fornisce molto altro, e, se si desidera costruire un'applicazione serverless interamente basata su Knative, è necessario aggiungere manualmente una serie di componenti esterni per completarlo, a partire dalla service mesh fino ad alcune API di alto livello che le altre piattaforme contengono *out-of-the-box*.
Inoltre, la struttura di una qualsiasi applicazione MapReduce necessita di un meccanismo efficace per la comunicazione tra i componenti, possibilmente a scambio di eventi, che

garantisca un buon livello di disaccoppiamento, una certa affidabilità e anche semplicità d'uso. Se si considera il componente Eventing di Knative, dopo alcuni test preliminari, questo è risultato essere tanto potente quanto generico, e soprattutto troppo poco integrato con le funzioni serverless. Ciò deriva dal fatto che i due blocchi, Serving e Eventing, sono stati pensati come entità separate, autocontenute e indipendenti. Se a questi aspetti si aggiungono l'andamento particolarmente irregolare che si è potuto osservare nei test sulla memoria (Figura 39) e la dilatazione dei tempi di risposta a seguito della sequenzializzazione di parte delle richieste, Knative non appare più come candidato ideale, specie in un progetto che implica la necessità di interagire in tempo reale con degli utenti.
Infine, un ultimo aspetto che fa dissuadere dalla scelta di Knative riguarda una questione prettamente implementativa: durante la ricerca degli strumenti da utilizzare in fase di sviluppo, si è deciso di affidarsi a una delle librerie più diffuse e consolidate per la lavorazione dei PDF, *iText* [94]. Questo potente strumento è però disponibile soltanto per ambiente Java, il che risulta essere problematico in Knative, il quale, a causa di un bug ancora aperto, attualmente riscontra problemi proprio nell'istanziazione di funzioni Java.

La seconda scelta ricadrebbe su OpenWhisk, piattaforma con un solido supporto alle spalle, che, nonostante pecchi di alcune limitazioni sul fattore della scalabilità, in parte per colpa delle risorse a disposizione in questo progetto, è capace di gestire significativi carichi di richieste altamente memory-bound. Il consumo di memoria che ne deriva non è esiguo, ma è quantomeno lineare ed elastico. Inoltre, tenuta presente la considerevole quantità di risorse disponibili tipicamente all'interno di un cluster industriale, l'installazione e i consumi di OpenWhisk non dovrebbero presentare un limite eccessivamente restrittivo, soprattutto se si considera l'ottima capacità del sistema di rilasciare le risorse nel breve termine. In determinati scenari di produzione potrebbe infatti risultare più invalidante un picco minore di utilizzo, ma che non riesce a essere smaltito, piuttosto che una fase transitoria di carico maggiore.
Bisogna poi tenere conto di aspetti che vanno oltre le sole performance sotto stress, ma che dipendono principalmente dalle funzionalità offerte dalle piattaforme. In particolare, in un modello come quello del MapReduce in cui è necessario un buon sistema di comunicazione tra i servizi, sicuramente la completezza offerta dal componente Kafka di OpenWhisk consentirebbe una certa semplicità e consistenza in fase di modellazione, implementazione e test, anche grazie alla possibilità di lanciare una funzione tramite il meccanismo a *trigger* basato sui topic di eventi.
Altro aspetto fondamentale è la presenza di client, disponibili in vari linguaggi, capaci di sfruttare agilmente tutte le API del framework. A ciò si aggiunge anche un ottimo supporto al linguaggio Java, che permetterebbe l'utilizzo senza ostacoli della libreria per PDF selezionata.
Infine, l'adozione di OpenWhisk in un'azienda può rivelarsi oggi una scelta vincente, poiché, in base a quanto dichiarato da Matt Rutkowski, CTO di *Serverless Technologies*

and Advocacy in IBM, OpenWhisk sta già avviando un processo di integrazione con Knative che gli consentirà di risolvere i problemi che ancora lo affliggono, sfruttando i vantaggi del layer di Knative al di sopra di Kubernetes, realizzando così workload serverless nativi e completi [95]. Per tutte queste ragioni, nell'implementazione dell'applicazione MapReduce serverless in contesto FinTech, si è scelto di affidarsi a OpenWhisk.

Alcuni degli esempi presentati, presi come spunto per la definizione delle interazioni tra i componenti da realizzare, uniti agli studi e ai test effettuati sulla piattaforma selezionata, hanno condotto alla formalizzazione del modello descritto nella Figura 54.

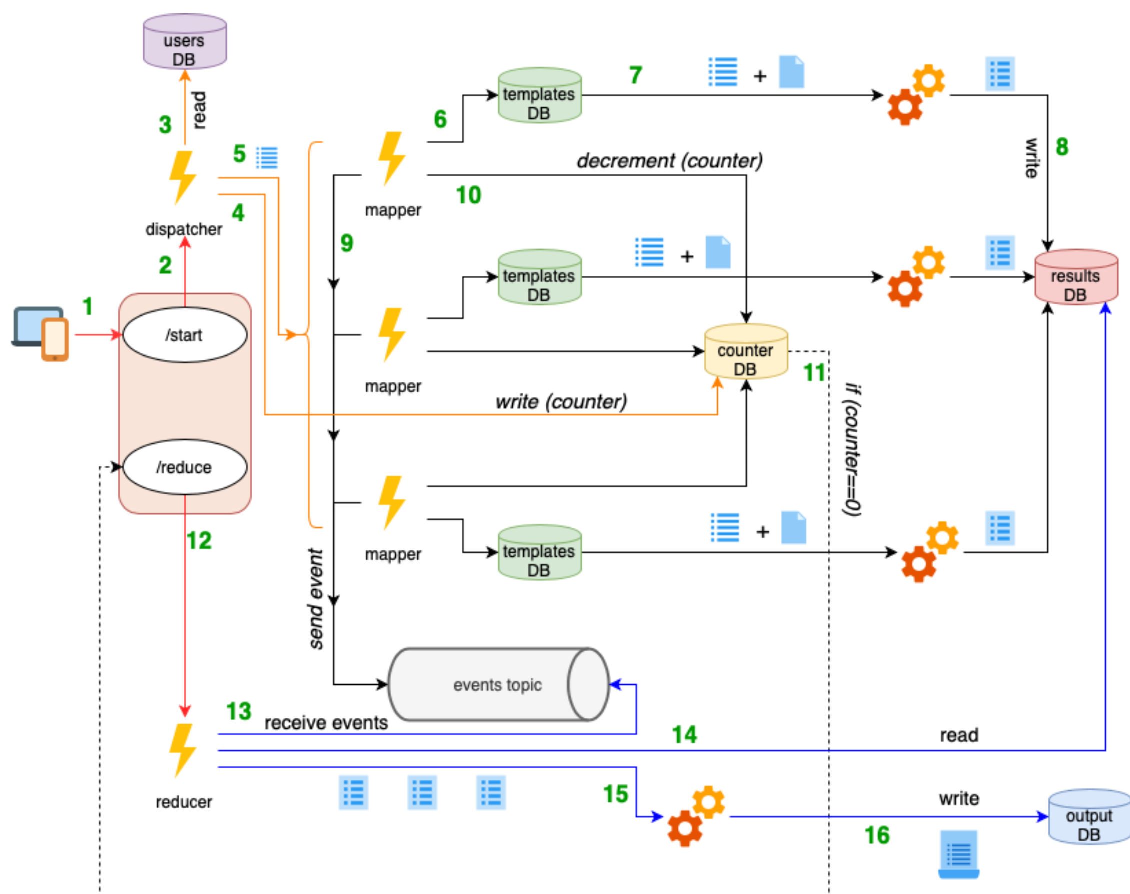

Figura 54 - Modello del progetto serverless MapReduce

A monte del processo è prevista un'entità **server** in ascolto delle richieste di inizio operazione, che avvia una prima FaaS **dispatcher**, la quale si occupa di calcolare i subtask da eseguire e di avviare i **mapper**; sarà poi compito dell'ultimo *mapper* far scaturire l'esecuzione del **reducer**.

Viene ora analizzato l'intero processo, per ognuno dei quattro componenti principali, con riferimento agli step numerati nella figura. Il primo componente attivato sarà il *server*:

1) Un utente effettua una richiesta per la generazione di un documento PDF (*/start*).
2) Il *server*, alla ricezione della richiesta, attiva la funzione *dispatcher.*

Il *dispatcher*, una volta messo in esecuzione:

3) Legge i dati dell'utente da un db e calcola il numero di *mapper* necessari per portare a compimento l'operazione; per ipotesi a ogni utente sono associate più collezioni di dati, ognuna delle quali contiene le informazioni necessarie a compilare i campi di una pagina del PDF finale.
4) Scrive il numero di *mapper* che verranno attivati in un db accessibile a tutti i componenti dell'applicazione (contatore).
5) Attiva i *mapper* in parallelo, indicandogli i dati da inserire, dove reperire il template del PDF da compilare, un numero di sequenza, il nome del topic al quale dovranno inviare il messaggio di completamento e l'indirizzo del contatore.

Ogni *mapper* avviato eseguirà le medesime operazioni per generare un risultato parziale, in particolare:

6) Scarica il template di sua competenza.
7) Compila il template con i dati ricevuti come parametro.
8) Salva la il documento generato in un db (risultato parziale).
9) Manda un messaggio di completamento sul topic indicato, che include il numero di sequenza e l'indirizzo del risultato parziale.
10) Decrementa il contatore dei *mapper* in esecuzione.
11) Se il nuovo valore del contatore è "0", allora avvisa il *server* (*/reduce*) di avviare il *reducer*, altrimenti termina.

Il *server*, quando riceve indicazione del completamento dell'operazione di mapping (*/reduce*):

12) Attiva il *reducer*, indicandogli il numero totale di *mapper*, ovvero quanti risultati parziali dovrà leggere e elaborare.

Il *reducer*, infine, si occuperà di concatenare i risultati parziali fino alla generazione di un documento unico:

13) Si sottoscrive al topic passatogli come parametro e legge i messaggi presenti finché non li ha ricevuti tutti, essendo a conoscenza del numero totale di *mapper* precedentemente lanciati.

14) Da ogni messaggio estrapola le informazioni necessarie a reperire il risultato parziale relativo, con le quali va ad interrogare il db dei risultati.
15) Mette in ordine i pezzi scaricati e li concatena in un unico documento.
16) Carica il risultato finale sul db di output.

Scendendo maggiormente nei dettagli implementativi, si presentano adesso gli effettivi strumenti utilizzati nel progetto, motivandone le scelte, fino alla realizzazione delle funzioni, con codice annesso.

6.3.1 Apache Kafka e Zookeeper

Come detto, in un ambiente MapReduce è necessario un meccanismo consistente che abiliti le comunicazioni tra entità disaccoppiate e indipendenti, realizzabile con efficacia grazie all'utilizzo di un sistema ad eventi.
Il sistema ad eventi selezionato è **Apache Kafka** [68] e la sua presenza nativa in OpenWhisk ha giocato un ruolo importante nella selezione di questa piattaforma. *Apache Kafka* è un sistema open source di messaggistica istantanea molto diffuso, che consente la gestione di un elevato numero di operazioni in tempo reale con migliaia di client, sia in lettura che in scrittura. Viene realizzato dal team di *LinkedIn* a partire dal 2010 per far fronte al problema della gestione di grandi quantità di dati, in scenari in cui era necessaria un'elaborazione in batch in real time per rendere immediatamente disponibili i dati agli utenti.

Kafka viene definito più precisamente come "piattaforma di streaming distribuito", essendo progettato per eseguire su un cluster multinodo ed essendo dotato di tre funzionalità chiave:

- Pubblicazione e sottoscrizione a stream di dati, similmente a come si fa con una coda di messaggi.
- Mantenimento di tali stream di dati su supporti di memorizzazione persistente come meccanismo di fault-tolerance.
- Processamento degli stream di dati al loro arrivo.

