POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

Function–as–a–service on Kubernetes

Workload optimization with Kubeless and Knative

Relatore
prof. Riccardo Sisto

Candidato
Enrico Franco

Supervisori aziendali
dott. Guido Vicino, dott. Gioacchino Martino
Blue Reply S.r.l.

Anno accademico 2019 – 2020

Sommario

Il paradigma di cloud computing si è evoluto negli ultimi anni partendo da architetture on–premises con macchine virtuali, passando per i modelli IaaS, PaaS e SaaS, fino ad arrivare ai paradigmi serverless e Function–as–a–Service che permettono di distribuire applicazioni in modo trasparente per i team di sviluppo, i quali possono concentrare il loro lavoro sull'innovazione senza doversi occupare della configurazione e messa in campo dell'applicazione. L'impiego di container Docker, opportunamente gestiti da orchestratori di livello superiore come Kubernetes, ha reso possibile lo sviluppo ed erogazione di applicazioni come microservizi e funzioni, indipendenti tra loro in termini di sviluppo e testing, seppur possibilmente cooperanti.

Soluzioni commerciali, tra cui AWS Lambda e Microsoft Azure Functions, sono disponibili da alcuni anni per la realizzazione pratica di tale paradigma e negli ultimissimi anni si sono affiancate soluzioni open–source, quali Kubeless e Knative. Entrambi questi framework si pongono, a livello architetturale, *on–top* dell'infrastruttura Kubernetes e ne estendono risorse e funzionalità offerte, ridefinendone ed ampliandone alcune componenti in modo da permettere una più rapida messa in campo del codice, corroborata con politiche di autoscaling ed un approccio *event–driven*.

L'obiettivo di questo lavoro è il confronto di un'applicazione realizzata con architettura monolitica ed una equivalente – capace cioè di adempiere allo stesso insieme di compiti – messa in campo con tecnologia serverless e FaaS, in termini di performance e di risorse utilizzate.

Al fine di confrontare praticamente tali metodologie, è necessario sviluppare e mettere in campo tali applicazioni. Prendendo quindi come punto di partenza un'applicazione monolitica, è necessario scomporre questa in diversi moduli i quali globalmente costituiscono un'applicazione a microservizi. Ciascuno di questi può essere messo in campo con metodologie diverse, sottostando a ragionevoli ipotesi che mirino ad utilizzare le peculiarità del framework scelto e del paradigma serverless in generale, in modo tale da evidenziare quanto possibile le differenze tra le due architetture, insieme con i principali pregi e difetti di ognuna.

Prima di procedere però con tale implementazione è fondamentale una fase di analisi e confronto tra le soluzioni open–source sopra citate, in modo tale da individuarne le caratteristiche distintive principali, per poi procedere con la realizzazione pratica sfruttando uno tra i due framework.

Le applicazioni sviluppate e messe in campo durante questa tesi forniscono un esempio rappresentante un limitato scenario. Il lavoro qui presentato si presta infatti ad ulteriori sviluppi, in maniera tale da costituire una reale alternativa all'interno del panorama di possibili soluzioni in ambiente enterprise.

Ringraziamenti

Innanzitutto ritengo doveroso spendere qualche parola per ringraziare tutte le persone che mi hanno supportato (e sopportato) durante la stesura di questo lavoro e più in generale durante il mio intero percorso di studi.

Ringrazio i miei genitori e la mia intera famiglia per avermi sempre sostenuto durante l'intero corso dei miei studi, per avermi fornito tutti i mezzi e le opportunità per concludere questo percorso,

Ringrazio i miei amici, nonché fedeli compagni di vita, specialmente Christopher e Giovanni, con i quali ho condiviso intensi momenti di studio ed indispensabile svago,

Ringrazio i miei colleghi del Politecnico, soprattutto Luca, Stefano ed Emanuele, per avermi agevolato questo percorso e per la loro preziosa collaborazione,

Ringrazio Matteo, caro amico dai tempi dell'Istituto Tecnico, collega al Politecnico e sempre leale compagno di banco, per avere creduto in me ed avermi spronato a dare del mio meglio,

Ringrazio Domenico, collega del Politecnico e collega in Blue Reply, per i nostri confronti e per avermi particolarmente sostenuto, accompagnato, consigliato durante l'intero periodo di ricerca, sviluppo e scrittura di questa tesi,

Ringrazio Marta, fedelissima amica ritrovata negli ultimi anni, per le nostre costruttive e talvolta anche accese discussioni, per essermi stata vicina nei momenti difficili ed avermi stimolato ad esprimere il massimo delle mie potenzialità,

Ringrazio l'intero organico di Blue Reply S.r.l., in particolar modo Guido, Gioacchino, Olga e Dario per l'opportunità, per avermi dato fiducia ed aver condiviso spazi, strumenti e soprattutto competenze e conoscenze durante la realizzazione di questo lavoro.

Indice

Elenco delle figure V

Elenco delle tabelle VII

1 Introduzione 1

1.1 Modelli di cloud computing . . . 1

1.1.1 Serverless computing e Function–as–a–Service . . . 2

1.1.2 Contesto development . . . 3

1.2 Docker . . . 4

1.2.1 Oggetti di Docker . . . 5

1.2.2 Architettura . . . 5

1.3 Kubernetes . . . 7

1.3.1 Kubernetes Control Plane . . . 8

1.3.2 Oggetti Kubernetes . . . 9

1.3.3 Ingress . . . 12

1.4 Obiettivo . . . 13

2 Software e tecnologie utilizzati 14

2.1 Kubeless . . . 14

2.1.1 Funzioni . . . 15

2.1.2 Meccanismi publish/subscribe . . . 16

2.1.3 Esposizione HTTP . . . 16

2.2 Knative . . . 17

2.2.1 Serving . . . 18

2.2.2 Eventing . . . 20

2.2.3 Apache Kafka e Knative . . . 24

2.3 Software ausiliari . . . 25

2.3.1 Istio . . . 25

2.3.2 Envoy . . . 26

2.3.3 CloudEvents . . . 26
2.3.4 Apache Kafka . . . 27
2.3.5 Prometheus . . . 28
2.3.6 Grafana . . . 29
2.3.7 Apache JMeter . . . 29

3 Applicazione 31
3.1 Scelta del framework . . . 31
3.1.1 Kubeless . . . 32
3.1.2 Knative . . . 33
3.1.3 Popolarità . . . 34
3.1.4 Istio . . . 35
3.1.5 Apache Kafka . . . 36
3.2 Applicazione monolitica . . . 36
3.3 Applicazione serverless . . . 38
3.3.1 Architettura . . . 40
3.3.2 Integrazione con Apache Kafka . . . 41
3.3.3 Integrazione con Istio . . . 41
3.4 Transizione da applicazione monolitica ad applicazione serverless . . . 43
3.5 Confronto applicazioni a riposo . . . 43

4 Test 47
4.1 Inserimento singolo . . . 47
4.1.1 Applicazione serverless . . . 47
4.1.2 Applicazione monolitica . . . 49
4.2 Inserimento multiplo . . . 49
4.2.1 Applicazione serverless . . . 50
4.2.2 Applicazione monolitica . . . 51
4.3 Lettura di dati . . . 52
4.3.1 `Deployment` Kubernetes vs. Applicazione monolitica . . . 53
4.3.2 `Knative Service` inattivo vs. Applicazione monolitica . . . 55
4.4 Utilizzo misto . . . 60
4.4.1 Lettura libri e lettura clienti . . . 60
4.4.2 Letture ed inserimenti contemporanei . . . 65
4.5 Autoscaling . . . 69
4.5.1 In Kubernetes . . . 69
4.5.2 In Kubeless . . . 70
4.5.3 In Knative . . . 70

5 Calcoli e stime 71
5.1 Raccolta dati . . . 71
5.1.1 Ricerca libri . . . 72
5.1.2 Inserimento prestiti e resi . . . 73
5.1.3 Ricerca clienti . . . 73
5.1.4 Inserimento clienti . . . 74
5.1.5 A riposo . . . 74
5.2 Stima scenario giornaliero . . . 75

6 Sviluppi futuri 77

7 Conclusioni finali 79

A Codice applicazione 81
A.1 Kubeless . . . 81
A.2 Applicazione monolitica . . . 82
A.3 Applicazione serverless . . . 84
A.3.1 Servizio di logging . . . 84
A.3.2 Servizio Knative standard . . . 85
A.3.3 Servizio Knative in ascolto su topic Apache Kafka . . . 87
A.3.4 Generazione di eventi in GoLang . . . 88

Elenco delle figure

1.1 Confronto tra virtualizzazione e container 4
1.2 Grafico Google Trends di "Docker" degli ultimi quattro anni 5
1.3 Grafico Google Trends di "Kubernetes" degli ultimi quattro anni 7
1.4 Grafico Google Trends di "Docker" e "Kubernetes" negli ultimi quattro anni 8
1.5 Architettura di Kubernetes 8

2.1 Grafico Google Trends di "Kubeless" degli ultimi tre anni 15
2.2 Grafico Google Trends di "Knative" degli ultimi tre anni 17
2.3 Knative Serving – Architettura 19
2.4 Knative Eventing – Broker & Trigger 21
2.5 Knative Eventing – `Sequence` 23
2.6 Knative Eventing – `Parallel` 23
2.7 Knative Eventing – Consegna diretta di eventi 24
2.8 Knative Eventing – Consegna multipla di eventi 24
2.9 Prometheus – Architettura 29

3.1 Kubeless – Utilizzo CPU 32
3.2 Kubeless – Utilizzo memoria 33
3.3 Knative – Utilizzo CPU 34
3.4 Knative – Utilizzo memoria 34
3.5 Grafico Google Trends di "Kubeless" e "Knative" negli ultimi tre anni 35
3.6 Istio – Risorse utilizzate 36
3.7 Apache Kafka – Risorse utilizzate 36
3.8 Inserimento tramite messaggio su topic Apache Kafka 40
3.9 Applicazione serverless – `Sequence add-customer` attivata da un messaggio sul topic Apache Kafka *customers* 41
3.10 Applicazione serverless – Configurazione degli `Ingress Gateway` di Istio 42
3.11 Applicazione monolitica – Risorse utilizzate a riposo 44
3.12 Applicazione serverless – Risorse utilizzate da un `Deployment` Kubernetes a riposo 44

3.13 Applicazione serverless – Risorse utilizzate da un `Knative Service` a riposo 45
3.14 Applicazione serverless – Utilizzo CPU di un `Knative Service` a riposo 46
3.15 Applicazione serverless – Utilizzo memoria di un `Knative Service` a riposo 46

4.1 Applicazione serverless – Risorse utilizzate all'inserimento di un singolo utente . . 48
4.2 Applicazione serverless – Utilizzo CPU all'inserimento di un singolo utente 48
4.3 Applicazione serverless – Utilizzo memoria all'inserimento di un singolo utente . . 49
4.4 Applicazione serverless – Risorse utilizzate all'inserimento di prestiti 50
4.5 Applicazione serverless – Utilizzo CPU all'inserimento di prestiti 50
4.6 Applicazione serverless – Utilizzo memoria all'inserimento di prestiti 51
4.7 Applicazione monolitica – Utilizzo risorse all'inserimento di prestiti 52
4.8 Applicazione monolitica – Latenza all'inserimento di prestiti 52
4.9 Lettura di libri – Latenza 54
4.10 Lettura di libri – Distribuzione latenza 54
4.11 Lettura di libri – Utilizzo CPU 55
4.12 Lettura di libri – Utilizzo memoria 55
4.13 `Knative Service` inattivo vs. Applicazione monolitica – Latenza risposte 56
4.14 `Knative Service` inattivo vs. Applicazione monolitica – Latenza futura (Media) . 57
4.15 `Knative Service` inattivo vs. Applicazione monolitica – Latenza futura (Deviazione standard) 58
4.16 `Knative Service` inattivo vs. Applicazione monolitica – Latenza terminato il transitorio 58
4.17 `Knative Service` inattivo vs. Applicazione monolitica – Utilizzo CPU 59
4.18 `Knative Service` inattivo vs. Applicazione monolitica – Utilizzo memoria 60
4.19 Lettura di libri e clienti – Latenza applicazione monolitica 61
4.20 Lettura di libri e clienti – Latenza applicazione serverless 61
4.21 Lettura di libri e clienti – Confronto latenza endpoint `GET /books` 62
4.22 Lettura di libri e clienti – Confronto latenza endpoint `GET /customers` 62
4.23 Lettura di libri e clienti – Confronto utilizzo CPU 63
4.24 Lettura di libri e clienti – Confronto utilizzo memoria 63
4.25 Lettura di libri e clienti – Risorse `Deployment get-books` 64
4.26 Lettura di libri e clienti – Risorse `Deployment get-customers` 64
4.27 Lettura libri ed inserimento prestiti – Latenza endpoint `GET /books/{isbn}` . . . 66
4.28 Lettura libri ed inserimento prestiti – Latenza endpoint `POST /loans` 67
4.29 Lettura libri ed inserimento prestiti – Utilizzo CPU 68
4.30 Lettura libri ed inserimento prestiti – Utilizzo memoria 68
4.31 `Horizontal Pod Autoscaler` 69

Elenco delle tabelle

3.1 Biblioteca – API REST 37
3.2 Applicazione serverless – Architettura 39
3.3 Applicazione serverless – Mapping tra topic Apache Kafka e `Sequence` Knative . . 41
3.4 Applicazione serverless – Mapping tra URI e servizio 42

4.1 Lettura di libri e clienti – Statistiche stress test 65

5.1 Risorse utilizzate 75
5.2 Risorse utilizzate giornalmente 76

Capitolo 1

Introduzione

Nel corso degli anni, il panorama delle tecnologie informatiche si è evoluto attraversando diverse fasi, partendo da scenari quali il *grid computing*, fino ad arrivare alle moderne soluzioni in cloud. Le più recenti architetture di computing in cloud, che si basano sui concetti di IaaS, PaaS e SaaS, sono emerse come soluzione tecnologica con l'obiettivo di abbattere gli elevati costi delle risorse hardware lato server.

Il *grid computing* consiste in una rete distribuita di calcolatori connessi e coordinati tra loro, utilizzati per la risoluzione di un problema complesso. Quest'ultimo, infatti, viene suddiviso in task più piccoli, chiamati *grid*, ciascuno gestito da un singolo nodo del cluster. In questo modo, utilizzando tale pattern collaborativo, il gruppo di calcolatori si comporta come un supercomputer virtuale che fornisce l'accesso ad una rete di risorse geograficamente distribuite, fornendo una singola interfaccia di accesso a questa macro risorsa.

Il *cloud computing*, invece, centralizza le risorse e la computazione, utilizzando un modello di accesso client–server. Tale architettura risulta quindi più flessibile, maggiormente scalabile e più facilmente fruibile poiché sfrutta protocolli web standard. Poiché al giorno d'oggi ogni azienda ha la necessità di spazio per poter immagazzinare tutti i dati che essa produce, il cloud computing negli ultimi anni ha preso piede in quanto efficace modello per ottenere l'accesso a risorse condivise, sfruttando la rete internet, in modalità on–demand e con elevata disponibilità [1]. Tali risorse possono essere messe in campo e rilasciate in modo rapido con un costo minimo e senza la minima interazione con il provider del servizio. Il paradigma cloud ha iniziato ad avere successo nel momento in cui i produttori di hardware ed gli sviluppatori di software hanno combinato le loro forze nella creazione di una rete di server connessi tra loro, in grado di sfruttare pienamente l'elevata potenza di calcolo disponibile.

1.1 Modelli di cloud computing

Storicamente, i modelli di cloud computing si sono evoluti passando da IaaS, PaaS e SaaS, fino ad arrivare, più recentemente ai paradigmi serverless e FaaS.

Infrastructure–as–a–Service Il paradigma IaaS consiste nell'utilizzo di un'infrastruttura di calcolo fornita e gestita attraverso la rete internet. Questo paradigma permette di modificare le risorse on–demand ed evitare l'installazione, gestione e manutenzione delle piattaforme fisiche e della loro infrastruttura di rete, abbattendo drasticamente i costi. Infatti, è il fornitore del

servizio che si occupa di ospitare e gestire tutte l'infrastruttura hardware ed assicurarne l'elevata disponibilità, garantendo continuità aziendale e procedure di disaster recovery, difficilmente ottenibili con un'infrastruttura on–premises, permettendo di focalizzare il proprio lavoro sull'attività principale, senza curarsi delle risorse IT.

Platform–as–a–Service Il paradigma PaaS consiste nell'utilizzo di un ambiente completo di sviluppo e distribuzione, messo a disposizione in cloud. Una soluzione PaaS integra una soluzione IaaS che include server, archiviazione e rete, ed offre servizi di più alto livello, quali middleware, strumenti di supporto e business intelligence, permettendo il supporto e la gestione dell'intero ciclo di vita di un'applicazione web.

Il modello PaaS evita spese e gestione di licenze software, poiché è il fornitore del servizio ad occuparsi sia dell'infrastruttura hardware, sia degli strati applicativi di base, permettendo una riduzione del tempo per la scrittura di codice, un utilizzo di strumenti spesso complessi ad un prezzo contenuto ed un'efficiente gestione del ciclo di vita dell'applicazione.

Software–as–a–Service Il paradigma SaaS consiste nell'utilizzo di applicazioni complete basate su cloud e residenti in esso, accessibili attraverso internet. Questo modello maschera completamente l'infrastruttura hardware ed i middleware software sottostanti, garantendo un'elevata disponibilità e sicurezza dell'applicazione, nonché dei dati in essa contenuti. La disponibilità in cloud ed il continuo aggiornamento dell'applicazione sono garantiti dal fornitore del servizio ed acquistati come servizio, permettendo una maggiore fruibilità dell'applicazione da qualunque dispositivo connesso ad internet.

1.1.1 Serverless computing e Function–as–a–Service

Negli ultimissimi anni questi modelli di computazione in cloud si sono evoluti nel paradigma *Function–as–a–Service*, erroneamente noto anche come *serverless computing*.

La computazione *serverless* consiste nell'esecuzione di applicazioni in modo trasparente rispetto all'architettura sottostante e non ha nulla a che vedere con l'idea di una rete che non utilizza server per l'elaborazione dei dati. Infatti, il nome sta ad indicare che le attività di provisioning, scalabilità e gestione del server sul quale sono messe in campo ed operano le applicazioni, vengono eseguite automaticamente da qualche piattaforma o framework intermedio, ottimizzando le risorse dell'organizzazione in modo trasparente allo sviluppatore e permettendo a quest'ultimo di concentrarsi sulla business logic, valorizzando maggiormente il cuore applicativo ed accelerando di conseguenza la messa in campo dei prodotti, focalizzando l'attenzione sull'innovazione.

Al contrario, nel paradigma *Function–as–a–Service*, viene definita *funzione* un'unità operativa di base eseguita alla ricezione di eventi, che ne gestisce l'elaborazione e l'eventuale risposta. La funzione è l'unica componente che lo sviluppatore si occupa di scrivere e configurare, poiché la sua messa in campo è demandata al provider, il quale ha il compito di allocare le risorse necessarie in base al carico di lavoro richiesto, portando ad una riduzione del *time–to–market*. In questo modo, vengono anche ridotti i costi riguardanti la manutenzione e/o l'acquisto dell'infrastruttura, i quali vengono infatti monitorati dal provider cloud in base all'effettivo utilizzo di risorse.

Tuttavia, l'intrinseca assenza di stato delle funzioni, porta alla necessità di utilizzare un qualche meccanismo di storage persistente per supportare il flusso dell'applicazione, collegare eventi tra loro o coordinare i diversi microservizi. È necessario tenere in considerazione una possibile perdita di performance, a causa della possibile completa de–allocazione delle risorse effettuata dal cloud provider che causa un ritardo iniziale nella fase di avvio della runtime che esegue l'applicazione, il

quale aggiunge inevitabilmente un'ulteriore latenza al meccanismo. Inoltre, potrebbe essere sconveniente cedere completamente il controllo dell'infrastruttura al cloud provider, poiché l'assenza di controllo dello stack può portare ad un completo monopolio del cloud provider, limitando la flessibilità di tale architettura e, ad esempio, la migrazione verso un altro provider a causa dei possibili costi derivanti dall'adattamento dell'applicazione alle nuove specifiche.

Le più grandi aziende operanti nel mondo del cloud computing offrono soluzioni commerciali nella forma di piattaforme per il cloud computing, tra cui AWS Lambda, Google Cloud Functions, Microsoft Azure Functions e IMB Cloud Functions [2]. Tali soluzioni però richiedono che le funzioni rispettino alcune caratteristiche peculiari della piattaforma, causando possibile *vendor lock–in*. Per ovviare a questo problema, sono stati sviluppati diversi framework che permettono di mettere in campo applicazioni in modalità FaaS sfruttando la propria infrastruttura, tra cui Kubeless, Apache OpenWhisk, OpenFaaS e Knative [3].

1.1.2 Contesto development

Le prime applicazioni realizzate in informatica venivano eseguite ognuna su una macchina fisica dedicata, con conseguenti costi elevati durante le fasi di acquisto, configurazione e manutenzione di tale hardware, spesso accompagnati da un grande inutilizzo delle risorse allocate.

La *virtualizzazione hardware* ha risolto tale problema, permettendo di allocare e condensare i servizi su poche macchine server fisiche, portando ad una riduzione dei costi e migliorando soprattutto la manutenibilità di tali architetture. Un software particolare, noto come *hypervisor* è responsabile del collegamento con l'hardware e permette di condividere un singolo sistema host fisico tra diversi ambienti, separati e ben distinti tra loro, noti come *macchine virtuali*, che si affidano al software hypervisor per la spartizione delle risorse fisiche disponibili. La virtualizzazione hardware risolve alcuni tra i problemi sopra elencati, permettendo infatti di creare facilmente ambienti di sviluppo o test con la corretta configurazione hardware o migrare intere macchine virtuali. Tuttavia, questo approccio non risulta essere la soluzione finale, perché richiede un'elevata configurazione e si presenta comunque piuttosto pesante in termini di utilizzo di risorse, in quanto ogni macchina virtuale deve emulare un intero sistema operativo e le risorse risultano perennemente allocate dal punto di vista del sistema host [4].

La *virtualizzazione software* si presenta come una soluzione più leggera rispetto alla virtualizzazione hardware, poiché il *container*, unità fondamentale di questo approccio, viene lanciato in esecuzione direttamente a livello del sistema operativo e condivide questo strato con tutti gli altri container. Rispetto ad una macchina virtuale, un container risulta quindi meno invasivo in termini di risorse ed è possibile interagire con esso attraverso interfacce standard che permettono di avviarlo, stopparlo e definire eventuali parametri come variabili d'ambiente. Oltre a mantenere separate le applicazioni, per intrinseca natura, è possibile costruire un'applicazione grande e complessa come insieme di container eterogenei e, di conseguenza, organizzare applicazioni multi–container che interagiscono attraverso diverse infrastrutture cloud.

Un *container* in ambiente Unix può essere visto come un insieme di processi isolati dal resto del sistema. Tutti i file necessari per la sua esecuzione sono contenuti al suo interno e forniti tramite un'immagine, rendendo i container portabili e consistenti nella loro evoluzione nel tempo, dallo sviluppo, alla fase di test, fino alla fase di messa in campo in un ambiente di produzione.

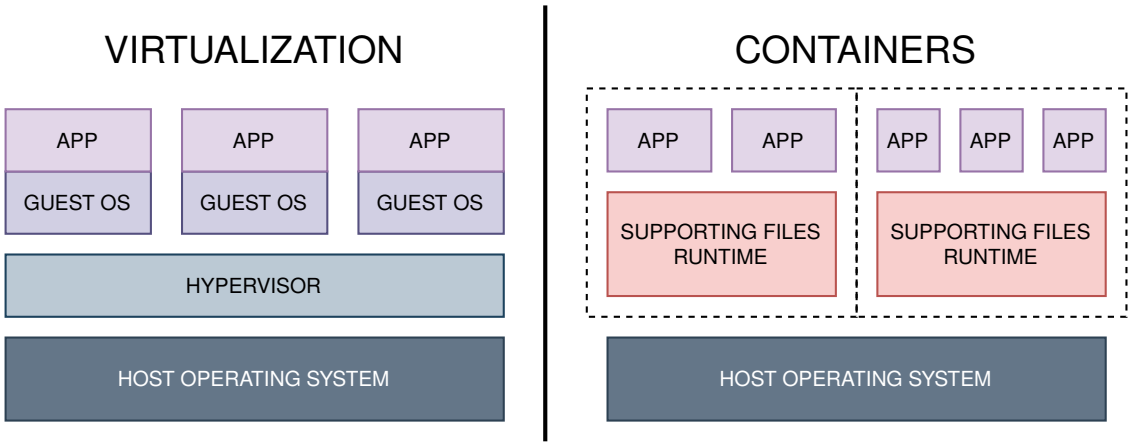

Figura 1.1. Confronto tra virtualizzazione e container

Le applicazioni, oggigiorno, sono spesso costruite a partire da componenti esistenti, utilizzando ed affidandosi ad altri servizi ed altre applicazioni già esistenti per sfruttare componenti standard. Ognuna di queste componenti possiede, a sua volta, le proprie dipendenze, generando talvolta possibili conflitti. Unendo a monte tutti i componenti, ciascuno con le proprie dipendenze, è possibile risolvere questo problema. L'approccio a container permette di installare un'applicazione in un nuovo ambiente in modo più efficiente ed efficace, in quanto è sufficiente eseguire un gestore di container, poiché tutte le dipendenze vengono dichiarate all'interno di un particolare file ed impacchettate insieme con l'applicazione stessa all'interno di un container, permettendo di conseguenza l'esecuzione dell'applicazione in diversi sistemi. Inoltre, l'approccio a container permette di risolvere i conflitti tra dipendenze: nel caso in cui siano necessarie diverse versioni dello stesso applicativo, infatti, è sufficiente che queste siano eseguite in container diversi.

1.2 Docker

Docker [5] nasce all'inizio del 2013 come un progetto open-source scritto in Go di dotCloud, un'azienda incentrata sulle Platform–as–a–Service in cloud, come un'estensione della tecnologia che l'azienda ebbe sviluppato per eseguire e monitorare il proprio cluster di server. L'azienda nei mesi successivi cambia il proprio nome in Docker Inc., in seguito all'unione con la Linux Foundation [6] e nel gennaio 2020 conta più di 2500 star su GitHub ed oltre 3500 fork.

Docker è una tecnologia open–source per la costruzione, spostamento, distribuzione e rilascio di applicazioni basate su container, distribuita sotto Licenza Apache Common 2.0, disponibile per tutte le maggiori piattaforme in grado di eseguire container in ambienti Linux oppure Windows. La struttura stessa dei container garantisce il loro isolamento, permettendo l'esecuzione multipla su un singolo host, sfruttando direttamente il kernel della macchina senza la necessità di un server hypervisor. In questo modo, i container sono il centro dello sviluppo dell'applicazione, la quale viene costruita come insieme di container per esprimere le dipendenze, permettendone una facile distribuzione e testing in ambienti diversi, fino alla messa in campo in produzione in un data center locale, presso un cloud provider o una soluzione ibrida tra le precedenti, sfruttando un sistema di orchestrazione, come Kubernetes, illustrato nella sezione 1.3.

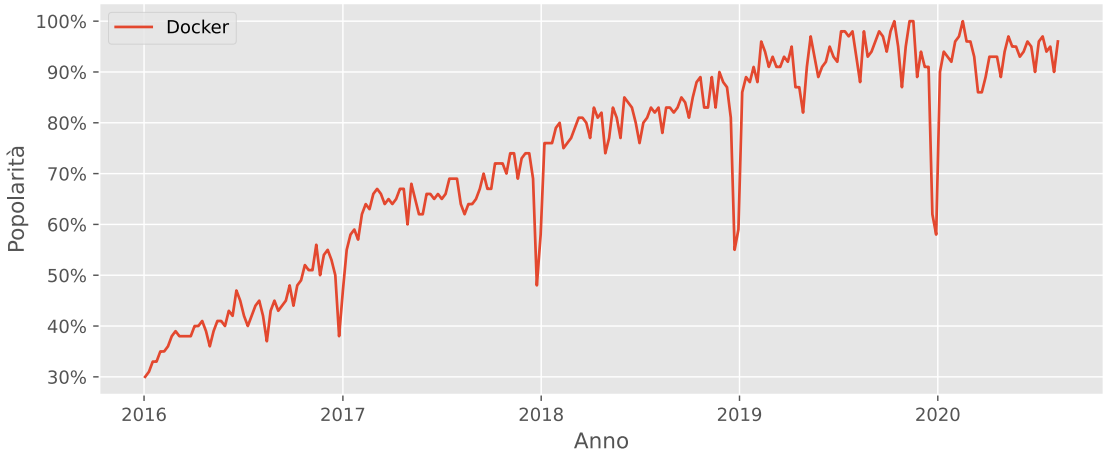

Figura 1.2. Grafico Google Trends di "Docker" degli ultimi quattro anni

1.2.1 Oggetti di Docker

Docker utilizza e crea differenti tipi di oggetti, come immagini e container. Un *container* costituisce un'istanza di un'immagine in esecuzione, con il quale è possibile interagire direttamente dalla macchina host, e può essere visto come un'evoluzione di un processo, isolato dagli altri container e dal sistema host.

Un'*immagine*, invece, costituisce un template di sola lettura, utilizzato per la creazione di un container Docker. Solitamente, un'immagine viene costruita sulla base di un'altra immagine alla quale vengono aggiunte le configurazioni necessarie e peculiari dell'applicazione. La definizione di un'immagine avviene all'interno di un file, chiamato `Dockerfile`, nel quale vengono indicate le istruzioni, ognuna delle quali aggiunge un *layer* all'interno dell'immagine finale, in modo tale che, di volta in volta, vengano ricompilati solo gli strati che effettivamente sono stati modificati.

1.2.2 Architettura

Docker utilizza un'architettura client–server, nella quale il *Docker client* comunica, attraverso una API REST, con il *Docker daemon* che ha il compito di mettere in campo, eseguire e distribuire i container Docker.

Docker Engine

Docker Engine costituisce il cuore della piattaforma e si comporta come un'applicazione client–server che espone una serie di API di tipo REST che i programmi possono sfruttare per comunicare con il processo demone `dockerd` che costituisce la parte server della piattaforma. Il client, invece, è un'interfaccia a linea di comando composta dal comando `docker` e suoi sottocomandi. Il client Docker ed il processo demone `dockerd` possono essere eseguiti sullo stesso sistema, oppure è possibile utilizzare un demone remoto al quale si effettua una connessione attraverso socket ed interfacce di rete.

Registry Docker

I *registry Docker* hanno il compito di memorizzare e storicizzare le immagini Docker. Tra questi, *Docker Hub* [7] è un repository pubblico gratuito che permette agli sviluppatori di trovare e condividere immagini di container con il proprio team, oppure con la comunità Docker, tra cui immagini ufficiali di alta qualità revisionate dal team *Official Images* che ha il compito di analizzare vulnerabilità in modo da garantire elevati standard di sicurezza. L'utilizzo di immagini ufficiali è sempre consigliato, perché accompagnate da ottima documentazione e progettate per la maggior parte dei casi d'uso.

Architettura interna

Internamente, Docker sfrutta i *namespace* per garantire l'isolamento tra i container, creando un insieme di namespace durante l'esecuzione di un container. Ogni componente di un container utilizza un namespace separato ed il suo accesso è limitato solamente a tale namespace.

Per limitare l'accesso a determinati tipi di risorse, Docker utilizza i *control group*, denominati `cgroup` nel mondo Unix, permettendo al Docker Engine la condivisione di risorse hardware e l'applicazione di limiti e vincoli.

Al fine, invece, di creare differenti strati di un'immagine in modo leggero, Docker utilizza un particolare tipo di file system, denominato *union file system*. Docker Engine ha quindi il compito di combinare opportunamente namespace e control group in un oggetto chiamato *container format*.

1.3 Kubernetes

Kubernetes [8] è una piattaforma open–source eseguibile in infrastrutture on–premises, ibride o cloud ideata per automatizzare la messa in campo, lo scaling e la gestione di applicazioni a container, raggruppando tali container in unità logiche di più alto livello che ne facilitano la gestione e la manutenzione. Kubernetes è un progetto scritto in Go, inizialmente sviluppato da Google, attualmente parte della Cloud Native Computing Function, costruito sulla base dei principi che permettono a Google di eseguire e gestire miliardi di container ogni settimana, garantendo un'ottima scalabilità al crescere delle dimensioni del proprio team.

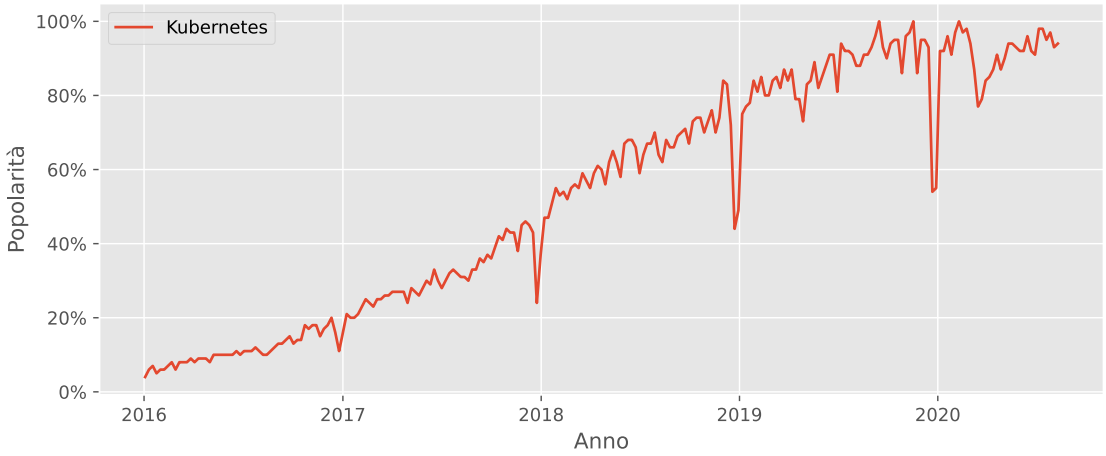

Figura 1.3. Grafico Google Trends di "Kubernetes" degli ultimi quattro anni

Kubernetes utilizza un insieme di oggetti fruibili tramite API per descrivere lo stato desiderato del cluster, indicando, ad esempio, quali applicazioni eseguire, quali immagini utilizzare, il numero di repliche da istanziare, quali risorse di rete e di spazio su disco rendere disponibili. L'interazione con tali oggetti è possibile attraverso un'interfaccia a linea di comando costituita dal comando `kubectl` e suoi sottocomandi, che hanno il compito di contattare le API nel modo appropriato. Il *Kubernetes Control Plane*, costituito da un insieme di processi all'interno del cluster, ha il compito di modificare e mantenere lo stato del cluster coerente con lo stato desiderato, attraverso una serie di operazioni automatiche, tra cui il riavvio di container oppure lo scaling del numero di repliche.