Questi tre aspetti fanno del framework qualcosa di ben più completo e complesso di un semplice sistema ad eventi.
Per quanto riguarda la parte di messaggistica, utilizzata nello specifico in questo progetto, uno stream di dati in Kafka è rappresentato da un insieme di *record*, mantenuti in categorie denominate *topic*. Una volta creato il topic, ogni nodo può assumere il ruolo di produttore (*Producer API*) per quel topic o di consumatore (*Consumer API*). Il

meccanismo a topic di Kafka permette di ottenere un forte disaccoppiamento nello spazio e nel tempo tra queste due entità, consentendo di sottoscriversi a un topic per il quale non sono ancora presenti produttori o di produrre messaggi verso un topic che non ha ancora consumatori registrati. Inoltre, essendo un sistema costruito appositamente per ambienti distribuiti, conscio della presenza di situazioni di parallelismo, esso fornisce built-in meccanismi affidabili di ordinamento e garanzia nella gestione di comunicazioni molti-a-molti.

Oltre a essere la funzionalità sfruttata in fase di sviluppo, il sistema ad eventi è sicuramente la modalità di utilizzo della piattaforma più diffusa. Tuttavia, come detto, Kafka offre molto di più, grazie alle caratteristiche di persistenza che consentono di scrivere i messaggi su dischi replicati per sopperire a eventuali errori e inconsistenze, ma soprattutto alla possibilità di realizzare pipeline che prendono in ingresso flussi continui di dati, li elaborano e producono in output stream di dati trasformati su determinati topic di destinazione.

In ogni cluster Kafka deve però essere presente un controllore, che gestisca le partizioni distribuite e questa mansione è affidata a un secondo servizio open source, chiamato **Zookeeper** [69], anch'esso ovviamente presente in OpenWhisk (Figura 55). Zookeeper è un servizio di configurazione, sincronizzazione e registro di nomi per applicazioni distribuite. Senza scendere troppo nei dettagli, esso si occupa di mantenere informazioni chiave relative sia ai produttori che ai consumatori di messaggi. Per i produttori, ad esempio, mantiene lo stato di salute dei nodi e gestisce le repliche in caso di fallimenti, mentre per i consumatori mantiene l'offset, ovvero un indicatore della posizione in cui ogni consumatore è arrivato nella lettura dei messaggi per un certo topic. Molto importante, infine, il suo ruolo di registro dei topic, mantenuto per entrambi le entità, in modo che sia consumatori che produttori possano essere a conoscenza dei topic presenti nel sistema.

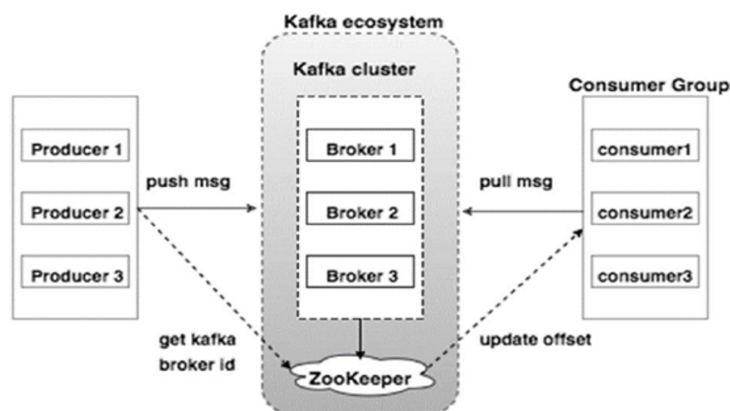

Figura 55 - Architettura di Kafka e Zookeeper

In OpenWhisk, Kafka e Zookeeper rendono possibile lo scambio di eventi e l'avvio di funzioni tramite *trigger*, permettendo così invocazioni affidabili, efficienti e disaccoppiate. La Figura 56 rappresenta il modello ad eventi realizzabile in OpenWhisk, in cui, una volta creata una funzione (*action*), è sufficiente definire un *trigger* che comunichi con la sorgente di eventi (*event source* e *feed*) e una regola (*rule*) che colleghi la *action* al *trigger*.

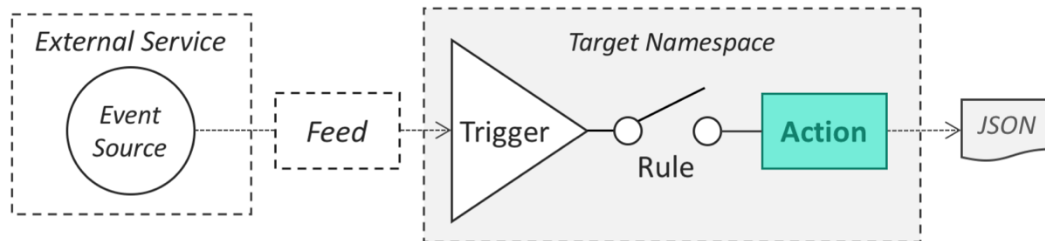

Figura 56 - Modello ad eventi di OpenWhisk

6.3.2 MinIO

Nello sviluppo di questo progetto è stata necessaria la presenza di un object storage per salvare i risultati parziali e finali delle funzioni. Non esistono vincoli particolari nella scelta di una particolare implementazione, tuttavia, come si è visto nella letteratura presentata, tipicamente in scenari simili si tende ad utilizzare un'istanza di *S3*.
Esiste però un'alternativa gratuita, open source e accessibile sia online che in locale, chiamata **MinIO** [96], ideale per effettuare dei test di upload e download di file di medie-grandi dimensioni, lavorando con stream di byte. MinIO è stato progettato per affermarsi come lo standard per gli object storage nei cloud privati ed è pertanto una soluzione ampiamente utilizzata, con una solida e numerosa community alle spalle.

Il servizio viene spesso definito come una versione gratuita di *S3* che può essere eseguita localmente. Difatti, MinIO è un server di object storage che implementa le stesse API pubbliche del servizio di Amazon, e ciò significa che le applicazioni progettate per S3 sono automaticamente configurate per dialogare con MinIO.
L'object storage offre la possibilità, come del resto fa S3, di creare un *bucket* personale, una sorta di cartella remota per dati non strutturati, in cui inserire qualsiasi tipo di file, come ad esempio foto, video, o, nel caso in esame, documenti PDF. La dimensione dei singoli oggetti memorizzati può variare da pochi Kb fino a un massimo di 5Tb (come in S3), e l'interazione con questi risulta essere particolarmente ottimizzata, con velocità di lettura e scrittura dichiarate rispettivamente di *183 Gb/s* e *171 Gb/s*. Nelle fase di sviluppo sono state testate sia la versione remota che quella in locale, ed è stato possibile

apprezzare l’effettiva potenzialità di questa soluzione, con sessioni di download e upload rapide e consistenti.

Oltre al fatto di essere efficiente e gratuito, un altro dei motivi che ha spinto maggiormente verso questa scelta è che MinIO è stato appositamente progettato per eseguire su container leggeri orchestrati da Kubernetes (Figura 57).

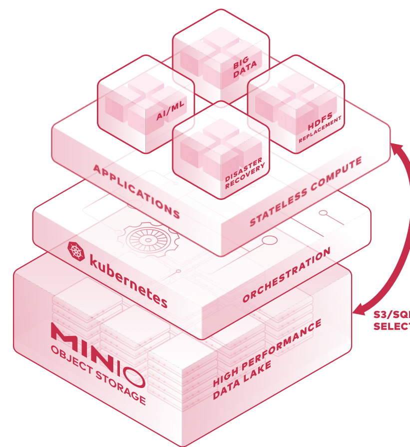

Figura 57 - Struttura a livelli in MinIO

Per effettuarne il deployment locale è sufficiente istanziare un volume di Kubernetes da agganciare a un Kubernetes Service, a sua volta applicabile con semplici file *.yaml* o tramite un *Helm chart* ufficialmente rilasciato dal team di MinIO.

6.3.3 Restdb

Il database in cui viene salvato e decrementato il contatore dei *mapper* attivi può essere idealmente un qualunque db, preferibilmente NoSQL, accessibile tramite REST API.
La prima ipotesi è ricaduta ovviamente sul componente **CouchDB** [67], offerto nativamente dalla piattaforma OpenWhisk, che avrebbe pertanto permesso una latenza ridotta al minimo, rimanendo all’interno dell’ambiente cluster circoscritto. Tuttavia, dopo aver studiato le possibilità da questo offerte, confrontandole con le esigenze progettuali, ci si è resi conto che l’attuale versione non fornisce un’API capace di decrementare direttamente un valore numerico e di ricevere come risposta il valore aggiornato. Questo aspetto è di particolare importanza, in quanto, una feature del genere, consentirebbe ad ogni *mapper* di aggiornare il contatore e di conoscerne il nuovo valore attraverso un’unica chiamata. Ciò renderebbe il decremento del contatore un’azione atomica, senza la necessità di sincronizzare i vari *mapper* e senza il rischio che più funzioni contemporaneamente vedano il contatore azzerato e scaturiscano l’attivazione del

reducer.

Partendo da questa considerazione, si è cercata una soluzione che fosse allo stesso tempo gratuita, efficiente e dotata di questa funzionalità chiave. Fortunatamente ci si è presto imbattuti in **Restdb** [97], un servizio facilmente utilizzabile online, senza alcuna installazione e con un'ottima documentazione che ne ha permesso un utilizzo semplice e rapido.
Sul piano implementativo, il servizio si presenta come una delle tante soluzioni presenti nel panorama NoSQL, con interazioni CRUD realizzate tramite REST API, dati trattati sotto forma di oggetti JSON e complete funzionalità di controllo degli accessi basato sui ruoli (RBAC).

Ciò che però lo distingue da molti altri servizi competitor è proprio il motivo principale per il quale è stato scelto, ovvero un ricco set di API capaci di manipolare i dati sulla base di istruzioni precise e complete. Di seguito viene riportato un esempio di chiamata per decrementare il contatore, che mostra quanto sia rapido il processo che porta un *mapper* a determinare la fine della fase di mapping:

```
Unirest.put("COUNTER_DB_URL" + counterId)
                .header("x-apikey", "X-API_KEY")
                .header("cache-control", "no-cache")
                .body("{\"$inc\":{\"counter\": -1}}").asJson();

        int newValue = (int) response.getBody()
                                        .getObject()
                                        .get("counter");

        // If TRUE, I'm the last mapper
        if(newValue == 0) {
           //END OF MAPPING PHASE
}
```

Il tutto avviene tramite un'unica richiesta HTTPS con metodo PUT, che rende possibili la modifica del dato senza alcuna conoscenza pregressa del suo valore e la lettura immediata dello stato aggiornato, direttamente inserito nella risposta.

Data la semplicità e la comodità d'uso, si è deciso di appoggiarsi a questo servizio non solo per la persistenza del contatore, ma anche per memorizzare alcuni utenti fittizi di test. Restdb offre infatti anche un ottimo sistema di generazione randomica di dati e collezioni, che ha accelerato di molto le fasi di sperimentazione. È stato così possibile, in breve tempo, realizzare un meccanismo per cui a ogni utente viene associato un insieme di collezioni, e a ognuna di queste collezioni viene associato un template PDF. In questo modo, con una sola chiamata GET per un dato utente, si ottengono tutte le informazioni

necessarie a generare il relativo documento, già strutturate per essere collocate in diverse pagine del PDF finale.

6.3.4 Realizzazione delle funzioni

Partendo dalla componente non serverless, si sottolinea innanzitutto che il *server* è stato realizzato in NodeJS avvalendosi del modulo **Express** [98], ormai divenuto il framework per NodeJS standard per le interazioni web, data l'estrema semplicità con cui è possibile istanziare un server HTTP che gestisca agilmente il traffico web in entrata e in uscita. La scelta di porre un'entità server a monte del processo è motivata dal bisogno di avere un punto di controllo durevole e non effimero, che possa donare una maggiore solidità al sistema.

Anche la action (FaaS) *dispatcher* è stata realizzata in ambiente Javascript, poiché il suo comportamento prevede principalmente l'attivazione dei *mapper*, facilmente realizzabile tramite il client **Openwhisk-client-js** [99], consigliato nella documentazione di OpenWhisk stesso. Tramite questo strumento si possono invocare tutte le API necessarie a generare e invocare le action, sia in modo diretto, che indirettamente attraverso i trigger, per un maggiore disaccoppiamento.

Le action (FaaS) dei *mapper* e del *reducer* sono state invece realizzate in Java, così da poter utilizzare la popolare libreria per la manipolazione di documenti PDF **iText** [94]. Con questa è possibile, tra le altre cose, modificare agilmente i file in formato PDF, grazie a un processo che estrapola e compila con poche istruzioni i campi presenti in un template. La libreria in questione non copre soltanto le operazioni eseguite dai *mapper*, ma è in grado anche di realizzare la logica del *reducer*, offrendo funzionalità adeguate a concatenare più documenti in un unico output, risultando quindi particolarmente adatta allo scopo del progetto. Nelle operazioni appena descritte è necessario fornire i documenti sotto forma di stream di byte, il che è l'ideale date le modalità con le quali si effettuano il download e l'upload dei file sullo storage ad oggetti selezionato.

Segue ora la presentazione del codice che implementa il comportamento dei quattro componenti principali, *server*, *dispatcher*, *mapper* e *reducer*.

Server.js

```
const express = require("express")
var openwhisk = require('openwhisk')
var bodyParser = require('body-parser')
```

```
const PORT = 'PORT_NUMBER'
const app = express()
var jsonParser = bodyParser.json()

// Connect to OpenWhisk platform
var options = {apihost: 'API_HOST', api_key: 'API_KEY'}
var ow = openwhisk(options)

// HP: the user ID to be processed is passed as argument
var userId = process.argv.slice(2)

// Wait for start requests
app.get("/start", (req, res) => {
    var params = {
      topicName: 'TOPIC_NAME',
      userId: userId
    }
    // Invoke dispatcher
    ow.triggers.invoke({name:"dispatcherTrigger", params})
    .then(result => {
      console.log("dispatcher triggered")
    }).catch(err => {
      console.error("failed to fire trigger", err)
    })
});

// The last mapper will signal the end of the Map phase
app.post("/reduce", jsonParser, (req, res) => {
  var params = {
    userId: userId,
    bucketName: req.body.bucketName,
    topicName: req.body.topicName,
    totalMappers: req.body.totalMappers
  }
  // Invoke reducer
  ow.triggers.invoke({name:"reducerTrigger", params})
  .then(result => {
    console.log("reducer triggered")
  }).catch(err => {
    console.error("failed to fire reduce trigger", err)
  })
});

app.listen(PORT, () => {
  console.log("Server is listening on port: " + PORT);
});
```

Dispatcher.js

```
function main(params) {
  var openwhisk = require('openwhisk')
  var request = require("request")

  // Connect to OpenWhisk platform
  var options = {apihost: 'API_HOST', api_key: 'API_KEY'}
  var ow = openwhisk(options)

  // Read parameters
  var userId = params.userId
  var topicName = params.topicName
  var bucketName = 'BUCKET_NAME'

   // GET the passed user from a DB
  var options = {
    method: 'GET',
    url: 'USERS_DB_URL' + userId,
    headers: {
      'cache-control': 'no-cache',
      'x-apikey': 'X-API_KEY'
    }
  };
  request(options, function (error, response, body) {
    if (error) throw new Error(error);

    var parsedBody = JSON.parse(body)

    // Calculate the number of mappers to be triggered
    // One mapper for each subcollection of the user
    // -1 is because we have to exclude the userId field
    mappersNumber = Object.keys(parsedBody).length - 1

    // Store the counter in a DB
    var options = {
      method: 'POST',
      url: 'COUNTER_DB_URL',
      headers:
       { 'cache-control': 'no-cache',
         'x-apikey': 'X-API_KEY',
         'content-type': 'application/json' },
      body: { counter: mappersNumber },
      json: true };
    request(options, function (error, response, body) {
        var counterId = body["_id"]
        console.log("counterId: " + counterId)
```

```
        // Invoke mappers
        var k = 0
        Object.keys(parsedBody).forEach(function(key) {
          if(key != "_id") {
            var params = {
              sequenceNumber: k,
              bucketName: bucketName,
              //the template has the same name of the subcollection
              templateName: key + ".pdf",
              counterId: counterId,
              topicName: topicName,
              totalMappers: mappersNumber,
              //pass the collection with which the mapper will work
              data: parsedBody[key]
            }
            k++
            ow.triggers.invoke({name:"mapperTrigger", params})
            .then(result => {
              console.log("trigger fired!")
            }).catch(err => {
              console.error("failed to fire trigger", err)
            })
          }
       })
    });
  });
}
exports.main = main
```

Mapper.java

```java
public class Mapper {

        public static JsonObject main(JsonObject args) {

        // Prepare response object
        JsonObject response = new JsonObject();

        // Parse parameters
        int sequenceNumber;
        String bucketName;
        String templateName;
        String counterId;
        String topicName;
        int totalMappers;
        JsonObject data;
        try {
            sequenceNumber = args
                                   .getAsJsonPrimitive("sequenceNumber")
                                    .getAsInt();
            bucketName = args
                                .getAsJsonPrimitive("bucketName")
                                .getAsString();
            templateName = args
                                .getAsJsonPrimitive("templateName")
                                .getAsString();
            counterId = args
                                .getAsJsonPrimitive("counterId")
                                .getAsString();
            topicName = args
                                .getAsJsonPrimitive("topicName")
                                .getAsString();
            totalMappers = args
                                .getAsJsonPrimitive("totalMappers")
                                .getAsInt();
            data = args.getAsJsonObject("data");

        } catch(Exception e) {
           response.addProperty("Exception", "Incorrect parameter");
           return response;
        }

        // Get fields from args
        HashMap<String, String> fields = new Gson()
                                                 .fromJson(data,
                                                     HashMap.class);
```

```
// Connect to object storage
MinioClient minioClient;
try {
    minioClient = new MinioClient("MINIO_URL");

} catch (InvalidEndpointException
        | InvalidPortException e) {
    response.addProperty("Exception", "Minio exception");
    return response;
}


// Download PDF template from the object storage as stream
InputStream inStream;
try {
    inStream = minioClient.getObject(bucketName,
                                    templateName);

} catch (Exception e) {
    response.addProperty("Exception", "Minio exception");
    return response;
}

// Initialize outputStream
ByteArrayOutputStream os = new ByteArrayOutputStream();

// Open PDf template stream
PdfStamper pdfStamper;
try {
    pdfStamper = new PdfStamper(new PdfReader(inStream),
                                        os);

} catch (DocumentException | IOException e) {
    response.addProperty("Exception", "Stamper exception");
    return response;
}

// Extract compilable fields from template
AcroFields pdfFormFields = pdfStamper.getAcroFields();

// Compile fields
for(Entry<String, String> entry : fields.entrySet()){
    try {
        pdfFormFields.setField(entry.getKey(),entry
                                        .getValue());

    } catch (IOException | DocumentException e) {
```

```
            response.addProperty("Exception", "Field exception");
            return response;
        }
    }

    // Flatten changes
    pdfFormFields.setGenerateAppearances(true);
    pdfStamper.setFormFlattening(true);

    // Free resources
    try {
        pdfStamper.close();
        inStream.close();

    } catch (DocumentException | IOException e) {
        response.addProperty("Exception", "Stream exception");
        return response;
    }

    // Upload compiled PDF to the remote storage
    ByteArrayInputStream bais = new ByteArrayInputStream(
                                        os.toByteArray());
    String compiledFileName = "COMPILED_FILE_NAME";
    try {
        minioClient.putObject(bucketName, compiledFileName,
                                bais);

    } catch (Exception e) {
        response.addProperty("Exception", "Minio exception");
        return response;
    }

    // Free resources
    try {
        bais.close();

    } catch (IOException e) {
        response.addProperty("Exception", "Resource exception");
        return response;
    }

    // Put new PDF info into Kafka for the reducer
    try {
        produceCompletionRecord(topicName, sequenceNumber,
                                    compiledFileName);

    } catch (ClassNotFoundException e) {
        response.addProperty("Exception", "Kafka exception");
        return response;
```

```
        }

        // Decrement mappers counter
        try {
            decrementCounter(counterId, bucketName,
                                    topicName, totalMappers);

        } catch (UnirestException e) {
            response.addProperty("Exception", "Unirest exception");
            return response;
        }

        response.addProperty("Action", "Completed");

        return response;
    }

    public static void decrementCounter(String counterId,
                                        String bucketName,
                                        String topicName,
                                        int totalMappers)
                                           throws UnirestException {

        // Decrement the mappers counter on a remote DB
        HttpResponse<JsonNode> response = Unirest
                                            .put("COUNTER_DB_URL"
                                                + counterId)
                .header("content-type", "application/json")
                .header("x-apikey", "X-API_KEY")
                .header("cache-control", "no-cache")
                .body("{\"$inc\":{\"counter\": -1}}").asJson();

        int newValue = (int) response.getBody()
                                        .getObject()
                                        .get("counter");

        // If TRUE, I'm the last mapper
        if(newValue == 0) {
            Unirest.post("SERVER_URL/reduce")
            .header("content-type", "application/json")
            .header("cache-control", "no-cache")
            .body("{\"totalMappers\":" + totalMappers
                    + ", \"topicName\":" + "\"" + topicName + "\""
                    + ", \"bucketName\":" + "\"" + bucketName + "\""
                    + "}").asJsonAsync();
        }
    }

    public static void produceCompletionRecord(String topicName,
```

```
                                                    int sequenceNumber,
                                                    String fileName) {

        // Configure properties for Kafka
        Properties props = new Properties();
        props.put("bootstrap.servers", "KAFKA_URL");
        props.put("acks", "all");
        props.put("retries", 0);
        props.put("batch.size", 16384);
        props.put("linger.ms", 1);
        props.put("buffer.memory", 33554432);

        KafkaProducer<String, String> producer =
                            new KafkaProducer<String, String>(props);
        producer.send(new ProducerRecord<String, String>(topicName,
                                                         sequenceNumber,
                                                         fileName));

        producer.close();
    }
}
```