Diffusione In figura 1.4 è mostrato il confronto tra Docker e Kubernetes in termini di ricerche su Google. Appare subito evidente come le due tecnologie siano sempre state altamente correlate in termini di ricerche, di interesse e quindi, verosimilmente, di utilizzo. La differenza sostanziale è insita nel fatto che Docker può essere utilizzato anche dai singoli sviluppatori, a qualunque livello, come risorsa utile durante le fasi di sviluppo e testing, permettendo di emulare componenti dell'applicazione senza richiederne l'installazione completa, mentre Kubernetes si propone come orchestratore di container e quindi utile per lo più in ambienti enterprise nei quali è necessario mettere in campo e conseguentemente gestire un elevato numero di container.

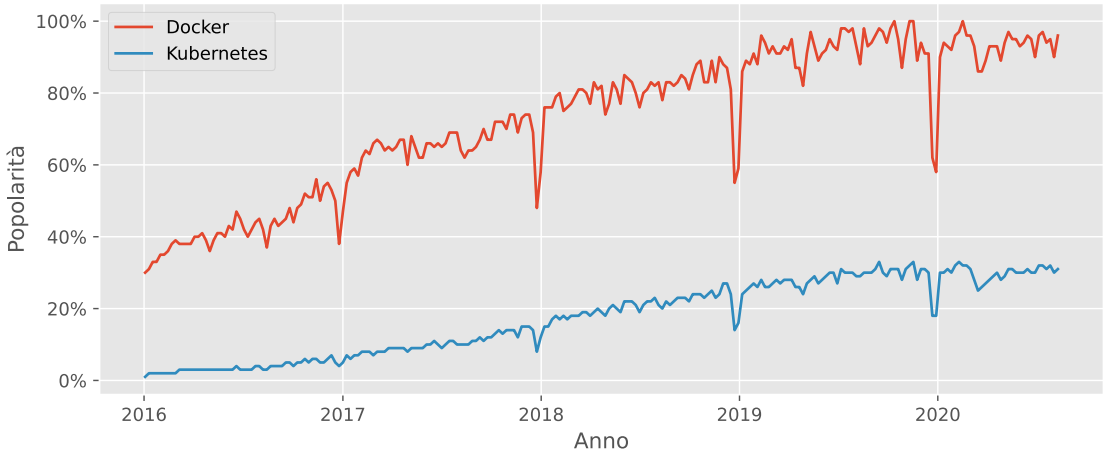

Figura 1.4. Grafico Google Trends di "Docker" e "Kubernetes" negli ultimi quattro anni

1.3.1 Kubernetes Control Plane

Il *Kubernetes Control Plane* è composto da un insieme di processi in esecuzione all'interno del cluster, con il compito di storicizzare tutti gli oggetti Kubernetes, confrontando lo stato attuale con lo stato desiderato ed effettuando le modifiche necessarie al raggiungimento di quest'ultimo.

L'architettura di Kubernetes è mostrata in figura 1.5.

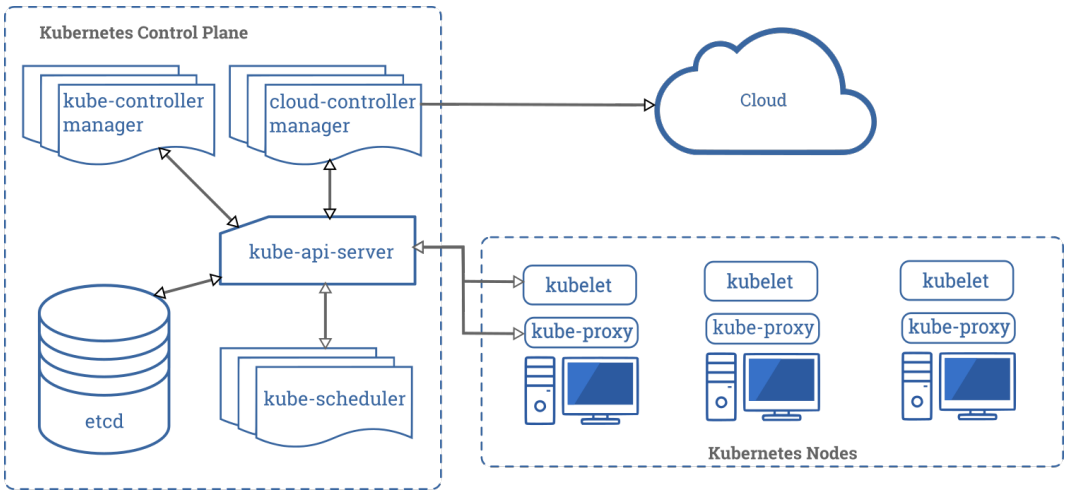

Figura 1.5. Architettura di Kubernetes

Master Kubernetes

Il nodo *master* è responsabile di mantenere coerente lo stato corrente con lo stato desiderato, attraverso un insieme di processi:

`kube-apiserver` Il server API Kubernetes ha il compito di validare e configurare i dati verso l'interfaccia REST, fornendo un'interfaccia per interrogare e modificare lo stato del cluster.

`kube-controller-manager` Il controller manager di Kubernetes è un processo demone che si comporta come un controllore, osservando lo stato del cluster attraverso le API ed effettuando le modifiche necessarie allo stato corrente per raggiungere lo stato desiderato.

`kube-scheduler` Lo scheduler Kubernetes ha il compito di assegnare i processi ai differenti nodi del cluster, tenendo conto della topologia di rete, delle politiche, dei vincoli e dei requisiti da rispettare. La sua efficienza ed efficacia hanno un elevato impatto sulle prestazioni dell'intero cluster.

`etcd` `etcd` [9] costituisce un meccanismo distribuito di storage di coppie chiave–valore, che fornisce un servizio ad alta affidabilità per la memorizzazione di dati negli ambienti distribuiti o nei cluster.

Solitamente, tali processi vengono eseguiti da un singolo nodo all'interno del cluster, ma è possibile ridondare il master su più nodi per garantire maggiore affidabilità e disponibilità.

Nodi Kubernetes

Gli altri nodi del cluster sono macchine con le quali raramente si interagisce, che hanno il compito di eseguire effettivamente le applicazioni, controllate dal nodo master, sulle quali sono in esecuzione i seguenti processi:

`kubelet` Il processo demone `kubelet` permette la comunicazione con il nodo master.

`kube-proxy` Il proxy Kubernetes si comporta come un canonico proxy, con il compito di replicare i servizi di rete definiti nelle API Kubernetes all'interno di ogni nodo.

1.3.2 Oggetti Kubernetes

Kubernetes offre una varietà di astrazioni per descrivere lo stato del proprio sistema. Questi oggetti costituiscono entità persistenti per rappresentare lo stato del cluster e descrivono quali applicazioni sono in esecuzione e su quali nodi, quali risorse sono allocate per tale applicazione e le politiche adottate. Per istanziare un oggetto, molto spesso si utilizza un file `.yaml`[1] che ne descrive le specifiche ed altre proprietà basilari.

Alla creazione di un oggetto Kubernetes, viene modificato lo stato corrente creando un nuovo stato desiderato che il sistema si occupa di raggiungere, in primo luogo assicurando che tale oggetto esista. In ogni istante, il Kubernetes Control Plane ha il compito di gestire in modo attivo lo stato attuale dell'oggetto, in modo tale che questo rispecchi lo stato desiderato.

Ciascun oggetto è caratterizzato da due campi che ne descrivono la configurazione:

Specifiche Le *specifiche* descrivono lo stato desiderato dell'oggetto, cioè le caratteristiche che si vogliono ottenere.

[1] YAML [10] Ain't Markup Language. YAML è un formato *human friendly* utilizzato per la serializzazione di dati.

Stato Lo *stato* descrive lo stato attuale dell'oggetto e viene automaticamente aggiornato da Kubernetes.

Kubernetes offre alcuni oggetti basilari, tra cui `Pod`, `Service`, `Volume`, `Namespace` ed oggetti che offrono un maggiore livello di astrazione, tra cui `Deployment`, `DaemonSet`, `StatefulSet`, `ReplicaSet` e `Job` che forniscono funzionalità aggiuntive, affidando la loro gestione ad alcuni controllori.

`Pod`

Un *Pod* costituisce la più semplice, piccola e basilare unità di esecuzione in Kubernetes e rappresenta un processo all'interno del cluster. Incapsula uno o più container fortemente accoppiati che condividono le risorse per la memorizzazione dei dati, un indirizzo IP univoco ed una serie di opzioni che descrivono il comportamento dell'applicazione.

Raramente si creano `Pod` direttamente in Kubernetes, perché questi sono concepiti come entità effimere. Solitamente, infatti, questi vengono creati da *controller* di più alto livello, come `Deployment`, `StatefulSet` o `DaemonSet`, che ne gestiscono l'intero ciclo di vita.

`Pod` che esegue un singolo container Il modello *one–container–per–Pod* è il più utilizzato e permette di visualizzare il `Pod` come un wrapper intorno al semplice container.

`Pod` che esegue più container Un `Pod` che esegue più container accoppiati e cooperanti può invece essere visualizzato come un wrapper che gestisce tali container ed un insieme di risorse tra loro condivise come un'unica entità esecutiva. In questo caso, i container vengono schedulati insieme sulla stessa macchina del cluster, possono comunicare tra di loro utilizzando `localhost` e possono condividere dati attraverso i `Volume`, i quali permettono anche di rendere persistenti i dati in modo che siano visibili al `Pod` anche dopo un suo riavvio o rescheduling.

`Service`

Un *Service* in Kubernetes ha il compito di esporre un'applicazione in esecuzione su diversi `Pod` come servizio di rete. Un `Service` costituisce un'astrazione definendo un insieme di `Pod` ed alcune policy per regolare l'accesso a tali risorse.

I `Pod` in Kubernetes sono entità effimere a cui viene assegnato un indirizzo IP che può variare nel tempo, il cui ciclo di vita è solitamente gestito da entità di livello superiore, ad esempio da `Deployment`. Un `Service` costituisce l'interfaccia verso l'insieme di `Pod` o il `Deployment` che li sta astraendo.

Kubernetes offre diversi tipi di `Service`:

`ClusterIP` Espone il `Service` con un indirizzo IP interno al cluster, rendendo il `Service` raggiungibile solo dall'interno del cluster stesso.

`NodePort` Espone il `Service` su ciascun indirizzo IP corrispondente ad un nodo del cluster, su una porta scelta automaticamente, uguale per tutti i nodi.

`LoadBalancer` Espone un `Service` all'esterno utilizzando un bilanciatore.

`ExternalName` Mappa un `Service` su un nome DNS ti tipo `CNAME`.

Un'ulteriore modalità per esporre un `Service` è l'utilizzo di un `Ingress`, i cui dettagli sono discussi nella sezione 1.3.3, il quale si comporta come un punto di ingresso al cluster e permette la scrittura di regole per il routing e l'esposizione di differenti servizi sullo stesso indirizzo IP.

`Volume`

I *volumi* risolvono il problema della persistenza dei dati e della condivisione dei dati tra container all'interno dello stesso `Pod`. I file all'interno dei container infatti sono effimeri, quindi quando un container viene interrotto i suoi dati vengono persi ed il container viene riavviato partendo da uno stato pulito.

Il concetto di `Volume` è già presente in Docker, ma in maniera meno strutturata. In Docker, infatti, un `Volume` è semplicemente una directory del sistema host o di un altro container che il container utilizza come storage. Un `Volume` in Kubernetes ha un ciclo di vita identico a quello del `Pod` che lo utilizza, permettendo la persistenza dei dati tra i diversi riavvii dei container. Kubernetes supporta diversi tipi di `Volume` e ciascun `Pod` ne può utilizzare un numero arbitrario simultaneamente.

Al fine di ottenere un livello di persistenza indipendente dal ciclo di vita dei `Pod`, Kubernetes offre il concetto di `Persistent Volume`. In questo caso, il `Pod` ha il compito di dichiarare l'interesse attraverso un `PersistentVolumeClaim` e Kubernetes si occupa di soddisfare tale richiesta associando un `Persistent Volume`. Tale associazione può avvenire in *modo statico* se l'amministratore del cluster crea manualmente i `Persistent Volume`, oppure in *modo dinamico* attraverso la definizione di `StorageClass`, le quali hanno il compito di descrivere la tipologia di storage offerta, astraendone la reale struttura su disco.

`Namespace`

Un *Namespace* può essere visto come un cluster virtuale. Il loro utilizzo è consigliato quando l'ambiente Kubernetes è condiviso tra molti utenti, possibilmente appartenenti a team o progetti differenti.

I `Namespace` non possono essere annidati l'uno dentro l'altro e costituiscono un ambiente all'interno del quale i nomi delle risorse devono essere univoci, permettendo di dividere logicamente il cluster Kubernetes in gruppi di risorse. Il nome del `Namespace` viene utilizzato per assegnare un nome DNS a ciascun `Service`, in modo che questi siano raggiungibili non solamente tramite il loro indirizzo IP, che potrebbe mutare, ma tramite un hostname, se richiamati all'interno dello stesso `Namespace`, o tramite il loro FQDN[2] se richiamati da un `Namespace` differente.

`Deployment`

Un *Deployment*, spesso abbreviato come *Deploy*, fornisce un approccio dichiarativo per la creazione e la modifica di `Pod` e `ReplicaSet`. Il `Deployment` descrive lo stato desiderato, ed il *Deployment Controller* si occupa di modificare lo stato attuale al fine di raggiungere lo stato desiderato, creando, modificando o rimuovendo `ReplicaSet` e `Pod`.

Un `Deployment` attraversa diversi stati nel suo ciclo di vita. In particolare, la sua creazione è costituita dalla seguente procedura:

[2]Fully Qualified Domain Name.

1. Il `Deployment` crea un nuovo `ReplicaSet`;
2. Il `Deployment` attiva il nuovo `ReplicaSet` impostando il numero di repliche desiderato;
3. Il `Deployment` disattiva il `ReplicaSet` precedente, se presente, impostando il numero di repliche a zero e quindi eliminando i `Pod` precedenti;
4. I nuovi `Pod` vengono messi in campo e, quando i loro container sono disponibili, entrano in esecuzione.

`DaemonSet`

Un *DaemonSet* ha il compito di assicurare che su tutti i nodi del cluster, o su un sottoinsieme di questi, sia in esecuzione una copia di un determinato `Pod`. Ad ogni inserimento o rimozione di un nodo del cluster, il `DaemonSet` si occupa di aggiungere o eliminare il `Pod` specificato.

I `DaemonSet` vengono tipicamente impiegati quando è necessario eseguire un processo demone su tutti i nodi, ad esempio *Prometheus Node Exporter*, che ha il compito di esportare metriche riguardanti l'hardware ed il sistema operativo del nodo, o *Weave* [11], che ha il compito di creare una rete virtuale per la connessione tra i diversi `Pod` all'interno del cluster.

`StatefulSet`

Un *StatefulSet* ha il compito di gestire applicazioni stateful e controllare la messa in campo e lo scaling di un insieme di `Pod`, garantendone l'ordinamento e l'unicità. In modo simile ad un `Deployment`, uno `StatefulSet` gestisce una serie di `Pod` identici, ma a questi viene assegnato un identificatore univoco che viene mantenuto anche dopo la loro distruzione o il loro riavvio.

`ReplicaSet`

Un *ReplicaSet* ha il compito di mantenere attivi un insieme di `Pod` e viene spesso utilizzato per garantire la presenza di un numero specifico di `Pod` identici. Tuttavia, l'utilizzo diretto di un `ReplicaSet` è sconsigliato ed è preferibile utilizzare un `Deployment`, in quanto quest'ultimo costituisce un oggetto di livello più alto, definito con un approccio dichiarativo che automaticamente gestisce un `ReplicaSet`.

`Job`

Un *Job* crea uno o più `Pod` ed ha il compito di garantire che uno specifico numero di questi `Pod` termini con successo, portando quindi a termine con successo il compito per il quale questi sono stati istanziati.

1.3.3 Ingress

Per poter sfruttare le risorse `Ingress`, è necessario che all'interno del cluster sia presente un *Ingress Controller*. Poiché questi controller non sono avviati in modo automatico durante l'avvio del cluster, al contrario di altri controller facenti parte dell'eseguibile `kube-controller-manager`, è necessario installare una soluzione custom a seconda delle proprie esigenze [12]. Attualmente,

Kubernetes supporta e manutiene attivamente i controller GCE[3] e nginx, ma supporta l'installazione di altri controller quali Ambassador, Gloo, Istio, Kong e Traefik. È possibile installare diversi Ingress Controller contemporaneamente all'interno del cluster, annotando opportunamente l'oggetto `Ingress` per indicare quale classe utilizzare. In linea generale, tutti gli Ingress Controller dovrebbero soddisfare le stesse specifiche, ma questi possono operare in modo leggermente diverso a causa di diverse scelte implementative.

In generale, un oggetto `Ingress` in Kubernetes si occupa della gestione dell'esposizione dei servizi all'esterno del cluster, tipicamente con il protocollo HTTP, e dell'accesso agli stessi. Un `Ingress` può fornire funzionalità di *load balancing*, terminazione SSL e *name–based virtual hosting* [13].

Un *ingress controller* ha quindi il compito di rendere disponibile l'`Ingress`, solitamente attraverso un bilanciatore, oppure configurando ulteriori meccanismi che possano aiutare nella gestione del traffico. Un `Ingress` non espone direttamente le porte con i relativi protocolli arbitrariamente, poiché il compito di esporre all'esterno del cluster servizi diversi da HTTP e HTTPS è svolto da `Service` di tipo `NodePort` o `LoadBalancer`.

1.4 Obiettivo

L'obiettivo di questo lavoro è il confronto di un'applicazione realizzata con architettura monolitica ed una equivalente – capace cioè di adempiere allo stesso insieme di compiti – messa in campo con tecnologia serverless e FaaS, in termini di performance e di risorse utilizzate.

Al fine di poter confrontare a livello pratico tali metodologie e paradigmi, è necessario sviluppare e mettere in campo tali applicazioni. Prendendo quindi come punto di partenza un'applicazione monolitica, è necessario scomporre questa in diversi moduli i quali globalmente costituiscono un'applicazione a microservizi ed erogare ognuno di questi con l'ausilio di un framework costruito on–top dell'architettura Kubernetes. Ciascuna componente può quindi essere messa in campo con metodologie diverse, sottostando a ragionevoli ipotesi che mirino ad utilizzare le peculiarità del framework scelto e del paradigma serverless in generale, in modo tale da evidenziare quanto possibile le differenze tra le due architetture, insieme con i principali pregi e difetti di ognuna.

Innanzitutto, prima di procedere con l'implementazione, è fondamentale una fase di analisi e confronto tra alcuni framework open–source, in modo tale da individuarne le caratteristiche distintive principali, per poi procedere con la realizzazione pratica scegliendone uno tra quelli esaminati. Durante questa fase di analisi è bene tenere in considerazione anche eventuali dipendenze di tali framework.

A messa in campo avvenuta, è necessaria una fase di analisi e comparazione delle performance delle due soluzioni. Per fare ciò è possibile effettuare batterie di test in modo tale da "stressare" le applicazioni ed analizzare, ove possibile, la risposta dell'applicazione in termini di latenza e l'utilizzo di risorse quali CPU e memoria RAM.

[3] Google Compute Engine.

Capitolo 2

Software e tecnologie utilizzati

Durante la realizzazione di questa tesi sono stati utilizzati due diversi framework open–source per la messa in campo di un'applicazione serverless: Kubeless e Knative, ciascuno con le proprie peculiarità. Questi framework serverless si propongono come alternative a soluzioni commerciali, permettendo di mettere in campo porzioni di codice in modo nativo su un cluster Kubernetes, estendendo alcune sue caratteristiche e permettendo ai team di sviluppo di occuparsi solamente della scrittura di codice senza doversi preoccupare del provisioning o manutenzione dell'infrastruttura e dell'architettura sottostante.

L'obiettivo di questo capitolo è presentare i software e le tecnologie utilizzati, illustrandone le principali caratteristiche, concentrando il focus sulle proprietà distintive di ognuno e maggiormente utilizzate.

2.1 Kubeless

Kubeless [14] è un framework open–source basato su Kubernetes per realizzare applicazioni serverless sviluppato da Bitnami[1]. Si propone come un'alternativa open–source alle alternative commerciali che implementano architetture serverless, come AWS Lambda, Azure Functions o Google Cloud Functions. Il progetto, scritto in Go, disponibile su GitHub, conta più di 5000 star e più di 500 fork.

Kubeless permette di mettere in campo piccole porzioni di codice, chiamate *funzioni*, senza doversi occupare dell'infrastruttura sottostante. Lavorando on top di un cluster Kubernetes, il framework riesce a sfruttare tutti i vantaggi forniti dall'architettura Kubernetes per fornire funzionalità di auto–scaling, routing delle API e monitoraggio.

Al fine di creare le proprie funzioni come risorse Kubernetes, Kubeless definisce diverse *Custom Resource Definition* all'interno del cluster Kubernetes ed utilizza il meccanismo dei controller per monitorare tali risorse, avviando routine per mantenerne allineato lo stato e per rendere disponibili tali funzioni attraverso il protocollo HTTP o meccanismi publish/subscribe.

[1] **Bitnami** [15] è una compagnia della società VMWare Inc., leader nel packaging di applicazioni. In questo modo, la gestione in ambienti multi–cloud e multi–piattaforma viene semplificata, fornendo infrastrutture ed applicazioni ottimizzate e quindi permettendo un elevato grado di standardizzazione tra piattaforme.

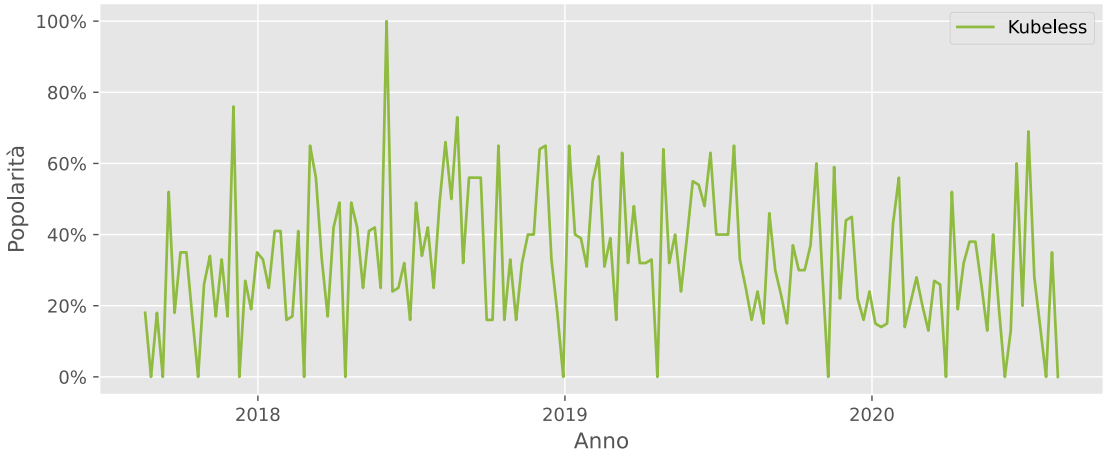

Figura 2.1. Grafico Google Trends di "Kubeless" degli ultimi tre anni

2.1.1 Funzioni

Le *funzioni*, denominate `Function` nella terminologia Kubeless, rappresentano l'entità principale in Kubeless e possono essere messe in campo attraverso la Command Line Interface `kubeless`, oppure direttamente come risorsa Kubernetes `Function`.

I parametri distintivi di una funzione sono i seguenti:

Runtime Specifica la runtime da utilizzare e la sua versione. Deve corrispondere ad una delle runtime offerte da Kubeless.

Timeout Specifica il timeout massimo per la funzione. Al termine, l'esecuzione della funzione termina.

Handler Specifica quale funzione deve essere esposta ed il nome file all'interno del quale si trova, attraverso una notazione del tipo `<file_name>.<function_name>`.

Deps Specifica quali sono le dipendenze della funzione, nel formato specifico della routine.

Checksum Specifica l'hash SHA–256, autocalcolato, del contenuto della funzione.

Function Content Type Specifica il tipo della funzione, eventualmente con il flag `+zip` se si tratta di un archivio. I formati sono supportati sono `base64`, `text` e `url`.

Function Specifica il contenuto della funzione, a livello di codice, eventualmente codificato base64, oppure come URL.

Le funzioni per default vengono eseguite come utente non privilegiato, evitando quindi che abbiano permessi di root. Questo comportamento può comunque essere modificato indicando un diverso *Security Context* all'interno dei file di configurazione della funzione.

Interfaccia

Indipendentemente dalla routine utilizzata, tutte le funzioni utilizzano la stessa interfaccia e ricevono due parametri:

`event` Contiene le informazioni sulla sorgente dell'evento che la funzione ha ricevuto.

`context` Contiene informazioni generali riguardanti la funzione, ad esempio il suo nome o il timeout massimo.

Le funzioni devono restituire una stringa, che verrà incapsulata nella risposta HTTP.

2.1.2 Meccanismi publish/subscribe

Ciascuna funzione Kubeless può essere attivata attraverso un meccanismo di tipo PubSub, ponendosi in ascolto su un determinato topic di un sistema di messaggistica e consumando tali messaggi come proprio input. Attualmente, Kubeless supporta questo meccanismo utilizzando i sistemi di messaggistica Apache Kafka e NATS.

In questa tesi è stato utilizzato un cluster Apache Kafka. Per una breve descrizione di questa tecnologia, fare riferimento alla sezione 2.3.4.

Tra le varie *Custom Resource Definition* create da Kubeless, viene definita la risorsa `KafkaTrigger` che permette di associare una funzione Kubeless ad un topic Apache Kafka. In questo modo la funzione si pone in ascolto sul topic indicato e, al momento della produzione di un messaggio su tale topic, viene automaticamente attivata ricevendo come input il messaggio stesso.

2.1.3 Esposizione HTTP

Kubeless utilizza gli `Ingress` forniti da Kubernetes per fornire funzionalità di routing verso le funzioni.

Una funzione Kubeless appena messa in campo, per default, è associata ad un Kubernetes `Service` utilizzando il suo *ClusterIP*, in questo modo non viene esposta all'esterno del cluster. Per questo motivo, Kubeless offre strumenti per esporre pubblicamente tale funzione.

Al fine di poter creare route verso le funzioni in Kubeless, è necessario istanziare un *Ingress controller* in Kubernetes.

Kubeless supporta nativamente diverse soluzioni:

- Nginx Ingress;
- Kong Ingress;
- Traefik Ingress.

Una volta disponibile l'Ingress controller, è possibile creare oggetti del tipo HTTP Trigger che possono essere associati ad una funzione Kubeless. In questo modo, quando viene invocato l'host sul quale è esposto l'`Ingress`, eventualmente specificandone il path, viene attivata la corrispondente funzione Kubeless.

È inoltre possibile, con ulteriori configurazioni, abilitare il protocollo TLS inserendo in fase di messa in campo il certificato X.509 necessario, oppure abilitare il meccanismo di HTTP Basic Authentication.

2.2 Knative

Knative [16] è un framework basato su Kubernetes per mettere in campo e gestire applicazioni e carichi di lavoro sfruttando un'architettura serverless. Knative estende Kubernetes al fine di fornire un insieme di componenti per sviluppare applicazioni basate su container, che posso essere messe in campo in locale, in cloud oppure ancora in data center di terze parti.

In modo simile a Kubeless, i componenti Knative sono costruiti on top di Kubernetes, astraendo i complicati dettagli e permettendo agli sviluppatori di focalizzare l'attenzione sui proprio compiti specifici. Knative nasce con l'idea di uniformare le best practices condivise tra le implementazioni reali che hanno riscosso un elevato successo, risolvendo tutti i compiti più complessi riguardanti le fasi di messa in campo e gestione di servizi cloud. In particolare:

1. Messa in campo di un container;
2. Instradamento e gestione del traffico, ad esempio con approccio blue/green;
3. Scaling automatico e dimensionamento del carico di lavoro in relazione al numero di richieste;
4. Collegamento tra servizi già avviati e diversi sistemi ad eventi.

Le caratteristiche principali di questo framework sono:

- API con un elevato livello di astrazione per lo sviluppo di un'applicazione nei casi più comuni;
- Rapida messa in campo di un servizio serverless scalabile e sicuro;
- Funzionalità e moduli lascamente accoppiati, permettendo l'installazione delle sole componenti necessarie;
- Portabilità, poiché viene installato come un componente custom on top di un'installazione Kubernetes esistente.

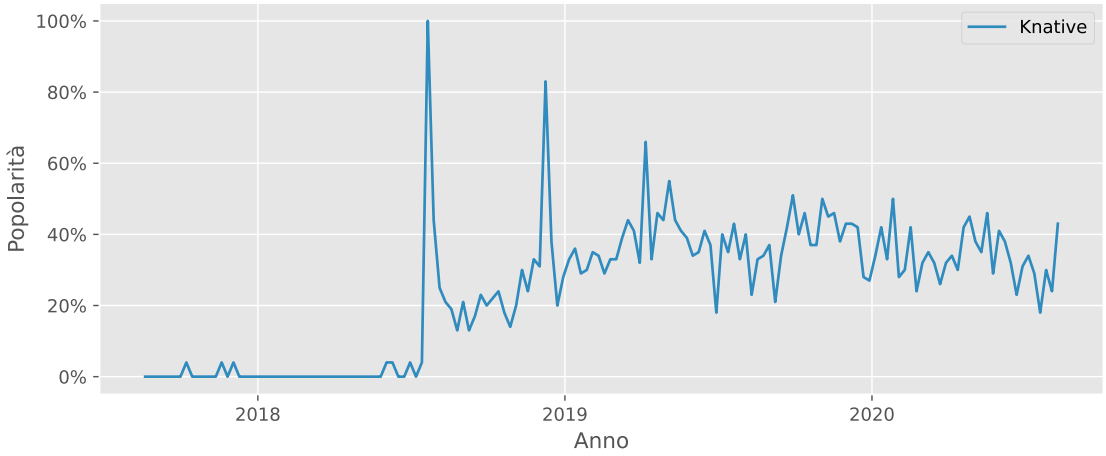

Figura 2.2. Grafico Google Trends di "Knative" degli ultimi tre anni

Knative necessita di un Ingress o Gateway capace di instradare le richieste ai servizi Knative. Al momento, tali funzionalità sono offerte dai seguenti framework:

- Ambassador [17] – *Ambassador* è un Gateway API open–source basato su Envoy (per i dettagli, fare riferimento alla sezione 2.3.2), nativo in Kubernetes. Ambassador permette di gestire solamente il traffico nord–sud[2], quindi il suo utilizzo è consigliato se non è necessario avere a disposizione una service mesh ed è quindi possibile evitare tale overhead;
- Contour [18] – *Contour* è un Ingress Controller open–source basato su Envoy. Contour soddisfa tutte le specifiche di rete e supporta tutte le feature offerte da Knative;
- Gloo [19] – *Gloo* è un Gateway API open–source basato su Envoy. Gloo si propone come un'alternativa più leggera, ma comunque completa rispetto ad Istio, in grado di supportare tutte le specifiche di Knative. Il suo utilizzo è consigliato se non è necessario avere a disposizione una service mesh e le risorse hardware sono limitate;
- Istio [20] – *Istio* è una Service Mesh open–source basata su Envoy che include un Ingress compatibile con Knative. Il suo utilizzo è consigliato se si vogliono sfruttare tutte le caratteristiche di una service mesh.

In questa tesi è stato utilizzato Istio, in seguito ad un'analisi della documentazione ufficiale, confrontata con lo scopo di questa tesi. Infatti, oltre ad essere la soluzione più completa e probabilmente più utilizzata, è necessario utilizzare una service mesh, ed è necessario utilizzare un Ingress Gateway per direzionare il traffico entrante nel cluster. I dettagli di Istio, sono illustrati nella sezione 2.3.1.

Knative è diviso in due componenti:

Serving Esegue i container in modalità serverless e si occupa dei dettagli a livello di connettività di rete, permettendo autoscaling e tracciamento delle revision. In questo modo, l'utente Knative può concentrarsi sulla logica.

Eventing Permette di costruire applicazioni che sfruttano stream di dati, occupandosi di iscrizione, consegna e gestione degli eventi attraverso un approccio dichiarativo e fornendo un modello ad oggetti a disposizione degli sviluppatori.

2.2.1 Serving

Knative Serving è un sistema costruito on top di Kubernetes ed Istio, progettato per supportare la messa in campo e l'esposizione di applicazioni e funzioni in modalità serverless. Questa componente offre le seguenti primitive:

- Rapida messa in campo di container serverless;
- Scaling automatico;
- Instradamento e configurazione di rete utilizzando i componenti di Istio;

[2]Con *traffico nord–sud* si intende il traffico tra il cluster e l'esterno del cluster. La terminologia deriva probabilmente dal modo in cui solitamente, nei diagrammi di rete, vengono illustrate la rete interna e la rete esterna.

- Snapshot del codice messo in campo con le relative configurazioni.

Knative Serving definisce un insieme di oggetti *Custom Resource Definition* in Kubernetes, mostrati nella figura 2.3, che vengono utilizzati per la definizione e per il controllo del comportamento delle applicazioni serverless messe in campo all'interno del cluster.

`Service` Gestisce in modo automatico l'intero ciclo di vita dell'applicazione, controllando la creazione ed il ciclo di vita di altri oggetti per garantire che l'applicazione possegga una `Route`, una `Configuration` ed una nuova `Revision` ad ogni cambiamento del `Service` in modo tale che il `Service` possa instradare il traffico all'ultima `Revision`, oppure ad una specifica.

`Route` Associa un endpoint ad una o più `Revision`, permettendo di gestire il traffico con diverse politiche, come il frazionamento del traffico e route con nome.

`Configuration` Controlla lo stato dell'applicazione messa in campo e ha il compito di mantenerne aggiornato lo stato, fornendo una chiara separazione tra il codice eseguito e la configurazione, in accordo con la *Metodologia dei dodici fattori*.

`Revision` Costituisce uno snapshot del codice e della relativa configurazione per ogni modifica apportata all'applicazione.

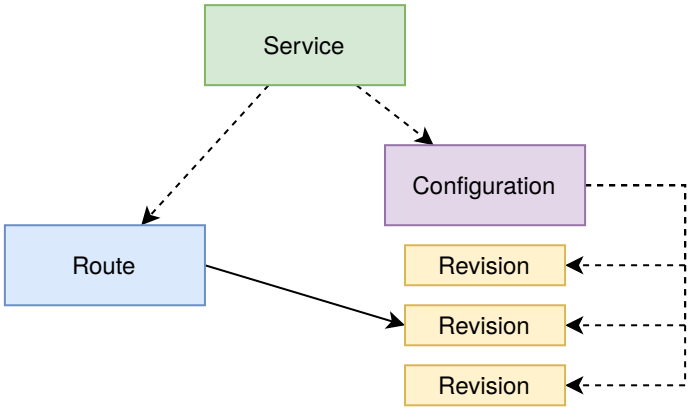

Figura 2.3. Knative Serving – Architettura

Partendo da un'immagine Docker, è possibile mettere in campo un `Knative Service` – oggetto diverso e più complesso rispetto ad un `Service` in Kubernetes – ed automaticamente Knative si occupa dei seguenti passaggi:

1. Creazione di una `Revision` immutabile per ogni versione dell'applicazione;
2. Creazione di una serie di risorse di rete per l'applicazione, quali `Route`, `Ingress` e corrispondente `Service` di Kubernetes;
3. Scaling automatico dei `Pod` in relazione al traffico, fino ad un numero massimo stabilito di repliche ed un numero minimo che può anche essere nullo in assenza di traffico per un determinato intervallo di tempo.

Architettura

L'infrastruttura di Knative Serving permette di raggiungere gli obiettivi indicati precedentemente. Questa componente richiede di impostare un nome di dominio che verrà utilizzato automaticamente da Knative per esporre all'esterno del cluster i `Service` Knative.

Activator Il servizio `activator` è responsabile della ricezione e dell'accodamento delle richieste dirette alle `Revision` non attive e si occupa di inviare metriche all'`autoscaler`. Inoltre, ha il compito di ritentare la consegna ad una `Revision` quando l'`autoscaler` modifica una `Revision` in base alle metriche riportate.

Autoscaler Il servizio `autoscaler` riceve le metriche relative alle richieste e abilita un adeguato numero di `Pod` per la corretta gestione di tale carico.

Controller Il servizio `controller` ha il compito di riconciliare e mantenere aggiornato lo stato degli oggetti pubblici di Knative. Ogni qual volta un utente installa un nuovo `Knative Service`, il servizio `controller` è responsabile di convertire la configurazione in una serie di `Revision`, e quindi in `Deployment` e Knative Pod Autoscaler.

Webhook Il servizio `webhook` intercetta le chiamate API dirette a Kubernetes e tutte le richieste di inserimento e modifica di *Custom Resource Definition*. Ha il compito di definire ed impostare valori di default, rifiutare oggetti non corretti o inconsistenti, validare e modificare le chiamate alle API di Kubernetes.

2.2.2 Eventing

Knative Eventing è un sistema progettato per ottenere un basso accoppiamento tra sorgenti di eventi e consumatori di eventi, con i seguenti obiettivi:

- Servizi lascamente accoppiati, in modo tale che questi possano essere messi in campo su diverse piattaforme, come Kubernetes, macchine virtuali, SaaS, FaaS;
- Produttori e consumatori di eventi indipendenti, in modo tale che ogni sorgente possa produrre eventi senza necessariamente avere un consumatore pronto a riceverli e che, analogamente, ogni consumatore possa dichiarare il proprio interesse ad un insieme di eventi senza necessariamente avere una sorgente che stia generando eventi;
- Servizi esterni possono essere connessi al sistema, permettendo alle nuove applicazioni di essere messe in campo senza modificare gli attuali produttori e consumatori;
- Interoperabilità tra i servizi, garantita poiché Knative Eventing è consistente con le specifiche Cloud Events sviluppate dalla Cloud Native Computing Foundation.

Sorgenti

Ciascuna *sorgente* offerta da Knative Eventing è una specifica *Custom Resource Definition* in Kubernetes, che permette di definire singolarmente gli argomenti ed i parametri specifici necessari ad istanziare la sorgente stessa.

Alcune delle sorgenti offerte da Knative sono:

- Apache Kafka – Permette di importare messaggi Apache Kafka all'interno di Knative;
- CronJob – Permette di produrre eventi periodicamente;
- Kubernetes – Permette di importare eventi del server API di Kubernetes all'interno di Knative;
- BitBucket, GitHub, GitLab – Permettono di importare eventi specifici generati dai repository all'interno di Knative.

Consumatori

Knative Eventing, al fine di permettere la consegna a differenti tipi di servizi, definisce due interfacce generiche che possono essere implementati da diverse risorse Kubernetes:

Addressable Oggetti in grado di ricevere ed accettare eventi trasmessi ad un indirizzo, definito nel campo `status.address.url`, attraverso il protocollo HTTP. I `Service` Kubernetes, come caso particolare, soddisfano tale interfaccia.

Callable Oggetti in grado di ricevere un evento trasmesso attraverso HTTP, trasformarlo ed inviare al più un nuovo evento nella risposta HTTP. Tali eventi, possono essere ulteriormente elaborati, nello stesso modo in cui vengono elaborati gli eventi generati da una sorgente esterna.

Broker e Trigger

A partire dalla versione 0.5 di Knative Eventing, sono stati introdotti gli oggetti *Broker* e *Trigger*, per facilitare l'inoltro ed il filtraggio degli eventi.

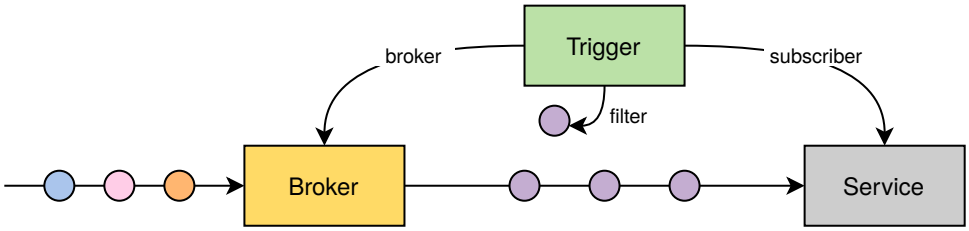

Figura 2.4. Knative Eventing – Broker & Trigger

- Un Broker riceve eventi e li invia ai subscriber, corrispondenti ad uno o più Trigger;
- Un Trigger descrive un filtro su qualche attributo degli eventi ed inoltra tale evento ad un oggetto `Addressable`.

Canali

Knative Eventing definisce un oggetto *Channel*, anche questo come una *Custom Resource Definition* di Kubernetes, con il compito di fornire un livello di persistenza che si occupa dell'inoltro di eventi.

Gli eventi vengono indirizzati ai servizi oppure inoltrati ai canali utilizzando le `Subscription`. I canali possono essere di differenti tipologie, in modo tale che messaggi di tipo diverso possano servirsi di tipi diversi di canali, in base a differenti requisiti.

I canali supportati da Knative Eventing sono attualmente:

- GCP PubSub – I canali corrispondono a Google Cloud Platform PubSub;
- InMemoryChannel – I canali sono di tipo *best effort*. Sono uno strumento utile in fase di sviluppo, ma non dovrebbero essere utilizzati in produzione. Questi, infatti, non supportano meccanismi di persistenza, non garantiscono la consegna, non tentano la riconsegna e non garantiscono l'ordinamento nell'invio e ricezione dei messaggi;
- KafkaChannel – I canali corrispondono a topic Apache Kafka;
- NatsChannel – I canali corrispondono a Streaming NATS.

In questa tesi, come suggerito dalla documentazione di Knative Eventing, sono stati utilizzati i canali in memoria nelle prime fasi di test ed i canali Kafka durante l'implementazione vera e propria.

Costrutti di alto livello

Knative offre alcune risorse utili nei casi in cui è necessario connettere e far cooperare diverse funzioni.

Sequence Una *Sequence*, definita come *Custom Resource Definition* in Kubernetes, fornisce un modo per descrivere una lista ordinata di funzioni. Ogni funzione della sequenza riceve un evento che può filtrare oppure elaborare ed inoltrare alla funzione successiva. Internamente, una `Sequence` si occupa di creare i necessari `Channel` e le necessarie `Subscription` per realizzare tale concatenazione. La figura 2.5 rappresenta, a livello logico, una `Sequence` Knative.

Ogni `Sequence` è caratterizzata da alcuni parametri:

`Steps` Una lista ordinata di `Subscriber`, ognuno dei quali implementa l'interfaccia `Addressable`.

`ChannelTemplate` Un template utilizzato per creare i `Channel` che collegano i diversi `Step`.

`Reply` Un parametro opzionale per referenziare un oggetto da invocare al termine dell'elaborazione della `Sequence`, al quale inviare l'evento generato dall'ultimo `Step`.

Ogni `Sequence` è caratterizzata da uno stato:

`Conditions` Fornisce dettagli sullo stato generale della `Sequence`.

`ChannelStatuses` Fornisce dettagli sullo stato dei singoli `Channel`, sotto forma di array per facilitarne la lettura in relazione agli `Step`.

`SubscriptionStatuses` Fornisce dettagli sullo stato delle singole `Subscription`, sotto forma di array per facilitarne la lettura in relazione agli `Step`.

`AddressStatus` Fornisce informazioni sugli indirizzi utilizzati dalle risorse `Addressable`. L'invio di dati a questi indirizzi attiva il corrispondente `Channel` posto logicamente prima del relativo `Step`.

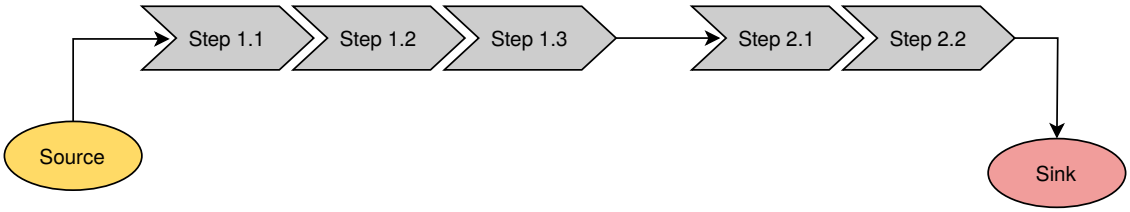

Figura 2.5. Knative Eventing – `Sequence`

Parallel Una *Parallel*, anch'essa definita come *Custom Resource Definition* in Kubernetes, fornisce un modo per descrivere una lista di *branch* per gli eventi.

Un oggetto di tipo `Parallel` definisce quali sono i suoi branch, ognuno dei quali è costituto da una lista di `Subscriber` ed opzionalmente da una `Reply` in modo simile ad una `Sequence`, un `ChannelTemplate` per indicare il tipo di `Channel` da utilizzare ed una `Reply` opzionale che raccoglie il risultato di ciascun branch che non possiede la propria `Reply`.

La figura 2.6 rappresenta, a livello logico, una `Parallel` Knative.

Ciascuna `Parallel` è caratterizzata da parametri simili ad una `Sequence` Knative e da una serie di valori che ne identificano lo stato corrente.

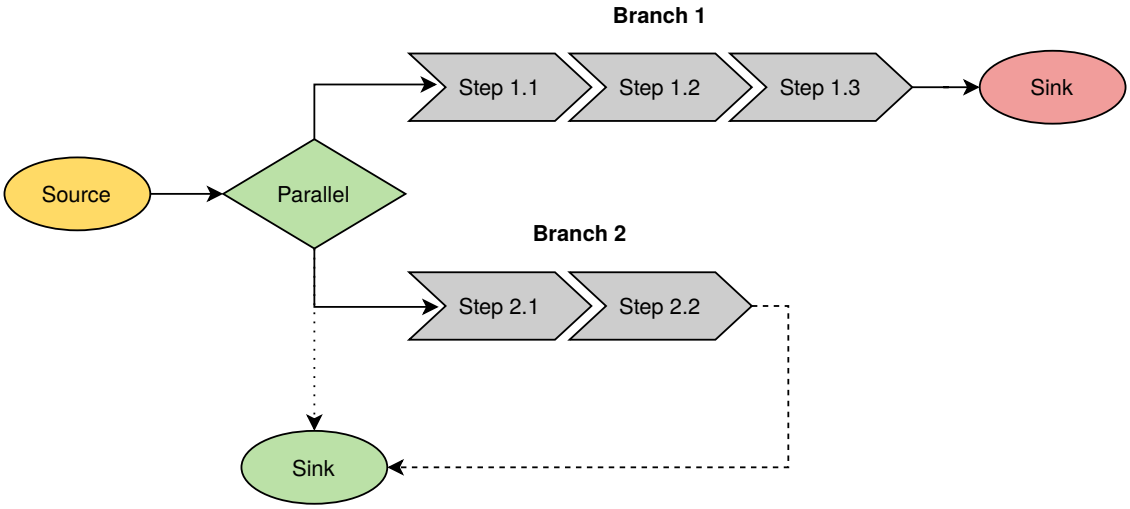

Figura 2.6. Knative Eventing – `Parallel`

Architettura

L'infrastruttura di Knative Eventing supporta due modalità per la consegna di eventi.

Consegna diretta La *consegna diretta* consiste nella consegna, da parte di una sorgente, ad un singolo servizio, il quale deve implementare l'interfaccia `Addressable`, ad esempio un `Knative Service` o un `Service` Kubernetes. La sorgente ha il compito di ritentare oppure accodare gli eventi se la destinazione non è disponibile.

Figura 2.7. Knative Eventing – Consegna diretta di eventi

Consegna multipla La *consegna multipla* consiste nella consegna, da parte di una sorgente, a differenti servizi, attraverso un sistema di `Channel` e `Subscription`. L'implementazione stessa dei `Channel` garantisce che i messaggi siano recapitati alla destinazione e ha il compito di accodare eventi se la destinazione non è disponibile.

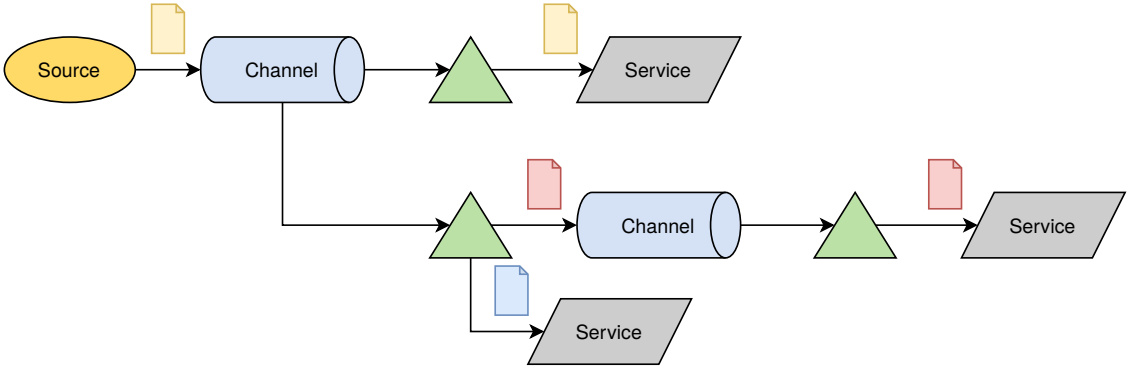

Figura 2.8. Knative Eventing – Consegna multipla di eventi

2.2.3 Apache Kafka e Knative

Apache Kafka può essere utilizzato in Knative sotto forma di `KafkaChannel` oppure `KafkaSource`, componenti che permettono di integrare Knative con il broker Apache Kafka.

Una risorsa `KafkaSource` svolge il compito di portare i messaggi Apache Kafka all'interno della rete Knative Eventing.

La risorsa `KafkaChannel` costituisce una delle possibili implementazioni di un canale Knative e si appoggia un topic Apache Kafka.

L'integrazione tra Knative ed Apache Kafka avviene attraverso le seguenti componenti Knative:

Kafka Channel Controller Ha il compito di mantenere allineata la configurazione delle componenti, controllando periodicamente ed eventualmente aggiornando opportunamente la configurazione delle risorse.s

Kafka Channel Dispatcher Ha il compito di ricevere e distribuire gli eventi agli opportuni consumatori.s

Kafka Webhook Ha il compito di validare ed impostare i parametri di default per gli oggetti `KafkaChannel`.s

Kafka Config Map Contiene i parametri di configurazione per accedere ai server di bootstrap di Apache Kafka.

2.3 Software ausiliari

Accanto ai framework sopra illustrati, sono state utilizzate alcune altre tecnologie accessorie per completare la messa in campo, il testing, il monitoraggio ed in generale l'analisi delle due applicazioni. In particolare, la realizzazione di questo lavoro richiede l'introduzione delle seguenti tecnologie:

- *Istio* è necessario per soddisfare le dipendenze di Knative e viene utilizzato come Ingress Gateway da e verso l'esterno del cluster e per realizzare una rete di servizi interna al cluster;
- Una descrizione delle specifiche *CloudEvents* è necessaria poiché il meccanismo ad eventi utilizzato da Knative nel mondo Knative rispetta tali specifiche;
- *Apache Kafka* viene utilizzato come meccanismo per l'attivazione di alcune funzioni nell'applicazione serverless e come canale per la trasmissione di eventi;
- *Prometheus* è necessario per raccogliere le metriche dell'applicazione ed interrogare i dati raccolti;
- *Grafana* viene utilizzato per la visualizzazione grafica, l'elaborazione delle metriche raccolte da Prometheus;
- *Apache JMeter* viene utilizzato per la fase di testing delle applicazioni, poiché permette l'invio di richieste HTTP e la raccolta di risultati.

2.3.1 Istio

Istio [20] è una piattaforma open–source, scritta in Go, che permette di connettere, gestire e securizzare microservizi, con lo scopo di integrare e gestire il traffico tra questi ultimi, implementare politiche di sicurezza ed aggregare dati di telemetria. Istio costituisce un livello di astrazione rispetto all'architettura del cluster sottostante, quale ad esempio Kubernetes.

Istio cerca di ridurre la complessità dello scenario DevOps, nel quale gli sviluppatori utilizzano i microservizi per garantire una maggiore portabilità e gli operatori gestiscono architetture ibride molto vaste e complesse, costruendo una mesh di servizi trasparente alle applicazioni distribuite sottostanti ed offrendo una serie di API che permettono di integrare piattaforme di logging e monitoring.

Istio permette di creare una rete di servizi con caratteristiche aggiuntive di bilanciamento del carico, mutua autenticazione tra servizi e monitoring, effettuando poche, talvolta nessuna, modifiche al codice sorgente dell'applicazione.

Le caratteristiche principali offerte da Istio per l'intera rete di servizi sono:

- Gestione del traffico – La configurazione delle regole ed il routing del traffico permettono di controllare il flusso del traffico e delle chiamate API tra i diversi servizi;
- Sicurezza – Le politiche di sicurezza offerte nativamente da Istio permettono agli sviluppatori di concentrarsi sulla sicurezza a livello applicativo, poiché è Istio ad occuparsi della sicurezza di canale, dell'autenticazione, dell'autorizzazione e della crittografia.

  L'integrazione di Istio con Kubernetes permette di securizzare la comunicazione tra i `Pod` o tra i `Service`, sia a livello di rete, sia a livello applicativo;

- Policy – Istio permette la configurazione di regole dinamiche, verificate a runtime, come il controllo dinamico del traffico, gestione di whitelist e blacklist per limitare l'accesso, re–indirizzamento e riscrittura di header;
- Osservabilità – I meccanismi di logging, monitoring e tracciamento offerti da Istio permettono di ottenere importanti informazioni e statistiche riguardanti i servizi sviluppati.

2.3.2 Envoy

Envoy [21] è un proxy progettato per le applicazioni in cloud, parte della Linux Foundation. Nasce per facilitare lo sviluppo di un'architettura a microservizi, che inevitabilmente incontra problemi nelle configurazioni di rete e nell'osservabilità del sistema, tipicamente con difficoltà molto superiori ad un'applicazione monolitica.

Envoy è un proxy distribuito ad alte prestazioni open–source realizzato da Lyft, scritto in C++, progettato per singoli servizi ed applicazioni, per bus di comunicazione ed anche per architetture a microservizi che necessitano una mesh di servizi. Envoy si ispira a soluzioni ben note, come NGINX, bilanciatori hardware e bilanciatori in cloud, astraendo i dettagli del livello di rete e fornendo una piattaforma comune indipendente. Il traffico all'interno della mesh realizzata da Envoy viene monitorato, permettendo una più facile individuazione dei problemi, ed eventuale calibratura delle prestazioni.

2.3.3 CloudEvents

CloudEvents [22], parte del gruppo di lavoro Serverless della Cloud Native Computing Foundation, propone una specifica, ancora in via di sviluppo, per la descrizione di eventi utilizzando un linguaggio comune, indipendente da vendor e piattaforma.

Il vasto utilizzo degli eventi porta ad una descrizione differente di questi in base al contesto. In assenza di una piattaforma comune, gli sviluppatori devono adattare le proprie conoscenze in base al contesto e l'utilizzo di librerie ed architetture universali risulta quindi impossibile, riducendo drasticamente portabilità, produttività ed interoperabilità.

Le specifiche CloudEvents definiscono mapping tra diversi protocolli e codifiche, mentre l'SDK CloudEvents copre varie runtime e linguaggi. Un singolo CloudEvent potrebbe essere instradato attraverso diversi hop che potrebbero fare uso di diversi protocolli, codifiche ed attributi. Le specifiche si limitando quindi a definire l'insieme di caratteri utilizzabili per ciascun attributo.

Attributi

Gli attributi, noti anche come *context attributes*, hanno il compito di descrivere l'evento e sono progettati per essere serializzati in modo indipendente dai dati, permettendo la loro ispezione senza dover deserializzare anche il campo dati.

Alcuni tra gli attributi definiti dalle specifiche sono obbligatori:

`id` Identifica l'evento.

Il produttore dell'evento ha il compito di garantire che la coppia di attributi `id` e `source` sia univoca per ciascun evento, poiché il consumatore potrebbe interpretare un evento con stessi `id` e `source` come duplicato di un evento precedente.

`source` Identifica il contesto nel quale l'evento viene scatenato. Questo attributo spesso include informazioni sulla sorgente dell'evento.

È compito dell'applicazione scegliere se utilizzare UUID, URN, nomi DNS autoritativi oppure schemi specifici definiti all'interno del contesto applicativo.

`specversion` Indica la versione delle specifiche CloudEvents utilizzata dall'evento, permettendo l'interpretazione del contesto.

`type` Descrive il tipo dell'evento in relazione all'oggetto che lo ha generato.

Questo attributo è spesso utilizzato per instradamento, osservabilità o applicazione di policy custom. Dovrebbe essere preceduto dal nome reverse–DNS.

Tra gli attributi opzionali è possibile inserire:

`datacontenttype` Definisce il tipo di dato contenuto nel campo `data`. In alcuni casi, questo valore può essere omesso per i formati più comuni come, ad esempio, il formato JSON.

`dataschema` Identifica lo schema che il campo `data` deve rispettare. Se presente, questo campo deve essere valorizzato ad un URI non vuoto.

`subject` Descrive il soggetto dell'evento prodotto dalla sorgente.

L'utilizzo di questo attributo è utile se sono presenti componenti intermediari che non hanno accesso al campo `data`, ad esempio perché protetto da crittografia.

`time` Timestamp dell'istante in cui l'evento è stato generato.

I CloudEvents possono includere informazioni specifiche relative al dominio applicativo. Quando presenti, queste devono essere incapsulate nel campo `data`, rispettare il formato specificato nel campo `datacontenttype` e lo schema definito nel campo `dataschema`, se questi due attributi sono presenti.

Privacy e sicurezza

Al fine di permettere l'interoperabilità, obiettivo primario delle specifiche, molte delle informazioni devono viaggiare in chiaro con evidenti problemi di potenziale perdita delle proprietà di sicurezza.

Per evitare pericolosi data leak, informazioni sensibili non dovrebbero essere inserite nei valori degli attributi in quanto ogni istanza della catena produttori–intermediari–consumatori può accedere a tali campi, ispezionarne e salvarne il contenuto. Al fine di permettere l'accesso ai dati solamente da parte dei sistemi ai quali questi sono indirizzati, è necessario utilizzare opportune tecniche crittografiche, le quali sono al di fuori delle specifiche. Infine, possono essere impiegati protocolli di sicurezza che garantiscano lo scambio in modo sicuro e fidato delle informazioni contenute nei CloudEvents.

2.3.4 Apache Kafka

Apache Kafka [23] è una piattaforma distribuita di streaming. Qualunque piattaforma di streaming possiede tre caratteristiche chiave:

1. Pubblicazione di messaggi su uno stream di eventi, sul quale ne viene permessa l'iscrizione;

2. Memorizzazione di stream di messaggi con tolleranza ai guasti e garanzia della durabilità delle informazioni;

3. Processamento di stream di messaggi.

Apache Kafka viene eseguito all'interno di un cluster in modo scalabile su uno o più server possibilmente anche dislocati geograficamente. Ciascun cluster è in grado di memorizzare stream di messaggi che vengono raggruppati in categorie chiamate *topic*. Ogni messaggio è identificato da una chiave, un valore – che corrisponde al suo contenuto – ed il suo timestamp.

Apache Kafka è costituito da quattro API fondamentali:

Producer API Permette ad un'applicazione di pubblicare messaggi su uno o più topic Apache Kafka.

Consumer API Permette ad un'applicazione di iscriversi ad uno o più topic Apache Kafka e processarne i messaggi prodotti.

Streams API Permette ad un'applicazione di comportarsi come stream processor, consumando i messaggi provenienti da uno o più topic Apache Kafka e producendo un messaggio su uno o più topic Apache Kafka, trasformando eventualmente i messaggi.

Connector API Permette di costruire ed eseguire produttori o consumatori riutilizzabili per connettere topic Apache Kafka ad applicazioni esistenti o sistemi di persistenza dei dati. Ad esempio un connector associato ad una base di dati relazionale è in grado di catturare ogni cambiamento in una specifica tabella.

In Apache Kafka, la comunicazione tra client e server avviene attraverso il protocollo TCP, che garantisce semplicità, alte prestazioni ed indipendenza dal linguaggio.

2.3.5 Prometheus

Prometheus [24] è una piattaforma di monitoring ed alerting open–source scritta in Go, progetto parte della Cloud Native Computing Foundation.

Prometheus offre un modello di dati multi–dimensionale, con serie temporali identificate da un nome ed una serie di coppie chiave–valore, che può essere interrogato con un linguaggio flessibile, noto come PromQL[3], adatto alla cardinalità ed alla quantità dei dati. Inoltre, permette di memorizzare le sequenze temporali in memoria oppure su disco locale e, caratteristica ancora più importante, permette di visualizzare tali dati in varie modalità, tra cui l'integrazione con Grafana, i cui dettagli sono riportati nella sezione 2.3.6.

Prometheus risulta particolarmente adatto quando è necessario lavorare con dati numerici ed è utile per la visualizzazione di statistiche globali del sistema. Non risulta invece adatto quando sono necessarie informazioni puntuali e dettagliate, in quanto le informazioni raccolte non sono sufficientemente complete. In questi ultimi casi, infatti, è suggerito l'utilizzo di altri sistemi per ottenere, aggregare ed analizzare i dati, mentre è sempre possibile utilizzare Prometheus come strumento per il monitoring del sistema.

[3]Prometheus Query Language.

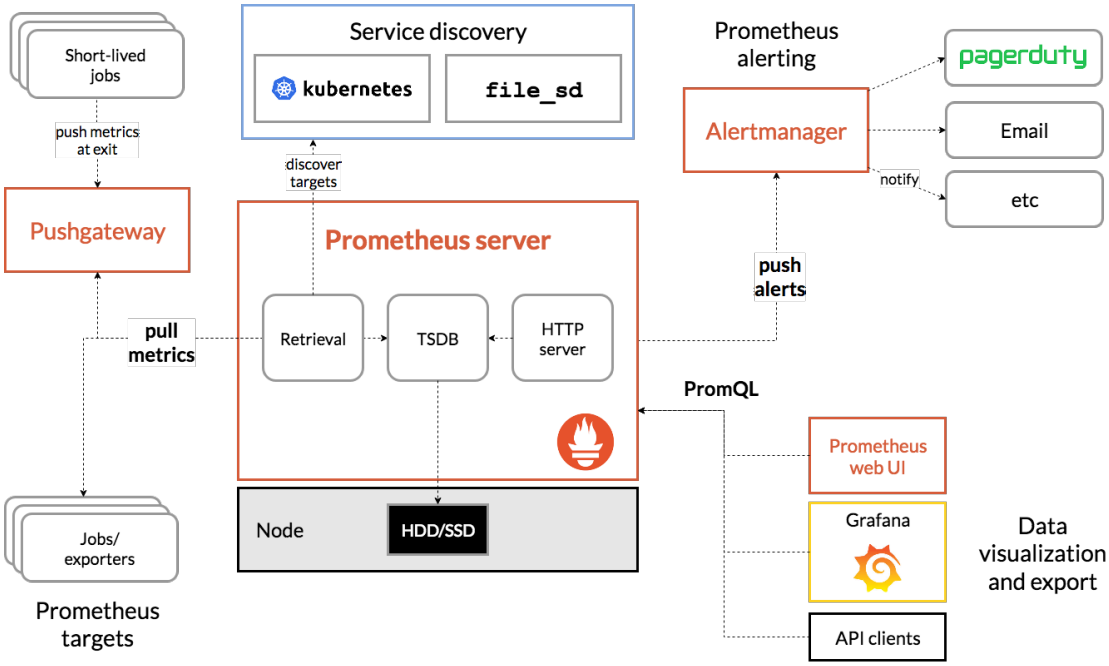

Figura 2.9. Prometheus – Architettura

La figura 2.9 mostra l'architettura di Prometheus ed i suoi componenti. Prometheus si occupa della lettura di metriche provenienti dai `Job` in Kubernetes, direttamente oppure attraverso un *Push gateway* intermediario, le memorizza localmente ed esegue su di esse una serie di operazioni per aggregare tali metriche, eventualmente creare nuove serie di dati, oppure lanciare alert al raggiungimento di determinate soglie. Grafana ed altri strumenti possono essere utilizzati per interrogare Prometheus, direttamente oppure attraverso API, in modo tale da visualizzarne, o ulteriormente elaborarne, i dati raccolti.

2.3.6 Grafana

Grafana [25] è una piattaforma open–source scritta in TypeScript e Go, volta ad analizzare e monitorare qualunque tipo di base di dati. Grafana permette di interrogare, visualizzare e generare avvisi in base a metriche, indipendentemente dal sistema su cui queste sono memorizzate, ad esempio su Prometheus.

Grafana offre strumenti come istogrammi, grafici a torta e mappe di calore per facilitare la visualizzazione dei dati, accessibili con diversi tipi di autenticazione in modo da garantire la separazione dei compiti secondo le politiche necessarie.

2.3.7 Apache JMeter

Apache JMeter [26] è un'applicazione open–source scritta interamente in Java, realizzata con l'obiettivo di testare il comportamento delle applicazioni web e misurarne le prestazioni. Può essere utilizzata per simulare pesanti carichi di lavoro lato server al fine di analizzarne la robustezza ed il comportamento in diversi scenari.

Apache JMeter permette, ad esempio, di testare applicazioni utilizzando il protocollo HTTP, permettendo di impostare header e corpo delle richieste e registrare i risultati, quali codice di stato della risposta, corpo della risposta e misure di performance quali il tempo di risposta.

Capitolo 3

Applicazione

Al fine di poter studiare il comportamento di un'architettura serverless e FaaS in un caso pseudo reale, è stata realizzata un'applicazione basilare che simula una biblioteca. L'applicazione è chiaramente un esempio, in quanto l'obiettivo primario di questa tesi non è mettere in campo un'applicazione completa, bensì lo scopo è evidenziare quali e quanti vantaggi, benefici e svantaggi si possono ottenere dall'utilizzo di un'architettura a funzioni. Entrambe le applicazioni sono state scritte in Go [27], in modo tale da sfruttare il più possibile il codice già scritto e non inficiare le prestazioni con possibili ottimizzazioni effettuate in fase di compilazione o runtime proprie del linguaggio di programmazione.

Partendo da un'applicazione monolitica per la gestione di prestiti di libri all'interno di una biblioteca che espone alcune API REST, è stata creata un'applicazione con funzionalità equivalenti sfruttando un'architettura mista serverless e FaaS. L'applicazione, per completezza, utilizza uno strato di persistenza basato su MongoDB [28]. La scelta di una tecnologia non relazionale è motivata dalla limitatezza e semplicità dello scenario, che non richiede uno schema rigido, anzi, l'assenza di tale schema ha permesso di modificare piccole parti del modello dei dati durante lo sviluppo dell'applicazione.

3.1 Scelta del framework

Dopo aver testato entrambi i framework, Kubeless e Knative, utili per lo sviluppo di un'architettura a funzioni, è stato necessario scegliere quale tra i due utilizzare nella costruzione del caso d'uso scelto.

In questa sezione vengono confrontate nel dettaglio le risorse utilizzate dai due framework illustrate in termini di utilizzo CPU e memoria RAM. È bene specificare che nel mondo Kubernetes, la misura dell'utilizzo CPU viene effettuata in *unità cpu* [29]. Un'*unità cpu* è equivalente ad una *virtual CPU* o *virtual core* per i cloud providers, oppure ad un *hyperthread* per i prodotti Intel. Spesso può essere utile utilizzare sottomultipli del *core*, definendo ad esempio il *millicore* come la millesima parte di un *core*, abbreviato come *mcore*, ad indicare che viene sfruttata solamente una parte della capacità computazionale di una CPU.