Reducer.java

```
public class Reducer {

        public static JsonObject main(JsonObject args) {

        // Prepare response object
        JsonObject response = new JsonObject();

         // Parse parameters
        String userId;
        String topicName;
        String bucketName;
        int totalMappers;
        try {
            userId = args.getAsJsonPrimitive("userId")
                                                .getAsString();
            topicName = args.getAsJsonPrimitive("topicName")
                                                .getAsString();
            bucketName = args.getAsJsonPrimitive("bucketName")
                                                .getAsString();
            totalMappers = args.getAsJsonPrimitive("totalMappers")
                                                .getAsString();

        } catch(Exception e) {
           response.addProperty("Exception", "Incorrect parameter");
            return response;
        }

        // Configure properties for Kafka
        Properties props = new Properties();
        props.put("bootstrap.servers", "KAFKA_URL");
        props.put("group.id", "test");
        props.put("enable.auto.commit", "true");
        props.put("auto.commit.interval.ms", "1000");
        props.put("session.timeout.ms", "30000");

        KafkaConsumer<String, String> consumer =
                           new KafkaConsumer<String, String>(props);

        // Subscribe to Kafka topic
        consumer.subscribe(Arrays.asList(topicName));

        // Wait for all the messages to be received
        int i = 0;
        TreeMap<Integer, String> fileNames =
                                         new TreeMap<Integer, String>();
        while (true) {
```

```java
            ConsumerRecords<String, String> records =
                                consumer.poll(Long.MAX_VALUE);
            for (ConsumerRecord<String, String> record : records) {
                fileNames.put(Integer.parseInt(record.key()),
                                                record.value());
                i++;
            }
            if(i >= totalMappers) {
                break;
            }
        }

        // Connect to object storage
        MinioClient minioClient;
        try {
            minioClient = new MinioClient("MINIO_URL");
        } catch (InvalidEndpointException |
                 InvalidPortException e) {
            response.addProperty("Exception", "MinioException");
            return response;
        }

        // Prepare the streams for PDF concatenation
        Document document = new Document();
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        PdfCopy copy;
        try {
            copy = new PdfCopy(document, os);
        } catch (DocumentException e) {
            response.addProperty("Exception", "PdfCopy exception");
            return response;
        }
        document.open();


        // Loop over the documents to be concatenated
        int n;
        for (int sequenceNumber : fileNames.keySet()) {

            // Download a PDF document
            InputStream inStream;
            try {
                inStream = minioClient.getObject(bucketName,
                                                fileNames
                                                .get(sequenceNumber)
                                                );
            } catch (Exception e) {
                response.addProperty("Exception", "MinioException");
                return response;
```

```
            }
            PdfReader reader;
            try {
                reader = new PdfReader(inStream);
            } catch (IOException e) {
                response.addProperty("Exception", "Pdf exception");
                return response;
            }

            // Loop over the pages in that document
            // and add them to the final document
            n = reader.getNumberOfPages();
            for (int page = 0; page < n; ) {
                try {
                    copy.addPage(copy.getImportedPage(reader,
                                                        ++page));
                } catch (BadPdfFormatException | IOException e) {
                    response.addProperty("Exception", "PdfCopy");
                    return response;
                }
            }
            try {
                copy.freeReader(reader);
                reader.close();
                inStream.close();
            } catch (IOException e) {
               response.addProperty("Exception", "StreamException");
               return response;
            }
        }

        document.close();

        // Upload the concatenated file to the remote object storage
        ByteArrayInputStream bais = new ByteArrayInputStream(
                                            os.toByteArray());
        String concatenatedFileName = "CONCATENATED_FILE_NAME";
        try {
            minioClient.putObject(bucketName,
                                    concatenatedFileName,
                                    bais);
        } catch (Exception e) {
            response.addProperty("Exception", "Minio Exception");
            return response;
        }

        try {
            bais.close();
        } catch (IOException e) {
```

```
            response.addProperty("Exception", "Resource exception");
            return response;
        }

        response.addProperty("Action", "Completed");
        return response;
    }
}
```

6.4 Analisi del workflow

Una volta confermata la fattibilità del modello, l'analisi e le successive sperimentazioni su questo eseguite hanno mirato innanzitutto a una valutazione globale dell'intero workflow da un punto di vista del modello di programmazione, confrontandolo con un modello sequenziale più standard. La prima cosa da evidenziare è che una struttura parallela in MapReduce, rispetto a un normale programma sequenziale, necessita di meccanismi di comunicazione e sincronizzazione non banali, in questo caso realizzati tramite l'ausilio di broker di eventi e database. Aggiungere tali componenti al processo implica senz'altro un aumento della complessità, che si riversa in uno sviluppo e un debugging meno agevole rispetto a una più semplice struttura sequenziale. Il problema principale sta appunto nel dover organizzare le entità in gioco in modo che queste possano ricongiungersi nella fase finale, nonostante siano distribuite su nodi dislocati e lascamente connessi. Entrando più nella specifica implementazione, la conseguenza di questo vincolo è che ogni *mapper* deve in qualche modo segnalare il proprio completamento al broker e salvare sullo storage il risultato parziale, ovvero la pagina PDF prodotta, prima di poter terminare. Al contrario, un servizio tradizionale può mantenere in memoria tutte le pagine, per poi concatenarle come ultima istruzione della sua esecuzione.

Da questo punto di vista, il modello di programmazione sequenziale risulta certamente più lineare e semplice, ma soprattutto con una probabilità di errore ridotta. A conferma di ciò, si è notato che, testando il workflow con template semplici e di dimensioni relativamente piccole, nell'ordine dei Kb, non è possibile apprezzare i vantaggi del modello MapReduce realizzato. Le due alternative arrivano perlopiù ad eguagliarsi, sia nei tempi di esecuzione che nelle performance, rendendo senz'altro preferibile la semplicità di un'architettura sequenziale. Tuttavia, bisogna considerare che, nel caso di documenti nell'ambito FinTech, un report contiene solitamente grafici, immagini e tabelle che vanno ad appesantire sensibilmente sia i template che le operazioni di compilazione. In una situazione simile, il dover mantenere in memoria significativi stream di byte per la durata dell'intero processo, senza poterli scaricare in una fase intermedia, unito anche alla latenza per il reperimento dei template da uno storage,

diventa facilmente un collo di bottiglia per l'esecuzione sequenziale, specie a fronte di un numero elevato di richieste concorrenti. Per quanto il servizio sequenziale possa essere replicato, essendo ogni singola istanza particolarmente *time-consuming* e *memory-consuming*, si arriverà prima o poi a un punto in cui le richieste dovranno essere accodate, dilatando ulteriormente i tempi di attesa.

Un altro fattore da considerare, in particolare per quanto riguarda la sola fase di compilazione dei template, è la variabilità del numero di template da elaborare. I documenti, infatti, a seconda dell'utente che li richiede o della tipologia della richiesta, possono essere composti da più o meno pagine. Questo dettaglio espone immediatamente un grande limite del programma sequenziale, i cui tempi di esecuzione aumenterebbero proporzionalmente al numero di pagine di elaborare, cosa che invece non accade in una struttura parallela, in cui si andrebbe semplicemente a partizionare il lavoro su un numero maggiore di istanze. Il modello realizzato va infatti a invocare una funzione per ogni pagina, dunque, una pagina in più da elaborare corrisponde soltanto a una funzione addizionale, che, eseguendo in parallelo con le altre, andrà a terminare il processo senza incidere sul tempo di esecuzione globale. Uno dei grandi vantaggi sta quindi proprio nel fatto che il MapReduce serverless può garantire un tempo di esecuzione che sia idealmente indipendente dalla lunghezza del PDF da generare.

Un terzo vantaggio consiste invece nella possibilità, che il modello offre nativamente e senza alcuna modifica, di dividere l'intero processo in due momenti diversi. Questo significa che, all'occorrenza, si potrebbe decidere di elaborare gli output parziali *out-of-band* per poi ricomporli solo in un secondo momento, con una separazione nello spazio e nel tempo.

Infine, una quarta e ultima feature della struttura MapReduce è il poter avviare idealmente la fase di reduce anche prima che la fase di map sia stata completata. Ciò non solo è facilmente realizzabile, ma va addirittura a semplificare il modello, eliminando la necessità del punto di sincronizzazione e dell'annessa logica del contatore. Basterebbe infatti far partire il componente *reducer* all'inizio dell'intero processo, così che, non appena nel topic arriva l'indicazione di completamento da parte del primo *mapper*, il *reducer* possa già elaborare il primo risultato parziale, e così via fino a completare l'intero documento pochi istanti dopo la terminazione dell'ultimo *mapper*. Ovviamente un comportamento del genere non è applicabile al caso sequenziale, in cui è necessario attendere l'elaborazione di tutte le pagine, prima di poterle mettere insieme.

6.5 Descrizione dei test

Come detto, l'effettiva differenza dei modelli sta nella fase centrale di compilazione dei template. La parte iniziale di scaricamento delle informazioni da imprimere nei template

è analoga nei due casi, con il solo e poco significativo overhead, nello scenario parallelo, per attivare le funzioni *mapper*. In particolare, i valori raccolti nelle sperimentazioni mostrano un ritardo addizionale $\Delta t \leq 1s$ per scaturire l'esecuzione delle FaaS. Anche la parte finale di concatenamento è molto simile, ma col vantaggio, nel modello sequenziale, di non dover reperire i risultati parziali, avendoli già pronti in memoria. Tuttavia, è già stata presentata una versione alternativa del *reducer*, che effettua un'elaborazione concorrentemente ai *mapper*, andando a favore dello schema parallelo.

Per dare un'idea più concreta dei vantaggi che si ottengono col modello MapReduce si riportano quindi alcuni risultati ottenuti dall'esecuzione isolata della fase di mapping, messa a confronto con le medesime operazioni eseguite in modo sequenziale. Per una comparazione più oggettiva, si è deciso di effettuare il deployment del servizio sequenziale sullo stesso cluster delle funzioni *mapper*, replicandolo su più nodi per capire fino a che punto esso riesca a servire più richieste contemporaneamente, prima di cominciare ad accodarle. Poiché, come già esposto, le differenze sono realmente apprezzabili solo a fronte di documenti almeno di medie-grandi dimensioni, i test sono stati effettuati su template di singole pagine PDF da circa 1Mb l'uno, contenenti immagini e grafici, fino a formare documenti composti dalle 3 alle 100 pagine: scelte giustificate sia dal bisogno di comprendere qual è il collo di bottiglia dei sistemi, ma anche da una rapida indagine sui diversi tipi di documentazione scaricabili da siti di home banking.

In particolare, i test che si sono susseguiti hanno puntato a una valutazione dei modelli andando a insistere prima sul numero di pagine da elaborare e poi sul numero di richieste inviate (numero di utenti fittizi generati), al fine di trovare il punto a partire dal quale il MapReduce parallelo risulta più performante del sequenziale. Le tre fasi di sperimentazione sono state le seguenti:

1) Aumento graduale del numero delle pagine da 3 a 100, ma inviando una sola richiesta (un solo utente).
2) Aumento graduale del numero delle richieste (utenti) da 3 a 100, mantenendo costante a 10 il numero di pagine da elaborare; si è scelto questo valore in quanto per documenti di ridotta dimensione non si sono apprezzate particolari differenze.
3) Aumento graduale della frequenza di invio delle richieste.

Il punto focale dei test è stata la misurazione dei tempi di esecuzione delle varie operazioni richieste, prevedendo risultati migliori per il servizio sequenziale fino a che il carico in ingresso non lo costringa ad accodare le richieste, con conseguente dilatazione dei tempi ed errori di timeout.
Durante l'esecuzione dei cicli di test sono inoltre stati monitorati alcuni aspetti legati al numero di errori riportati, al consumo di memoria, ai tempi di avvio delle operazioni e al fenomeno dei *cold start*.

Per completezza i test sul modello MapReduce sono stati sempre effettuati in doppia copia, una volta considerando la sola elaborazione dei PDF ("*without upload*" nei grafici che seguiranno), alla stregua del servizio sequenziale, e una volta con annesso caricamento dei risultati parziali sull'object storage di destinazione ("*with upload*" nei grafici che seguiranno). L'object storage in questione è ovviamente *MinIO*, di cui è stato fatto il deployment locale al cluster, riducendo drasticamente la latenza in upload e download. In questo modo si riescono anche ad avere dei risultati più oggettivi che non sono influenzati da fattori esterni o delay per le comunicazioni remote.

Per l'invio delle richieste è stato sfruttato anche in questo caso lo strumento *Apache JMeter*, rivolgendosi al medesimo cluster di 5 nodi utilizzato nei test visti precedentemente e la cui scheda tecnica è presentata in Tabella 3.

6.6 Risultati

Vengono dunque riportati i grafici che riassumono e schematizzano i risultati ottenuti nella fase di mapping. Come detto, questi mettono a confronto gli andamenti del programma sequenziale con quelli del MapReduce, proposto nelle due versioni differenti, con e senza upload del risultato parziale. Da un punto di vista pratico, la comparazione più oggettiva e corretta pretenderebbe l'osservazione della sola versione con upload, tuttavia è stato reputato di interesse avere una visione sull'esecuzione anche a prescindere da questo dettaglio, in quanto fattore variabile e dipendente dal servizio di persistenza che si sceglie di utilizzare.