3.1.1 Kubeless

Kubeless risulta essere molto più leggero nell'utilizzo di memoria e CPU. Inoltre non ha nessuna dipendenza esterna e quindi l'installazione risulta essere notevolmente più rapida. Offre un'interfaccia a riga di comando attraverso il comando `kubeless` che semplifica molto la messa in campo delle funzioni. Data la sua estrema semplicità, però, non dispone di costrutti di alto livello e risulta quindi non particolarmente adatto in un contesto in cui sia necessario mettere in successione funzioni logicamente collegate al fine di creare un flusso di operazioni o di dati. Per implementare una sorta di meccanismo di concatenazione di funzioni in Kubeless è infatti necessario inserire porzioni di codice che permettano ad una funzione di scrivere messaggi su un sistema di messaggistica, ad esempio Apache Kafka, sul quale la funzione successiva nella sequenza è posta in ascolto attraverso un Trigger coerente con il sistema di messaggistica utilizzato. In alternativa, si dovrebbe lanciare, sempre a livello programmatico, una richiesta HTTP diretta al microservizio da invocare, il quale deve essere posto in ascolto attraverso un Trigger di tipo HTTP.

Per utilizzare il meccanismo di concatenazione che sfrutta i messaggi Apache Kafka, è necessario installare, oltre al *controller manager* presente per default, il controller *Kafka trigger controller*. Il primo si occupa di mantenere coerente lo stato degli oggetti Kubernetes rispetto alle risorse create ed amministrate con il comando `kubeless`, mentre il secondo si occupa della gestione della comunicazione attraverso topic Apache Kafka. Le figure 3.1 e 3.2 mostrano il consumo di risorse di questi due controller, rispettivamente come utilizzo CPU ed utilizzo memoria, messi in campo come `Deployment` Kubernetes.

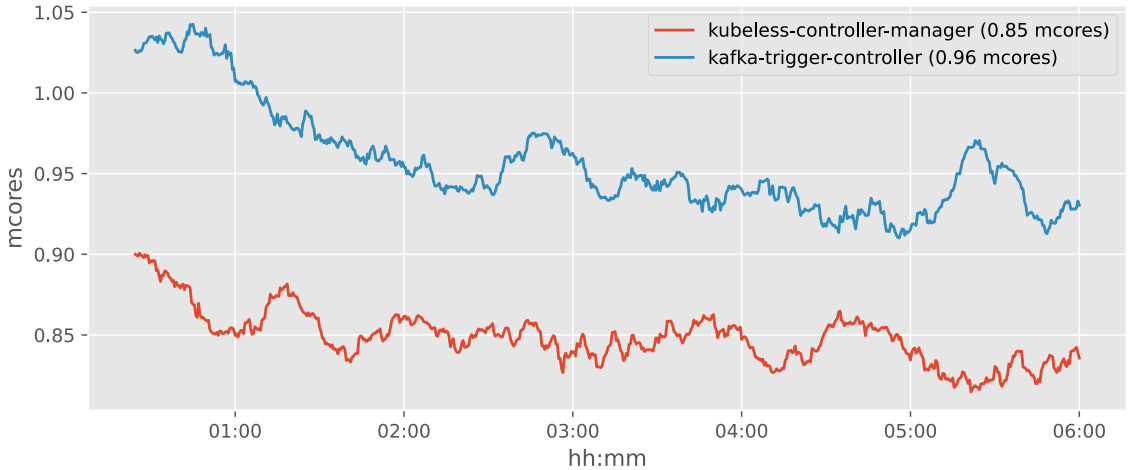

Figura 3.1. Kubeless – Utilizzo CPU

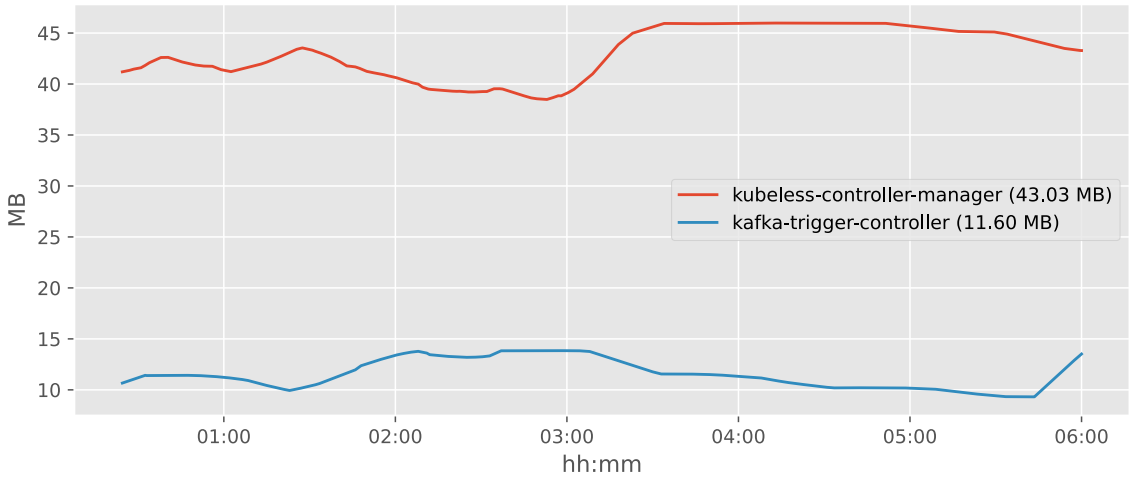

Figura 3.2. Kubeless – Utilizzo memoria

3.1.2 Knative

Nel framework Knative, invece, sono presenti costrutti di più alto livello come `Sequence` o `Parallel` che permettono di semplificare l'architettura dell'applicazione in quanto consentono l'utilizzo di un approccio dichiarativo anziché programmatico. Knative, inoltre, offre di base complesse politiche di autoscaling, messe in campo come sidecar container associati ai suoi `Deployment`, ed è facilmente estendibile con sorgenti di vario tipo, che forniscono una discreta varietà di modalità per attivare le funzioni. Di contro, data la complessa architettura, risulta molto più pesante nell'utilizzo di risorse, considerando anche la dipendenza dal framework Istio e la messa in campo di codice risulta più macchinosa e meno immediata, in quanto non è presente un'interfaccia a riga di comando ed è necessario effettuare manualmente la build di immagini Docker prima di poter procedere alla messa in campo del codice, compito svolto invece internamente dal comando `kubeless` per quanto riguarda il framework mostrato precedentemente.

Le figure 3.3 e 3.4 mostrano il consumo dei `Namespace` creati ed utilizzati da Knative per il suo completo funzionamento, ciascuno corrispondente ad una sua specifica funzionalità.

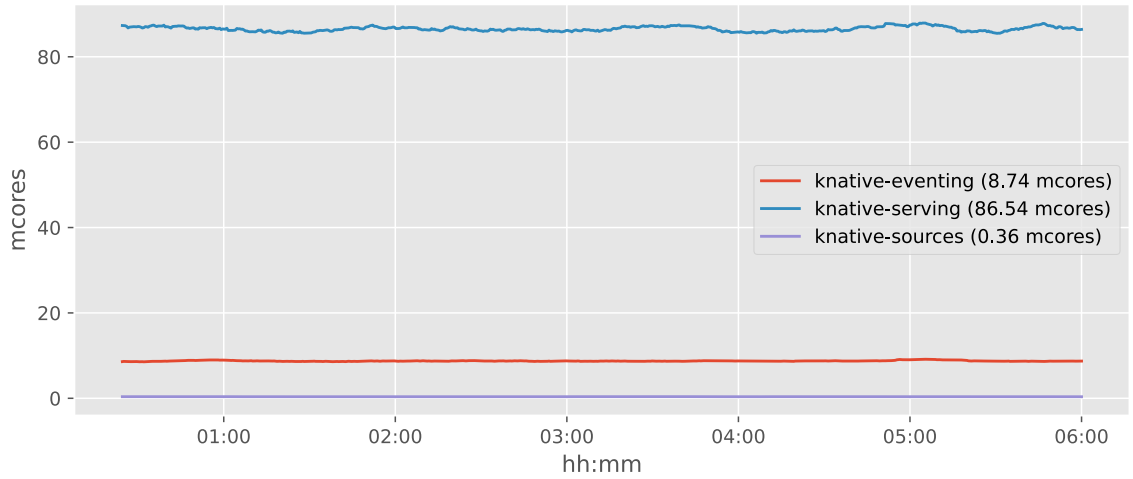

Figura 3.3. Knative – Utilizzo CPU

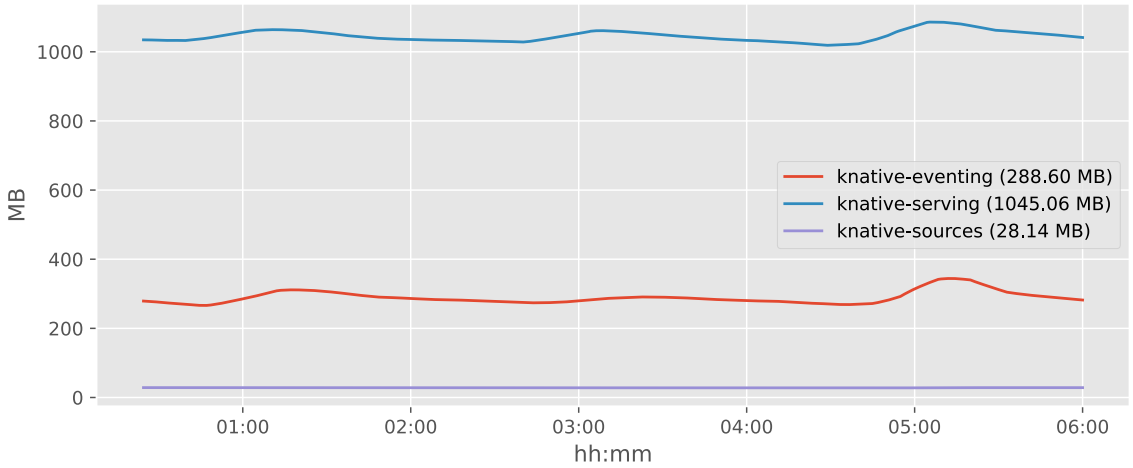

Figura 3.4. Knative – Utilizzo memoria

Dal confronto tra le figure precedenti è evidente come Knative richieda molte più risorse hardware, necessarie poiché si presenta come un framework più completo e versatile per la realizzazione di applicazione serverless e FaaS. Tra queste risorse, è necessario considerare anche le *Knative sources*, responsabili dell'attivazione delle funzioni, come mostrato nella sezione 2.2.2.

3.1.3 Popolarità

La figura 3.5 mette a confronto la popolarità dei due framework sopra citati in base alle ricerche Google. È subito evidente come Knative compaia più tardi nel panorama dei framework serverless/FaaS rispetto a Kubeless, ma riscuota dopo pochissimo tempo un grande interesse, favorito, probabilmente, in gran parte dal supporto di Google e sicuramente per la sua maggiore versatilità e completezza nello sviluppo di applicazioni. Nonostante ciò, Kubeless continua a riscuotere un

discreto e costante successo, seppur in modo decisamente minore, probabilmente a causa della sua rapidità e facilità di installazione e configurazione, combinati ad una minima richiesta di risorse.

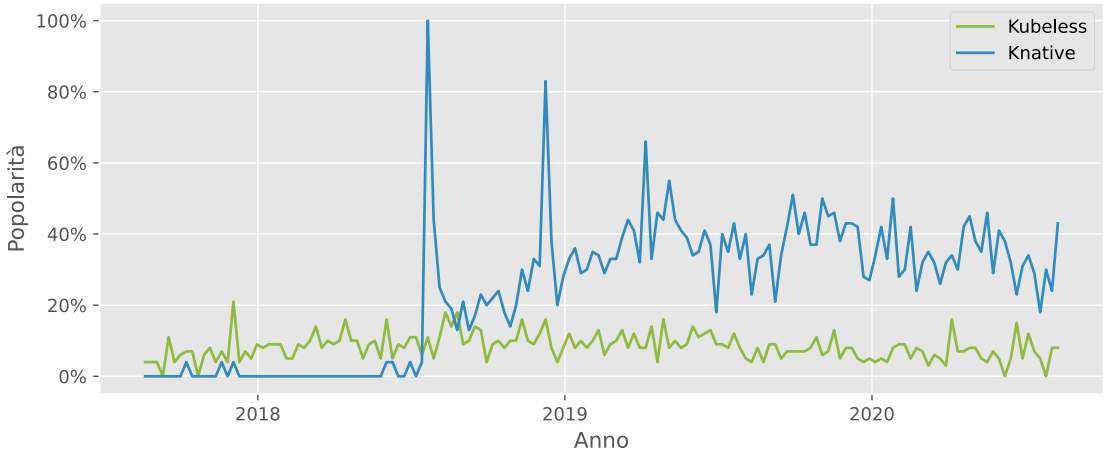

Figura 3.5. Grafico Google Trends di "Kubeless" e "Knative" negli ultimi tre anni

3.1.4 Istio

Il `Namespace` contenente Istio è stato volutamente omesso, in quanto questo servizio di gateway e routing da e verso l'esterno del cluster potrebbe essere già presente in una comune installazione di Kubernetes utilizzata in un ambiente reale. Inoltre, un Ingress gateway sarebbe necessario anche nell'installazione di Kubeless per poter beneficiare degli *HTTP Trigger*, i quali hanno il compito di intercettare le richieste in ingresso ed attivare le funzioni Kubeless.

Al fine di permettere il passaggio di traffico all'interno del cluster Knative, è necessario installare un ulteriore gateway locale fornito da Istio, più comunemente noto come *cluster local gateway*. In questo modo, è possibile configurare alcune `Revision` ad utilizzare `Route` locali, cioè disponibili solamente all'interno del cluster.

In figura 3.6 sono mostrate le risorse, in termini di CPU e memoria RAM utilizzate dall'installazione di Istio.

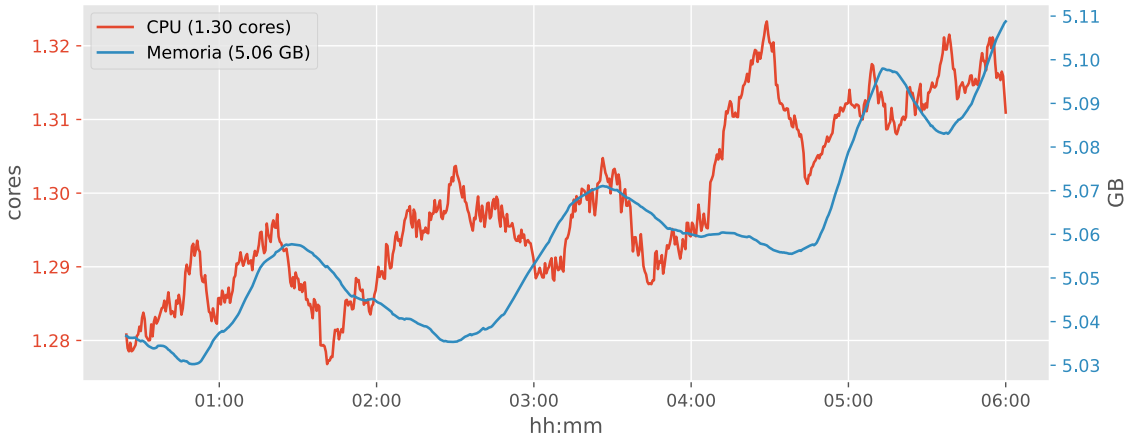

Figura 3.6. Istio – Risorse utilizzate

3.1.5 Apache Kafka

L'installazione di Apache Kafka è necessaria per permettere la comunicazione tra le varie funzioni, ed è una peculiarità di questa soluzione, quindi è necessario considerare questa dipendenza tra i costi, in termini di risorse, di questa architettura.

In figura 3.7 sono mostrate le risorse, in termini di CPU e memoria RAM utilizzate dall'installazione di Apache Kafka.

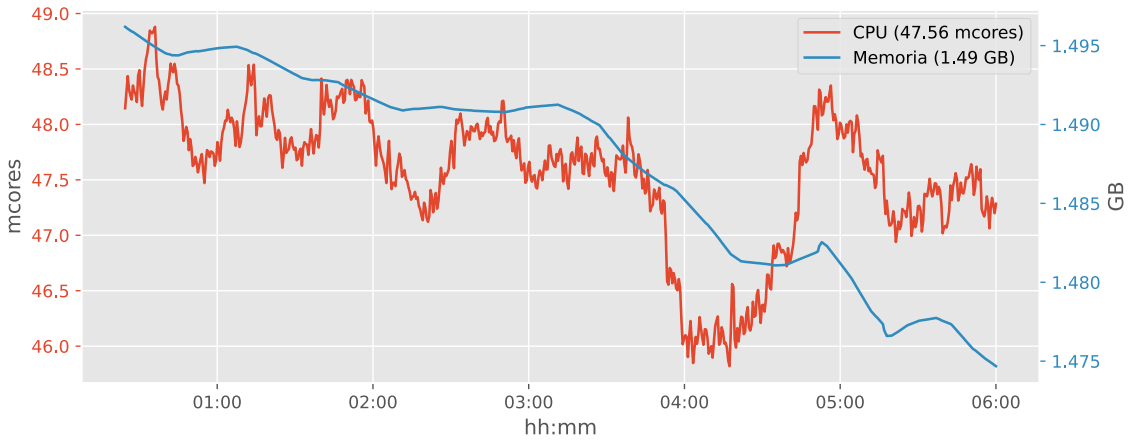

Figura 3.7. Apache Kafka – Risorse utilizzate

3.2 Applicazione monolitica

In questo scenario di esempio, si suppone che, attraverso le API esposte, il catalogo dei libri non possa mutare nel tempo e che i clienti non possano essere modificati o eliminati successivamente all'inserimento. Inoltre, per semplicità, non sono state messe in campo metodologie di sicurezza per proteggere l'accesso alle API, né a livello applicativo, né a livello di rete o trasporto.

La tabella 3.1 mostra gli endpoint di una API REST di tale applicazione:

- L'endpoint `GET /books` permette la lettura di tutti i libri presenti nella biblioteca, eventualmente filtrando i risultati con *Query string* `Author` e `Category` per indicare rispettivamente un autore del libro o una delle categorie di appartenenza. L'API risponde `200 Ok` ed una lista di libri, ciascuno di questi in accordo con il modello JSON mostrato nel riquadro 3.1;
- L'endpoint `GET /books/{isbn}` permette la lettura di un singolo libro, noto il suo codice univoco ISBN. L'API risponde `200 Ok` se il libro è presente nella biblioteca e le relative informazioni nel corpo della risposta, altrimenti risponde `404 Not found`;
- L'endpoint `GET /customers` permette la lettura di tutti gli utenti presenti nella biblioteca, eventualmente filtrando i risultati con *Query string* `FirstName` e `LastName` per indicare rispettivamente il nome oppure il cognome del cliente. L'API risponde `200 Ok` ed una lista di utenti, ciascuno di questi in accordo con il modello JSON mostrato nel riquadro 3.2;
- L'endpoint `GET /customers/{id}` permette la lettura di un singolo utente, noto il suo codice univoco all'interno della biblioteca. L'API risponde `200 Ok` se l'utente è presente nella biblioteca e le relative informazioni nel corpo della risposta, altrimenti risponde `404 Not found`;
- L'endpoint `POST /customers` permette di inserire un nuovo utente nella biblioteca. Il formato del corpo della richiesta deve rispettare il modello nel riquadro 3.2. L'API risponde `201 Created` se la richiesta ha successo, `400 Bad request` se la richiesta non è ben formata, altrimenti `409 Conflict` se l'utente è già presente nella biblioteca;
- L'endpoint `POST /loans` permette di inserire un nuovo prestito nella biblioteca. Il formato del corpo della richiesta deve rispettare il modello nel riquadro 3.3. L'API risponde `201 Created` se la richiesta ha successo, `400 Bad request` se la richiesta non è ben formata, `404 Not found` se l'utente o il libro indicati non sono presenti nella biblioteca, altrimenti `409 Conflict` se non è possibile prendere in prestito il libro per qualche conflitto, ad esempio perché già preso in prestito da un altro utente;
- L'endpoint `PUT /loans` permette di inserire un reso nella biblioteca. Il formato del corpo della richiesta deve rispettare il modello nel riquadro 3.4. L'API risponde `202 Accepted` se la richiesta ha successo, `400 Bad request` se la richiesta non è ben formata, altrimenti `404 Not found` se l'utente, il libro oppure il corrispondente prestito non è presente nella biblioteca.

Request				**Response**	
Method	URI	Parameters	Body	Code	Body
`GET`	`/books`	Query: `Author`		`200 Ok`	Lista di `Book`
		Query: `Category`			
`GET`	`/books/isbn`			`200 Ok`	`Book`
`GET`	`/customers`	Query: `FirstName`		`200 Ok`	Lista di `Customer`
		Query: `LastName`			
`GET`	`/customers/id`			`200 Ok`	`Customer`
`POST`	`/customers`		`Customer`	`201 Created`	
`POST`	`/loans`		`Loan`	`201 Created`	
`PUT`	`/loans`		`Return`	`202 Accepted`	

Tabella 3.1. Biblioteca – API REST

Di seguito, sono mostrati i modelli dei dati utilizzati da tale applicazione, in ingresso ed in uscita.

```
{
    "isbn": "ISBN del libro",
    "title": "Titolo del libro",
    "authors": [ ... ],
    "categories": [ ... ]
}
```

Modello 3.1. Libro – `Book`

```
{
    "id": "Identificativo dell'utente",
    "first_name": "Nome dell'utente",
    "last_name": "Cognome dell'utente",
    "phone": "Numero di telefono dell'utente",
    "email": "Indirizzo email dell'utente"
}
```

Modello 3.2. Cliente – `Customer`

```
{
    "start": "Data di inizio del prestito",
    "isbn": "ISBN del libro",
    "customer": "ID del cliente"
}
```

Modello 3.3. Richiesta di prestito – `Loan`

```
{
    "end": "Data di fine del prestito",
    "isbn": "ISBN del libro",
    "customer": "ID del cliente"
}
```

Modello 3.4. Richiesta di chiusura di prestito – `Return`

3.3 Applicazione serverless

Durante la fase di scrittura dell'applicazione serverless è necessario formulare alcune ipotesi, immaginando differenti scenari nell'utilizzo dell'applicazione. Al fine di enfatizzare il più possibile l'approccio serverless, cercando di sfruttare al massimo i vantaggi ed ipotizzarne gli svantaggi, ciascun endpoint dell'applicazione monolitica è stato trasformato in una funzione, messa in campo attraverso un canonico `Deployment` Kubernetes, oppure attraverso un `Knative Service`. Le ipotesi formulate sono le seguenti:

1. Si suppone che l'endpoint `GET /books`, utile alla ricerca dei libri eventualmente attraverso i filtri per autore e categoria, sia frequentemente utilizzato dagli utenti della biblioteca;
2. In modo analogo, l'endpoint `GET /customers`, utile alla ricerca dei clienti, insieme con il loro storico di prestiti, eventualmente attraverso filtri, si suppone frequentemente utilizzato dal personale della biblioteca;

3. Si suppone, invece, che l'endpoint `GET /books/{isbn}` sia scarsamente utilizzato per ricevere informazioni puntuali su un libro del quale già si conosce il codice univoco identificativo, oppure che l'utilizzo di questo endpoint possa accettare dei piccoli ritardi, imputabili anche ad un ritardo nella ricerca in un database reale contenente una grossa mole di dati;

4. In modo analogo, l'endpoint `GET /customers/{id}` si suppone scarsamente utilizzato, in quanto utile per ricevere informazioni puntuali su un cliente del quale già si conosce il codice identificativo, per altro reperibili con opportuni filtri attraverso l'endpoint `GET /customers`;

5. Si suppone, ancora, che durante l'inserimento di clienti, prestiti e resi, rispettivamente gestiti dagli endpoint `POST /customers`, `POST /loans` e `PUT /loans`, siano accettabili ritardi, che tali operazioni possano essere completate in modalità asincrona e che l'utilizzo di queste funzionalità non sia così frequente quanto le ricerche di libri e clienti.

A seguito delle ipotesi precedenti, durante la migrazione, che richiede una parziale riscrittura di codice, da applicazione monolitica ad applicazione serverless, sono state effettuate le seguenti scelte architetturali, schematizzate nella tabella 3.2:

- Gli endpoint `GET /books` e `GET /customers` sono stati trasformati in funzioni messe in campo come differenti `Deployment` Kubernetes. In questo modo, le risorse di questi endpoint risultano perennemente allocate, poiché il tempo impiegato durante la fase di allocazione delle risorse e messa in campo del container creerebbe un ritardo nell'utilizzo dell'applicazione probabilmente intollerabile;

- Gli endpoint `GET /books/{isbn}` e `GET /customers/{id}`, al contrario, sono stati trasformati in funzioni messe in campo come differenti `Knative Service`. In questo modo, le risorse di questi endpoint vengono allocate solo quando necessario, con conseguente ritardo in fase di messa in campo del container, a vantaggio di un utilizzo più efficiente delle risorse;

- Gli endpoint `POST /customers`, `POST /loans` e `PUT /loans`, infine, sono stati trasformati in funzioni messe in campo come differenti `Knative Service` in ascolto su differenti topic Apache Kafka. In questo modo gli endpoint sono attivati da un messaggio inviato sul topic al quale viene dichiarato interesse e le risorse di ciascun endpoint vengono allocate solamente quando necessario.

Attivazione	**Allocazione risorse**	**Implementazione**
`GET /books`	Perenne	`Deployment` Kubernetes
`GET /books/{isbn}`	Al bisogno	`Knative Service`
`GET /customers`	Perenne	`Deployment` Kubernetes
`GET /customers/{id}`	Al bisogno	`Knative Service`
Messaggio sul topic *customers*	Al bisogno	`Knative Service`
Messaggio sul topic *loans*	Al bisogno	`Knative Service`
Messaggio sul topic *returns*	Al bisogno	`Knative Service`

Tabella 3.2. Applicazione serverless – Architettura

A differenza dell'applicazione monolitica, non è possibile tenere traccia degli inserimenti di utenti, prestiti e resi nei log dell'applicazione, poiché le risorse dei container che si occupano di tali

operazioni vengono rilasciate al completamento dell'operazione e non è possibile quindi accedere ai messaggi di log generati dal container. Per ovviare a questo problema, è stato costruito un servizio di logging che viene invocato dopo ciascun inserimento

A livello logico, il flusso dell'applicazione durante un inserimento è il seguente, mostrato schematicamente nella figura 3.8:

1. Inserimento di un messaggio sul topic Kafka;
2. Inserimento dell'oggetto nel database;
3. Inserimento di una riga in un file di log.

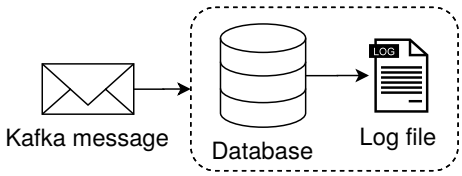

Figura 3.8. Inserimento tramite messaggio su topic Apache Kafka

3.3.1 Architettura

Entrando nel dettaglio dell'architettura dell'applicazione serverless, l'inserimento nel database è effettuato da una funzione attivata quando viene esplicitamente invocata, mentre l'inserimento dei dettagli in un file di log è effettuato da un `Pod` che aggiunge una riga di testo su un file per simulare un meccanismo di tracciamento, monitoring ed eventuale produzione di statistiche. È importante evidenziare che, per default, i servizio istanziati da Knative sono contattabili con richieste HTTP su un URL pubblico dipendente dal nome di dominio impostato per il cluster. Per tutelarsi da questa eventualità, è possibile impostare l'etichetta `serving.knative.dev/visibility` con il valore `cluster-local` per rendere visibile il `Knative Service` solamente dall'interno del cluster.

Per poter rendere accessibile il file di log generato direttamente dal sistema host, è necessario utilizzare un `Persistent Volume` e si è scelto di utilizzare un ordinario `Deployment` Kubernetes piuttosto che un `Knative Service`. Tale scelta è vincolata dall'implementazione dei servizi Knative, i quali non permettono di utilizzare le risorse `Persistent Volume`, ma risulta una scelta più che accettabile in quanto il meccanismo di logging richiede, sia in linea generale, sia in questo caso particolare, poche risorse computazionali. Inoltre, questo servizio risulta essere invocato da diversi endpoint, ulteriore motivazione che ha portato a realizzare tale servizio in modalità standard, le cui risorse risultano quindi perennemente allocate.

Il collegamento tra il servizio che si occupa di inserire i dati nel database, dopo avere effettuato i necessari controlli per garantire i vincoli legati al contesto applicativo, ed il servizio di logging è effettuato utilizzando la risorsa `Sequence` offerta da Knative. La risorsa `Sequence` (2.2.2) estende l'interfaccia `Addressable` di Kubernetes ed indica al suo interno una serie di servizi, anch'essi che estendono l'interfaccia `Addressable`, i quali vengono invocati nell'ordine dichiarato. In questo caso sono state create tre `Sequence` praticamente identiche, poiché svolgono le stesse funzioni di inserimento dati e successivo logging.

3.3.2 Integrazione con Apache Kafka

L'ultima fase durante la costruzione di quest'applicazione richiede l'integrazione delle risorse Kubernetes/Knative create con il sistema di messaggistica Apache Kafka. In modo simile ai `Knative Service`, la `Sequence` generata risulta contattabile solamente attraverso HTTP e per evitare questa possibilità si può limitarne la visibilità, in modo analogo rispetto ad un `Knative Service`, impostando opportunamente l'etichetta `serving.knative.dev/visibility`.

Un oggetto Knative di tipo `KafkaSource` è in grado di attivare una qualunque risorsa Kubernetes che estende l'interfaccia `Addressable` quando rileva un nuovo messaggio su un topic Apache Kafka sul quale è in ascolto, generando un evento all'interno della service mesh di Knative di tipo `dev.knative.kafka.event` e con label `kafkasources.sources.knative.dev/key-type`. La risorsa `Sequence` è quindi utilizzata come destinazione del messaggio, che viene inoltrato come *Request Body* al primo elemento della sequenza. Questo approccio è stato utilizzato per ogni procedura di inserimento e permette, a livello logico, il collegamento tra *topic* e `Sequence`, come mostrato in tabella 3.3.

Topic Apache Kafka	`Sequence` **Knative**
customers	`add-customer`
loans	`add-loans`
returns	`add-returns`

Tabella 3.3. Applicazione serverless – Mapping tra topic Apache Kafka e `Sequence` Knative

Quindi, un client intenzionato ad inserire un oggetto nell'applicazione, semplicemente accoda un messaggio su un topic Apache Kafka anziché inviare una richiesta `POST`. Il messaggio inviato su tale topic ha la stessa struttura, mostrata nei modelli 3.2, 3.3 e 3.4, della analoga richiesta HTTP.

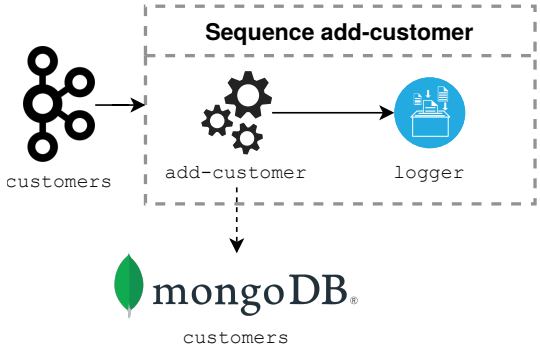

Figura 3.9. Applicazione serverless – `Sequence add-customer` attivata da un messaggio sul topic Apache Kafka *customers*

3.3.3 Integrazione con Istio

La costruzione dell'applicazione con `Knative Service` e `Service` visibili solamente all'interno del cluster porta con sé la necessità di esporre manualmente le API all'esterno del cluster. È quindi necessario anteporre, a livello logico, alcuni gateway che indirizzino opportunamente il traffico verso il corretto servizio, mostrati nella tabella 3.4.

URI	**Servizio**
`GET /library/v2/books`	`Deployment` *GET Books*
`GET /library/v2/books/*`	`Knative Service` *GET Book*
`GET /library/v2/customers`	`Deployment` *GET Customers*
`GET /library/v2/customers/*`	`Knative Service` *GET Customer*

Tabella 3.4. Applicazione serverless – Mapping tra URI e servizio

Per realizzare il mapping presentato in tabella 3.4, è necessario istanziare diverse risorse `Ingress Gateway`, messe a disposizione da Istio. Un oggetto `Ingress Gateway` ha la funzione di intercettare il traffico ricevuto sul gateway del cluster ed instradare le richieste che rispettano determinate proprietà all'interno di uno specifico `Service` Kubernetes, identificato da un URL locale al cluster, eventualmente riscrivendone alcuni header o proprietà.

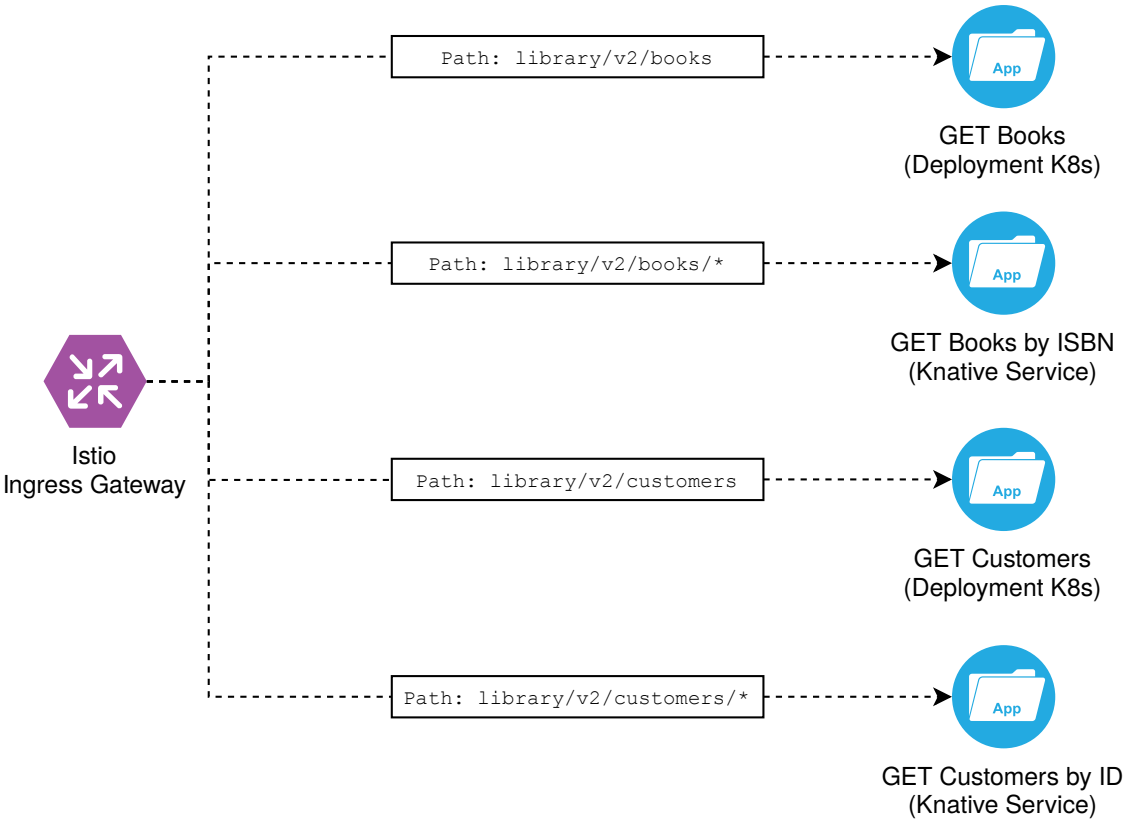

Figura 3.10. Applicazione serverless – Configurazione degli `Ingress Gateway` di Istio

3.4 Transizione da applicazione monolitica ad applicazione serverless

Concettualmente, la scrittura di un'applicazione realizzata su un'architettura serverless ha senso quando la rispettiva applicazione monolitica risulta troppo pesante e con un consumo di risorse troppo elevato. Al fine di simulare un'applicazione sufficientemente complessa, l'applicazione monolitica è stata espansa inserendo:

1. Computazione crittografica per simulare operazioni che coinvolgano in modo importante la CPU ed aumentare il tempo medio di esecuzione;
2. Strutture dati di grandi dimensioni per simulare un ampio spazio occupato in RAM dall'applicazione.

L'applicazione monolitica, caratterizzata da n endpoint, è dunque stata divisa in n moduli, ciascuno responsabile di un singolo endpoint della corrispondente API REST, a seguito delle seguenti ipotesi:

1. Dato M il quantitativo di RAM occupato dall'applicazione monolitica, questo è stato diviso tra le varie funzioni dell'applicazione serverless in quantitativi $m_1, m_2, \ldots, m_n$ tali che:

$$\sum_{i=1}^{n} m_i = M$$

   Inoltre, sono state appesantite maggiormente le funzioni che si occupano di inserimenti o aggiornamenti nel database, ipotizzando che, in un caso reale, queste operazioni siano, in linea generale, più lente poiché soggette ad un numero maggiore di controlli back–end per forzare policy personalizzate e business rule specifiche del contesto in cui opera l'applicativo.

2. Dato c_i il consumo, in termini di CPU, di un modulo dell'applicazione monolitica, questo è stato riportato in modo identico nella relativa funzione dell'applicazione serverless, in quanto il costo computazionale del modulo entra in gioco solamente nel momento in cui questo viene invocato, indipendentemente dall'architettura sulla quale viene messo in campo, sia questa monolitica oppure FaaS.

3.5 Confronto applicazioni a riposo

Al fine di poter mettere a confronto le due soluzioni realizzate, è necessario prima di tutto valutare quali siano le risorse impiegate dalle due applicazioni a riposo, prendendo in considerazione un periodo all'interno del quale non vengono ricevute ed elaborate richieste.

Come discusso nella sezione 3.4, per simulare un caso reale nel quale l'approccio a microservizi possa essere preso in considerazione come valida alternativa ad un'applicazione monolitica, è necessario che quest'ultima applicazione abbia un discreto consumo di risorse. Infatti, come mostrato nella figura 3.11, viene imposto un consumo in termini di memoria RAM non trascurabile. Come è evidente dalla figura, l'attività in termini di CPU risulta comunque minima, mentre il consumo di memoria RAM risulta costante ed intorno al valore imposto a priori.

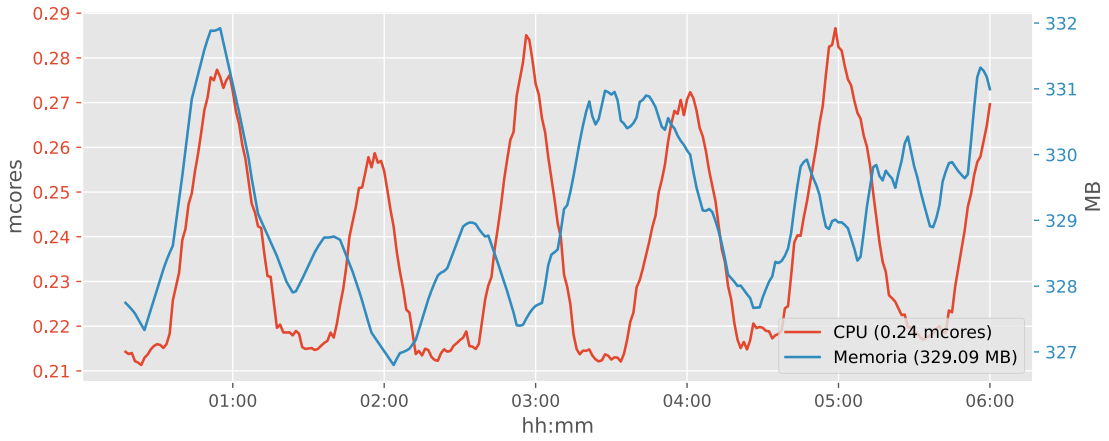

Figura 3.11. Applicazione monolitica – Risorse utilizzate a riposo

Riguardo invece all'applicazione realizzata con architettura FaaS, solo alcuni moduli, in particolare soltanto gli endpoint `GET /books` e `GET /customers`, secondo le ipotesi enunciate nella sezione 3.3, risultano costantemente attivi e quindi solo per alcuni endpoint le risorse risultano perennemente allocate. Questi endpoint sono messi in campo come ordinari `Deployment` Kubernetes, senza alcun ausilio delle funzionalità offerte del framework Knative.

La figura 3.12 mostra le risorse assegnate all'endpoint `GET /books` in particolare, ma analoghe considerazioni sono valide per l'endpoint `GET /customers` poiché soggetto alle stesse ipotesi enunciate nella sezione 3.3. Anche in questo caso, è bene notare come l'utilizzo di CPU sia decisamente trascurabile, poiché non viene, di fatto, eseguita alcuna computazione quando l'applicazione è a riposto, mentre risulta degno di nota soltanto il consumo di memoria, anche in questo caso costante e vicino al valore imposto a priori.

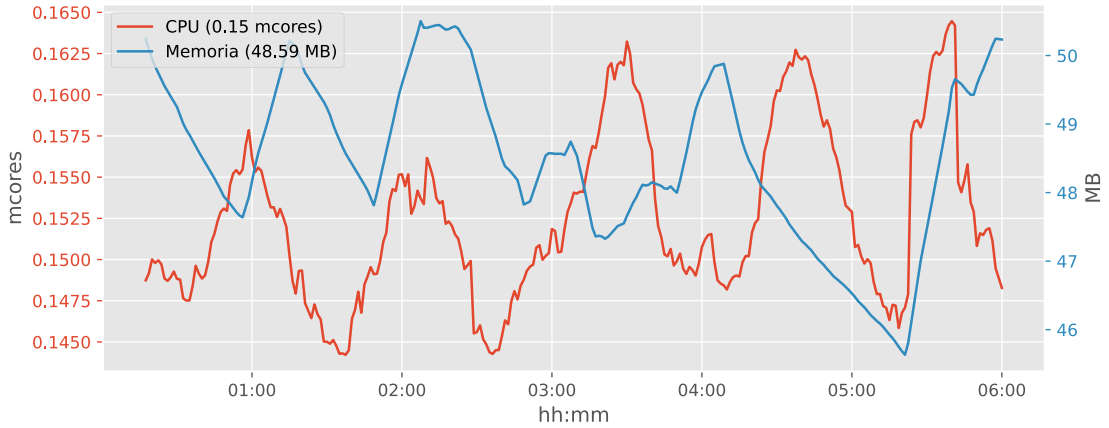

Figura 3.12. Applicazione serverless – Risorse utilizzate da un `Deployment` Kubernetes a riposo

La figura 3.13 mostra, invece, le risorse assegnate all'endpoint `GET /books/{isbn}` in particolare, ma analoghe considerazioni sono valide per l'endpoint `GET /customers/{id}` poiché soggetto alle

stesse ipotesi enunciate nella sezione 3.3. In questo caso, l'idea è stata quella di valutare le risorse assegnate ad un endpoint della soluzione FaaS, messo in campo come `Knative Service` se questo fosse costantemente attivo, con conseguente allocazione perenne delle risorse. In questo caso, Knative, in quanto responsabile della messa in campo dei `Knative Service`, implementa altre funzionalità rispetto ad un canonico `Deployment` Kubernetes ed è infatti possibile notare come l'utilizzo di risorse sia molto diverso tra le due soluzioni.

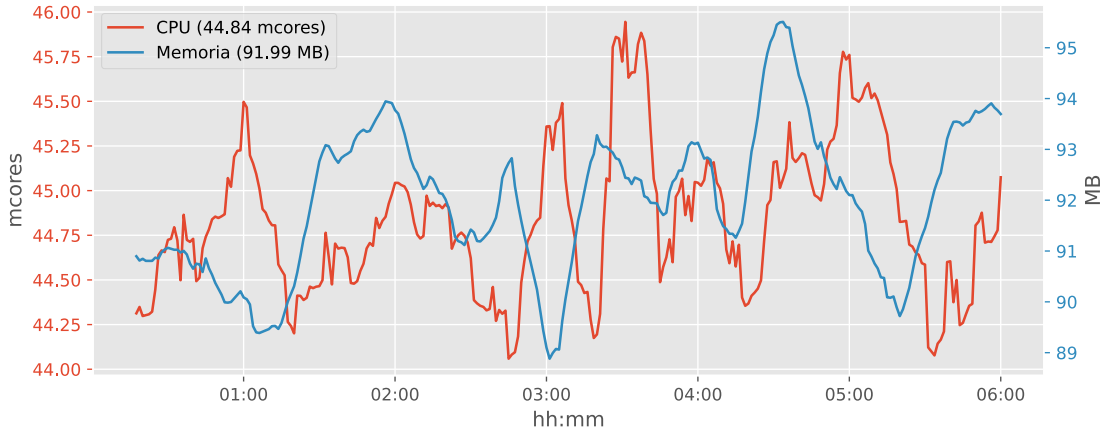

Figura 3.13. Applicazione serverless – Risorse utilizzate da un `Knative Service` a riposo

È fondamentale osservare, con l'ausilio delle figure 3.14 e 3.15, come il container `queue-proxy`, iniettato in modo automatico da Knative, occupi una quantità di risorse tutt'altro che trascurabile nella messa in campo di `Knative Service`, sia in termini di memoria RAM, sia, e soprattutto, in termini di utilizzo di CPU anche durante un periodo nel quale non vengono ricevute richieste.

Come nel caso precedente, l'utilizzo di CPU risulta trascurabile per il container `get-book`, responsabile dell'esecuzione dell'effettiva logica di ricerca dei dati e risposta, poiché non viene, di fatto, eseguita alcuna computazione quando l'applicazione è a riposto, mentre risulta degno di nota soltanto il consumo di memoria, anche in questo caso costante e vicino al valore imposto a priori. Tali considerazioni sono del tutto analoghe al caso precedente, poiché la logica interna del container `get-book` risulta del tutto simile alla logica del container `get-books`.

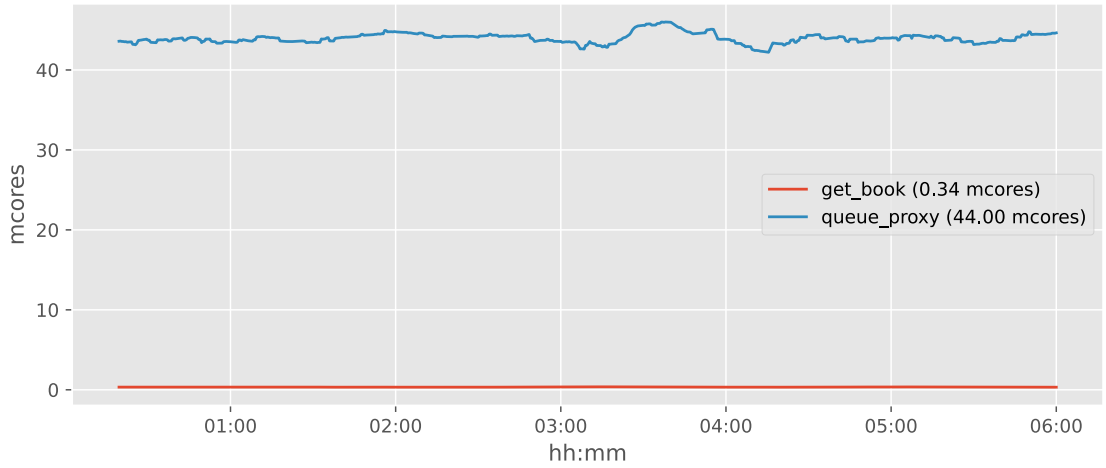

Figura 3.14. Applicazione serverless – Utilizzo CPU di un `Knative Service` a riposo

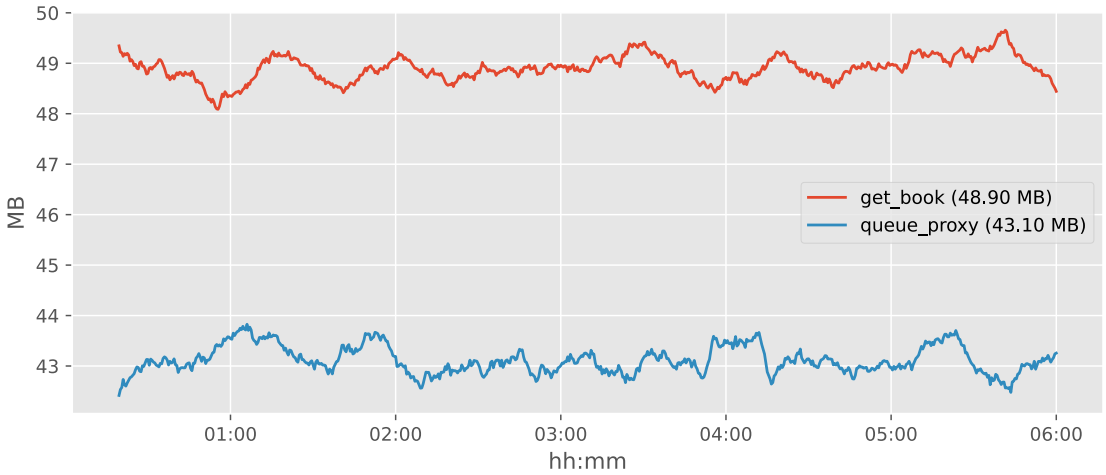

Figura 3.15. Applicazione serverless – Utilizzo memoria di un `Knative Service` a riposo

In ultimo, è importante ricordare che gli endpoint responsabili dell'inserimento dei dati all'interno del database vengono attivati solo quando necessario e quindi il loro contributo in termini di utilizzo CPU e memoria è da considerare nullo in fase di riposo.

Capitolo 4

Test

I test dell'applicazione sono stati effettuati utilizzando l'applicativo Apache JMeter (2.3.7). In particolare, si sono voluti ricreare differenti scenari di testing in modo da simulare picchi di lavoro per l'applicazione. In ottica di valutazione delle performance è importante confrontare i tempi di risposta dell'applicazione, considerando eventualmente la percentuale di fallimento nell'adempimento della richiesta, in relazione con l'impatto in memoria ed in termini computazionali dell'applicazione.

Le figure seguenti, esportate da Grafana come dati grezzi ed elaborate in seguito, riportano gli utilizzi delle risorse hardware, in termini di sfruttamento CPU e memoria RAM delle diverse funzioni. Dove possibile, è stato inoltre inserito il dato della latenza, fornito dalle statistiche e dai risultati di Apache JMeter. In tutte le figure successive, dove presente, viene riportato tra parentesi il valore medio della misurazione, in modo tale da favorire un'indicazione quantitativa di massima e favorire il confronto tra i diversi grafici.

4.1 Inserimento singolo

La maggiore peculiarità della soluzione FaaS è rappresentata dall'inserimento di dati – in questo caso particolare di clienti, prestiti e resi – in modo asincrono attraverso messaggi su topic Apache Kafka che attivano funzioni silenti, allocandone le risorse solamente quando necessario.

L'obiettivo di questo test è valutare la risposta dell'applicazione FaaS rispetto ad un'applicazione monolitica, in seguito all'inserimento di un singolo dato, fornito tramite una richiesta `POST` nel caso di applicazione monolitica, oppure come messaggio Apache Kafka nel caso di applicazione FaaS.

4.1.1 Applicazione serverless

Come evidenziato dalla figura 4.1, il `Deployment` che risponde alla richiesta necessita di alcuni secondi per poter essere messo in campo ed iniziare l'elaborazione della richiesta vera e propria e rimane attivo per alcuni minuti al termine dell'effettiva elaborazione, come mostrato dalla discrepanza temporale tra l'utilizzo di CPU e l'utilizzo di memoria, per tentare, a livello molto basilare da parte di Knative, di ottimizzare le risorse impiegate tra richieste successive in un ristretto lasso di tempo.

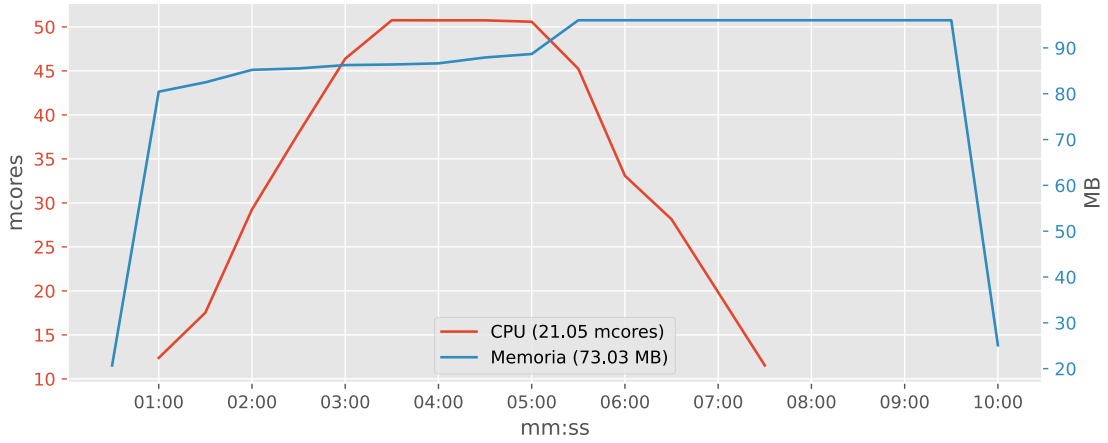

Figura 4.1. Applicazione serverless – Risorse utilizzate all'inserimento di un singolo utente

Le figure 4.2 e 4.3, invece, mostrano nel dettaglio le risorse utilizzate dal `Pod` che viene invocato dopo l'inserimento di un messaggio su un topic Apache Kafka e si occupa effettivamente di eseguire l'inserimento di un dato all'interno della base dati.

Come è possibile osservare nelle figure, Knative si occupa di iniettare un secondo container, denominato `queue-proxy` accanto a quello effettivamente responsabile dell'esecuzione dell'operazione desiderata, in questo caso `add-customer`, responsabile del concreto inserimento di un utente all'interno della biblioteca. Analoghe considerazioni, naturalmente, varrebbero per l'inserimento di prestiti e resi.

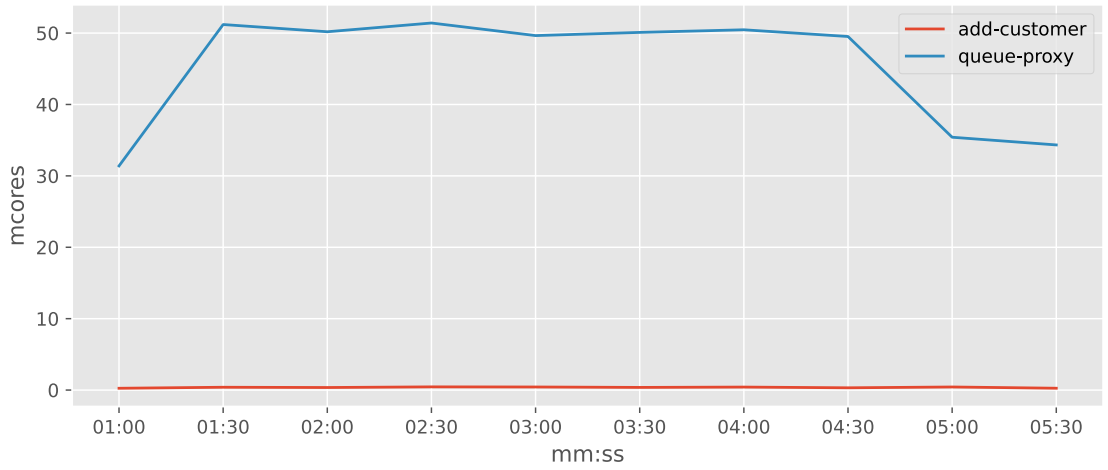

Figura 4.2. Applicazione serverless – Utilizzo CPU all'inserimento di un singolo utente

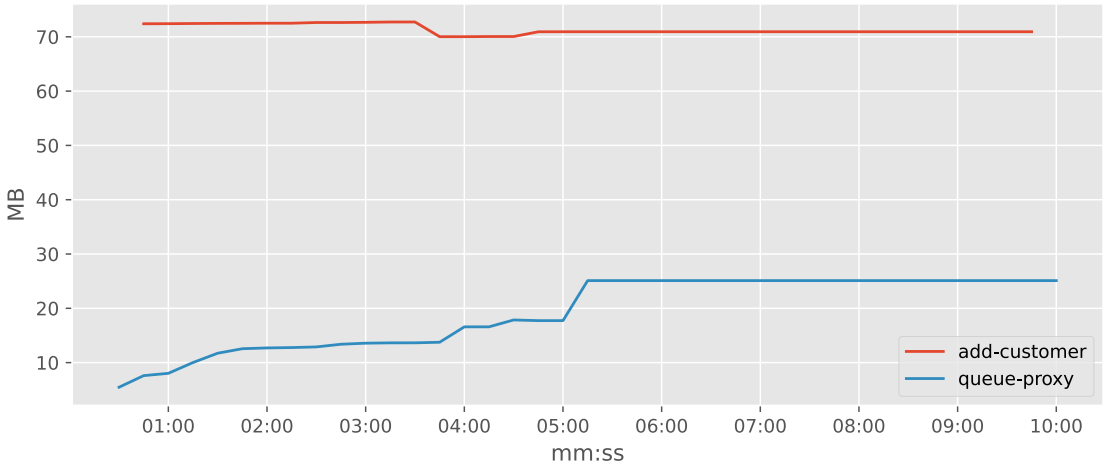

Figura 4.3. Applicazione serverless – Utilizzo memoria all'inserimento di un singolo utente

Il container `queue-proxy` viene automaticamente iniettato dal framework Knative ed ha la funzione di forzare policy che limitino la concorrenza tra le diverse richieste ricevute.

4.1.2 Applicazione monolitica

Per quanto concerne invece l'applicazione monolitica, l'inserimento di un singolo dato attraverso una richiesta HTTP `POST` non risulta modificare in modo significativo le risorse utilizzate. In questo caso, è quindi possibile, nei calcoli successivi, utilizzare il valore medio dell'applicazione a riposo, riportato nella figura 3.11.

4.2 Inserimento multiplo

L'obiettivo di questo test è valutare la risposta dell'applicazione FaaS rispetto ad un'applicazione monolitica, in seguito all'inserimento di una serie di dati concentrati in un arco temporale piuttosto ristretto, per simulare un picco di carico nell'utilizzo dell'applicazione. Prendendo in considerazione il caso d'uso di una biblioteca scolastica oppure universitaria, infatti, è possibile ipotizzare che durante l'arco dell'intera giornata, molteplici richieste di prestiti o di resi siano concentrate in alcune particolari fasce orarie.

Altro scenario possibile che necessita di un inserimento multiplo potrebbe riguardare l'introduzione all'interno della base dati di una lista di nuovi utenti o libri, oppure ancora l'aggiornamento in batch dei dati di utenti, prestiti o resi, con opportune modifiche all'applicazione e sviluppo di endpoint adeguati.

Anche in questo caso, analogamente allo scenario discusso nella sezione 4.1, è necessario fornire i dati attraverso una serie di richieste `POST` nel caso di applicazione monolitica, oppure attraverso messaggi Apache Kafka nel caso di applicazione FaaS.

Per effettuare questo test, sono state lanciate 1000 richieste in un arco temporale di 1000 secondi.

4.2.1 Applicazione serverless

Nella figura 4.4 è possibile osservare come, anche in uno scenario caratterizzato da molteplici inserimenti, il container impieghi diversi secondi prima di essere effettivamente operativo. A differenza del caso precedente, nel quale si osservava l'inserimento di un singolo dato, il tempo di attivazione e messa in campo del container viene totalmente assorbito durante l'intero test e risulta, quindi, quasi trascurabile.

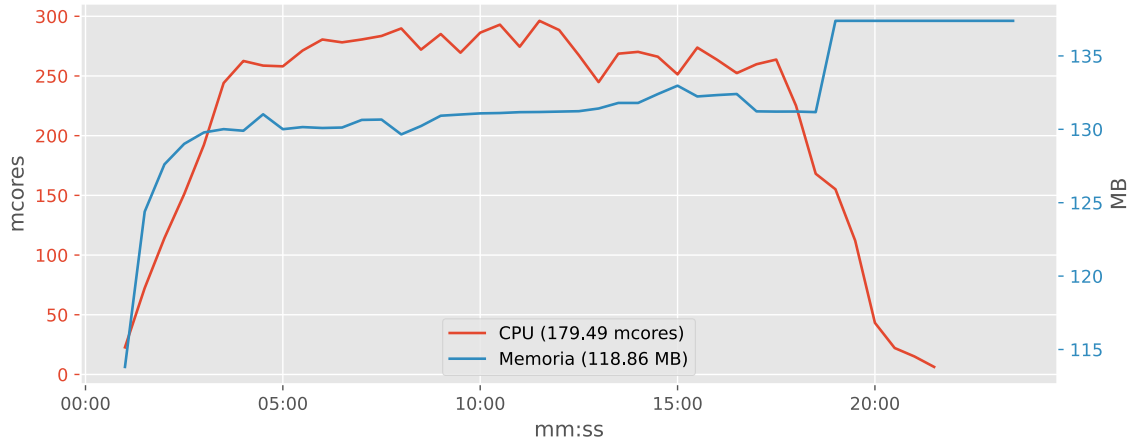

Figura 4.4. Applicazione serverless – Risorse utilizzate all'inserimento di prestiti

Con considerazioni simili, è possibile notare, grazie alle figure 4.5 e 4.6 come l'impatto del container `queue-proxy` risulti, in questa situazione, quasi marginale, ed in ogni caso con un'incidenza decisamente minore rispetto allo scenario mostrato sopra. Anche in questa situazione, in modo analogo alla precedente, le risorse rimangono allocate per qualche minuto al termine dell'esecuzione dell'ultima richiesta, in maniera tale da mantenere attivo il container ancora qualche istante prima della sua definitiva distruzione con conseguente completa deallocazione delle risorse da esso utilizzate.

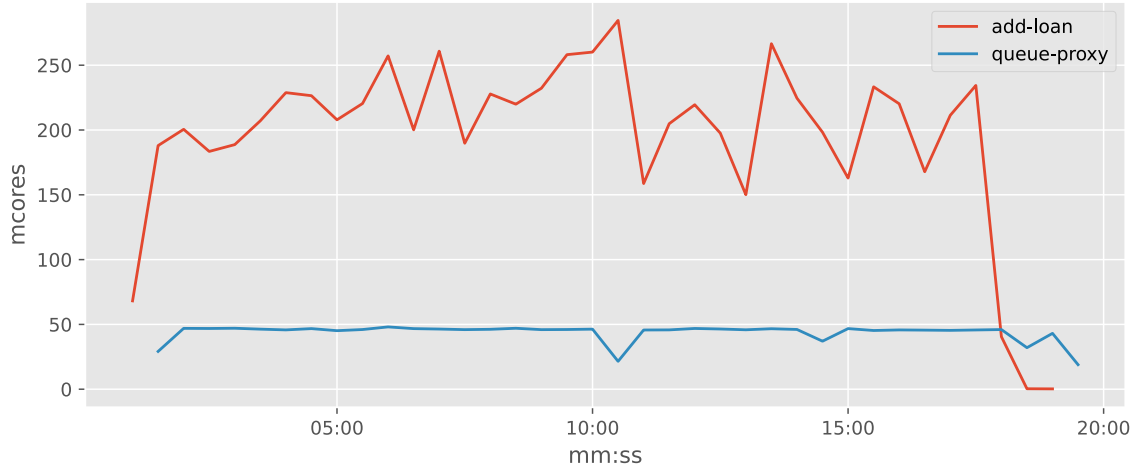

Figura 4.5. Applicazione serverless – Utilizzo CPU all'inserimento di prestiti

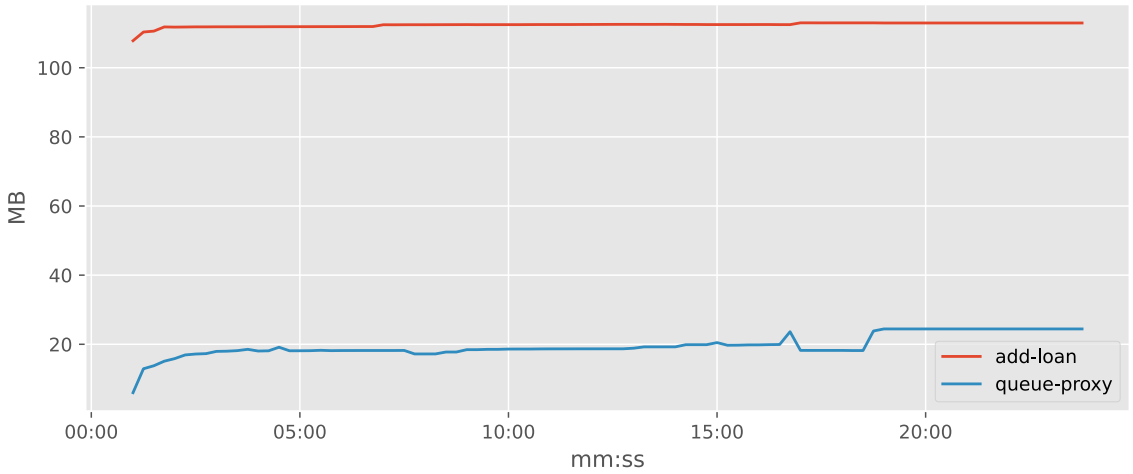

Figura 4.6. Applicazione serverless – Utilizzo memoria all'inserimento di prestiti

Da questo confronto, inoltre, è possibile evidenziare come il container sidecar `queue-proxy` influisca in un maniera sostanzialmente costante rispetto al carico di lavoro eseguito dal container a cui questo è affiancato. Infatti, le risorse utilizzate dal container `queue-proxy` risultano essere piuttosto simili confrontando le figure 4.2 e 4.3 e le figure 4.5 e 4.6, permettendo quindi di affermare che le risorse impiegate dal container sidecar sono indipendenti dal container responsabile dell'effettiva esecuzione delle operazioni.

4.2.2 Applicazione monolitica

Osservando invece il comportamento dell'applicazione monolitica, è possibile notare grazie alla figura 4.7 come, anche in questo caso, aumenti naturalmente l'utilizzo di CPU, il quale passa da un'attività praticamente nulla ad un attività comparabile con la soluzione FaaS. Aumenta significativamente anche l'utilizzo di memoria, con un utilizzo circa quadruplo rispetto alla soluzione a microservizi, poiché, anche in condizioni stazionarie, l'applicazione monolitica alloca un discreto quantitativo di memoria.

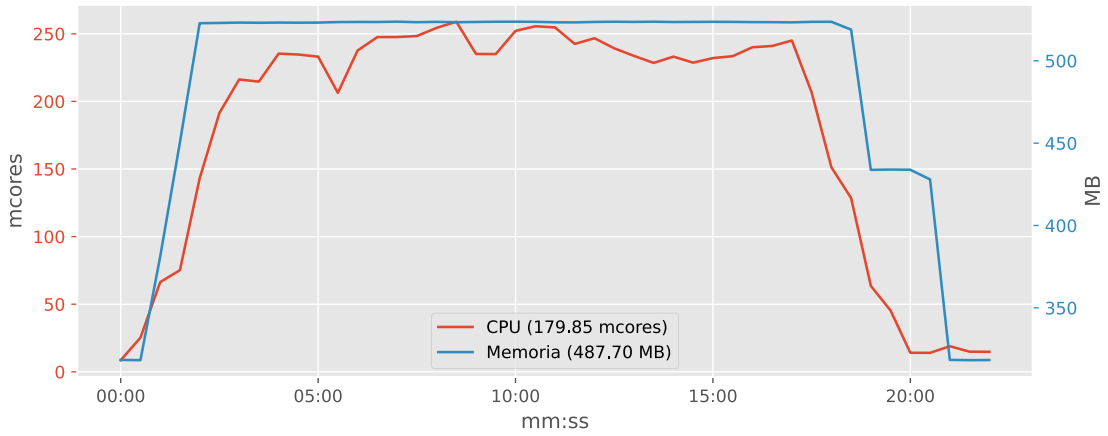

Figura 4.7. Applicazione monolitica – Utilizzo risorse all'inserimento di prestiti

La figura 4.8 mostra invece il tempo di risposta dell'applicazione monolitica, dato non rilevabile per l'applicazione serverless in quanto gli inserimenti in quest'ultima avvengono in modo asincrono per scelta progettuale. Si può notare come il tempo di risposta risulti pressoché costante per tutta la durata del test, tranne in un breve momento nel quale probabilmente l'applicazione non riesce a smaltire completamente le richieste e queste rimangono accodate in attesa di essere servite o di essere inviate come risposta.