6.6.1 Risultati test con pagine incrementali

Nel grafico in Figura 58 viene presentato il tempo di esecuzione, misurato in secondi, nel caso di una singola invocazione per la compilazione di documenti di complessità crescente, da un minimo di 3 pagine, fino a un massimo di 100. I tempi riportati per il MapReduce corrispondono ai tempi massimi registrati all'interno di ogni gruppo di *mapper* lanciato, ricordando che a 3 pagine corrispondono un gruppo di 3 funzioni *mapper*, a 5 pagine un gruppo di 5 funzioni e così via.

Alla luce di quanto detto, il confronto significativo sta tra il modello sequenziale, in blu nel grafico, e il MapReduce con upload, in rosso, in quanto, sebbene il modello sequenziale non effettui alcun upload verso l'object storage di destinazione, è giusto che il modello parallelo venga inficiato da questo dettaglio operativo, essendo il salvataggio dei risultati parziali uno svantaggio intrinseco delle strutture MapReduce da tenere in considerazione.

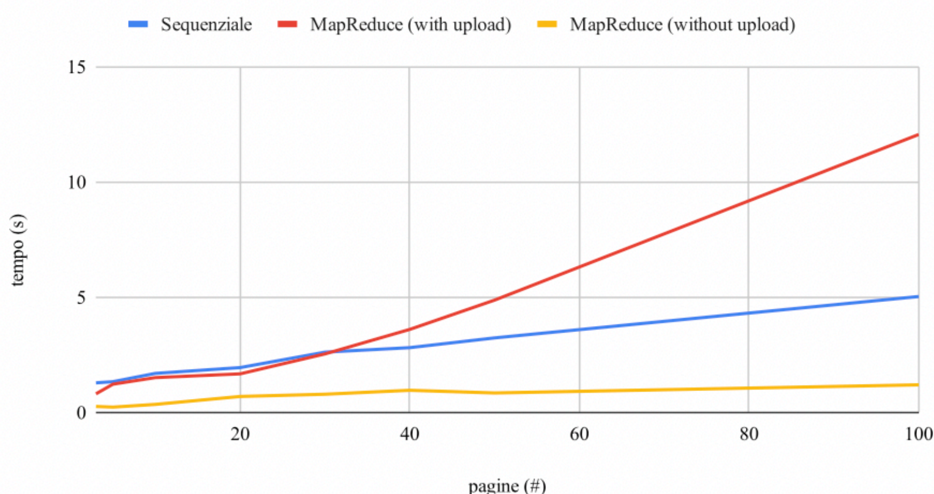

Figura 58 - Tempi di esecuzione con pagine incrementali

Tuttavia, la linea in giallo relativa alla sola compilazione dei template è stata ugualmente riportata per evidenziare come, in assenza di upload dei dati, il modello parallelo presenta una certa costanza, senza risentire in alcun modo del numero crescente delle pagine da elaborare, come del resto era stato ipotizzato precedentemente. Questo deriva dal fatto che un numero di pagine maggiore corrisponde a un numero di funzioni maggiori da avviare, che però, eseguendo in parallelo, non vanno a influenzare il tempo globale di esecuzione, rimasto sempre al di sotto della soglia del secondo. Questo non avviene nel caso con upload, soprattutto a causa della concorrenza negli accessi allo storage, risolvibile, in uno scenario di produzione, fornendo un'infrastruttura di storage dedicata e più avanzata di quella disponibile in fase di sviluppo.

Tornando al confronto tra gli altri due andamenti, è possibile notare come, in una situazione in cui vi è un numero limitato di utenti che richiede la generazione di documenti di importanti dimensioni, il paradigma sequenziale risulti ancora preferibile in quanto a prestazioni e semplicità guadagnata. Il problema sta appunto nella difficoltà maggiore che riscontra il modello MapReduce nell'effettuare il salvataggio concorrente di una quantità di dati sempre maggiore nel volume Kubernetes, che si riversa in un andamento che cresce di circa 1 secondo ogni 10 pagine addizionali, rispetto a quello del sequenziale che è aumentato mediamente di solo mezzo secondo tra un test e l'altro.
La percentuale di errori riscontrata viene qui omessa, in quanto, in questa fase, non si sono mai presentati fallimenti delle richieste.

Per quanto concerne invece l'occupazione di memoria e l'utilizzo della banda, questi sono stati simili nei due casi, ma con la differenza che, nel caso del MapReduce, sono stati ripartiti su tutti i nodi del cluster, anziché gravare sui soli nodi in cui si trovava il servizio

sequenziale. Nella Figura 59 si può osservare questo comportamento, considerando che, seppure il servizio sequenziale venga replicato a mano su più nodi, il meccanismo delle FaaS in OpenWhisk, che prevede l'automatica distribuzione delle operazioni su potenzialmente tutti i nodi del cluster, risulta comunque molto più comodo ed efficace.

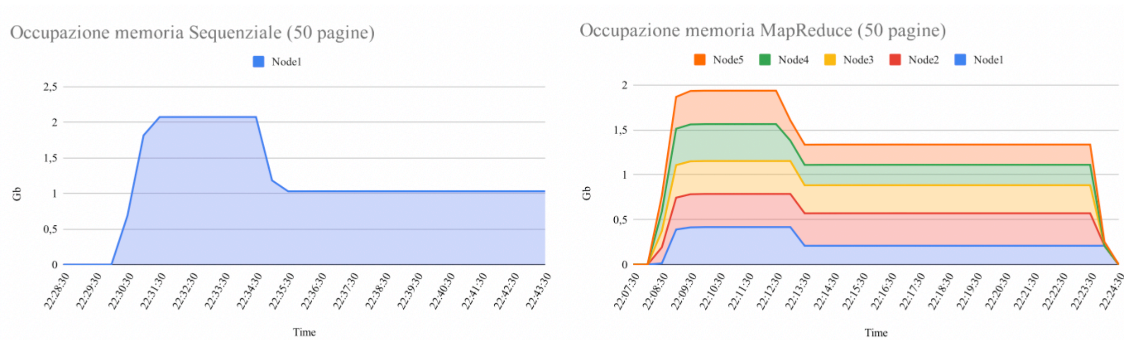

Figura 59 - Confronto occupazione memoria

Infine, effettuando uno zoom sui risultati ottenuti, in Figura 60 viene mostrata la differenza nei tempi di esecuzione dei soli *mapper*, nel caso esemplificativo di elaborazione di 10 pagine, rispettivamente con e senza *cold start*.

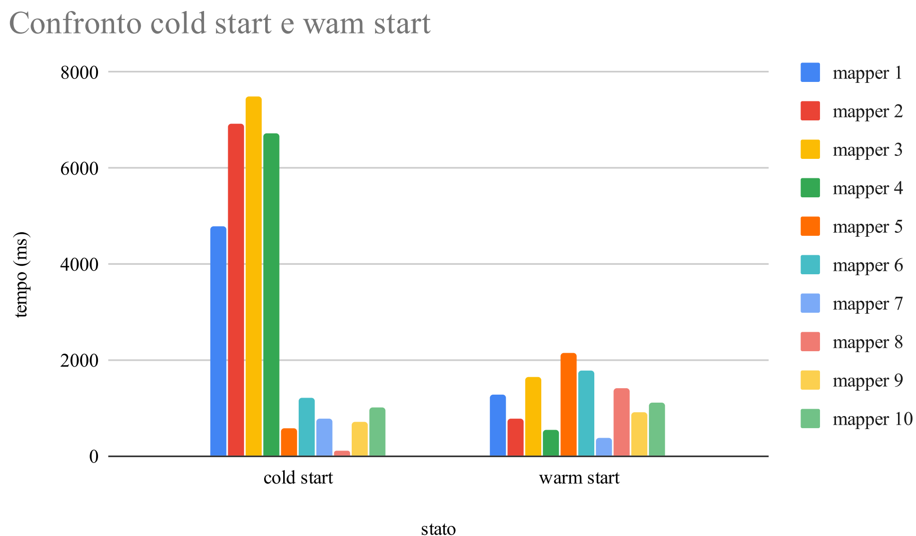

Figura 60 - Confronto cold start e warm start

Questa comparazione è realizzabile effettuando due richieste identiche in successione, in modo che la prima necessiti della creazione preliminare di container per eseguire le

funzioni, mentre la seconda trovi già pronte tutte le risorse di cui ha bisogno. La differenza nei due casi è significativa, con il primo gruppo di *mapper*, sulla sinistra, che ritarda la sua esecuzione a causa dello *startup delay* dovuto all'istanziazione dei nuovi pod. In particolare, 4 delle 10 funzioni vengono messe in attesa, mentre per le restanti 6 vengono sfruttate repliche già generate nel test precedente.
In generale, ognuna delle funzioni in *warm* termina le proprie operazioni nel giro di pochi millisecondi, da un minimo di *119ms* a un massimo di *2136ms*, mentre il tempo necessario alla creazione dei container con l'environment Java richiesto comporta un sostanziale ritardo nell'esecuzione globale, da un minimo di circa *4,5s* fino a un massimo di *7,5s*. Ad ogni modo, l'influenza dei *cold start* è stata variabile durante le sessioni di test, motivo per cui si è deciso di riportare, alla fine di questa sezione, un'indagine più approfondita e globale del fenomeno.

Alla luce di quanto visto dopo questa prima fase di test, si può affermare che le FaaS non sono particolarmente adatte a implementare operazioni richieste raramente, in quanto si incapperebbe sempre nel fenomeno dei *cold start* che ne dilata i tempi di esecuzione, mentre il loro punto di forza sta nello smaltire importanti flussi di richieste ravvicinate, tipici degli scenari industriali, grazie all'ottimo livello di scalabilità raggiungibile nel serverless. È proprio sul numero di richieste che va ad insistere la successiva sessione di test.

6.6.2 Risultati test con richieste incrementali

Poiché fintanto che si è lavorato su documenti di media complessità, intorno alle 10 pagine, nei test precedenti i due modelli hanno ottenuto valori comparabili, si è deciso, in questa seconda fase di test, di mantenere fissata questa quantità e agire stavolta sul numero di utenti, andando a stressare i sistemi con carichi gradualmente maggiori, partendo da 3 richieste concorrenti fino ad arrivare a 100.
In Figura 61 vengono riassunti i risultati ottenuti.

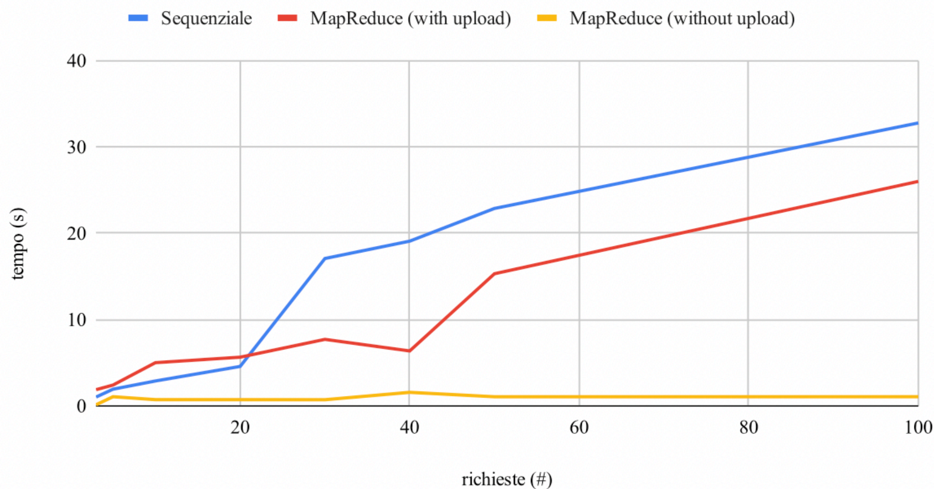

Figura 61 - Tempi di esecuzione con richieste incrementali

Partendo ancora una volta dall'analisi del caso isolato di MapReduce senza upload, di nuovo in giallo nel grafico, sono sempre più evidenti i vantaggi dell'elaborazione parallela dei documenti, che risulta dunque essere non solo indipendente dal numero di pagine, ma anche dal numero di utenti. Anche in questo caso, nessuna delle esecuzioni è mai durata più di un secondo. Un risultato molto convincente, che già di per sé potrebbe spingere verso l'adozione di un modello sequenziale.