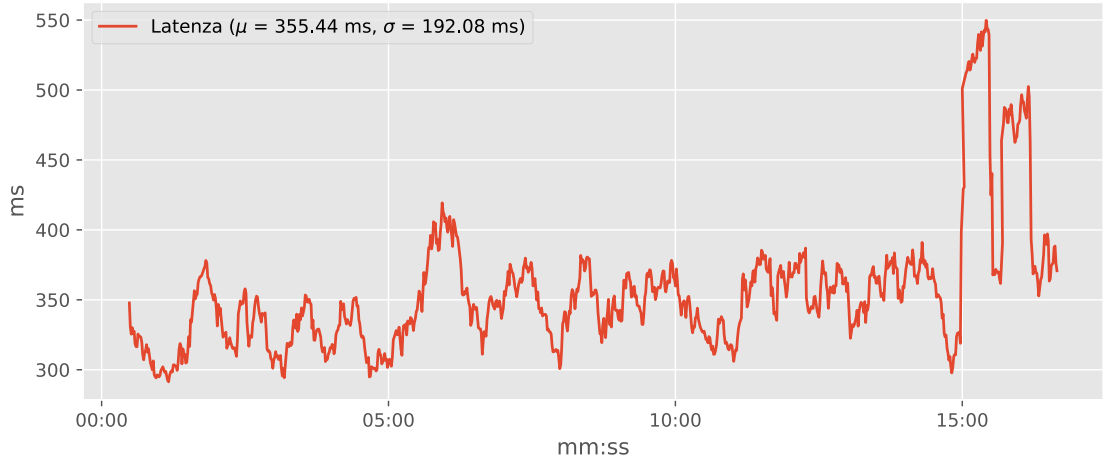

Figura 4.8. Applicazione monolitica – Latenza all'inserimento di prestiti

4.3 Lettura di dati

L'obiettivo di questa serie di test è quello di confrontare le risposte delle due applicazioni in seguito ad una serie di richieste `GET` per la lettura di una lista di libri, concentrati in un arco temporale piuttosto ristretto, al fine di simulare un picco di carico nell'utilizzo dell'applicazione. Prendendo

in considerazione il caso d'uso di una biblioteca scolastica oppure universitaria, infatti, è possibile ipotizzare che durante l'arco dell'intera giornata, gli utenti richiedano informazioni in particolari fasce orarie.

Le due applicazioni vengono contattate sugli endpoint `GET /books` oppure `GET /books/{isbn}`, valutando la latenza media della risposta e la variabilità di questa misura, effettuando diversi test che si differenziano tra loro per numero di richieste, durata e conseguentemente numero di richieste al secondo. Apache JMeter misura la *latenza* come l'intervallo temporale che trascorre tra l'istante appena precedente all'invio della richiesta e l'istante appena successivo alla ricezione dell'inizio della risposta. La latenza, quindi, include anche il tempo necessario all'assemblaggio della richiesta e all'elaborazione della prima parte della risposta. In questo modo, la latenza misurata da JMeter dovrebbe essere decisamente simile a quella sperimentata da un utente che utilizza un browser o altre applicazioni lato client.

Inoltre, considerando che la lettura dei libri è implementata in modo diverso a seconda dell'endpoint, come ordinario `Deployment` Kubernetes nel caso di lettura di un insieme di libri oppure come `Knative Service` nel caso di lettura puntuale di un singolo dato, in questa sezione viene anche confrontato il diverso comportamento delle due soluzioni architetturali, in termini di performance, stabilità nella risposta e di risorse utilizzate.

Analoghe considerazioni sono valide per quanto riguarda la lettura delle informazioni sui clienti della biblioteca, rese disponibili contattando le applicazioni sugli endpoint `GET /customers` oppure `GET /customers/{id}`, poiché questi seguono le stesse logiche ed eseguono sostanzialmente le stesse operazioni lato backend, cioè una ricerca nella base dati, seppur su collezioni di dati diverse.

4.3.1 `Deployment` Kubernetes vs. Applicazione monolitica

In questa sezione vengono valutate le performance di una specifica funzione, messa in campo come `Deployment` Kubernetes, e confrontate tali prestazioni con la corrispondente funzione resa disponibile da un'applicazione monolitica. Al fine di realizzare questo test, è necessario inviare alle due applicazioni un insieme di richieste di tipo `GET` sull'endpoint `/books`, le quali restituiscono l'elenco completo dei libri disponibili nella biblioteca. Come descritto nella sezione 3.3 e riassunto nella tabella 3.2, l'endpoint serverless è implementato come un `Deployment` Kubernetes perennemente allocato, poiché si suppone frequentemente utilizzato e quindi latenze dovute al tempo di messa in campo del container non sarebbero tollerabili. In particolare, durante questo test, sono state inviate 10000 richieste in un arco temporale di 2000 secondi.

Latenza

Osservando la figura 4.9 è possibile notare come le due soluzioni reagiscano in modo estremamente simile per quanto concerne il ritardo nella risposta, avendo una latenza media che si discosta di solamente qualche punto percentuale. È bene, inoltre, considerare che l'applicazione serverless presenti una latenza leggermente più variabile.

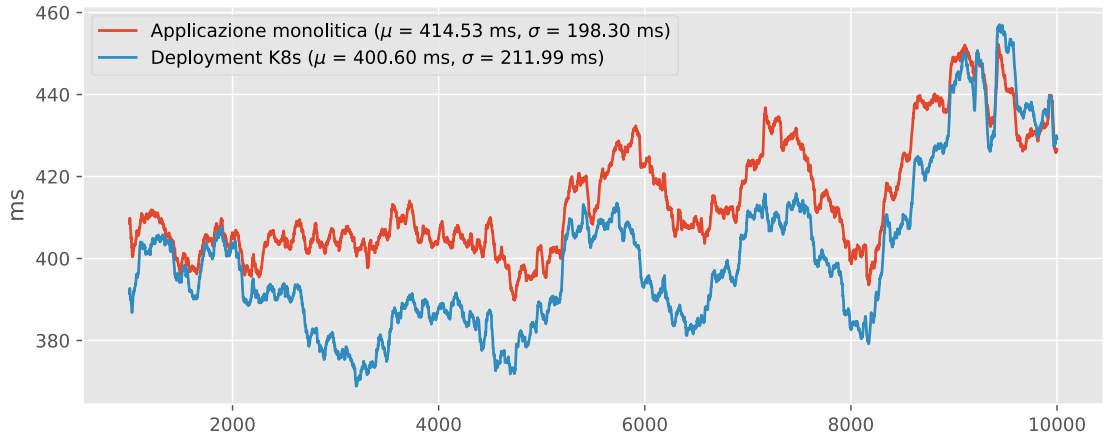

Figura 4.9. Lettura di libri – Latenza

Dalla figura 4.10 è possibile osservare come le due soluzioni applicative abbiano una distribuzione della latenza molto simile, ad ulteriore conferma della figura precedente, ed è pertanto possibile affermare che le applicazioni hanno prestazioni decisamente simili per quanto riguarda il tempo di risposta.

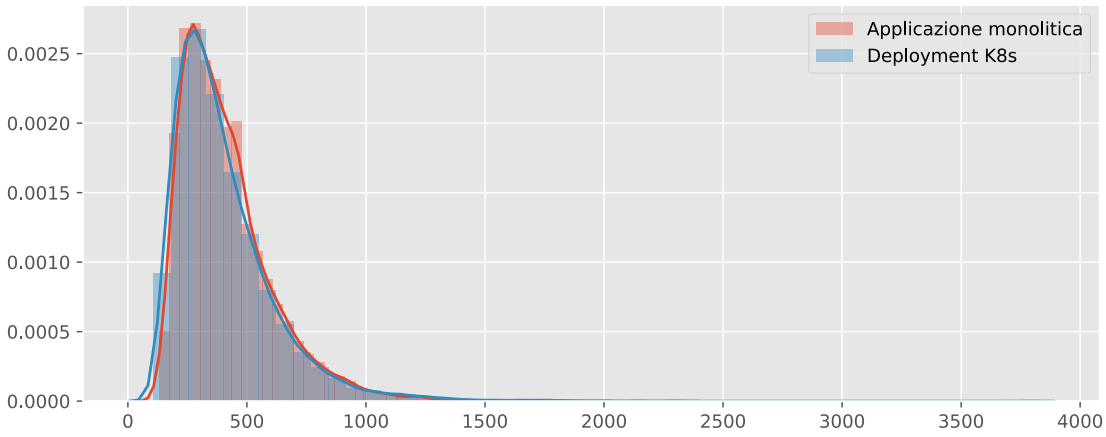

Figura 4.10. Lettura di libri – Distribuzione latenza

Risorse computazionali

In questo scenario, è bene tenere in considerazione anche il consumo di risorse computazionali delle due differenti architetture.

Per come è stata costruita l'applicazione, è importante osservare, grazie alla figura 4.11, come l'utilizzo di CPU sia sostanzialmente identico. Infatti, le due soluzioni effettuano una ricerca del tutto simile all'interno della base dati per reperire il catalogo dei libri, a seguito di una computazione crittografica, effettuata al fine di simulare un qualche tipo di carico lato CPU, come descritto nella sezione 3.4.

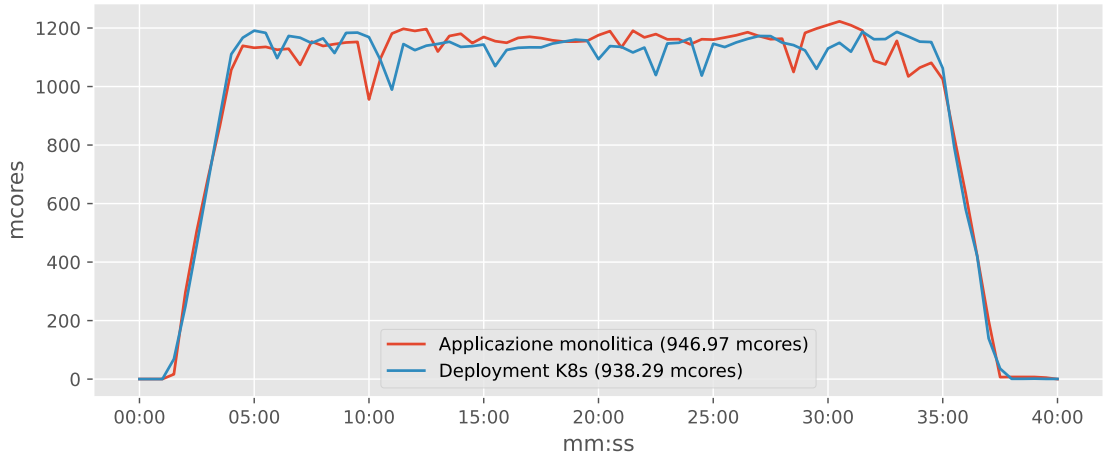

Figura 4.11. Lettura di libri – Utilizzo CPU

Osservando invece la figura 4.12, si può notare come l'utilizzo di memoria segua, anche in questo caso, un andamento estremamente simile tra le due soluzioni.

Da questa figura si nota come, rispetto ad una situazione di riposo, mostrata nella sezione 3.5, il consumo di memoria aumenti di circa il 70% nell'applicazione serverless e del 60% nell'applicazione monolitica. Da questo primo confronto, si può immediatamente osservare come l'utilizzo di memoria si discosti, tra le due soluzioni ed inizino ad emergere le peculiarità di ciascuna e le loro differenze.

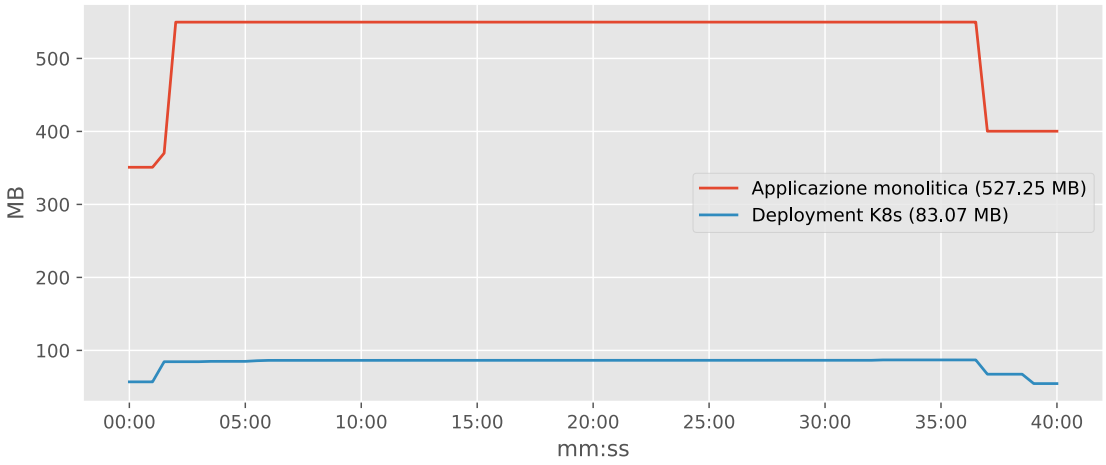

Figura 4.12. Lettura di libri – Utilizzo memoria

4.3.2 `Knative Service` inattivo vs. Applicazione monolitica

In questa sezione viene valutata la risposta di un `Knative Service` inizialmente a riposo, che quindi non utilizza alcun tipo di risorsa, né CPU, né memoria, e viene messo in campo nel momento

in cui riceve una richiesta. In questo caso, è il framework Knative ad occuparsi dell'istanziazione di una replica di tale `Deployment` che gestisce ed elabora tale richiesta.

Latenza

È immediato notare che una politica di questo genere porta ad avere dei tempi di risposta esageratamente elevati per le prime richieste, poiché a queste corrisponde un tempo, decisamente non trascurabile, in cui il container deve essere istanziato ed avviato. La figura 4.13 mostra esattamente questa situazione, nella quale è evidente come le prime richieste dirette al `Knative Service` abbiano una latenza decisamente fuori scala rispetto alle successive dirette alla stessa, ed anche rispetto alle richieste dirette all'applicazione monolitica. Non è un caso se, infatti, l'applicazione serverless, proprio a causa del periodo transitorio di messa in campo del container, presenti una maggiore variabilità nella latenza, dato confermato dalla deviazione standard σ mostrata in figura.

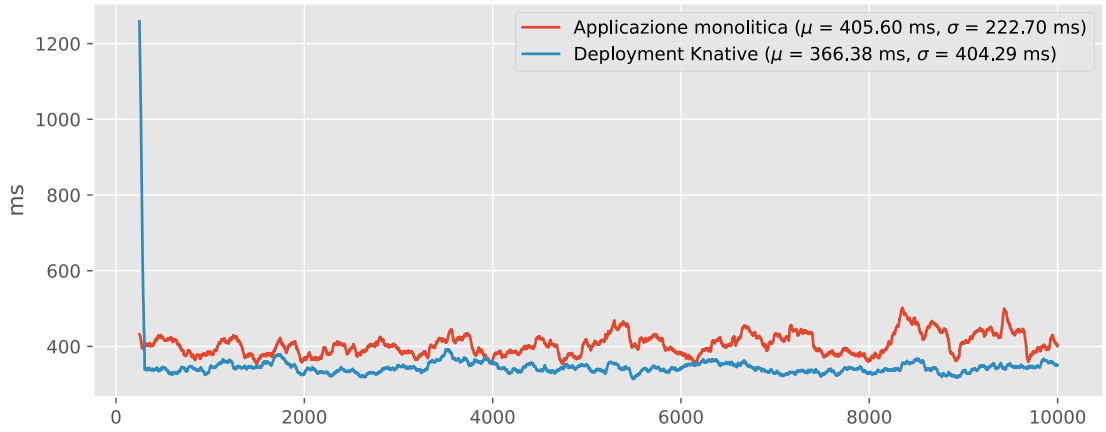

Figura 4.13. `Knative Service` inattivo vs. Applicazione monolitica – Latenza risposte

Dalla figura 4.13 è inoltre possibile intravedere come, a transitorio esaurito, l'applicazione serverless sembri reagire meglio alle richieste, rispondendo con una latenza più bassa, visibile chiaramente dal grafico, e confermato da una latenza media inferiore, anche considerando il transitorio stesso.

Esclusione del transitorio

Per poter effettivamente mettere a confronto le due differenti architetture, è necessario eliminare il periodo transitorio di messa in campo ed inizializzazione del container. Per fare ciò, è stato deciso di valutare la *latenza media futura* e la *deviazione standard futura della latenza.*

La *media futura* rispetto al campione i è stata definita come la media dei campioni successivi al campione i, includendo il campione stesso. In altre parole, dati N campioni, numerati da 0 a $N-1$ e la media futura con il simbolo μ^{F}:

$$\mu_i^{\mathrm{F}} = \frac{\sum_{j=i}^{N-1} x_j}{N-i} \tag{4.1}$$

In modo analogo, è stata definita la *deviazione standard futura* con il simbolo σ^{F}:

$$\sigma_i^{\mathrm{F}} = \sqrt{\frac{\sum_{j=i}^{N-1}\left(x_j - \mu_i^{\mathrm{F}}\right)}{N-i}} \tag{4.2}$$

L'idea che giustifica questa scelta è quella di valutare da quale campione in avanti la media e la deviazione standard della stessa misura non sono più soggette ad ampie variazioni. Infatti, nel momento in cui queste due misure rimangono pressoché costanti, si può considerare estinto il transitorio.

Le figure 4.14 e 4.15 mostrano esattamente quanto detto sopra. Da questi due grafici, è possibile notare come l'applicazione monolitica mantenga sempre costante i propri valori di media futura e deviazione standard futura della media, esattamente come ci si aspetta, poiché le sue risorse risultano perennemente allocate e non è soggetta ad alcun transitorio iniziale. D'altro canto, queste misurazioni si abbassano drasticamente osservando l'andamento dell'applicazione serverless, fino a stabilizzarsi tra la cinquantesima e la settantacinquesima richiesta.

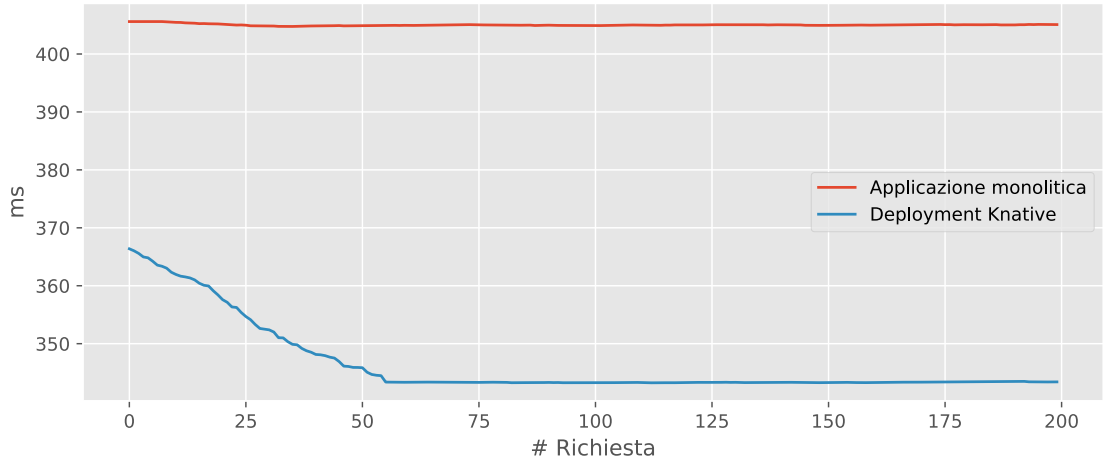

Figura 4.14. `Knative Service` inattivo vs. Applicazione monolitica – Latenza futura (Media)

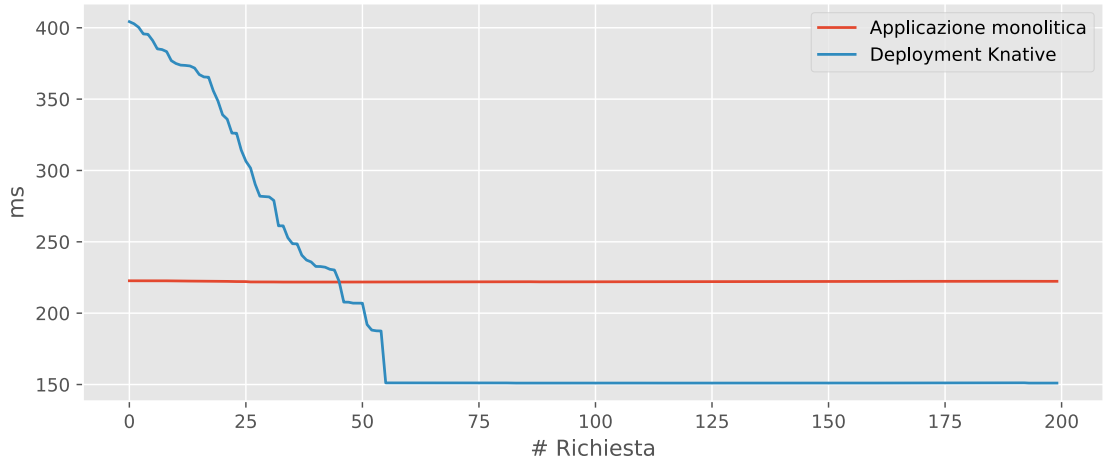

Figura 4.15. `Knative Service` inattivo vs. Applicazione monolitica – Latenza futura (Deviazione standard)

Riprendendo quindi in considerazione i grafici della latenza, ma questa volta eliminando i primi 100 campioni, si ottiene la figura 4.16, la quale effettivamente conferma quanto detto precedentemente e mostra come l'applicazione serverless, se realizzata utilizzando il supporto offerto dal framework Knative, sia più performante poiché più rapida nel rispondere alle richieste dei client ed anche più stabile.

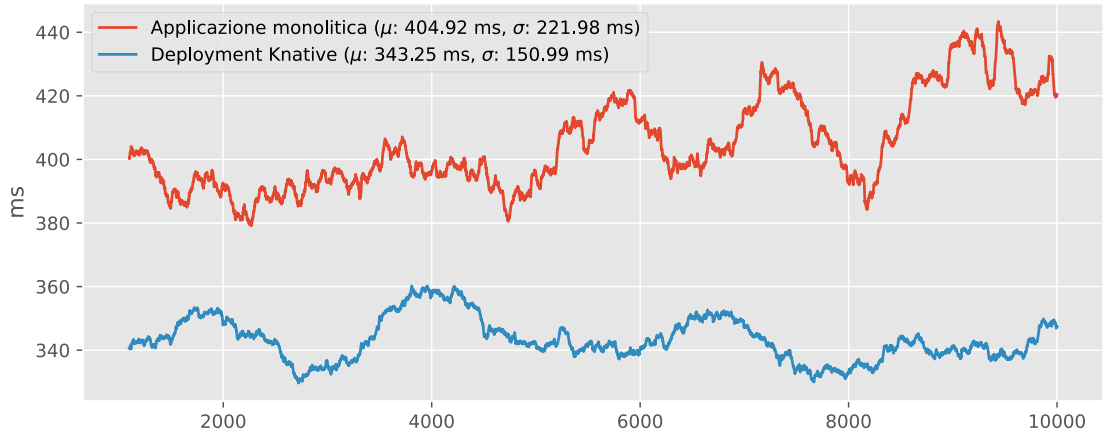

Figura 4.16. `Knative Service` inattivo vs. Applicazione monolitica – Latenza terminato il transitorio

Tempo di messa in campo del container

Per stimare, in termini temporali, la durata del periodo transitorio per un container messo in campo come `Knative Service`, è possibile procedere considerando il numero di richieste nell'unità di tempo, o meglio l'intervallo temporale trascorso tra due richieste successive e, conseguentemente, valutare la quantità di richieste smaltite dall'applicazione prima di ottenere valori stazionari per

la media e la deviazione standard. Considerando che questo test esegue 10000 richieste in un tempo $T = 2000$ secondi, l'intervallo che intercorre tra una richiesta e la successiva equivale a $\Delta t = 0.2\,\mathrm{s}$. Dai grafici precedenti risulta evidente come le misurazioni si stabilizzino completamente tra la sessantesima e la settantacinquesima richiesta. Di conseguenza, si può concludere che il *periodo transitorio* che intercorre tra la ricezione della prima richiesta, messa in campo e l'effettiva e completa operatività del container equivale ad un tempo T_t compreso tra due estremi:

$$60\,\Delta t \leq T_t \leq 75\,\Delta t$$
$$12\,\mathrm{s} \leq T_t \leq 15\,\mathrm{s}$$

Risorse computazionali

Per quanto concerne invece le risorse computazionali utilizzate, è necessario osservare le figure 4.17 e 4.18. Da queste figure si può immediatamente notare come l'attività in termini di CPU sia decisamente simile tra le due architetture, esattamente come ci si aspetta, poiché eseguono le stesse operazioni.

In termini di memoria, invece, è naturale che l'endpoint realizzato come `Knative Service` utilizzi decisamente meno risorse, comparabili con la soluzione messa in campo come `Deployment` Kubernetes.

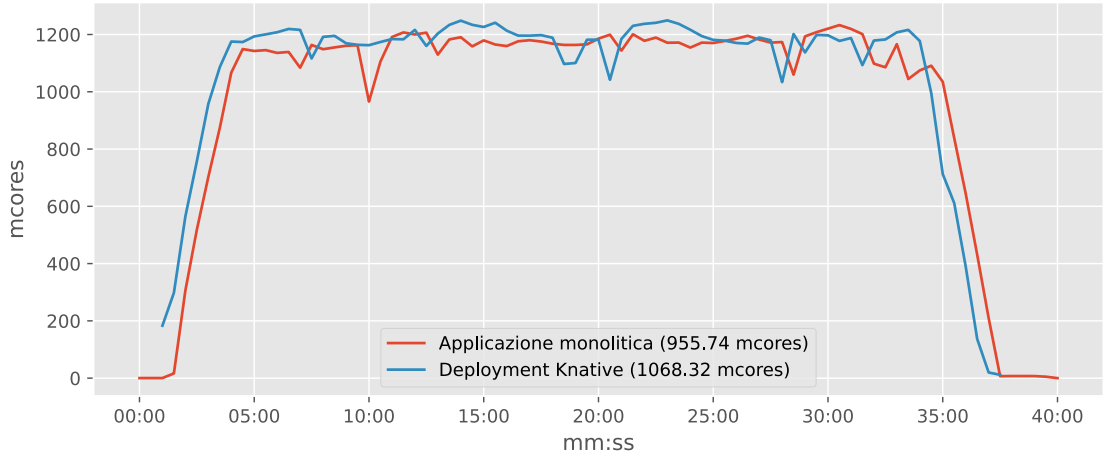

Figura 4.17. `Knative Service` inattivo vs. Applicazione monolitica – Utilizzo CPU

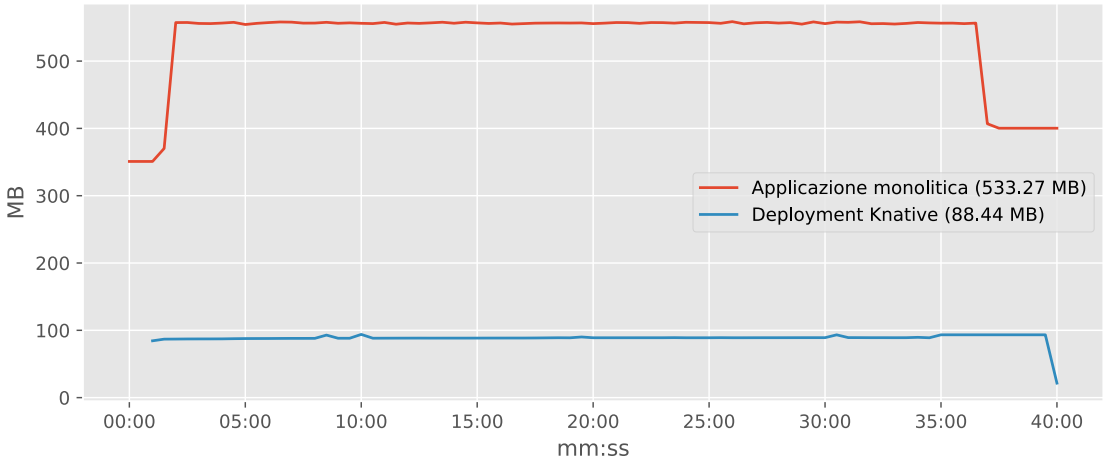

Figura 4.18. `Knative Service` inattivo vs. Applicazione monolitica – Utilizzo memoria

In entrambe le osservazioni, sia in termini di utilizzo CPU che in termini di utilizzo memoria, è bene prendere nota del fatto che queste risorse siano sostanzialmente identiche a quelle impiegate da un ordinario `Deployment` Kubernetes, soggetto allo stesso stimolo, come mostrato nella sezione 4.3.1, in particolare con le figure 4.11 e 4.12. Il contributo del sidecar container `queue-proxy` risulta quindi decisamente meno significativo rispetto a quanto mostrato per l'inserimento di utenti nella sezione 4.2.1, poiché l'impatto di questo container in termini di risorse computazionali, rispetto ad una classica soluzione con `Deployment` Kubernetes, risulta completamente assorbito durante il reale utilizzo dell'endpoint stesso.

4.4 Utilizzo misto

Sino a questo punto sono state effettuate diverse osservazioni con l'obiettivo di emulare semplici casi d'uso. Tuttavia, un'applicazione reale non è soggetta solamente a singoli stimoli su specifici endpoint, bensì è soggetta ad un insieme di richieste eterogenee durante lo stesso intervallo di tempo.

4.4.1 Lettura libri e lettura clienti

In questo scenario viene ipotizzato il caso d'uso in cui vengono contemporaneamente effettuate richieste di lettura libri e di lettura clienti, ad esempio per costruire un'applicazione frontend che integri in qualche modo questi dati.

Per la realizzazione di questo test vengono contattati i seguenti endpoint con le modalità indicate:

- `GET /books` con 5000 richieste in 1000 secondi;
- `GET /customers` con 5000 richieste in 1000 secondi.

In questo modo viene mantenuto lo stesso numero di richieste al secondo come nei test precedenti per la lettura dei libri, discussi nella sezione 4.3.

Latenza

Le figure 4.19 e 4.20 mostrano l'andamento della latenza per le due soluzioni nei diversi endpoint. Queste figure sono necessarie per accertarsi che ciascuna delle due applicazioni si comporti allo stesso modo tra i propri endpoint. Osservando i grafici, infatti, è possibile notare come gli andamenti delle due latenze seguano un andamento decisamente simile, a tratti anche interlacciato.

Da queste figure, come già illustrato nella figura 4.9, è immediato osservare come la latenza sia mediamente minore[1] per la soluzione a microservizi, seppur con una maggiore variabilità[2].

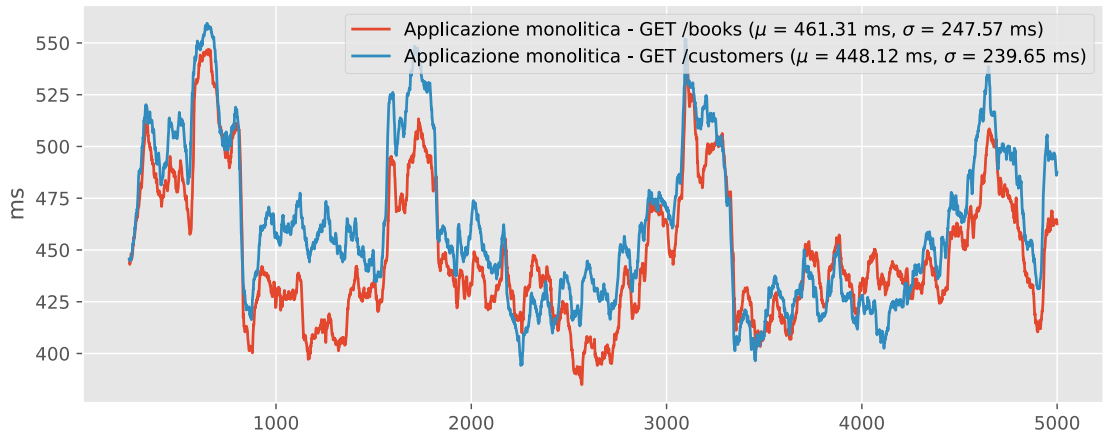

Figura 4.19. Lettura di libri e clienti – Latenza applicazione monolitica

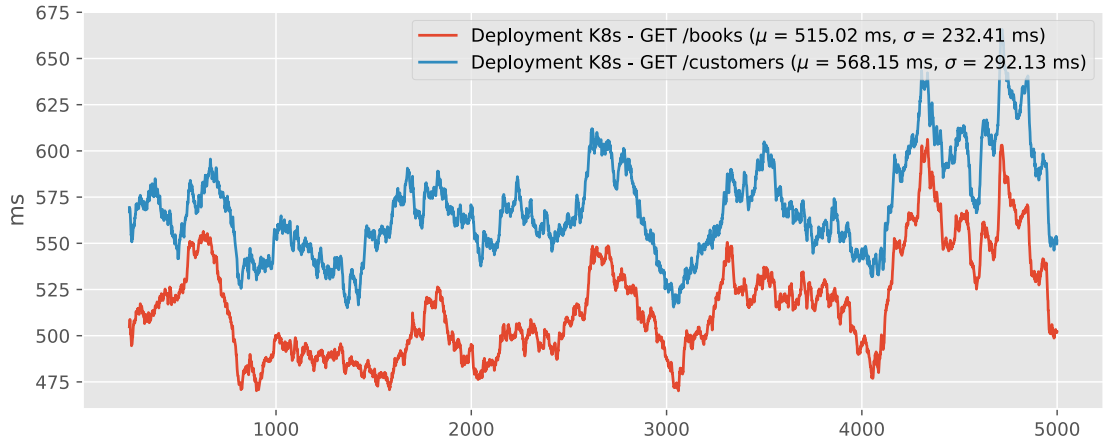

Figura 4.20. Lettura di libri e clienti – Latenza applicazione serverless

[1] La media della latenza è indicata in figura con il simbolo μ.

[2] La variabilità è misurata come deviazione standard, indicata in figura con il simbolo σ.

Le figure 4.21 e 4.22, invece, sono utili per confrontare rispettivamente le risposte degli endpoint `GET /books` e `GET /customers` delle due differenti soluzioni. Queste figure, nuovamente, mostrano come la soluzione a microservizi presenti una latenza *sempre* inferiore alla soluzione monolitica.

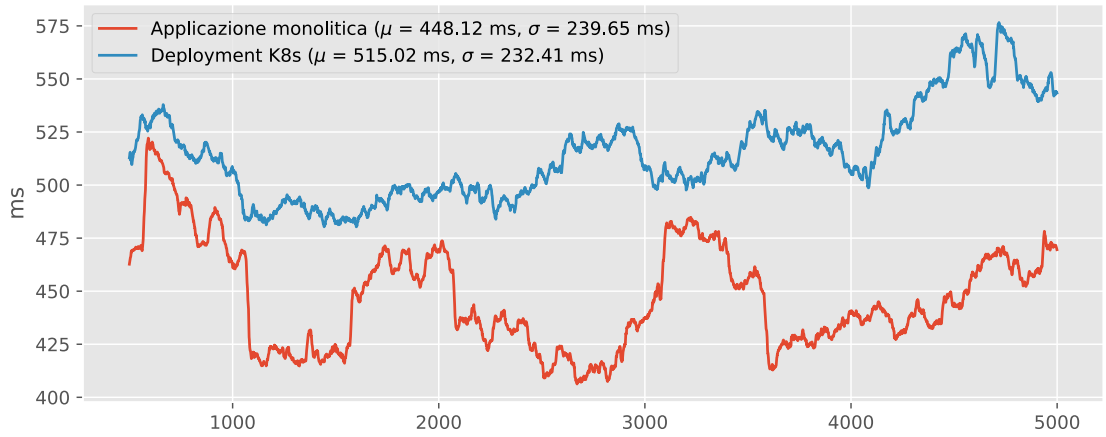

Figura 4.21. Lettura di libri e clienti – Confronto latenza endpoint `GET /books`

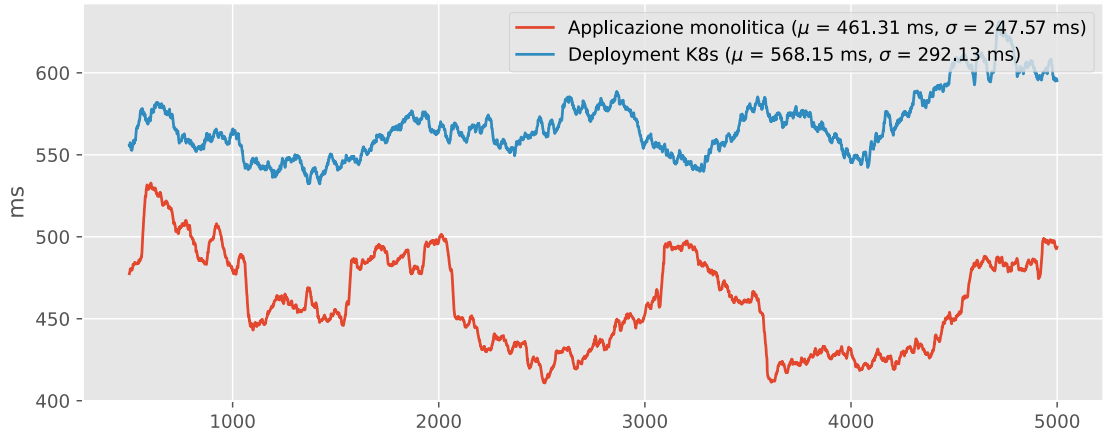

Figura 4.22. Lettura di libri e clienti – Confronto latenza endpoint `GET /customers`

Risorse computazionali

Le figure 4.23 e 4.24 mostrano il confronto delle risorse utilizzate dalle due architetture. In questi grafici i dati provenienti dai `Deployment` `get-books` e `get-customers`, responsabili rispettivamente della gestione delle richieste `GET /books` e `GET /customers` nella soluzione serverless, vengono sommati in modo tale da permetterne il confronto con l'architettura monolitica.

In particolare, dalla figura 4.23 è possibile notare, come effettivamente ci si aspetta, un utilizzo totale di CPU decisamente simile tra le due soluzioni, decisamente superiore, di circa un 50%, rispetto al test in cui vengono effettuate solo richieste `GET /books`, mostrato nella sezione 4.3,

poiché in questo caso vengono contattati più endpoint contemporaneamente e la computazione crittografica risulta quindi doppia.

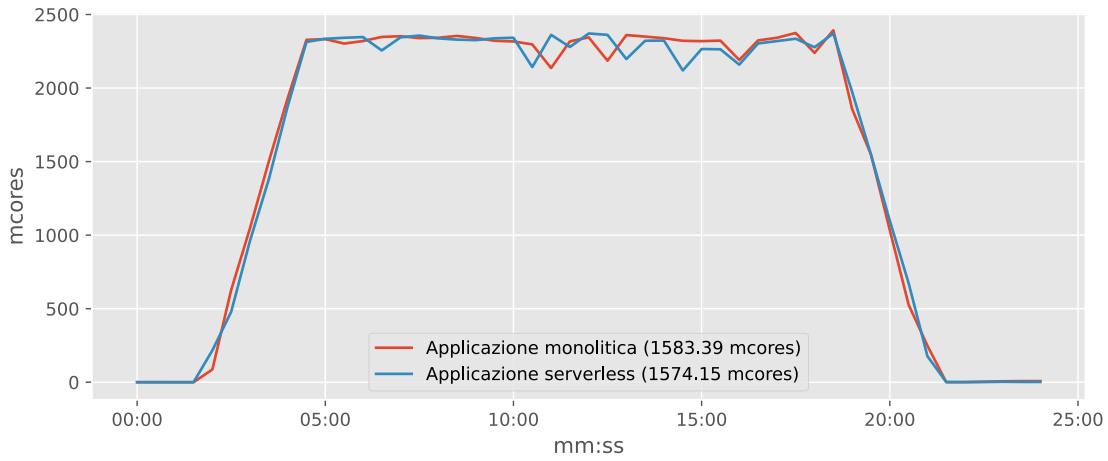

Figura 4.23. Lettura di libri e clienti – Confronto utilizzo CPU

Per quanto riguarda invece il consumo di memoria, dalla figura 4.24 è possibile notare un utilizzo minore di questa risorsa, giustificato dal fatto che, probabilmente, le richieste, essendo in numero minore su ciascun endpoint, vengono servite più in fretta e quindi ne rimangono accodate un numero inferiore lato backend in attesa di essere processate.

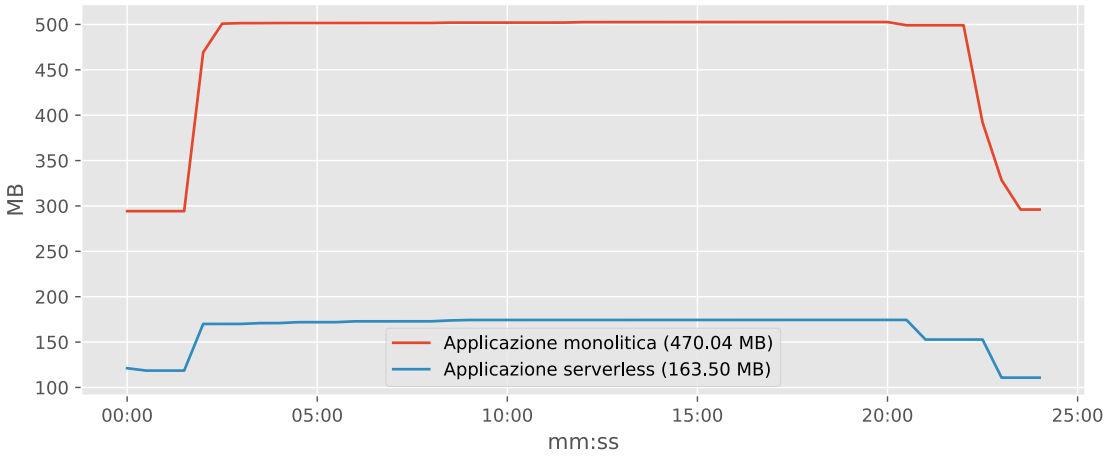

Figura 4.24. Lettura di libri e clienti – Confronto utilizzo memoria

Risorse computazionali – Soluzione serverless

Le figure 4.25 e 4.26 mostrano rispettivamente il dettaglio delle risorse utilizzate dagli endpoint `GET /books` e `GET /customers` realizzati come `Deployment` Kubernetes per l'architettura serverless.

Dal confronto di questi grafici è possibile affermare che gli endpoint utilizzano sostanzialmente lo stesso ammontare di risorse computazionali.

Questa situazione è giustificata da due evidenti fattori:

1. Gli endpoint sono soggetti alle stesse ipotesi, illustrate nella sezione 3.3, realizzati e messi in campo con le stesse modalità;
2. Gli endpoint sono soggetti allo stesso stimolo, poiché ricevono lo stesso numero di richieste nello stesso intervallo di tempo.

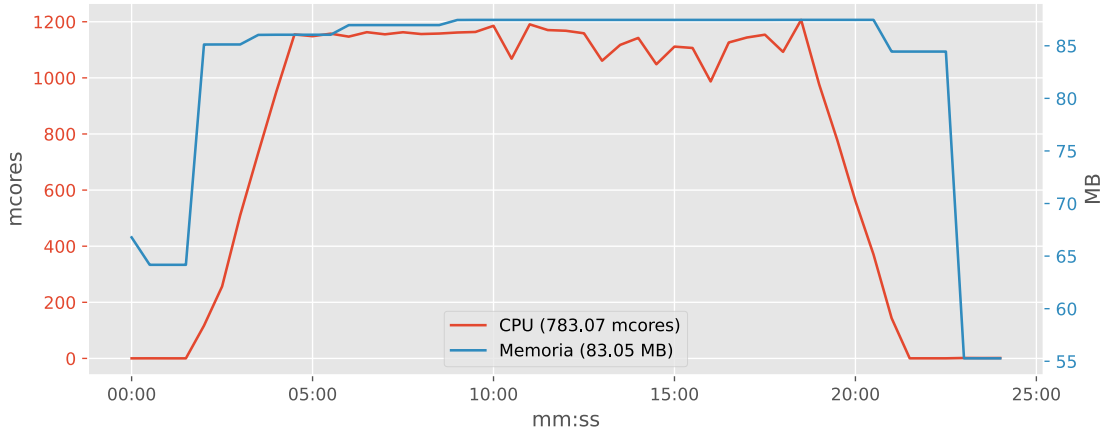

Figura 4.25. Lettura di libri e clienti – Risorse `Deployment get-books`

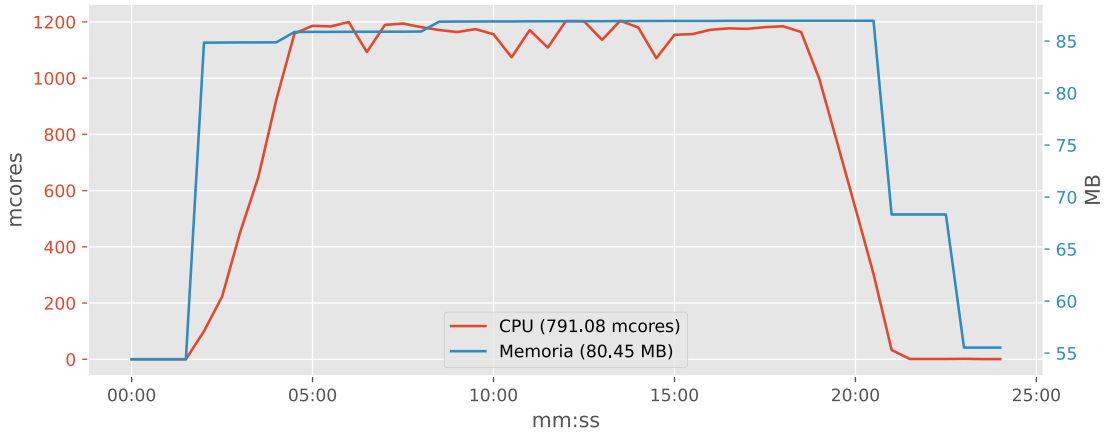

Figura 4.26. Lettura di libri e clienti – Risorse `Deployment get-customers`

Stress test

Con l'obiettivo di cercare il massimo numero di richieste che le due applicazioni riescono a gestire, è necessario effettuare uno *stress test*. In particolare, gli endpoint vengono contattati come segue:

- `GET /books` con 7500 richieste in 1000 secondi;
- `GET /customers` con 7500 richieste in 1000 secondi.

Il numero di richieste complessive è quindi maggiore del 50% rispetto al test precedente, con un numero totale di richieste inviate per ogni secondo pari a 15.

Da questo test, a livello prettamente qualitativo, si può affermare che le risorse utilizzate aumentano in misura proporzionale rispetto al carico. Quest'ultimo cresce del 50% e con esso anche CPU e memoria utilizzati, poiché con l'aumento del numero di richieste, cresce conseguentemente la quantità di CPU utilizzata a causa della computazione crittografica inserita e cresce la quantità di memoria considerando che le richieste accodate in attesa di essere servite saranno in numero maggiore.

Per quanto concerne invece la latenza, questa misura cresce decisamente, spostando il valore medio intorno al secondo ma mantenendo pressoché invariato il valore dello scarto quadratico medio.

Ciò che appare decisamente interessante è il numero di richieste a cui le due applicazioni rispondono con successo. Infatti, da questo test, i cui risultati sono mostrati nella tabella 4.1, emerge come l'applicazione monolitica inizi ad avvicinarsi al suo *punto di rottura*.

	Applicazione monolitica		**Applicazione serverless**	
	`GET /books`	`GET /customers`	`GET /books`	`GET /customers`
`200 Ok`	6940 (92.53%)	6928 (92.53%)	7500 (100%)	7499 (99.99%)
`503 Service unavailable`	560 (7.47%)	572 (7.63%)	0 (0%)	1 (0.01%)

Tabella 4.1. Lettura di libri e clienti – Statistiche stress test

In conclusione, è quindi possibile sostenere che l'applicazione a microservizi risulti più efficace nello smaltire una grossa mole di richieste vicine temporalmente tra loro, poiché queste funzioni nascono come entità indipendenti, che fanno quindi uso di risorse in modo parallelo. Inoltre, tali microservizi non si devono occupare internamente del routing della richiesta, compito che invece spetta all'applicazione monolitica, poiché questo viene effettuato a monte da Istio e da esso direzionato verso il `Deployment` opportuno.

4.4.2 Letture ed inserimenti contemporanei

In questo scenario, viene ipotizzato il caso d'uso in cui vengono contemporaneamente effettuate richieste di lettura libri e di inserimento prestiti, poiché il reale utilizzo di una biblioteca, tendenzialmente, si compone di una grande quantità di ricerche e contestualmente una minore richieste di prestiti. Per la realizzazione di questo test vengono contattati i seguenti endpoint con le modalità indicate:

- `GET /books/{isbn}` con 5000 richieste in 1000 secondi;
- `POST /loans` con 1000 richieste in 1000 secondi. Per quanto riguarda l'applicazione serverless, naturalmente, l'inserimento di un prestito non viene effettuato tramite una richiesta `HTTP POST`, ma attraverso un messaggio sul topic *loans* in Apache Kafka.

Latenza

Al fine di confrontare la latenza tra le due soluzioni, è possibile procedere in modo analogo a quanto riportato precedentemente nella sezione 4.3.2, cioè valutando latenza media e deviazione standard futuri in maniera tale da escludere i primi campioni, i quali sono soggetti a latenze decisamente fuori scala per l'applicazione serverless, a causa del tempo necessario per la messa in campo del container.

La figura 4.27 mostra l'andamento della latenza, opportunamente filtrato come indicato sopra, per le due soluzioni. Come ci si può aspettare, le due soluzioni hanno un andamento decisamente simile, con valori non troppo diversi al test per la sola lettura di libri mostrato nella figura 4.16. Rispetto a questa figura, i valori medi delle latenze si abbassano, poiché vengono ricevute complessivamente meno richieste, che vengono quindi processate in un tempo minore evitando tempi di attesa prima che queste vengano servite.

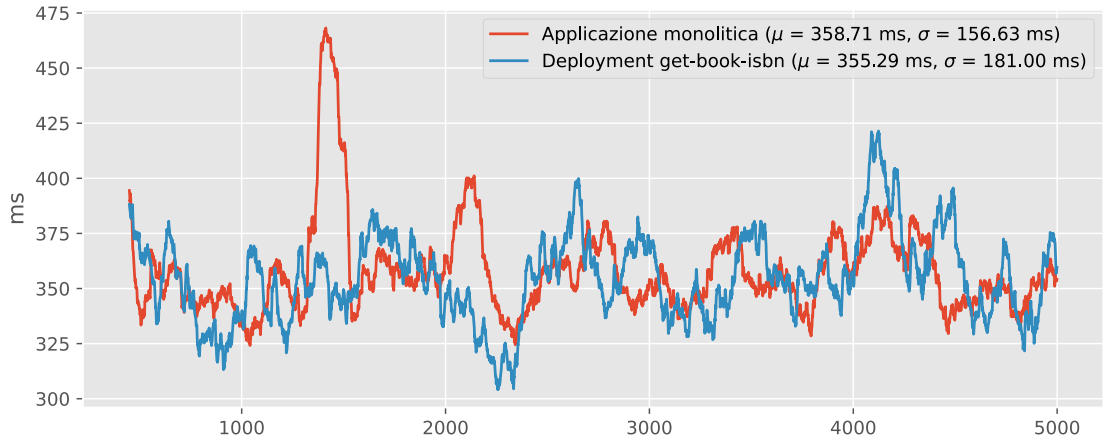

Figura 4.27. Lettura libri ed inserimento prestiti – Latenza endpoint `GET /books/{isbn}`

La figura 4.28 mostra invece l'andamento della latenza dell'endpoint `POST /loans` dell'applicazione monolitica. Dal confronto di questa figura con la figura 4.8, relativa al solo inserimento di prestiti, si osserva come gli andamenti della latenza siano pressoché uguali in entrambi i casi, con valori decisamente simili per quanto riguarda la media e la deviazione standard.

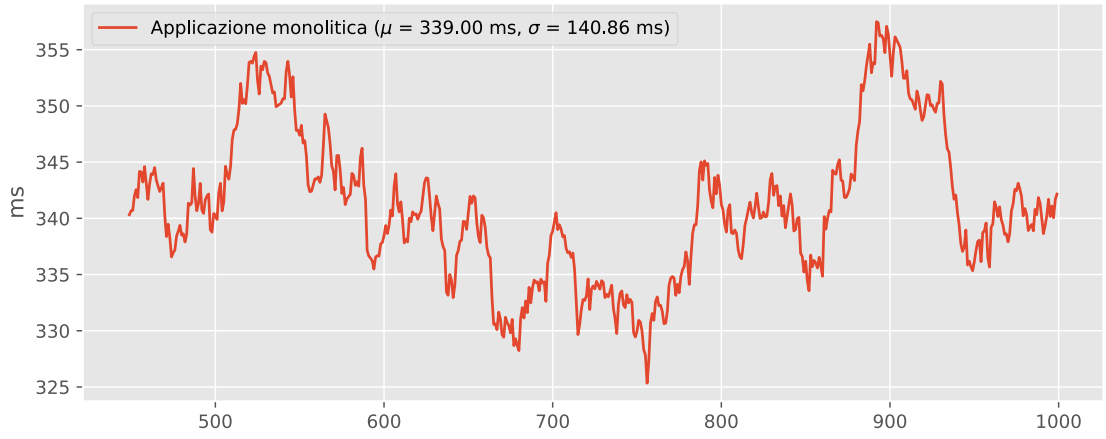

Figura 4.28. Lettura libri ed inserimento prestiti – Latenza endpoint `POST /loans`

Da questi due confronti, è possibile affermare che con questi carichi l'applicazione monolitica non subisca cali di prestazioni, misurate come latenza avvertita lato client, nel servire richieste contemporanee su diversi endpoint.

Risorse computazionali

Per quanto riguarda invece la quantità di risorse impiegate, è necessario osservare le figure seguenti, nelle quali vengono confrontate CPU e memoria tra le due architetture. In questi grafici, la curva etichettata come *Applicazione serverless* costituisce la somma delle risorse utilizzate dai due endpoint dell'architettura a microservizi.

Dalla figura 4.29 si può notare che la quantità di CPU utilizzata è complessivamente molto simile tra le due architetture, poiché vengono effettuate le stesse operazioni. L'applicazione serverless risulta avere un utilizzo lievemente superiore poiché vengono messi in campo anche due container `queue-proxy`, uno per ciascun `Knative Service` avviato in questo test.

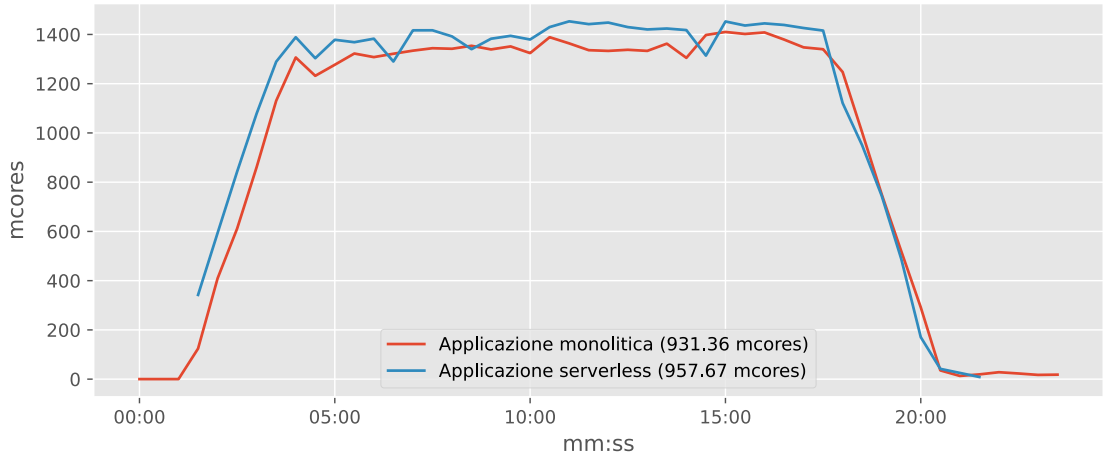

Figura 4.29. Lettura libri ed inserimento prestiti – Utilizzo CPU

La figura 4.30 mostra invece la quantità di memoria sfruttata dalle due architettura. È immediato notare un andamento decisamente simile rispetto alla figura 4.12, nella quale veniva osservata la quantità di memoria impiegata durante la lettura di diversi libri, sempre attraverso l'endpoint `GET /books/{isbn}`.

La quantità di memoria utilizzata dall'applicazione monolitica, infatti, risulta comparabile con quanto visto nei test precedenti. Per quanto concerne l'applicazione serverless, invece, esattamente come ci si aspetta, la quantità di memoria totale utilizzata risulta essere, con buonissima approssimazione, la somma della quantità di memoria impiegata dai due `Knative Service`, mostrati nei test precedenti con le figure 4.4 e 4.18, ad ulteriore conferma del fatto che i moduli dell'applicazione serverless lavorino in modo indipendente.

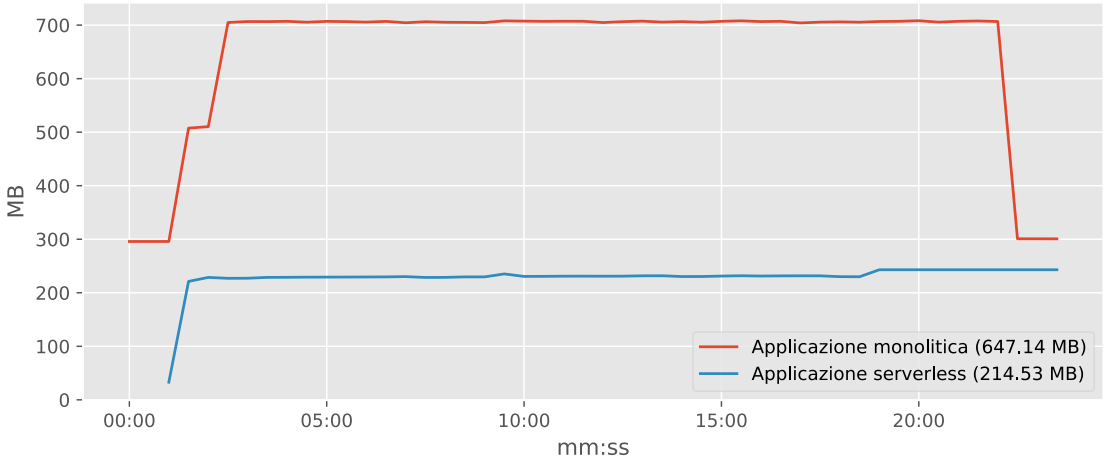

Figura 4.30. Lettura libri ed inserimento prestiti – Utilizzo memoria

4.5 Autoscaling

Durante le fasi di test dei due applicativi, sono anche state testate politiche di *autoscaling*. Tali policy di autoscaling hanno senso quando il carico di lavoro per l'applicazione risulta variabile nel tempo in modo tale da adattare dinamicamente il numero di `Pod` messi in campo in base ad esso.

In generale, il meccanismo di autoscaling risulta utile in tutte quelle casistiche in cui il traffico diretto verso il cluster non costituisce un flusso costante e regolare, ad esempio per processi batch di manutenzione, avviati in orari specifici, oppure per tutti i meccanismi di *Continuous Integration/Continuous Development* per i quali i test risultano meno utili durante il weekend o le ore notturne.

4.5.1 In Kubernetes

Dalla versione 1.3, Kubernetes mette a disposizione la risorsa `Horizontal Pod Autoscaler` che permette di impostare un valore percentuale di carico medio in termini di CPU, un numero minimo ed un numero massimo di repliche per i `Pod` istanziati dal relativo `Deployment`. Per poter impostare una percentuale di carico, è necessario innanzitutto configurare gli attributi `resources.limits.cpu` e `resources.requests.cpu`, i quali indicano rispettivamente il limite massimo di CPU che ciascun `Pod` può utilizzare ed il limite minimo che ciascun `Pod` richiede per poter essere messo in campo. In questo modo, l'`Horizontal Pod Autoscaler`, confrontando il carico attuale con il carico desiderato, riesce ad incrementare o decrementare il numero di repliche, agendo sugli attributi del `Deployment` che questo monitora, in modo da mantenere circa costante il carico di CPU sui diversi `Pod`.

Può succedere, soprattutto per un'applicazione medio–grande, come ad esempio la biblioteca in versione monolitica presentata, che le risorse minime per istanziare un nuovo `Pod` non siano disponibili sul cluster, soprattutto su un cluster di prova come quello utilizzato in questa tesi. Seppur molto utile teoricamente, risulta, a livello pratico, piuttosto difficile configurare in modo efficiente ed efficace un `Horizontal Pod Autoscaler` per tale applicazione, poiché il dimensionamento dei parametri `resources.limits.cpu` e `resources.requests.cpu` richiederebbe una fase di analisi e studio approfondito riguardanti il carico previsto per l'applicazione.

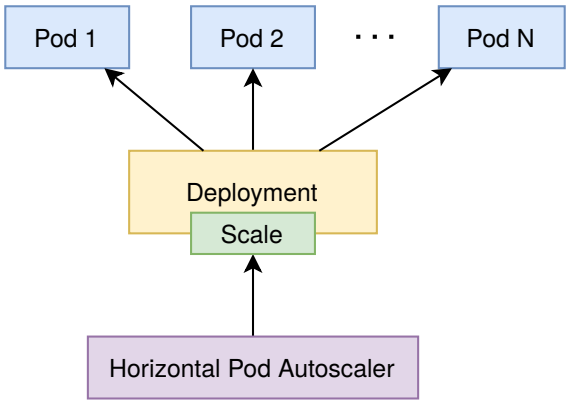

Figura 4.31. `Horizontal Pod Autoscaler`

Per quanto riguarda invece il decremento del numero di repliche dei `Pod`, noto anche come *scaling down*, questo risulta solitamente molto più lento, se confrontato con l'incremento, noto anche

come *scaling up*. Infatti, se sono sufficienti pochissimi minuti per lo *scaling up*, lo *scaling down* può addirittura richiedere una decina di minuti, perché il meccanismo di autoscaling cerca di assicurarsi che tali risorse non siano realmente più necessarie prima di eliminare i `Pod` messi in campo.

Funzionamento

In Kubernetes, l'`Horizontal Pod Autoscaler` implementa il meccanismo di un controllore, il quale ha il compito di intercettare le metriche inviate dalle risorse a cui è interessato ad intervalli di tempo regolari. Ottenute queste metriche, che possono essere standard oppure custom definite dallo sviluppatore, che deve attenersi ad opportune specifiche, l'`Horizontal Pod Autoscaler` effettua una serie di operazioni matematiche per calcolare la percentuale di utilizzo della risorsa ed il numero di repliche necessarie a smaltire tale carico, confrontando questo valore calcolato con il valore di target configurato.

4.5.2 In Kubeless

Kubeless implementa politiche di autoscaling utilizzando l'oggetto `Horizontal Pod Autoscaler` fornito da Kubernetes, in modo tale da scalare il numero di repliche in base a metriche ben definite per il carico di lavoro. Anche in Kubeless è quindi possibile impostare l'autoscaling in base a metriche standard, come ad esempio l'utilizzo di CPU, oppure in base a metriche custom implementate dallo sviluppatore dell'applicazione.

Essendo dotato del tool `kubeless` da riga di comando, è possibile implementare rapidamente un oggetto `Horizontal Pod Autoscaler`, in quanto questo comando, insieme con tutti i suoi sottocomandi e relative opzioni, funge come wrapper per la creazione e la gestione di risorse Kubernetes.

4.5.3 In Knative

Knative, oltre a supportare il classico meccanismo di autoscaling presente in Kubernetes appena mostrato, attraverso oggetti `Horizontal Pod Autoscaler`, realizza il proprio meccanismo di autoscaling con oggetti `Knative Pod Autoscaler` disponibili nella componente Serving, implementando logiche più complesse in modo tale da gestire il numero di repliche in modo più efficace in base alla quantità di richieste.

Nella maggior parte dei casi, i meccanismi implementati da `Knative Service` e `Knative Pod Autoscaler` sono più che sufficienti, ma è comunque possibile impostare configurazioni particolari per esigenze non standard. Ad esempio, è possibile impostare un meccanismo di *downscaling*, utilizzato in questa tesi per tutti i `Knative Service` illustrati, fino a zero repliche se non viene ricevuto traffico e, associato a questo, il periodo di tempo minimo e massimo che il sistema deve attendere prima di scalare le repliche a zero, offrendo quindi diverse possibilità per ottimizzare il carico di lavoro in base alle proprie necessità. È inoltre possibile impostare limiti per il numero di richieste simultanee attraverso etichette ed annotazioni.

Tutti questi meccanismi sono possibili grazie alle logiche implementate dal container `queue-proxy`, il quale viene automaticamente iniettato come *sidecar container* accanto al container principale di ciascun `Knative Service` ed ha il compito di forzare tali policy di concorrenza tra richieste.

Capitolo 5

Calcoli e stime

In questo capitolo si cerca di stimare e confrontare in termini quantitativi, e non solamente qualitativi come invece è avvenuto nel capitolo precedente, il costo di un'architettura monolitica e di un'architettura serverless ipotizzando il caso d'uso di una biblioteca scolastica.

Prima di poter procedere con i calcoli numerici, è necessario formulare diverse ipotesi, ragionevoli con lo scenario scelto, immaginando una giornata tipo:

1. Tre fasce orarie in cui vengono cercati libri dagli utenti della biblioteca, distribuite lungo la giornata;
2. Due fasce orarie in cui vengono inserite richieste di prestito e/o resi, ad esempio una a fine ed una ad inizio giornata;
3. Una fascia oraria in cui in cui vengono effettuate ricerche per le quotidiane operazioni da parte degli operatori, simulate come ricerche di clienti o libri;
4. Una fascia oraria in cui in cui vengono effettuati inserimenti o aggiornamenti di dati, oppure vengono effettuate operazioni di quotidiana manutenzione dagli operatori, simulate come inserimento di clienti.

5.1 Raccolta dati

Al fine di confrontare nel modo più oggettivo possibile i consumi in termini di CPU e memoria RAM, è necessario utilizzare la media ponderata come criterio di calcolo:

$$\bar{x} = \frac{\sum_i \bar{x_i} \Delta t_i}{T}$$

dove $\bar{x_i}$ indica il valore medio durante il periodo i, di durata Δt_i considerato e $T = 24$ ore.

Quindi, si ha

$$\overline{\text{cpu}} = \frac{\sum_i \overline{\text{cpu}_i} \Delta t_i}{T} \tag{5.1}$$

e

$$\overline{\text{memory}} = \frac{\sum_i \overline{\text{memory}_i} \Delta t_i}{T} \tag{5.2}$$

I valori medi $\overline{\text{cpu}_i}$ e $\overline{\text{memory}_i}$ utilizzati sono quelli presenti nelle figure del capitolo precedente, indicati tra le parentesi nei grafici, talvolta identificati con il simbolo μ.

Per ragioni pratiche, i test dell'applicazione sono stati effettuati su intervalli di tempo limitati. Per effettuare i calcoli e simulare scenari più dilatati nel tempo, si è scelto di utilizzare comunque tali valori medi, sottostando alla ragionevole ipotesi che, con un *rate* di richieste simile a quello riportato nei test, le due applicazioni, in termini di risorse impiegate, possano avere comportamenti simili.

5.1.1 Ricerca libri

Ipotizzando tre fasce orarie, ciascuna della durata di un'ora, in cui vengono cercati libri, è possibile confrontare le due architetture come segue.

In questo scenario è necessario scegliere su quale endpoint effettuare il calcolo per quanto riguarda l'applicazione serverless, poiché, a livello teorico `Deployment` Kubernetes e `Knative Service` sfruttano risorse computazionali in modo diverso. A livello pratico, come è invece mostrato dal confronto delle coppie figure 4.11 e 4.17 riguardanti la CPU e 4.12 e 4.18 riguardanti la memoria, le risorse impiegate non sono così diverse ed è quindi stato scelto di utilizzare il caso peggiore, cioè quello in cui viene messo in campo un `Knative Service`, poiché questo necessita di una maggiore quantità di risorse, soprattutto in condizioni di riposo.

$$\overline{\text{cpu}}_{\text{FaaS}} = 1068.32\,\text{mcores}$$
$$\overline{\text{memory}}_{\text{FaaS}} = 88.44\,\text{MB}$$

$$\overline{\text{cpu}}_{\text{monolithic}} = 955.74\,\text{mcores}$$
$$\overline{\text{memory}}_{\text{monolithic}} = 533.41\,\text{MB}$$

Ponendo quindi $T = 3\,\text{ore}$, è possibile stimare la quantità giornaliera di risorse impiegate dell'endpoint `GET /books/{isbn}`, ed analogamente per l'endpoint `GET /customers/{id}`:

$$\overline{\text{cpu}}_{\text{FaaS}} = \frac{1068.32\,\text{mcores} \cdot 3\,\text{h}}{24\,\text{h}} = 133.54\,\text{mcores} \tag{5.3}$$
$$\overline{\text{memory}}_{\text{FaaS}} = \frac{88.44\,\text{MB} \cdot 3\,\text{h}}{24\,\text{h}} = 11.06\,\text{MB} \tag{5.4}$$

$$\overline{\text{cpu}}_{\text{monolithic}} = \frac{955.74\,\text{mcores} \cdot 3\,\text{h}}{24\,\text{h}} = 119.47\,\text{mcores} \tag{5.5}$$
$$\overline{\text{memory}}_{\text{monolithic}} = \frac{533.41\,\text{MB} \cdot 3\,\text{h}}{24\,\text{h}} = 66.68\,\text{MB} \tag{5.6}$$

Per estensione, tali valori possono essere impiegati anche per stimare l'utilizzo di risorse degli endpoint `GET /books`, ed analogamente per l'endpoint `GET /customers`, poiché, nonostante questi endpoint siano messi in campo come `Deployment` Kubernetes, sarebbe certamente possibile una loro implementazione utilizzando `Knative Service`, con costi computazionali simili.