Oltretutto, in questo caso, risulta essere preferibile perfino il modello parallelo con annesso upload dei risultati, effettuato in modo sufficientemente rapido da consentire di terminare l'esecuzione prima della controparte sequenziale. Quest'ultimo, infatti, seppure si comporti inizialmente meglio, fino al test con 20 richieste, è messo sotto sforzo dall'elevato numero di utenti nelle sessioni di test immediatamente successive, riscontrando un ripido aumento nei tempi già nella gestione di 30 richieste. Una tale dilatazione temporale è dovuta a un inevitabile accodamento delle richieste, non essendo il servizio capace di scalare con la medesima elasticità delle FaaS che implementano i *mapper*, seppure esso sia stato replicato su più nodi per rendere il confronto formalmente più corretto e realistico.
Osservando il grafico, è inoltre degno di nota il comportamento del MapReduce in corrispondenza del test con 40 richieste. È infatti possibile notare una riduzione dei tempi, rispetto al caso che lo precede. Il motivo è, anche in questo caso, dovuto ai *cold start* che, non essendosi verificati in quella particolare situazione, hanno permesso di raggiungere un tempo globale inferiore.

Questa fase di test è stata particolarmente significativa, perché ha permesso di trovare il punto a partire dal quale il modello realizzato va effettivamente a diventare preferibile al

modello standard sequenziale, non soltanto da un punto di vista dei tempi di esecuzione, ma anche e soprattutto da un punto di vista della correttezza nel servizio delle risposte. In questo caso, infatti, l'incremento del numero degli utenti fittizi è arrivato persino a causare alcuni errori di esecuzione, distinguibili in timeout delle richieste a causa degli accodamenti e in fallimenti nell'istanziazione dei container a causa di un'occupazione eccessiva delle risorse.
In Figura 62 si può osservare il report sul numero degli errori riscontrati nelle varie fasi del test, a seconda del numero di richieste effettuate.

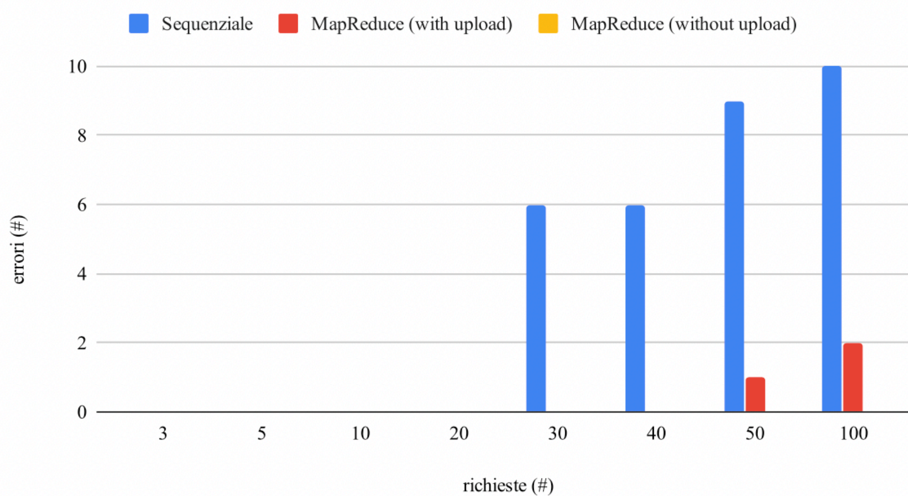

Figura 62 - Numero di errori con richieste incrementali

Senza l'overhead del salvataggio dei risultati parziali, il MapReduce privato del meccanismo di upload ha concluso l'intera schiera di test senza mai riportare errori. Per quanto riguarda invece la versione completa, in rosso, gli unici errori sono stati riportati negli ultimi due test, i più critici in termini di occupazione delle risorse: un solo *mapper* fallito su 500, nel test da 50 richieste per 10 pagine, e 2 *mapper* falliti su 1000, nel test da 100 richieste per 10 pagine. Le percentuali di errore sono dunque prossime allo zero e ciò che le ha causate è stato semplicemente l'eccessivo sovraccarico sul cluster, che non ha potuto rispondere al bisogno di creare un numero così elevato di container: errori dunque facilmente evitabili dotandosi di un cluster con capacità maggiori.
Non si può fare lo stesso discorso per il sequenziale, che invece è stato affetto da una percentuale considerevole di errori, tra il 10% e il 15%. In questo caso la maggior parte degli errori è stato dovuto all'occorrenza di multipli *response timeout* (fissati a 120s nei test). La ragione è da trovarsi nell'accodamento delle richieste che si è verificato a partire

dal test con 30 richieste, che si riflette tra l'altro nella crescita improvvisa dei tempi, visualizzabile in Figura 61 proprio in corrispondenza delle 30 richieste.

In generale in questa seconda fase di test, per quanto riguarda l'occupazione della memoria, si sottolinea che il numero crescente di documenti e pagine da elaborare si riflette, nel MapReduce, in un numero crescente di *mapper*, e quindi di container da istanziare, il che comporta un consumo della memoria maggiore rispetto al caso sequenziale, ma comunque equamente distribuito sui nodi. Mediamente è stato osservato un consumo del MapReduce pari a circa 2-3 volte quello del programma sequenziale. Anche gli effetti di questo aspetto negativo si risolverebbero, in un eventuale scenario di produzione, utilizzando un cluster di dimensioni maggiori che possa smistare più efficacemente il carico.

6.6.3 Risultati test con frequenze maggiorate

In tutti i test presentati, è stata impostata una frequenza di invio delle richieste pari a *1 req/s*. Un'ultima fase di test ha tentato di evidenziare eventuali cambiamenti andando aumentare tale frequenza. In generale, non sono state evidenziate particolari differenze, se non un carico leggermente maggiore sui nodi e una dilatazione, seppur minima, dei tempi. In particolare, dato il ridotto numero di operazioni che deve eseguire ogni *mapper*, un aumento nella frequenza delle richieste non ha comportato criticità. La maggiore difficoltà che ha invece incontrato il modello sequenziale nel gestire importanti flussi di richieste in ingresso ha fatto mantenere i valori negativi riportati nel test precedente. Date le marginali differenze, non è risultata di interesse un'analisi più approfondita di questo scenario.

Va infine fatta un'ultima considerazione generale che riguarda i tempi registrati nel MapReduce per tutti i test appena osservati. Sebbene la decisione di considerare il tempo massimo ottenuto in ogni gruppo di *mapper* sia formalmente la più corretta, è giusto sottolineare che, spesso, questa ha portato a registrare valori piuttosto elevati a causa anche di un solo *mapper* che, per motivi di sovraccarico o per *cold start*, ha terminato l'esecuzione molto dopo degli altri, influenzando negativamente il tempo globale. In particolare, per dare un'idea orientativa, si è visto che mediamente, in un gruppo di *mapper*, circa l'80-85% riusciva a completare il proprio lavoro tra i *200ms* e i *700ms*, mentre solo il restante 15-20% necessitava di secondi addizionali. Questa stima va dunque ulteriormente a far propendere verso il modello parallelo, tendenzialmente capace di terminare i propri task in tempi notevolmente minori rispetto alla controparte sequenziale.

6.6.4 Indagine sui cold start

Nel grafico in Figura 60 è ben visibile la disomogeneità presente tra le esecuzioni dei singoli *mapper*, derivante dal problema dei *cold start* che affligge le FaaS, come già esposto nelle sezioni precedenti. Più che come un problema esso andrebbe visto come una conseguenza necessaria e inevitabile se si vuole abbracciare a pieno il concetto di risorse attivate a runtime per il solo tempo di esecuzione e successivamente rilasciate. Ad ogni modo, per dare un'idea quantitativa, durante questa fase di sperimentazioni sono stati misurati i tempi di attivazione per le varie funzioni, dividendo i *cold start* e i *warm start*, anche grazie all'ausilio della piattaforma OpenWhisk che specifica, per ogni richiesta, se questa è stata presa in carico da una replica già presente (stato di *warm*) o da una nuova replica istanziata sul momento (stato di *cold*).

Non è semplice fornire un'indicazione precisa sulla differenza tra i due stati, in quanto i tempi di avvio risultano perlopiù variabili e influenzati da fattori di varia natura che riflettono la situazione dell'intera infrastruttura, a partire dal livello di occupazione di memoria nell'intero cluster, fino alle condizioni di rete e connessione, poiché, per istanziare un container, è spesso necessario scaricare prima l'immagine corrispondente da una repository. Tuttavia, è possibile effettuare una stima orientativa che riassuma i comportamenti osservati. Le indagini hanno condotto ai risultati mostrati nella Tabella 4, in cui sono indicati, misurati in secondi, alcuni valori limite registrati nelle numerose prove realizzate.

	MIN (solo avvio)	MAX (solo avvio)	MIN (esecuzione completa)	MAX (esecuzione completa)
Cold	1,02s	2,89s	1,22s	7,47s
Warm	3ms	6ms	87ms	5,75s

Tabella 4 - Indagine sui cold start nei mapper

I valori delle ultime due colonne si riferiscono ai tempi rispettivamente minimi e massimi impiegati per l'elaborazione di una singola pagina, senza sovraccarico causato dalle altre richieste e con una connessione allo storage locale. L'intento infatti è stato soltanto quello di valutare l'impatto dei *cold start*, isolando il più possibile lo scenario da influenze esterne. Le prime due colonne derivano invece da un'analisi atta a individuare i tempi minimi e massimi relativi ai soli avvii dei pod, senza l'aggiunta del tempo di effettiva esecuzione. La considerazione che ne deriva, specie osservando il divario tra i minimi e i relativi massimi, è che il *cold start* non è qualcosa che interessa esclusivamente la creazione del container, ma che, in caso di operazioni da eseguire e connessioni da

instaurare, l’avere un ambiente già stabilizzato da tempo (*warm*) può modificare in modo sostanziale l’intera esecuzione rispetto a un ambiente appena avviato (*cold*).
Valori approssimativamente simili sono stati ottenuti nell’analisi dei tempi di avvio del *reducer*, il che è comprensibile data l’omogeneità che intercorre tra i due ambienti: entrambi Java, dipendenti dal medesimo set di librerie, con connessioni allo stesso storage e allo stesso broker di eventi.

7 Conclusioni e sviluppi futuri

Molteplici ragioni, tecnologiche e sociali, hanno guidato la radicale trasformazione dei sistemi IT nel corso degli ultimi decenni. Spesso i processi innovativi in questo settore hanno completamente modificato le modalità con cui un dato servizio viene sviluppato, esposto e gestito. Altre volte, l'innovazione ha condotto alla creazione di strumenti, come il serverless computing, che non si pongono come sostituti di ciò che è già presente negli scenari distribuiti, ma come integrazione protesa verso l'elasticità, l'astrazione e la scalabilità dei sistemi.

Il serverless è attualmente il modello di servizio cloud con la diffusione più rapida, con un tasso di crescita annuale del 75%, in base a quanto riportato da *RightScale's 2018 State of the Cloud* [100], predicendo un'espansione ancora maggiore negli anni a venire, date le numerose adozioni avute nel 2019. Nel 2020 gli esperti prevedono innanzitutto un miglioramento nella sicurezza serverless, poiché questa non rispecchia ancora i canoni richiesti dalle grandi aziende del settore. Inoltre, si assisterà ad una semplificazione degli strumenti di sviluppo, monitoraggio e testing di applicazioni serverless, data la nota complessità nel controllo di architetture distribuite. Il tutto andrà certamente a beneficio delle piattaforme FaaS presenti attualmente sulla scena, sia proprietarie che open source, destinate a diventare sempre più complete e capaci di offrire un'esperienza d'uso che possa davvero far dimenticare agli sviluppatori la presenza dell'infrastruttura server.

In questo lavoro, i vantaggi del serverless, in particolar modo della componente FaaS nel mondo open source, sono stati analizzati da un punto di vista teorico per poi essere osservati in sperimentazioni che hanno abbracciato diversi aspetti e criticità di un'applicazione distribuita. Le rispettive indagini su scalabilità, consumo delle risorse e livelli di astrazione offerti dalle diverse piattaforme disponibili hanno orientato la selezione di un framework FaaS che fosse idealmente applicabile al caso d'uso, nell'ambito FinTech, per la generazione parallela di documenti.

In particolare, la scelta della piattaforma, ricaduta su OpenWhisk, nonostante in alcuni punti vada contro ciò che è stato mostrato dai test iniziali sulle performance, è stata motivata dall'importante supporto che il framework ha alle spalle, dalla presenza di componenti ben consolidati e adeguati alla modellazione del caso d'uso, da un utilizzo elastico della memoria, particolarmente adatto alla situazione esaminata e dalla promessa di un'integrazione con la piattaforma Knative. Quest'ultima, infatti, sarebbe stata la candidata eletta, sia per il livello di performance che riesce a raggiungere, sia per il modello orientato verso la standardizzazione dei workflow serverless su Kubernetes. Tuttavia, complici l'ancora lontana maturità e un'architettura volutamente generica, scarna e aperta all'inserimento di componenti e commodity esterne, la rendono poco fruibile per lo sviluppo di un'intera applicazione, in termini di tempi e costi.
Una volta che il processo di integrazione di Knative con la piattaforma OpenWhisk sarà

terminato e perfezionato, si potrà approfittare del doppio vantaggio di avere un sistema completo, ricco di funzionalità e realmente capace di rendere trasparente la presenza dei server, che sia allo stesso tempo poggiato al di sopra di un layer standard ed efficiente per l'esecuzione di processi serverless.

La modellazione della struttura MapReduce, realizzata tramite piattaforme FaaS open source, non ancora presente in letteratura, ha permesso di fornire un servizio reattivo e affidabile, capace di completare i propri task in tempi di esecuzione contenuti e teoricamente indipendenti sia dalla dimensione che dal numero dei documenti da elaborare, dimostrando i forti vantaggi che si ottengono rispetto a un sistema sequenziale.

Sebbene la prima fase di test, caratterizzata da una sola richiesta per documenti grandi, non abbia fatto risultare il modello sviluppato come preferibile, nelle ultime due fasi, con numerose richieste per documenti di medie dimensioni, si sono mostrati i vantaggi del modello parallelo. Considerando che in uno scenario di FinTech è tendenzialmente più probabile che si presentino situazioni con molti utenti che richiedono un numero limitato di pagine, rispetto a pochi utenti che ne richiedono invece un numero elevato, si ha la conferma che il modello MapReduce serverless realizzato è adatto allo scope individuato e che riesce a scalare proprio in base alla metrica che serve in questo ambito, ovvero in base al numero di utenti.

Sono stati inoltre proposti due modelli alternativi, come spunto per uno sviluppo successivo. Il primo di questi potrebbe migliorare significativamente il tempo globale di esecuzione del MapReduce, puntando su un'esecuzione parallela delle fasi di map e reduce. Oltre all'ottimizzazione in termini di performance, questa soluzione porterebbe anche una profonda semplificazione dell'intera struttura e delle sue interazioni, eliminando la necessità del punto di sincronizzazione.
Una seconda possibilità sarebbe quella di separare le due fasi del MapReduce in momenti distaccati, con *mapper* che possono quindi lavorare a prescindere dalle richieste in ingresso, e con un attivazione del *reducer* soltanto all'atto dell'effettiva richiesta real-time fatta da un utente. Ciò può risultare particolarmente utile nel caso in cui, in un ipotetico scenario di produzione, si preveda un aumento imminente del carico in ingresso, magari a fronte dell'aggiunta di un nuovo tipo di documentazione scaricabile da un sito di smart banking: il sistema aziendale in questione potrebbe far partire l'elaborazione *out-of-band* delle singole pagine, così da smaltire più agilmente l'intenso traffico di richieste, quando queste cominceranno ad arrivare, attivando esclusivamente i *reducer* che andranno a concatenare i risultati parziali già pronti.

Naturalmente, il vantaggio di un modello capace di scalare con una tale efficienza a fronte di numeri significativi di richieste non è da limitarsi al solo caso d'uso analizzato. Anche soltanto rimanendo all'interno dell'ambito FinTech, la stessa struttura potrebbe essere estesa a diversi altri bisogni, primo fra tutti il potenziamento delle app di *budgeting*,

utilizzate per tenere traccia delle spese e dei risparmi degli utenti, e la cui popolarità è cresciuta esponenzialmente con l'introduzione dello smart banking. Grazie all'elasticità delle FaaS, sarebbe infatti possibile, ad esempio, elaborare in parallelo liste di movimenti e pagamenti provenienti da sorgenti diverse, per poi ridurle a report che possano riassumere la situazione finanziaria di quantità considerevoli di utenti concorrenti, evitando colli di bottiglia, dilatazione dei tempi o saturazione delle risorse.

Anche al di fuori degli scenari finance, i casi d'uso caratterizzati dalla necessità di scalare in base al numero di interazioni sono innumerevoli e tutti trarrebbero enormi vantaggi dall'abbracciare la filosofia serverless. Si pensi ad esempio al batch processing di Big Data, con la possibilità smaltire agilmente task computazionalmente intensi, o ancora allo stream processing, particolarmente compatibile con la natura event-driven serverless: ogni evento presente in uno stream di dati potrebbe lanciare una nuova funzione, avviando così elaborazioni parallele in *bulk*, a ritmi serrati e con impatto minimo sul sistema.
Rimanendo sempre nell'ambito dei Big Data, lo stesso raccoglimento di tali dati da parte di dispositivi mobili o sensori IoT beneficerebbe di un'architettura altamente scalabile e a basso costo, considerando sia l'elevata frequenza di interazioni a cui questi sono sottoposti, ma soprattutto il fatto che il serverless possa essere implementato anche su infrastrutture semplici e senza particolari caratteristiche tecniche.

Ulteriori interessanti campi di applicazione si possono trovare negli scenari di image processing, con la possibilità di trasformazione parallela di file multimediali di grandi dimensioni, nelle logiche ad eventi di database auditing, nelle pipeline di *Continuous Integration and Continuous Deployment* (CI/CD), e perfino nella sfera dell'intelligenza artificiale, che ha ricevuto particolari attenzioni nell'ultimo anno. Sempre più assistenti vocali e sistemi di chat bot stanno infatti cominciando ad affacciarsi al paradigma serverless per offrire servizi elastici e leggeri, che possano giovare della scalabilità del serverless, visto l'aumento esponenziale degli utilizzatori da dover gestire.

I modelli e i casi d'uso citati, specie quelli di natura industriale, potrebbero in aggiunta venire affiancati da soluzioni cloud ibride, ottimizzando al massimo le performance nel privato e migliorando ulteriormente la scalabilità delle risorse del pubblico. La continua evoluzione del cloud verso implementazioni ibride, atta a combinare i vantaggi dei cloud pubblici con quelli dei cloud privati, è infatti già facilitata da Kubernetes, in grado di rendere omogenei più ambienti e trasparenti le interazioni tra questi. In questo contesto, il serverless si sta ponendo come il tassello mancante dell'hybrid cloud, astraendo definitivamente le complesse architetture tradizionali, con livelli di portabilità più che soddisfacenti, favorendo dunque la realizzazione di nuove strategie di business, con processi più economici, efficienti e rapidi.

In definitiva, crediamo che il serverless riuscirà a consolidare la propria presenza sul mercato, una volta superate le criticità attualmente presenti, continuando a riscuotere

successo in importanti scelte di produzione, spesso affiancando o sostituendo i sistemi tradizionali di diverse aree applicative, con un'ottimizzazione nell'utilizzo delle risorse di calcolo.

Bibliografia

[1] «Lambda - Serverless Compute,» [Online]. Available: https://aws.amazon.com/lambda/. [Consultato il giorno 22 12 2019].

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

[3] «What is cloud computing,» [Online]. Available: https://aws.amazon.com/it/what-is-cloud-computing/. [Consultato il giorno 21 01 2020].

[4] M. Raza, «Public Cloud vs Private Cloud vs Hybrid Cloud: What's The Difference?,» 12 09 2018. [Online]. Available: https://www.bmc.com/blogs/public-private-hybrid-cloud/. [Consultato il giorno 26 01 2020].

[5] S. Goyal, «Public vs Private vs Hybrid vs Community - Cloud Computing: A Critical Review,» 2014.

[6] «The Twelve-Factor App,» [Online]. Available: https://12factor.ne. [Consultato il giorno 23 12 2019].

[7] «Types of cloud computing,» [Online]. Available: https://aws.amazon.com/it/types-of-cloud-computing/?WICC-N=tile&tile=types_of_cloud. [Consultato il giorno 21 01 2020].

[8] «Serverless,» [Online]. Available: https://aws.amazon.com/serverless/. [Consultato il giorno 04 02 2020].

[9] «Serverless framework,» [Online]. Available: https://serverless.com. [Consultato il giorno 04 02 2020].

[10] «Auth0,» [Online]. Available: https://auth0.com. [Consultato il giorno 04 02 2020].

[11] «A Cloud Guru,» [Online]. Available: https://acloud.guru. [Consultato il giorno 04 02 2020].

[12] «I vantaggi del serverless computing,» [Online]. Available: https://www.redhat.com/it/topics/cloud-native-apps/what-is-serverless. [Consultato il giorno 04 02 2020].

[13] «What is BaaS? Backend-as-a-Service vs. Serverless,» [Online]. Available: https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/. [Consultato il giorno 01 02 2020].

[14] M. Fowler, «Serverless Architectures,» [Online]. Available: https://martinfowler.com/articles/serverless.html#unpacking-faas. [Consultato il giorno 06 01 2020].

[15] R. Ganguly, «The Definitive Guide to Serverless Architectures,» [Online]. Available: https://serverless.com/blog/definitive-guide-to-serverless-architectures/. [Consultato il giorno 27 01 2020].

[16] M. Roberts, «Serverless Architectures,» 22 05 2018. [Online]. Available: https://martinfowler.com/articles/serverless.html. [Consultato il giorno 25 01 2020].

[17] «Docker - Enterprise container platform,» [Online]. Available: https://www.docker.com/. [Consultato il giorno 22 12 2019].

[18] «Swarm mode overview,» [Online]. Available: https://docs.docker.com/engine/swarm/. [Consultato il giorno 18 09 2019].

[19] «What is Kubernetes,» [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Consultato il giorno 15 11 2019].

[20] M. Sergio, « Docker and containers,» 17 09 2019. [Online]. Available: https://www.html.it/pag/62783/docker-e-i-container/. [Consultato il giorno 13 10 2019].

[21] I. Eldridge, «What is Container Orchestration?,» 17 07 2018. [Online]. Available: https://blog.newrelic.com/engineering/container-orchestration-explained/. [Consultato il giorno 22 12 2019].

[22] «The Official YAML Web Site,» [Online]. Available: https://yaml.org . [Consultato il giorno 22 12 2019].

[23] «Nginx - High Performance Load Balancer, Web Server & Reverse Proxy,» [Online]. Available: https://www.nginx.com/ . [Consultato il giorno 22 12 2019].

[24] «Learn Kubernetes,» [Online]. Available: https://www.tutorialspoint.com/kubernetes/index.htm . [Consultato il giorno 15 11 2019].

[25] B. Burns, «How Kubernetes deployments work,» [Online]. Available: https://azure.microsoft.com/en-us/topic/what-is-kubernetes/ . [Consultato il giorno 21 09 2019].

[26] «CoreOs - Flannel,» [Online]. Available: https://github.com/coreos/flannel.

[27] S. Dinesh, «Kubernetes NodePort vs LoadBalancer vs Ingress? When should I use what?,» 11 05 2018. [Online]. Available: https://medium.com/google-cloud/Kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0 . [Consultato il giorno 15 11 2019].

[28] «Amazon EC2,» [Online]. Available: https://aws.amazon.com/ec2/. [Consultato il giorno 26 01 2020].

[29] «Virtual Machines,» [Online]. Available: https://azure.microsoft.com/en-us/services/virtual-machines/. [Consultato il giorno 06 01 2020].

[30] «Google Compute Engine,» [Online]. Available: https://cloud.google.com/compute/. [Consultato il giorno 26 01 2020].

[31] «RedHat,» [Online]. Available: https://www.redhat.com. [Consultato il giorno 26 01 2020].

[32] «VMWare,» [Online]. Available: https://www.vmware.com. [Consultato il giorno 26 01 2020].

[33] «Canonical,» [Online]. Available: https://canonical.com. [Consultato il giorno 26 01 2020].

[34] «Mirantis,» [Online]. Available: https://www.mirantis.com. [Consultato il giorno 26 01 2020].

[35] «Rancher,» [Online]. Available: https://rancher.com. [Consultato il giorno 06 01 2020].

[36] «Cloud Parks,» [Online]. Available: https://www.ibm.com/cloud/paks/. [Consultato il giorno 26 01 2020].

[37] «Google Cloud,» [Online]. Available: https://cloud.google.com/anthos. [Consultato il giorno 26 01 2020].

[38] «Tanzu,» [Online]. Available: https://cloud.vmware.com/tanzu. [Consultato il giorno 26 01 2020].

[39] «Arc,» [Online]. Available: https://azure.microsoft.com/services/azure-arc/.

[40] J. MSV, «How Kubernetes Has Changed The Face Of Hybrid Cloud,» 16 12 2019. [Online]. Available: https://www.forbes.com/sites/janakirammsv/2019/12/16/how-kubernetes-has-changed-the-face-of-hybrid-cloud/. [Consultato il giorno 26 01 2020].

[41] «Firecracker,» [Online]. Available: https://firecracker-microvm.github.io. [Consultato il giorno 25 02 2020].

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

[43] «Amazon Simple Storage Service (S3),» [Online]. Available: https://aws.amazon.com/s3/. [Consultato il giorno 27 12 2019].

[44] S. H. (. o. Maryland), A. S. (Frame.io), W. S. (Frame.io) e T. D. (. o. Maryland), «Go Serverless: Securing Cloud via Serverless Design Patterns,» 2018.

[45] «Cloud Functions - Event-driven Serverless Computing,» [Online]. Available: https://cloud.google.com/functions/. [Consultato il giorno 22 12 2019].

[46] «Azure Functions - Develop Faster with Serverless Computing,» [Online]. Available: https://azure.microsoft.com/en-us/services/functions/ . [Consultato il giorno 22 12 2019].

[47] L. W. (UW-Madison), M. L. (. S. University), Y. Z. (. S. University), T. R. (. Tech) e M. S. (UW-Madison), «Peeking Behind the Curtains of Serverless Platforms,» 2018.

[48] N. K. (. C. Labs), D. K. (. Labs) e M. M. (. C. Labs), «Towards Serverless as Commodity: a case of Knative,» 2019.

[49] «Open Container Initiative,» [Online]. Available: https://www.opencontainers.org. [Consultato il giorno 14 01 2020].

[50] «A specification for describing event data in a common way,» [Online]. Available: https://cloudevents.io. [Consultato il giorno 14 01 2020].

[51] «Knative,» [Online]. Available: https://cloud.google.com/knative/. [Consultato il giorno 22 12 2019].

[52] A. Palade, A. Kazmi e T. C. D. I. Siobh’an Clarke (School of Computer Science and Statistics, «An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge,» 2019.

[53] «Apache OpenWhisk,» [Online]. Available: https://openwhisk.apache.org/. [Consultato il giorno 22 12 2019].

[54] «OpenFaas introduction,» [Online]. Available: https://docs.openfaas.com/. [Consultato il giorno 15 11 2019].

[55] «Kubeless - A Bitnami Project,» [Online]. Available: https://kubeless.io. [Consultato il giorno 23 12 2019].

[56] «Serverless Functions for Kubernetes,» [Online]. Available: https://fission.io/ . [Consultato il giorno 22 12 2019].

[57] «Serverless Platform for Automated Data Science,» [Online]. Available: https://nuclio.io . [Consultato il giorno 23 12 2019].

[58] «OpenLambda,» [Online]. Available: https://github.com/open-lambda . [Consultato il giorno 23 12 2019].

[59] «GitHub,» [Online]. Available: https://github.com . [Consultato il giorno 23 12 2019].

[60] «OpenFaas for Kubernetes implementation,» [Online]. Available: https://github.com/openfaas/faas-netes . [Consultato il giorno 15 11 2019].

[61] «Prometheus,» [Online]. Available: https://prometheus.io. [Consultato il giorno 02 02 2020].

[62] «Stack,» [Online]. Available: https://docs.openfaas.com/architecture/stack/ . [Consultato il giorno 15 11 2019].

[63] «NATS - Open source messaging system,» [Online]. Available: https://nats.io/download/nats-io/nats-streaming-server/. [Consultato il giorno 01 26 2020].

[64] «Watchdog,» [Online]. Available: https://docs.openfaas.com/architecture/watchdog. [Consultato il giorno 15 11 2019].

[65] «Autoscaling,» [Online]. Available: https://docs.openfaas.com/architecture/autoscaling/. [Consultato il giorno 15 11 2019].

[66] «Functions,» [Online]. Available: https://cloud.ibm.com/functions/. [Consultato il giorno 27 01 2020].

[67] «Apache CouchDB,» [Online]. Available: https://couchdb.apache.org. [Consultato il giorno 02 02 2020].

[68] «Apache Kafka,» [Online]. Available: http://kafka.apache.org. [Consultato il giorno 02 02 2020].

[69] «Apache ZooKeeper,» [Online]. Available: https://zookeeper.apache.org. [Consultato il giorno 02 02 2020].

[70] S. G. (. Founder), Interviewee, [Intervista].

[71] «Extending the Kubernetes API with Custom Resources,» [Online]. Available: https://docs.okd.io/latest/admin_guide/custom_resource_definitions.html. [Consultato il giorno 23 11 2019].

[72] «Istio,» [Online]. Available: https://istio.io. [Consultato il giorno 23 12 2019].

[73] «Grafana - The open observability platform,» [Online]. Available: https://grafana.com/ . [Consultato il giorno 10 01 2020].

[74] «Ambassador,» [Online]. Available: https://www.getambassador.io. [Consultato il giorno 02 02 2020].

[75] «Gloo API Gateway,» [Online]. Available: https://www.solo.io/products/gloo/. [Consultato il giorno 02 02 2020].

[76] «Contour API Gateway,» [Online]. Available: https://github.com/projectcontour/contour. [Consultato il giorno 02 02 2020].

[77] «Fully Managed Enterprise Kubernetes Platform,» [Online]. Available: https://platform9.com/. [Consultato il giorno 23 01 2019].

[78] «Concepts,» [Online]. Available: https://docs.fission.io/docs/concepts/ . [Consultato il giorno 18 11 2019].

[79] «Controller,» [Online]. Available: https://docs.fission.io/docs/concepts/components/core/controller/. [Consultato il giorno 19 11 2019].

[80] «Router,» [Online]. Available: https://docs.fission.io/docs/concepts/components/core/router/. [Consultato il giorno 19 11 2019].

[81] «Executor,» [Online]. Available: https://docs.fission.io/docs/concepts/components/core/executor/. [Consultato il giorno 19 11 2019].

[82] «Function Pod,» [Online]. Available: https://docs.fission.io/docs/concepts/components/core/function-pod/. [Consultato il giorno 19 11 2019].

[83] «What is scalability testing?,» [Online]. Available: https://www.guru99.com/scalability-testing.html. [Consultato il giorno 15 10 2019].

[84] «The package manager for Kubernetes,» [Online]. Available: https://helm.sh. [Consultato il giorno 02 02 2020].

[85] «Bare metal load-balancer for Kubernetes,» [Online]. Available: https://metallb.universe.tf/ . [Consultato il giorno 08 01 2020].

[86] «Resource metrics pipeline,» [Online]. Available: https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline/. [Consultato il giorno 08 01 2020].

[87] «Apache JMeter Documentation,» [Online]. Available: https://jmeter.apache.org/. [Consultato il giorno 22 11 2019].

[88] «Big Data & Serverless (on K8s), Industry Architectures,» 07 2019. [Online]. Available: https://www.vamsitalkstech.com/?p=7855. [Consultato il giorno 14 01 2020].

[89] G. Esposito, «MapReduce: architettura e funzionamento,» 26 01 2015. [Online]. Available: https://www.html.it/pag/51684/map-reduce-architettura-e-funzionamento/. [Consultato il giorno 26 01 2020].

[90] S. Mallya, «Ad Hoc Big Data Processing Made Simple with Serverless MapReduce,» 2017. [Online]. Available: https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing- made-simple- with- serverless- mapreduce/. [Consultato il giorno 14 01 2020].

[91] B. Congdon, «Introducing Corral: A Serverless MapReduce Framework,» 05 2018. [Online]. Available: https://benjamincongdon.me/blog/2018/05/02/Introducing-Corral-A-Serverless-MapReduce-Framework/. [Consultato il giorno 14 01 2020].

[92] «Big Data Processing: Serverless MapReduce on Azure,» [Online]. Available: https://docs.microsoft.com/en-us/samples/azure-samples/durablefunctions-mapreduce-dotnet/big-data-processing-serverless-mapreduce-on-azure/. [Consultato il giorno 14 01 2020].

[93] «How to do fan-out and fan-in with AWS Lambda,» [Online]. Available: https://theburningmonk.com/2018/04/how-to-do-fan-out-and-fan-in-with-aws-lambda/ . [Consultato il giorno 15 01 2020].

[94] «Itext,» [Online]. Available: https://itextpdf.com/en/resources/examples/itext-7/text-pdf. [Consultato il giorno 03 02 2020].

[95] M. Melanson, «The New Stack - OpenWhisk Serverless Graduates Apache, Looks Ahead to Knative, Envoy and More,» 23 07 2019. [Online]. Available: https://thenewstack.io/the-openwhisk-serverless-platform-now-an-apache-top-level-project-builds-on-kubernetes-mesos/. [Consultato il giorno 15 02 2020].

[96] «MinIO,» [Online]. Available: https://min.io/. [Consultato il giorno 03 02 2020].

[97] «Restdb.io - Simple online NoSQL database backend,» [Online]. Available: https://restdb.io/. [Consultato il giorno 03 02 2020].

[98] «Express,» [Online]. Available: https://expressjs.com/it/. [Consultato il giorno 03 02 2020].

[99] «OpenWhisk client for Javascript,» [Online]. Available: https://github.com/apache/openwhisk-client-js. [Consultato il giorno 03 02 2020].

[100] RightScale, «2018 State of the Cloud Report».

[101] «Configure Service Accounts for Pods,» [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/ . [Consultato il giorno 2019 11 23].

[102] C. Null, «The state of serverless: 6 trends to watch,» [Online]. Available: https://techbeacon.com/enterprise-it/state-serverless-6-trends-watch. [Consultato il giorno 09 01 2020].

[103] T. Seals, «One-Fifth of Open-Source Serverless Apps Have Critical Vulnerabilities,» 06 04 2018. [Online]. Available: https://www.infosecurity-magazine.com/news/onefifth-of-serverless-apps/ . [Consultato il giorno 10 01 2020].