5.1.2 Inserimento prestiti e resi

Ipotizzando due fasce orarie, ciascuna della durata di un'ora, in cui vengono inseriti nella prima una serie di prestiti e nelle seconda una serie di resi, è possibile confrontare le due architetture come segue.

$$\overline{\text{cpu}}_{\text{FaaS}} = 179.49\,\text{mcores}$$
$$\overline{\text{memory}}_{\text{FaaS}} = 118.86\,\text{MB}$$

$$\overline{\text{cpu}}_{\text{monolithic}} = 179.85\,\text{mcores}$$
$$\overline{\text{memory}}_{\text{monolithic}} = 487.70\,\text{MB}$$

Ponendo quindi $T = 2\,\text{ore}$, è possibile stimare la quantità giornaliera di risorse impiegate dell'endpoint `POST /loans`, ed analogamente per l'endpoint `PUT /loans`:

$$\overline{\text{cpu}}_{\text{FaaS}} = \frac{179.49\,\text{mcores} \cdot 2\,\text{h}}{24\,\text{h}} = 14.96\,\text{mcores} \tag{5.7}$$
$$\overline{\text{memory}}_{\text{FaaS}} = \frac{118.86\,\text{MB} \cdot 2\,\text{h}}{24\,\text{h}} = 9.91\,\text{MB} \tag{5.8}$$

$$\overline{\text{cpu}}_{\text{monolithic}} = \frac{179.85\,\text{mcores} \cdot 2\,\text{h}}{24\,\text{h}} = 14.99\,\text{mcores} \tag{5.9}$$
$$\overline{\text{memory}}_{\text{monolithic}} = \frac{487.70\,\text{MB} \cdot 2\,\text{h}}{24\,\text{h}} = 40.64\,\text{MB} \tag{5.10}$$

5.1.3 Ricerca clienti

Ipotizzando una singola fascia oraria, della durata di un'ora, in cui vengono effettuate operazioni di lettura di vario tipo, è possibile confrontare le due architetture come segue.

In modo analogo alla sezione 5.1.1 è possibile estrarre gli stessi valori. Ponendo questa volta $T = 1\,\text{ora}$, è possibile stimare la quantità giornaliera di risorse impiegate dell'endpoint `GET /customers` o, analogamente, con le medesime motivazioni, dell'endpoint `GET /customers/{id}`:

$$\overline{\text{cpu}}_{\text{FaaS}} = \frac{1068.32\,\text{mcores} \cdot 1\,\text{h}}{24\,\text{h}} = 44.51\,\text{mcores} \tag{5.11}$$
$$\overline{\text{memory}}_{\text{FaaS}} = \frac{88.44\,\text{MB} \cdot 1\,\text{h}}{24\,\text{h}} = 3.69\,\text{MB} \tag{5.12}$$

$$\overline{\text{cpu}}_{\text{monolithic}} = \frac{955.74\,\text{mcores} \cdot 1\,\text{h}}{24\,\text{h}} = 39.82\,\text{mcores} \tag{5.13}$$
$$\overline{\text{memory}}_{\text{monolithic}} = \frac{533.41\,\text{MB} \cdot 1\,\text{h}}{24\,\text{h}} = 22.23\,\text{MB} \tag{5.14}$$

5.1.4 Inserimento clienti

Ipotizzando una singola fascia oraria, della durata di un'ora, in cui vengono inseriti o modificati clienti, oppure effettuate operazioni di manutenzione, è possibile confrontare le due architetture come segue.

In modo analogo alla sezione 5.1.2 è possibile estrarre gli stessi valori. Ponendo questa volta $T = 1\,\text{ora}$, è possibile stimare la quantità giornaliera di risorse impiegate dell'endpoint `POST /customers`:

$$\overline{\text{cpu}}_{\text{FaaS}} = \frac{179.49\,\text{mcores} \cdot 1\,\text{h}}{24\,\text{h}} = 7.48\,\text{mcores} \tag{5.15}$$

$$\overline{\text{memory}}_{\text{FaaS}} = \frac{118.86\,\text{MB} \cdot 1\,\text{h}}{24\,\text{h}} = 4.95\,\text{MB} \tag{5.16}$$

$$\overline{\text{cpu}}_{\text{monolithic}} = \frac{179.85\,\text{mcores} \cdot 1\,\text{h}}{24\,\text{h}} = 7.49\,\text{mcores} \tag{5.17}$$

$$\overline{\text{memory}}_{\text{monolithic}} = \frac{487.70\,\text{MB} \cdot 1\,\text{h}}{24\,\text{h}} = 20.32\,\text{MB} \tag{5.18}$$

5.1.5 A riposo

Utilizzando ancora l'ipotesi che tutti gli endpoint serverless vengano implementati come `Knative Service`, alcuni dei quali perennemente allocati a seguito di un'opportuna configurazione dei parametri del `Knative Pod Autoscaler`, l'applicazione a microservizi risulta costituita da quattro distinti endpoint costantemente allocati:

1. `GET /books`;
2. `GET /books/{isbn}`;
3. `GET /customers`;
4. `GET /customers/{id}`;

e da tre endpoint istanziati quando necessario che mappano i seguenti endpoint dell'applicazione monolitica:

1. `POST /customers`;
2. `POST /loans`;
3. `PUT /loans`.

Osservando i grafici 3.11 e 3.13 ed ipotizzando che tutti i servizi dell'applicazione serverless vengano implementati come `Knative Service`, è possibile stimare le risorse delle due applicazioni in condizioni di riposo:

$$\overline{\text{cpu}}_{\text{FaaS}} = 44.89\,\text{mcores} \cdot 4 = 179.56\,\text{mcores} \tag{5.19}$$

$$\overline{\text{memory}}_{\text{FaaS}} = 48.59\,\text{MB} \cdot 4 = 194.36\,\text{MB} \tag{5.20}$$

$$\overline{\text{cpu}}_{\text{monolithic}} = 0.24\,\text{mcores} \tag{5.21}$$
$$\overline{\text{memory}}_{\text{monolithic}} = 329.09\,\text{MB} \tag{5.22}$$

Avendo ipotizzato un periodo totale di lavoro di 7 ore, per le restanti 17 ore della giornata è possibile ipotizzare un periodo di riposo. Ponendo quindi $T = 7\,\text{ora}$, è possibile stimare la quantità giornaliera di risorse impiegate delle due applicazioni a riposo:

$$\overline{\text{cpu}}_{\text{FaaS}} = \frac{179.56\,\text{mcores} \cdot 17\,\text{h}}{24\,\text{h}} = 127.19\,\text{mcores} \tag{5.23}$$
$$\overline{\text{memory}}_{\text{FaaS}} = \frac{194.36\,\text{MB} \cdot 17\,\text{h}}{24\,\text{h}} = 137.67\,\text{MB} \tag{5.24}$$

$$\overline{\text{cpu}}_{\text{monolithic}} = \frac{0.24\,\text{mcores} \cdot 17\,\text{h}}{24\,\text{h}} = 0.17\,\text{mcores} \tag{5.25}$$
$$\overline{\text{memory}}_{\text{monolithic}} = \frac{329.09\,\text{MB} \cdot 17\,\text{h}}{24\,\text{h}} = 233.11\,\text{MB} \tag{5.26}$$

5.2 Stima scenario giornaliero

Ottenuti i dati dei diversi endpoint quando sottoposti a carico ed i dati delle applicazioni a riposo, è necessario combinare opportunamente tali valori. La tabella 5.1 riassume i valori precedentemente stimati, misurando l'utilizzo di CPU in mcores e l'utilizzo di memoria in MB.

	Applicazione serverless		**Applicazione monolitica**	
	CPU	**Memoria**	**CPU**	**Memoria**
Ricerca libri	133.54	11.06	119.47	66.68
Inserimento prestiti e resi	14.96	9.91	14.99	40.64
Ricerca clienti	44.51	3.69	39.82	22.23
Inserimento clienti	7.48	4.95	7.49	20.32
A riposo	127.19	137.67	0.17	233.11

Tabella 5.1. Risorse utilizzate

In uno scenario di un'applicazione reale, tale periodo di riposo potrebbe comunque essere soggetto a richieste di vario tipo, che con buona probabilità risulterebbero trascurabili in termini di risorse utilizzate. Tali richieste puntuali andrebbero comunque riportate su entrambe le applicazioni, spostando di conseguenza entrambe le stime a valori lievemente superiori, ma non ne modificherebbero il confronto a livello macroscopico.

$$\begin{aligned} \text{cpu}_{\text{FaaS}} &= \big(133.54 + 14.96 + 44.51 + 7.48 + 127.19\big)\,\text{mcores} \\ &= 327.68\,\text{mcores} \end{aligned} \tag{5.27}$$

$$\begin{aligned} \text{memory}_{\text{FaaS}} &= \big(11.06 + 9.91 + 3.69 + 4.95 + 137.67\big)\,\text{MB} \\ &= 167.28\,\text{MB} \end{aligned} \tag{5.28}$$

$$\begin{aligned} \text{cpu}_{\text{monolithic}} &= \big(119.47 + 14.99 + 39.82 + 7.49 + 0.17\big)\,\text{mcores} \\ &= 181.94\,\text{mcores} \end{aligned} \tag{5.29}$$

$$\begin{aligned} \text{memory}_{\text{monolithic}} &= \big(66.68 + 40.64 + 22.23 + 20.32 + 233.11\big)\,\text{MB} \\ &= 382.98\,\text{MB} \end{aligned} \tag{5.30}$$

La tabella 5.2 riassume questi ultimi calcoli, misurando l'utilizzo di CPU in mcores e l'utilizzo di memoria in MB.

	Applicazione serverless		**Applicazione monolitica**	
	CPU	**Memoria**	**CPU**	**Memoria**
Totale giornaliero	327.68	167.28	181.94	382.98

Tabella 5.2. Risorse utilizzate giornalmente

Da questi risultati, risulta evidente che l'applicazione realizzata con architettura serverless permetta di risparmiare, nello scenario ipotizzato, un quantitativo non trascurabile di memoria, pari a circa il 56%, a fronte di una maggiore richiesta di capacità computazionale di circa il 77%. Questa discrepanza è giustificata, per quanto riguarda l'utilizzo di memoria, dal numero ridotto di endpoint sempre attivi nell'applicazione serverless, nella quale viene quindi allocata una quantità minore di questa risorsa. Tali endpoint, però, essendo implementati come `Knative Service`, utilizzano ciascuno un maggiore quantitativo di CPU, che, anche in condizioni di riposo, risulta decisamente maggiore della quantità richiesta dall'applicazione monolitica.

Capitolo 6

Sviluppi futuri

Avendo costruito un'applicazione di esempio, i possibili sviluppi sono i più svariati. In una logica di business reale sarebbe necessario anteporre alla realizzazione dell'applicazione una oculata fase di analisi con l'obiettivo di individuare quali moduli sviluppare separatamente sfruttando un paradigma serverless o Function–as–a–Service e quali, eventualmente, implementare con una logica più tradizionale o non effettuare modifiche.

Le due applicazioni costituiscono esempi limitati nei quali la modifica dei dati non è permessa totalmente, a differenza di applicazioni complete e reali che implementano tali meccanismi. L'introduzione di nuovi endpoint che adempiano a tali compiti richiede una parziale aggiunta di codice per quanto riguarda la soluzione monolitica, con conseguente ricompilazione e distribuzione campo dell'intera applicazione. Il processo di estensione – e più in generale di evoluzione – risulta invece più snello per quanto riguarda l'applicazione serverless. Sarebbe infatti sufficiente mettere in campo il singolo microservizio in modo indipendente rispetto agli altri già resi disponili, poiché ciascuno di essi possiede il proprio ciclo di vita e le proprie peculiarità che ne caratterizzano la metodologia di erogazione.

Per rendere ancora più efficiente le politiche di autoscaling offerte da Knative, sarebbe possibile realizzare metriche custom che, intercettate dal `Knative Pod Autoscaler`, porterebbero ad ottenere meccanismi personalizzati facendo incontrare le specifiche esigenze dell'applicazione con gli strumenti automatici offerti.

Mettendo in campo più repliche di funzioni dello stesso tipo, sarebbe possibile realizzare direttamente di ridondanza all'interno dell'applicazione stessa. Ad esempio, ponendo in ascolto diverse funzioni sullo stesso topic Apache Kafka sarebbe possibile effettuare operazioni simultanee, come la scrittura di informazioni su diversi database back–end per ottenere meccanismi di *High Availability* e sfruttare eventualmente uno schema *multicloud*, purché, naturalmente, il sistema di messaggistica e la base di dati siano visibili e raggiungibili dai diversi cluster. In questo modo risulterebbe possibile migliorare ulteriormente la resilienza dell'architettura e la durabilità dei dati gestiti.

Inoltre, come mostrato per la logica di logging, sarebbe possibile sfruttare gli oggetti `Sequence` e `Parallel` forniti da Knative. Utilizzando queste due risorse è possibile implementare pipeline di funzioni complesse a piacimento, che possono includere arbitrariamente canonici `Service` Kubernetes e `Knative Service`. Questi oggetti, infatti, permettono di concatenare tra loro funzioni attraverso la generazione di eventi CloudEvents oppure elaborare tale dato in modo parallelo tra funzioni diverse, eventualmente poste in sequenza tra loro. Tali meccanismi sono disponibili attraverso una logica dichiarativa, racchiudendo la definizione di tali risorse in un file di configurazione,

che permette una più facile messa in campo di queste risorse, gestione, sviluppo e manutenzione delle differenti parti che le compongono.

Combinando opportunamente le caratteristiche illustrate è possibile, da un lato, scomporre problemi complessi in diverse componenti pressoché indipendenti tra loro, dall'altro, collegare tra loro queste componenti per realizzare meccanismi e logiche complesse.

Capitolo 7

Conclusioni finali

A livello generale, l'applicazione realizzata in questa tesi risulta essere un esempio, messo in campo con l'obiettivo di provare i diversi framework serverless messi a disposizione dalla comunità open–source. Da questo lavoro, si è cercato di estrarre le peculiarità di un'architettura a microservizi, con alcuni spunti del paradigma Function–as–a–Service e confrontare tali caratteristiche con una più classica e canonica architettura monolitica.

Prima di mettere in campo un'architettura a microservizi, è bene essere certi di riuscire ad abbattere, o quanto meno assorbire drasticamente, la quantità di risorse necessaria alla semplice installazione e messa in campo del framework di base. Prendendo in esempio Knative, questo richiede una quantità di risorse non così trascurabile, combinata ad una preliminare attività di configurazione, ed è necessario valutare se e quanto i benefici dell'architettura serverless possano compensare tali costi.

Senza dubbio, il tasto dolente di un'architettura a microservizi è la sua realizzazione pratica. Partendo da un'applicazione monolitica, infatti, è necessario innanzitutto analizzare con cura quali servizi possono essere scomposti, procedura non sempre banale e assolutamente non generalizzabile in quanto strettamente legata al contesto ed all'applicazione stessa. Inoltre. è bene tenere in considerazione che alcune parti di codice necessitano sicuramente di una riscrittura, quanto meno parziale, per poter essere adattate alla nuova architettura ed aderire alle specifiche imposte, ad esempio per utilizzare meccanismi di eventi, come quelli forniti da Knative Eventing. Tali frammenti di codice risultano quindi strettamente dipendenti dal framework utilizzato, rendendo praticamente impossibile la portabilità del codice sorgente e cioè vincolando, in modo piuttosto definitivo, la scelta del framework da utilizzare.

Il vantaggio più importante offerto da un'architettura a microservizi, come dice la parola stessa, è sicuramente rappresentato dalla divisione in moduli di un'applicazione, permettendo di ottenere componenti lascamente accoppiati tra loro. Si pensi, nel caso particolare dell'applicazione serverless realizzata, alla logica di logging, la quale risulta disaccoppiata da tutte le altre funzioni e condivisa tra i diversi endpoint, anziché essere replicata in più punti dell'applicazione. Questa viene semplicemente invocata attraverso un evento con opportuni attributi da rispettare, al quale il servizio indica interesse, che può essere generato da qualunque sorgente. D'altro canto, è possibile mettere in campo un numero arbitrario di funzioni che, se necessitano di una logica di logging, possono semplicemente generare un evento con alcuni specifici attributi, senza occuparsi di come, dove e da chi questa logica venga implementata.

Sviluppare una funzione ha di per sé un costo minimo, poiché costituisce un piccolo frammento di codice, ed è quindi possibile costruire applicazioni anche complesse componendo diversi moduli

oppure facilmente estendere applicazioni già messe in campo implementando ed integrando le sole componenti necessarie. Ciascun microservizio, inoltre, può essere messo in campo con attributi diversi per quanto riguarda le politiche di autoscaling, permettendo la massima personalizzazione in quanto a performance di un singolo endpoint, scenario non possibile durante la messa in campo di un'applicazione monolitica. Rendendo indipendenti tali moduli, risulta anche più facile il loro monitoraggio, sia in termini di performance e qualità del servizio, sia in termini di testing e tracciabilità.

Come mostrato precedentemente, in linea di massima, i servizi realizzati con tecnologia Knative, anziché con canonica tecnologia Kubernetes, risultano più scalabili ed elastici, oltre che stabili e performanti, anche se lievemente più dispendiosi in termini di risorse, soprattutto per quanto riguarda la capacità computazionale in termini di CPU utilizzata. In particolare, il `Knative Pod Autoscaler`, già nella sua configurazione standard, svolge molto bene il suo compito grazie all'utilizzo di sidecar container che elaborano metriche in modo più complesso rispetto all'`Horizontal Pod Autoscaler` offerto da Kubernetes e ne risolvono le problematiche illustrate, causate da una logica più semplice di quest'ultimo, che non sempre si rivela adatto e risulta di difficile configurazione. Per quanto riguarda il caso d'uso scelto

In conclusione, non è possibile dichiarare a livello globale e con certezza un'architettura migliore di un'altra, poiché la scelta architetturale dipende da molteplici fattori, quali il carico in memoria o capacità computazionale richiesta, la mole di traffico da smaltire e la sua distribuzione temporale. Ciò che risulta evidente è l'immenso panorama di possibilità che si aprono sfruttando un'architettura a microservizi, alcuni di questi implementati come Function–as–a–Service. Sfruttando questo nuovo paradigma, è possibile sviluppare infatti parti di codice più ridotte e snelle, cooperanti tra di loro attraverso una logica sempre più dichiarativa e meno programmatica, rendendo più facile i compiti di sviluppo e messa in campo di un'applicazione. Il paradigma è probabilmente ancora acerbo e trovare un caso d'uso reale implementabile in una logica di business risulta al momento difficile, ma la tecnologia è decisamente promettente grazie alla vastità di opzioni offerte dai diversi framework, in continuo miglioramento ed in continua espansione.

Appendice A

Codice applicazione

In questa appendice, vengono mostrati alcuni snippet di codice utilizzati per la configurazione, sviluppo o messa in campo delle applicazioni.

A.1 Kubeless

Il codice mostrato nel riquadro A.1 riporta un esempio di una funziona scritta in Python per il framework Kubeless.

Questa funzione è stata utilizzata per testare il meccanismo di pipeline delle funzioni in Kubeless. In questo esempio, viene letto il dato ricevuto ed inserito un messaggio con lo stesso contenuto su un topic Apache Kafka, attivando eventualmente un'ulteriore funzione in ascolto su tale topic.

```
from kafka import KafkaProducer
from kafka.errors import KafkaError

brokers = ["broker.kubeless:9092"]
topic = "kubeless-messages"

def handler(event, context):
    producer = KafkaProducer(bootstrap_servers = brokers)
    print("Connected? %s" % producer.bootstrap_connected())
    msg = event["data"].decode("UTF-8")
    print("Sending message \"%s\" in topic %s" % (msg, topic))

    ret_value = ""

    try:
        future = producer.send(topic, msg.encode("UTF-8"))
        record_metadata = future.get(timeout = 5)
        print("Record metadata: %s", record_metadata)
        ret_value = "Data \"%s\" correctly sent to kafka topic %s" % (msg,
            topic)
    except KafkaError as ke:
        print(ke)
        ret_value = "Error sending data \"%s\" to kafka topic %s" % (msg,
            topic)
    finally:
```

```
        producer.close()
        return ret_value
```

Codice A.1. Funzione Kubeless

Il comando `kubeless` permette di mettere in campo la funzione indicata. Tale comando funge da wrapper e si occupa di creare le opportune risorse Kubernetes.

```
$ kubeless function deploy kafka-producer --runtime python3.6 \
    --dependencies requirements.txt \
    --handler kafka-producer.handler \
    --from-file kafka-producer.py
```

La funzione posta in successione a questa nel flusso logico dell'applicazione dovrà essere messa in campo in modo tale che venga attivata quando viene pubblicato un messaggio sul topic *kubeless-messages*, lo stesso su cui scrive la funzione sopra riportata. È necessario quindi creare un oggetto `KafkaTrigger`, indicando le etichette della funzione da attivare attraverso l'opzione `function-selector` ed il topic Apache Kafka attraverso l'opzione `trigger-topic`.

```
$ kubeless trigger kafka create message-trigger --function-selector
    created-by=kubeless,function=kafka-producer --trigger-topic kubeless-
    messages
```

A.2 Applicazione monolitica

Il codice mostrato nel riquadro A.2 riporta il file YAML necessario per mettere in campo l'applicazione monolitica. Questo file definisce:

- Il `Deployment` Kubernetes, che utilizza l'immagine Docker costruita ed imposta alcune variabili d'ambiente come l'URI del database ed il quantitativo di memoria da allocare;
- Il `Service` Kubernetes, responsabile di rendere accessibile il `Deployment` all'interno del cluster Kubernetes;
- Il `VirtualService` di Istio, responsabile dell'esposizione del servizio sopra creato all'esterno del cluster attraverso il path con prefisso `library/v1/`. Questo oggetto si occupa di riscrivere alcuni attributi della richiesta originale e permette di inoltrare le richieste al `Service` `library`, sostituendo il prefisso indicato con il path `/`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: library
  namespace: tesi-faas
  labels: &labels
    app: library
spec:
  replicas: 1
  selector:
    matchLabels: *labels
  template:
    metadata:
```

```
      labels: *labels
    spec:
      containers:
      - name: library
        image: 10.39.10.33.nip.io:5000/library:cpu
        env:
        - name: MONGODB_URI
          value: mongodb.tesi-faas:27017
        - name: MEMORY
          value: "300"
---
apiVersion: v1
kind: Service
metadata:
  name: library
  namespace: tesi-faas
  labels: &labels
    app: library
spec:
  selector: *labels
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: external-library
  namespace: tesi-faas
spec:
  gateways:
  - knative-serving/knative-ingress-gateway
  hosts:
  - 10.39.10.33
  http:
  - name: library
    match:
    - uri:
        prefix: /library/v1/
    rewrite:
      uri: /
    route:
    - destination:
        host: library.tesi-faas.svc.cluster.local
        port:
          number: 80
```

Codice A.2. File YAML per la messa in campo dell'applicazione monolitica

A.3 Applicazione serverless

L'applicazione serverless costituisce la reale particolarità di questo lavoro ed è costituita da diverse parti, messe in campo in modo preciso per permetterne il collegamento e la corretta configurazione.

A.3.1 Servizio di logging

Il codice mostrato nel riquadro A.3 riporta il file YAML necessario per mettere in campo il servizio di logging, comune a tutti gli endpoint dell'applicazione serverless. Questo file definisce:

- Il `Deployment` Kubernetes, che utilizza l'immagine Docker costruita;
- Il `Service` Kubernetes, responsabile di rendere accessibile il `Deployment` all'interno del cluster Kubernetes;
- Il `PersistentVolumeClaim`, responsabile di associare il volume richiesto dal `Deployment logger` al successivo `Persistent Volume`;
- Il `Persistent Volume`, responsabile di dichiarare il tipo di volume, in questo caso `local`, cioè una directory del sistema, condivisa attraverso il protocollo NFS[1].

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: logger
  namespace: tesi-faas
  labels: &labels
    app: logger
spec:
  replicas: 1
  selector:
    matchLabels: *labels
  template:
    metadata:
      labels: *labels
    spec:
      containers:
      - name: logger
        image: 10.39.10.33.nip.io:5000/logger
        volumeMounts:
          - name: logger-storage
            mountPath: /data/library
      volumes:
      - name: logger-storage
        persistentVolumeClaim:
          claimName: logger-storage
---
apiVersion: v1
kind: Service
metadata:
```

[1] Network File System.

```yaml
  name: logger
  namespace: tesi-faas
  labels: &labels
    app: logger
spec:
  selector: *labels
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: logger-storage
  namespace: tesi-faas
  labels:
    app: logger
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: logger-storage
  namespace: tesi-faas
  labels:
    type: local
spec:
  capacity:
    storage: 1Gi
  accessModes:
  - ReadWriteOnce
  nfs:
    path: /mnt/logger-storage
    server: 10.39.10.33
```

Codice A.3. File YAML per la messa in campo del `Deployment` responsabile del logging nell'applicazione serverless

A.3.2 Servizio Knative standard

Il codice mostrato nel riquadro A.4 riporta il file YAML necessario per mettere in campo il servizio `get-book`, responsabile della gestione dell'endpoint `GET /books/{isbn}` nell'applicazione serverless. Questo file definisce:

- Il `Knative Service`, che utilizza l'immagine Docker costruita ed imposta alcune variabili d'ambiente come l'URI del database ed il quantitativo di memoria da allocare;

- Il `VirtualService` di Istio, responsabile dell'esposizione del servizio sopra creato all'esterno del cluster attraverso il path con prefisso `library/v2/books/`. Questo oggetto si occupa di riscrivere alcuni attributi della richiesta originale e permette di inoltrare le richieste al `Knative Service` sopra indicato, sostituendo il prefisso indicato con il path `/books/`.

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  labels:
    serving.knative.dev/visibility: cluster-local
  name: get-book
  namespace: tesi-faas
spec:
  template:
    spec:
      containers:
      - name: get-book
        image: 10.39.10.33.nip.io:5000/get-book:cpu
        env:
        - name: MONGODB_URI
          value: mongodb.tesi-faas:27017
        - name: MEMORY
          value: "30"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: external-get-book
  namespace: tesi-faas
spec:
  gateways:
  - knative-serving/knative-ingress-gateway
  - knative-serving/cluster-local-gateway
  hosts:
  - 10.39.10.33
  http:
  - name: get-book
    match:
    - uri:
        prefix: /library/v2/books/
    rewrite:
      authority: get-book.tesi-faas.svc.cluster.local
      uri: /books/
    route:
    - destination:
        host: cluster-local-gateway.istio-system.svc.cluster.local
        port:
          number: 80
```

Codice A.4. File YAML per la messa in campo del `Knative Service get-book` nell'applicazione serverless

A.3.3 Servizio Knative in ascolto su topic Apache Kafka

Il codice mostrato nel riquadro A.5 riporta il file YAML necessario per mettere in campo il servizio `add-customer`, responsabile della gestione dell'endpoint `POST /customers` nell'applicazione serverless, realizzato come funzione attivata con messaggi su topic Apache Kafka. Questo file definisce:

- Il `Knative Service`, che utilizza l'immagine Docker costruita ed imposta alcune variabili d'ambiente come l'URI del database ed il quantitativo di memoria da allocare;
- La `Sequence`, responsabile del collegamento logico, attraverso il meccanismo ad eventi, tra il `Knative Service add-customer` ed il `Service logger` mostrato precedentemente;
- La `KafkaSource`, responsabile di intercettare i messaggi accodati su un determinato topic Apache Kafka, dei quali vengono rispettivamente indicati il nome e l'indirizzo. Questi messaggi vengono direzionati alla `Sequence` sopra indicata, attivando la catena di funzioni.

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  labels:
    serving.knative.dev/visibility: cluster-local
  name: add-customer
  namespace: tesi-faas
spec:
  template:
    spec:
      containers:
      - name: add-customer
        image: 10.39.10.33.nip.io:5000/add-customer:cpu
        env:
        - name: MONGODB_URI
          value: mongodb.tesi-faas:27017
        - name: MEMORY
          value: "50"
---
apiVersion: flows.knative.dev/v1alpha1
kind: Sequence
metadata:
  name: add-customer
  namespace: tesi-faas
spec:
  channelTemplate:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: KafkaChannel
    spec:
      numPartitions: 3
      replicationFactor: 1
  steps:
    - ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: add-customer
  reply:
```

```
    ref:
      apiVersion: v1
      kind: Service
      name: logger
---
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: KafkaSource
metadata:
  name: add-customer
  namespace: tesi-faas
spec:
  bootstrapServers: kafka-kafka-bootstrap.kafka:9092
  consumerGroup: knative-group
  topics: customers
  sink:
    ref:
      apiVersion: flows.knative.dev/v1alpha1
      kind: Sequence
      name: add-customer
```

Codice A.5. File YAML per la messa in campo del `Knative Service add-customer` nell'applicazione serverless

A.3.4 Generazione di eventi in GoLang

Il codice mostrato nel riquadro A.6 riporta i frammenti caratteristici dell'implementazione di un `Knative Service`, scritto in Go, insieme con la generazione di eventi in Knative.

Dopo aver importato le librerie fondamentali, è necessario indicare la funzione responsabile di gestire le richieste. Tale funzione deve rispettare un certo prototipo nel quale viene indicato il contesto dell'evento, l'evento stesso ed un ulteriore parametro necessario per impostare correttamente la risposta. Tale funzione crea quindi un nuovo evento, impostandone opportunamente gli attributi ed il contenuto, che viene inserito nella risposta e quindi inviato nuovamente nel meccanismo di eventi di Knative.

```
package main

import (
    "context"
    "fmt"
    "log"
    "net/http"

    cloudevents "github.com/cloudevents/sdk-go"
    "github.com/google/uuid"
)

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

```
func AddCustomer(ctx context.Context, event cloudevents.Event, response *
    cloudevents.EventResponse) error {
    // ...

    message := Message{Message: fmt.Sprintf("Added customer %s", string(s)
        )}
    newEvent := cloudevents.NewEvent()
    newEvent.SetID(uuid.New().String())
    newEvent.SetSource("reply/tesi-faas/library/add-customer")
    newEvent.SetType("dev.reply.tesi-faas.library.add-customer")
    newEvent.SetData(message)
    response.RespondWith(http.StatusAccepted, &newEvent)

    log.Printf("Generated event: %+v", newEvent)

    return nil
}
```

Codice A.6. Generazione evento CloudEvents in GoLang

Bibliografia

[1] Seyyed Mohsen Hashemi e Amid Khatibi Bardsiri. «Cloud computing vs Grid computing». In: (ott. 2012).

[2] Y. Kim e G. Cha. «Design of the Cost Effective Execution Worker Scheduling Algorithm for FaaS Platform Using Two-Step Allocation and Dynamic Scaling». In: (nov. 2018), pp. 131–134. ISSN: null. DOI: 10.1109/SC2.2018.00027.

[3] A. Palade, A. Kazmi e S. Clarke. «An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge». In: 2642-939X (lug. 2019), pp. 206–211. ISSN: 2378-3818. DOI: 10.1109/SERVICES.2019.00057.

[4] Scott Hendrickson et al. «Serverless Computation with OpenLambda». In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, giu. 2016. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

[5] Docker Inc. *Docker*. 2020. URL: https://www.docker.com/.

[6] Dirk Merkel. «Docker: lightweight linux containers for consistent development and deployment». In: *Linux journal* 2014.239 (2014).

[7] Docker Inc. *Docker Hub*. 2020. URL: https://hub.docker.com/.

[8] The Kubernetes Authors. *Kubernetes*. 2020. URL: https://kubernetes.io/.

[9] The etcd authors. *etcd*. 2020. URL: https://etcd.io/.

[10] YAML. *YAML*. 2011. URL: https://yaml.org/.

[11] WeaveWorks. *Introducing Weave Net*. 2020. URL: https://www.weave.works/.

[12] Kubernetes Authors. *Ingress Controllers*. 2020. URL: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/.

[13] Kubernetes Authors. *Ingress*. 2020. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/.

[14] Bitnami Project. *Kubeless*. 2019. URL: https://kubeless.io/.

[15] Bitnami Project. *Bitnami*. 2020. URL: https://bitnami.com/.

[16] The Knative Authors. *Welcome, Knative*. 2020. URL: https://knative.dev/.

[17] Datawire.io. *Ambassador*. 2020. URL: https://www.getambassador.io/.

[18] Contour Authors. *Contour*. 2020. URL: https://projectcontour.io/.

[19] Inc Solo.io. *What is Gloo*. 2020. URL: https://docs.solo.io/gloo/latest.

[20] Istio Authors. *Istio*. 2019. URL: https://istio.io/.

[21] Envoy Project Authors. *Envoy*. 2020. URL: https://www.envoyproxy.io/.

[22] CloudEvents Authors 2020. *CloudEvents*. 2020. URL: https://cloudevents.io/.

[23] Apache Software Foundation. *Introduction – Apache Kafka is a distributed streaming platform. What exactly does that mean?* 2017. URL: https://kafka.apache.org/intro.

[24] Prometheus Authors. *Prometheus*. 2020. URL: https://prometheus.io/.

[25] Grafana Labs. *Grafana*. 2020. URL: https://grafana.com/.

[26] Apache Software Foundation. *Apache JMeter*. 2019. URL: https://jmeter.apache.org/.

[27] *Go*. 2020. URL: https://golang.org/.

[28] MongoDB Inc. *MongoDB*. 2020. URL: https://www.mongodb.com/.

[29] TheKubernetes Authors. *Managing Resources for Containers*. 2020. URL: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu.