Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

ANALISI COMPARATIVA DI SOLUZIONI SERVERLESS

Relatore:
Chiar.mo Prof.
Rossi Davide

Presentata da:
De Rosa Davide

II Sessione
Anno Accademico 2023/2024

Serverless

Cloud Computing

Google Cloud Functions

AWS Lambda

FaaS

Abstract

La crescente diffusione del **Serverless Computing** ha rivoluzionato il modo in cui vengono sviluppate e distribuite le applicazioni Cloud, offrendo scalabilità automatica e gestione trasparente delle risorse. Questa tesi si propone di analizzare due delle principali piattaforme serverless attualmente disponibili: *AWS Lambda* e *Google Cloud Functions*, confrontandole in termini di performance, costi e facilità di deploy.

Attraverso un caso studio sono state implementate API utilizzando due approcci architetturali differenti (*funzione unica per tutte le API* e *funzione per ogni chiamata API*). Le API sono state successivamente integrate con i database serverless *Amazon DynamoDB* e *Google Firestore*.

I risultati hanno dimostrato che *AWS Lambda* offre tempi di risposta inferiori, specialmente durante il *cold start*, e un costo per richiesta più vantaggioso. Tuttavia, *Google Cloud Functions* si distingue per la sua semplicità di configurazione, riducendo il numero di passaggi necessari per il deploy.

La tesi conclude che, sebbene *AWS Lambda* sia la scelta preferibile per progetti ad alte prestazioni e su larga scala, *Google Cloud Functions* rappresenta un'opzione decisamente valida per piccoli team o progetti che necessitano di una configurazione rapida e intuitiva.

Gli sviluppi futuri potrebbero includere l'analisi di altre piattaforme e l'applicazione del modello serverless in scenari più complessi, quali il *machine learning* e l'*intelligenza artificiale*.

Indice

Capitolo 1

Introduzione

Negli ultimi anni, il **Cloud Computing** ha subito un'enorme evoluzione, e una delle sue innovazioni più importanti è l'introduzione del modello **Serverless**. Questa tecnologia promette di rendere la gestione dell'infrastruttura più semplice ed economica, permettendo agli sviluppatori di concentrarsi esclusivamente sulla logica applicativa senza preoccuparsi delle risorse fisiche o virtuali sottostanti. I principali fornitori di servizi Cloud, come **Amazon Web Services** (**AWS**) e **Google Cloud** (**GCP**), hanno sviluppato soluzioni serverless ampiamente utilizzate nel settore accademico ed industriale, portando ad una rapida diffusione di questo modello.

1.1 Scopo della Tesi

L'obiettivo principale di questa tesi è quello di condurre un'*analisi comparativa* tra le due principali piattaforme per l'esecuzione di *funzioni serverless*: **AWS Lambda** e **Google Cloud Functions**.

La ricerca si concentrerà sul confronto tra le *performance*, i *costi* e la *facilità d'uso* delle due soluzioni, esaminando specificamente la loro integrazione con i *database serverless* e l'*implementazione delle API*.

Attraverso un caso studio, verranno evidenziati i punti di forza e di debolezza di ciascuna piattaforma, fornendo un quadro completo delle loro caratteristiche.

1.2 Metodologia

L'analisi è stata condotta seguendo un approccio sperimentale, basato su *test* e *misurazioni delle prestazioni* in contesti pratici. In particolare, sono stati esaminati

due approcci architetturali: la *funzione unica per tutte le API* e la *funzione dedicata per ogni chiamata API*. La sperimentazione ha incluso anche l'integrazione di *AWS Lambda* con **DynamoDB** e di *Google Cloud* con **Firestore**, due soluzioni di database serverless ampiamente utilizzate. I risultati sono stati valutati in termini di **latenza**, **costi** e **facilità di deploy**.

1.3 Struttura della Tesi

La tesi è suddivisa in *otto capitoli*.

Nel *Capitolo 2* verranno introdotte le nozioni di base del *serverless computing*, con un focus sui suoi principi fondamentali e sulle differenze rispetto ai modelli tradizionali.

Il *Capitolo 3* fornirà una panoramica dettagliata di *AWS Lambda* e *Google Cloud Functions*, spiegando il processo di deploy su entrambe le piattaforme.

Il *Capitolo 4* esplorerà l'integrazione con i *database serverless*, mentre nel *Capitolo 5* verranno analizzate le principali architetture per la gestione delle API.

Il *Capitolo 6* sarà dedicato al *caso studio*, seguito da una *discussione dei risultati* nel *Capitolo 7*, per poi concludere con le *riflessioni finali* nel *Capitolo 8*.

Capitolo 2

Nozioni di base su Serverless

2.1 Definizione e Concetti Fondamentali

Il *Serverless Computing* è una tecnologia in rapida crescita che sta avendo un impatto sempre più significativo sulla società, trovando ampia adozione sia nel mondo accademico che in quello industriale. La sua promessa principale è rendere i servizi informatici più accessibili, personalizzabili in base alle esigenze specifiche e a basso costo, delegando all'infrastruttura la gestione dei problemi operativi.

I principali fornitori di servizi cloud, come *Amazon*, *Microsoft*, *Google* e *IBM*, offrono piattaforme serverless già pronte all'uso, con ben definite responsabilità e prezzi.

Un sistema può essere considerato serverless se presenta le seguenti caratteristiche[1]:

- **Auto-scaling**. La capacità di adattarsi automaticamente alle variazioni del carico di lavoro, scalando sia orizzontalmente che verticalmente, è un elemento chiave. Un'applicazione serverless può ridurre il numero di istanze fino a zero, introducendo il concetto di *cold startup*, che può causare ritardi nel tempo di risposta dovuti alla necessità di avviare l'ambiente da zero e caricare il codice.
- **Pianificazione flessibile**. Non essendo vincolata a un server specifico, l'applicazione viene pianificata dinamicamente in base alle risorse disponibili nel cluster, garantendo bilanciamento del carico e prestazioni ottimali. Inoltre, la pianificazione tiene conto di più regioni geografiche per evitare interruzioni del servizio in caso di malfunzionamenti o crash.
- **Event-driven**. Le applicazioni serverless si attivano in risposta a eventi, come richieste HTTP, aggiornamenti di code di messaggi o nuove scritture su servizi di storage. Tramite l'associazione di trigger e regole agli eventi, il sistema

è in grado di rispondere in modo efficiente e flessibile alle diverse tipologie di input. Gli eventi possono essere attivati non solo da fonti esterne alla piattaforma cloud, ma anche internamente, attraverso i vari servizi offerti dalla piattaforma stessa. Questo permette agli sviluppatori di creare applicazioni distribuite che utilizzano diversi servizi cloud in modo integrato. Il serverless computing rappresenta una parziale realizzazione di un modello basato sugli eventi, dove le applicazioni vengono definite dalle azioni e dagli eventi che le attivano. Questo concetto richiama i sistemi di database attivi e riflette la letteratura sui sistemi event-driven, che da tempo teorizza l'esistenza di sistemi informatici generali in cui le azioni sono elaborate in modo reattivo ai flussi di eventi.

Le piattaforme serverless basate su funzioni adottano pienamente questa visione, utilizzando astrazioni semplici come le funzioni per definire le azioni e costruendo la logica di elaborazione degli eventi direttamente all'interno del cloud. In questo modo, il serverless computing offre un framework flessibile per la gestione e l'elaborazione degli eventi su larga scala.

- **Sviluppo trasparente**. Gli sviluppatori non devono più preoccuparsi della gestione delle risorse fisiche o dell'ambiente di esecuzione, poiché queste responsabilità sono delegate ai provider cloud. Questi ultimi si occupano di garantire la disponibilità delle risorse fisiche, la sicurezza e la potenza di calcolo, rendendo tutto ciò trasparente agli sviluppatori, facilitando così il processo di sviluppo e distribuzione.
- **Pagamento in base al consumo**. Il serverless trasforma il costo della capacità di calcolo da una spesa di capitale a una spesa operativa, eliminando la necessità per gli utenti di acquistare server dedicati per i picchi di carico. Il modello *pay-as-you-go* permette di pagare solo per le risorse effettivamente utilizzate.

Un modello di calcolo che soddisfa queste cinque caratteristiche è considerato serverless. La sua crescente diffusione è dovuta in parte al fatto che gli sviluppatori possono pagare solo in base all'uso effettivo delle risorse, piuttosto che per una capacità preallocata.

Oggi, il serverless computing viene utilizzato principalmente in scenari backend per lavori batch, come l'analisi dei dati, attività di machine learning e applicazioni web basate su eventi. [2]

2.2 Funzioni Serverless

La modalità tradizionale di distribuzione nell'ambito dell'*Infrastructure-as-a-Service* (*IaaS*) richiede che un server sia attivo a lungo termine per garantire servizi continui. Tuttavia, questa allocazione esclusiva implica che le risorse vengano mantenute anche quando l'applicazione non è in esecuzione. Di conseguenza, l'utilizzo delle risorse nei data center è generalmente basso, attestandosi in media intorno al 10%, specialmente per i servizi online con un uso prevalentemente diurno. Questa inefficienza ha portato allo sviluppo di un modello di servizio on-demand gestito dalla piattaforma, che ha l'obiettivo di migliorare l'utilizzo delle risorse e ridurre i costi del cloud computing.

Al momento non esiste una definizione ufficiale di serverless computing. Tuttavia, le definizioni comunemente accettate, come quelle proposte dal Berkeley View, lo descrivono come segue:

- *Serverless Computing = FaaS (Function-as-a-Service) + BaaS (Backend-as-a-Service).* Esiste un malinteso comune secondo cui il termine *serverless* può essere usato in modo intercambiabile con *FaaS*. In realtà, entrambi sono elementi fondamentali per il serverless computing. Il modello FaaS permette l'isolamento e l'invocazione delle singole funzioni, mentre il modello BaaS fornisce il supporto backend necessario per i servizi online.
- Nel modello *FaaS* - noto anche come paradigma *Lambda* - un'applicazione viene scomposta in funzioni o microservizi a livello di funzione. Gli aspetti principali che caratterizzano una funzione includono l'identificatore della funzione, il runtime del linguaggio, il limite di memoria per ciascuna istanza e l'URI (Uniform Resource Identifier) che definisce il codice della funzione.
- *BaaS* rappresenta un insieme di servizi essenziali su cui si basano le applicazioni. Alcuni esempi includono lo storage, i servizi di notifica dei messaggi e gli strumenti per il DevOps.

In sintesi, serverless computing combina sia il modello FaaS che quello BaaS, fornendo una struttura versatile per lo sviluppo e l'esecuzione di applicazioni senza la necessità di gestire direttamente l'infrastruttura sottostante.

Le funzioni cloud rappresentano quindi il fondamento del serverless computing e promuovono un modello di programmazione più semplice e versatile per il cloud. Grazie alla loro capacità di essere eseguite in risposta a eventi e richieste, le funzioni cloud permettono agli sviluppatori di concentrarsi sulla logica applicativa senza preoccuparsi della gestione dell'infrastruttura sottostante. Questo approccio semplificato consente di sviluppare, distribuire e scalare applicazioni in modo più effi-

ciente, aprendo la strada a un paradigma di programmazione cloud più flessibile e accessibile.

Per comprendere al meglio il modello di elaborazione serverless, si consideri un esempio basato su un'invocazione asincrona di una funzione serverless. In questo scenario, il sistema serverless riceve le chiamate API e le invocazioni vengono gestite attraverso un sistema di notifica. Il sistema serverless processa le richieste API inviate dagli utenti, le valida e avvia le funzioni, creando nuove sandbox per le invocazioni (*cold startup*) oppure riutilizzando sandbox già attive (*warm startup*). Ogni invocazione di funzione viene eseguita in isolamento, all'interno di un container individuale o di una macchina virtuale, che viene assegnata da un controllore di accesso per garantire la sicurezza e l'isolamento tra le invocazioni.

Grazie alla natura *event-driven* e all'elaborazione basata su singoli eventi, il sistema serverless può essere attivato su richiesta, creando istanze isolate in risposta agli eventi e scalando orizzontalmente in base al carico effettivo dell'applicazione. Successivamente, ogni worker che esegue le funzioni accede a un database di backend per memorizzare i risultati dell'elaborazione. Gli utenti possono inoltre personalizzare l'esecuzione di applicazioni complesse configurando trigger aggiuntivi e definendo interazioni tra eventi, costruendo pipeline di eventi interni per gestire flussi di lavoro articolati.[2]

2.3 Serverless contro approccio tradizionale

In una piattaforma serverless, l'utente si limita a scrivere una funzione cloud in un *linguaggio di alto livello* e a specificare l'evento che ne deve innescare l'esecuzione, ad esempio il caricamento di un'immagine nello storage cloud o l'inserimento di una miniatura in una tabella del database. Il sistema serverless si occupa poi di gestire tutto il resto, inclusi la selezione dell'istanza, la scalabilità, la distribuzione, la tolleranza ai guasti, il monitoraggio, la registrazione e l'applicazione di patch di sicurezza.

L'approccio tradizionale - che può essere definito anche *serverfull computing* - può essere visto come la programmazione in un linguaggio assembly di basso livello, mentre il serverless computing ricorda la programmazione in un linguaggio di alto livello, come Python. Un programmatore che utilizza un linguaggio assembly per calcolare un'espressione semplice come $c = a + b$ deve scegliere i registri, caricare i valori nei registri, eseguire l'operazione aritmetica e infine memorizzare il risultato. Questo processo riflette molte delle fasi del serverful computing nel cloud: prima si allocano o identificano le risorse, poi si caricano con il codice e i dati necessari, si eseguono i calcoli, si memorizzano i risultati e infine si rilasciano le risorse. L'o-

biettivo del serverless computing è quello di semplificare questo processo, offrendo ai programmatori cloud vantaggi simili a quelli della programmazione in linguaggi di alto livello.

Altre caratteristiche degli ambienti di programmazione avanzati trovano un parallelo naturale nel serverless computing. Ad esempio, la gestione automatica della memoria evita ai programmatori di gestire le risorse di memoria; allo stesso modo, il serverless computing libera i programmatori dal dover gestire direttamente le risorse del server.

In particolare, esistono tre differenze fondamentali tra il serverless computing e quello serverful:

- **Calcolo e storage disaccoppiati**. Nel serverless computing, lo storage e la computazione scalano in modo indipendente e vengono forniti e tariffati separatamente. In genere, lo storage è gestito tramite un servizio cloud dedicato, mentre la computazione avviene in modo stateless.
- **Esecuzione del codice senza gestione delle risorse**. L'utente non deve più preoccuparsi di allocare risorse. Basta fornire il codice, e il cloud si occupa automaticamente di assegnare le risorse necessarie per l'esecuzione.
- **Pagare per le risorse effettivamente utilizzate**. La fatturazione avviene in base alle risorse effettivamente consumate, come il tempo di esecuzione, anziché alle risorse preallocate, come la dimensione e il numero di macchine virtuali.

Usando queste differenze, si può spiegare come l'approccio serverless si distingue da soluzioni simili, presenti e passate.

L'attuale serverless computing con funzioni cloud si differenzia dai suoi predecessori per diversi aspetti essenziali: migliore autoscaling, forte isolamento, flessibilità della piattaforma e supporto dell'ecosistema dei servizi. Tra questi fattori, l'autoscaling offerto da *AWS Lambda* ha segnato un netto distacco da quanto fatto in precedenza. Il carico di lavoro viene gestito con una fedeltà maggiore rispetto alle tecniche di autoscaling precedentemente utilizzate, offrendo una scalabilità più rapida quando necessario e portando a zero le risorse ed i costi in assenza di domanda. La tariffazione è molto più precisa, con un incremento minimo di fatturazione di *100 ms* in un periodo in cui altri servizi di autoscaling prevedono la tariffazione oraria.

In particolare, il cliente viene addebitato per il tempo in cui il codice *viene effettivamente eseguito*, non per le risorse riservate all'esecuzione del programma. Questa distinzione garantisce che il cloud provider sia "coinvolto nel gioco" dell'autoscaling e di conseguenza fornisce incentivi per garantire un'allocazione efficiente delle risorse.

Per i cloud provider, il serverless computing favorisce la crescita del business, in quanto rendere il cloud più facile da programmare aiuta ad attirare nuovi clienti e a far sì che i clienti esistenti utilizzino maggiormente le offerte cloud.

Il breve tempo di esecuzione, il ridotto utilizzo della memoria e la natura stateless migliorano il multiplexing statistico, facilitando ai fornitori di servizi cloud la ricerca di risorse inutilizzate per eseguire questi compiti. I fornitori di cloud possono anche sfruttare hardware meno recente, come server più vecchi che potrebbero risultare meno interessanti per i clienti dei servizi di cloud tradizionali. Questi vantaggi contribuiscono ad aumentare il reddito derivante dalle risorse esistenti.

I clienti, dal canto loro, traggono vantaggio da una maggiore produttività nella programmazione e, in molti casi, possono anche ottenere risparmi sui costi grazie al miglior utilizzo delle risorse sottostanti. Anche se il serverless computing consente una maggiore efficienza l'uso del cloud potrebbe aumentare piuttosto che diminuire, poiché l'efficienza superiore stimola una maggiore domanda e l'ingresso di nuovi utenti.

Inoltre, il serverless computing eleva il livello di astrazione del cloud dal codice macchina *x86* (che rappresenta il 99% dei computer cloud) ai linguaggi di programmazione di alto livello, permettendo così innovazioni sull'architettura. Se architetture come *ARM* o *RISC-V* offrono migliori prestazioni in termini di costi rispetto a *x86*, il serverless computing facilita il passaggio a nuovi set di istruzioni. I fornitori di cloud potrebbero persino adottare ricerche su ottimizzazioni basate sui linguaggi e su architetture specifiche per domini, al fine di accelerare l'esecuzione di programmi scritti in linguaggi come Python.[3]

Capitolo 3

Introduzione ad AWS Lambda e Google Cloud Functions

Le grandi aziende tecnologiche come *Amazon*, *Google* e *Microsoft* offrono piattaforme serverless sotto diversi marchi. Sebbene i dettagli dei servizi possano variare, il principio fondamentale è lo stesso: con il modello di calcolo a consumo, il serverless computing mira a garantire l'autoscaling e a offrire servizi di calcolo a costi contenuti.

Sono state introdotte diverse implementazioni commerciali di successo. Amazon ha lanciato *Lambda* nel 2014, seguita da *Google Cloud Functions*, *Microsoft Azure Functions* e *IBM OpenWhisk* nel 2016.[4]

Tutte queste infrastrutture seguono il paradigma della *programmazione funzionale*: una funzione rappresenta un'unità di software che può essere distribuita sull'infrastruttura cloud del provider ed eseguire un'unica operazione in risposta a un evento esterno. Le funzioni possono essere attivate da diversi tipi di eventi, come[5]:

- un evento generato dall'infrastruttura cloud, ad esempio una modifica in un database cloud, il caricamento di un file in un object store, l'inserimento di un nuovo elemento in un sistema di messaggistica o un'azione programmata a un orario specifico;
- una richiesta diretta da parte dell'applicazione tramite HTTP o chiamate API del cloud.

3.1 AWS Lambda

3.1.1 Panoramica di AWS Lambda

Uno dei primi servizi di serverless computing è *AWS Lambda*, che permette di eseguire funzioni stateless scritte in uno dei linguaggi di programmazione supportati (come Node.js, Java, C# e Python) in risposta a eventi, su larga scala, con la possibilità di gestire fino a *3000 invocazioni in parallelo*.

Diversamente dai tradizionali servizi Cloud *IaaS*, AWS Lambda elimina la necessità per gli utenti di gestire direttamente i server, offrendo un'elasticità automatica gestita dalla piattaforma. Le funzioni Lambda sono progettate per essere *stateless*, ovvero non dipendono dall'infrastruttura sottostante. Il notevole livello di parallelismo supportato è una delle caratteristiche distintive di AWS Lambda.[6]

3.1.2 Deploy su AWS Lambda

Una volta creato e configurato il proprio account *AWS* (*Amazon Web Services*), si può accedere alla propria *console*, il centro di controllo di tutti i servizi messi a disposizione dalla piattaforma.

Cercando nei servizi offerti il servizio *Lambda* si accede alla sua console. Qui è possibile creare nuove funzioni o selezionarne una creata in precedenza.

Il processo di creazione è piuttosto intuitivo. Viene offerta la possibilità di utilizzare un piano (per casi comuni preconfigurati) o di fornire un'immagine di un container per distribuire la funzione. Nel caso base, invece, è possibile configurare il *nome della funzione*, il *runtime* (sono disponibili diversi runtime per i linguaggi Python, NodeJS, Java, Ruby e .NET) e l'*architettura* del set di istruzioni desiderata (x86_64 o arm64).

Vengono richieste inoltre le *autorizzazioni* della funzione. In caso di semplici funzioni non sarà necessario aggiungere ulteriori autorizzazioni oltre a quelle predefinite. Nel nostro *caso studio* utilizzeremo un ruolo personalizzato per consentire alla funzione di accedere ad un database serverless. Si parlerà quindi successivamente della creazione di ruoli per aggiungere autorizzazioni alla funzione.

Creata la funzione, si può accedere alla sua pagina dedicata. Oltre ad una panoramica della funzione, la quale mostra se sono collegati *trigger* o *destinazioni* alla nostra funzione, è possibile modificare il *codice della funzione* attraverso un comodo editor di testo. E' possibile eseguire *test*, monitorare lo stato tramite *CloudWatch*, gestire *alias* e *versioni*, e modificare le diverse configurazioni della funzione.

Per esporre la funzione tramite una semplice richiesta HTTP, bisogna aggiungere un *trigger* alla nostra *Lambda Function*. E' possibile utilizzare il servizio *API Gateway* offerto da AWS. Accedendo al servizio è possibile, come per *Lambda*, creare nuove API o selezionarne una creata in precedenza.

Il processo di creazione richiede inizialmente di scegliere il tipo di API. I tipi disponibili sono *API HTTP*, *API WebSocket*, *API REST* ed *API REST privata*. Nel nostro caso, la prima è più che sufficiente. Selezionato il tipo, verrà richiesto il *nome dell'API* e l'*integrazione* (servizi di back-end con cui comunicherà l'API). L'integrazione da scegliere corrisponde alla *Lambda Function* creata in precedenza.

Creata l'API, sarà possibile configurare i suoi diversi *instradamenti*, scegliendo il *metodo HTTP* ed il *nome del percorso*. Per completare l'integrazione tra la funzione ed il suo instradamento, bisogna collegare ogni singolo instradamento alla funzione, selezionando l'integrazione nell'apposita schermata.

Completati questi step, la nostra *Lambda Function* sarà accessibile dall'esterno tramite un *endpoint API*.

3.2 Google Cloud Functions

3.2.1 Panoramica di Google Cloud Functions

Google ha lanciato *Google Cloud Functions* in modo piuttosto discreto nel febbraio 2016. Pensata principalmente per i servizi di *Google Cloud*, Google evidenzia diversi casi d'uso specifici per Google Cloud Functions, come *backend per applicazioni mobili*, sviluppo di *API* e *microservizi*, *elaborazione dati*, *webhook* (per rispondere a *trigger* di terze parti) e *applicazioni IoT*.[7]

Come per le *Lambda Function*, viene eliminata la necessità per gli utenti di gestire direttamente i server, delegando la gestione dell'infrastruttura sottostante alla piattaforma.

3.2.2 Deploy su Google Cloud Functions

Una volta creato e configurato il proprio account *GCP* (*Google Cloud Platform*), si dovrà creare o selezionare un *progetto*, all'interno del quale verranno gestiti tutti i servizi utilizzati.

Selezionato il progetto, si può procedere con la ricerca dei servizi offerti dalla piattaforma attraverso la console di GCP che, simile ad AWS, funge da centro di controllo di tutti i servizi disponibili.

Cercando nei servizi offerti il servizio *Funzioni Cloud Run* si accede alla console dedicata. E' possibile quindi creare nuove funzioni o selezionarne una creata in precedenza.

Per creare una nuova funzione, si deve selezionare l'*ambiente* (la tipologia di *cloud run function*), il *nome della funzione*, la *regione geografica* ed il *tipo di trigger* (sono disponibili *HTTPS*, *Cloud Pub/Sub*, *Cloud Storage*, *Cloud Firestore*, *altri trigger*). Selezionato il trigger, viene richiesto il *tipo di autenticazione* alla funzione, potendo scegliere tra *Consenti chiamate non autenticate* e *Autenticazione necessaria*.
E' possibile inoltre configurare aspetti più tecnici, come impostazioni di *runtime*, *build*, *connessioni* e *repository per sicurezza e immagini*.

Configurato il tutto, un *editor di codice* permette di scegliere il *linguaggio di runtime* (sono disponibili diversi runtime per i linguaggi Python, NodeJS, Java, Ruby, .NET, Go e PHP) e di scegliere l'*entry point* della cloud function (il nome della funzione da eseguire). Qui sarà possibile scrivere o incollare il codice della funzione, modificabile anche dopo la creazione.

Creata la funzione, si può accedere alla sua pagina dedicata. Vengono quindi mostrare le *metriche* dettagliate della funzione, i *dettagli*, il *codice* (modificabile in qualsiasi momento), le diverse *variabili* utili per *runtime* e *build*, i *trigger* (come sarà possibile osservare a breve non c'è bisogno di alcun tipo di configurazione extra, a differenza di AWS), le *autorizzazioni*, i *log* ed i *test*.

Cloud Run, a differenza di *Lambda*, non necessità di ulteriori configurazioni per esporre API. Viene fornito direttamente un *URL* dedicato alla funzione appena creata. Tutta la configurazione dell'instradamento dovrà essere effettuata all'interno del codice.

Capitolo 4

Integrazione con Database Serverless

4.1 Introduzione ai Database Serverless

Le limitazioni nell'utilizzo dei database nelle applicazioni basate su serverless computing richiedono un approccio del tutto innovativo alla progettazione dei database. Con il passaggio dell'architettura dal modello *monolitico* ai *microservizi*, e ora a un insieme di funzioni autonome, anche i database devono seguire gli stessi principi: *assenza di manutenzione*, *pagamento basato solo sull'uso effettivo* e *distribuzione globale*. Questo porta all'adozione dei *database serverless*.[8]

Alcune delle soluzioni più popolari al momento sono *DynamoDB* (*AWS*) e *Firestore* (*GCP*).

4.2 Amazon DynamoDB

4.2.1 Panoramica su DynamoDB

DynamoDB è un servizio di *database cloud NoSQL* progettato per offrire prestazioni rapide e costanti a qualsiasi scala. Questo servizio, parte integrante dell'infrastruttura AWS, supporta centinaia di migliaia di clienti utilizzando una vasta rete di server distribuiti globalmente.

Gli utenti apprezzano DynamoDB per la sua capacità di rispondere alle richieste con una latenza costantemente bassa, con l'obiettivo di mantenere i tempi di risposta nell'ordine di pochi millisecondi.

DynamoDB si distingue per l'integrazione di alcune caratteristiche fondamentali, pensate per semplificare la vita di clienti e sviluppatori:

- **Servizio completamente gestito**. Grazie all'API di DynamoDB, le applicazioni possono creare tabelle, leggere e scrivere dati senza doversi preoccupare della gestione fisica o del mantenimento del database. DynamoDB si occupa di tutto: provisioning delle risorse, recupero da guasti, crittografia dei dati, aggiornamenti software, backup e altre operazioni di gestione.
- **Architettura multi-tenant**. DynamoDB ospita i dati di diversi clienti sulle stesse risorse fisiche, ottimizzando l'uso delle risorse e permettendo un risparmio sui costi che viene trasferito ai clienti. Ogni carico di lavoro è isolato grazie al monitoraggio dell'utilizzo e al provisioning accurato delle risorse.
- **Scalabilità illimitata**. Non ci sono limiti predefiniti sulla quantità di dati che una tabella può contenere. DynamoDB si espande elasticamente in base alle esigenze delle applicazioni, scalando da pochi a migliaia di server per soddisfare la domanda di storage e throughput.
- **Prestazioni prevedibili**. L'API semplificata di DynamoDB consente di gestire le richieste con bassa latenza, che generalmente rimane nell'ordine di millisecondi per i dati archiviati nella stessa regione AWS. Le latenze restano stabili anche quando le tabelle crescono da pochi MB a centinaia di TB, grazie alla distribuzione dei dati e al bilanciamento del carico.
- **Alta disponibilità**. I dati sono replicati automaticamente su più *Availability Zone* di AWS, garantendo continuità anche in caso di guasti a dischi o nodi, soddisfacendo così i requisiti più stringenti di disponibilità e durabilità.
- **Flessibilità d'uso**. DynamoDB non impone uno schema fisso per le tabelle, permettendo agli sviluppatori di creare modelli di dati personalizzati, inclusi attributi multivalore. Supporta modelli di dati chiave-valore o documento.

Queste caratteristiche fanno di DynamoDB una soluzione ideale per chi cerca un database cloud altamente scalabile, performante e affidabile. [9]

4.2.2 Integrazione di AWS Lambda con DynamoDB

L'integrazione di *DynamoDB* con le *Lambda Functions* parte proprio dalla creazione di queste ultime. Per poter accedere al servizio di database bisogna concedere alcune autorizzazioni alla funzione. Questo può essere fatto durante la fase di creazione, come menzionato in precedenza.

Per creare un nuovo ruolo, bisogna accedere al servizio *IAM*. Andando nella sezione *Ruoli* sarà possibile creare un nuovo ruolo o modificarne uno creato in precedenza.

Durante il processo di creazione, bisogna selezionare il *tipo di entità attendibile* (nel nostro caso *Servizio AWS*, il quale permette ai servizi AWS di eseguire operazioni sull'account) ed il *caso d'uso* (nel nostro caso *Lambda*, permettendo alle Lambda Functions di richiamare servizi AWS).

Successivamente, andranno selezionate le *policy* da collegare al nuovo ruolo. Nel nostro caso specifico andranno selezionate le policy *AmazonDynamoDBFullAccess* e *CloudWatchFullAccess*.

Per concludere la creazione sarà necessario dare un *nome ed una descrizione* al ruolo, e se necessario aggiungere un *tag*.

Una volta creato il ruolo personalizzato ed associato alla Lambda, il database sarà accessibile dalla nostra funzione.

Prendendo come esempio una funzione scritta in Python, basterà importare la libreria *boto3*, la quale permette il collegamento al database. Specificando il nome della tabella, sarà possibile interagire con il nostro database DynamoDB.

Alcuni esempi di codice verranno mostrati successivamente nel *caso studio*.

4.3 Google Cloud Firestore

4.3.1 Panoramica su Firestore

Firestore è un *database serverless schemaless* con funzionalità di notifica in tempo reale, cosa che semplifica notevolmente lo sviluppo di applicazioni web e mobili. È in grado di gestire milioni di query al secondo e petabyte di dati memorizzati.

Inoltre, con un basso consumo di QPS (query al secondo) e di storage, Firestore ha un costo irrisorio.

Alcuni aspetti chiave del successo di Firestore sono[10]:

- **Facilità d'uso**. Lo sviluppo di applicazioni moderne trae vantaggio dalla rapidità dell'iterazione e del deployment. Il modello di dati schemaless di Firestore, le transazioni *ACID*, la forte coerenza e l'indicizzazione di tutti i dati predefiniti consentono agli sviluppatori di concentrarsi principalmente sui dati che desiderano memorizzare e presentare all'utente finale, senza preoccuparsi dei dettagli della configurazione del database.
- **Funzionamento serverless e scalabilità rapida**. Alcune applicazioni diventano virali e questo comporta problemi di scalabilità dell'infrastruttura con l'aumento del carico, dello storage e dei costi. Con Firestore, lo sviluppatore dell'applicazione deve solo creare una pagina web (statica) o un'applicazione

e inizializzare un database Firestore per consentire agli utenti finali di archiviare e condividere i dati. Le richieste di database degli utenti finali vengono indirizzate direttamente a Firestore, senza la necessità di un server dedicato che esegua il controllo degli accessi grazie alle regole di sicurezza impostate dallo sviluppatore. L'API di Firestore incoraggia un utilizzo che scala indipendentemente dalle dimensioni del database e dal traffico. La tariffazione a consumo di Firestore, insieme a una quota giornaliera gratuita, garantisce che gli aumenti di fatturazione riflettano il successo dell'applicazione.

- **Query in tempo reale flessibili ed efficienti**. Un'applicazione ha spesso bisogno di inviare notifiche veloci a sottoinsiemi potenzialmente ampi di dispositivi web o mobili per molte ragioni, come la comunicazione tra utenti.

4.3.2 Integrazione di Google Cloud Functions con Firestore

L'integrazione di Firestore con le Google Cloud Functions risulta essere molto lineare. Prendendo come esempio una funzione scritta in Python, basterà dichiarare nei *requirements.txt* la libreria *google-cloud-firestore*. Una volta dichiarata, all'interno dello script potrà essere utilizzato il modulo *firestore*, il quale permetterà di stabilire una connessione con Firestore.

Specificando il nome della *collection*, sarà possibile interagire con il nostro database Firestore. Non è necessaria alcuna configurazione di ruoli per accedere al database, a differenza di DynamoDB.

Alcuni esempi di codice verranno mostrati successivamente nel *caso studio*.

Capitolo 5

Architettura delle API Serverless

Nel mondo delle funzioni serverless esistono molti *pattern architetturali* utili per la scrittura di codice pulito, efficiente e sicuro. In questo capitolo verranno discussi due pattern per la scrittura di funzioni, ognuno con i propri punti di forza e debolezze.

I due approcci in questione possono essere definiti come:

- Funzione unica per tutte le API
- Funzione per ogni chiamata API

Esempi dei due approcci saranno presenti nel *caso studio*, dove saranno confrontati.

5.1 Funzione unica per tutte le API

Con il seguente approccio viene creata una singola funzione, nella quale è presente tutto il codice per gestire le diverse chiamate API presenti. Per decidere cosa inserire nella stessa funzione, piuttosto di crearne una nuova, si possono usare principi di sviluppo tradizionali come il *low coupling* e la *high cohesion*.

I vantaggi di questo approccio sono:

- Tutta la logica relativa al contesto viene raggruppata in un unico luogo, rendendo il codice più leggibile.
- Il codice può essere riutilizzato tra le diverse funzioni.
- Il *security footprint* è ridotto, aggiornando un singolo file permette l'aggiornamento di molte funzioni.

Gli svantaggi invece sono:

- Difficoltà nel capire quando creare una nuova funzione. Ogni byte di codice extra rallenta il tempo di *cold start* della funzione.
- Aumento del raggio d'azione delle modifiche sul codice. La modifica di una singola riga di codice potrebbe far crollare una sezione dell'infrastruttura piuttosto di una singola funzione.

5.2 Funzione per ogni chiamata API

Questo approccio rappresenta la forma più pura dei *pattern serverless*. Ogni funzione ha un suo file, ed esegue una singola chiamata API.

I vantaggi di questo approccio sono:

- Massima riusabilità del codice. Ogni funzione ha una singola responsabilità.
- Si viene spinti a scrivere codice maggiormente testabile.
- Minor carico cognitivo per gli sviluppatori che modificano la specifica funzione.
- Miglior ottimizzazione dei tempi di esecuzione, e di conseguenza dei costi.

Gli svantaggi invece sono:

- Approccio funzionante solo per architetture completamente *event-driven*.
- Soffermandosi sul quadro generale, il carico cognitivo aumenta quando si parla di modifiche a livello di sistema.
- La manutenzione aumenta man mano che le funzioni crescono di numero.

Capitolo 6

Caso Studio: Confronto tra le Piattaforme

Prima di discutere dei risultati ottenuti dalle prove eseguite sulle piattaforme bisogna introdurre la *soluzione software* utilizzata e descrivere il *processo di testing* che ci porta ad ottenere tali risultati.

6.1 Descrizione delle Soluzioni Software

Il *caso studio* viene accompagnato da diversi script *Python*, leggermente diversi tra le due piattaforme e tra i due approcci.

L'obiettivo della soluzione software proposta è quella di fornire funzionalità *CRUD* ipotizzando la presenza di un *inventario* da riempire con dei *prodotti*. Tutti i dati inerenti all'inventario verranno quindi salvati in un database.

La soluzione presenta i seguenti metodi HTTP:

- **GET /items**: permette di ottenere la lista di tutti i prodotti presenti nell'inventario.
- **PUT /items**: permette di inserire o modificare un prodotto presente nell'inventario.
- **GET /items/{id}**: permette di ottenere un singolo prodotto dell'inventario specificando un *identificativo* univoco, se presente.
- **DELETE /items/{id}**: permette di eliminare un singolo prodotto dell'inventario specificando un *identificativo* univoco.

6.2 Implementazione su AWS Lambda

6.2.1 Deploy dell'Approccio Funzione Unica

Come anticipato nel *Capitolo 5*, il seguente approccio si basa su una singola funzione nella quale è presente tutto il codice per gestire le diverse chiamate API presenti.

Per quanto riguarda il database, viene utilizzato il precedentemente citato *DynamoDB*. Per consentire il collegamento ai dati alla *Lambda Function* occorre aggiungere alla funzione stessa il ruolo personalizzato discusso nel *Capitolo 4*.

Viene quindi utilizzato il seguente script:

```
1  import json
2  import boto3
3
4  client = boto3.client('dynamodb')
5  dynamodb = boto3.resource("dynamodb")
6  table = dynamodb.Table('product-inventory')
7
8  def lambda_handler(event, context):
9      body = {}
10     statusCode = 200
11
12     try:
13         if event['routeKey'] == "DELETE /items/{id}":
14             table.delete_item(Key={'id': event['
                   pathParameters']['id']})
15             body = 'Deleted item ' + event['pathParameters'
                   ]['id']
16         elif event['routeKey'] == "GET /items/{id}":
17             body = table.get_item(Key={'id': event['
                   pathParameters']['id']})
18             body = body["Item"]
19             responseBody = [{'price': body['price'], 'id':
                   body['id'], 'name': body['name'], '
                   description': body['description']}]
20             body = responseBody
21         elif event['routeKey'] == "GET /items":
22             body = table.scan()
23             body = body["Items"]
24             responseBody = []
```

```
25              for items in body:
26                  responseItems = [{'price': items['price'], '
                        id': items['id'], 'name': items['name'],
                        'description': items['description']}]
27                  responseBody.append(responseItems)
28              body = responseBody
29          elif event['routeKey'] == "PUT /items":
30              requestJSON = json.loads(event['body'])
31              table.put_item(
32                  Item={
33                      'id': requestJSON['id'],
34                      'price': requestJSON['price'],
35                      'name': requestJSON['name'],
36                      'description': requestJSON['description'
                            ]
37                  })
38              body = 'Put item ' + requestJSON['id']
39      except KeyError:
40          statusCode = 400
41          body = 'Unsupported route: ' + event['routeKey']
42
43      body = json.dumps(body)
44      res = {
45          "statusCode": statusCode,
46          "headers": {
47              "Content-Type": "application/json"
48          },
49          "body": body
50      }
51
52      return res
```

Lo script è strutturato in maniera lineare. Dopo aver stabilito la connessione con il database (grazie alla libreria *boto3*) si definisce l'entry-point della *Lambda Function*.

Il cuore della funzione controlla a quale metodo HTTP appartiene la richiesta e lo gestisce di conseguenza, restituendo il risultato dell'operazione. In caso di richiesta errata, viene gestito l'errore restituendo un messaggio di errore.

E' possibile quindi osservare l'approccio a funzione unica, con tutta la logica presente all'interno di una singola *Lambda*. I diversi percorsi creati in *API Gateway* puntano

alla stessa funzione, che gestisce tutti i metodi HTTP.

6.2.2 Deploy dell'Approccio Funzione per Ogni Chiamata

L'approccio a funzione per ogni chiamata si distingue dal precedente per la sua suddivisione in diverse *Lambda Function*, ognuna delegata alla gestione di un singolo metodo HTTP.

Per il database vale il discorso del precedente approccio.

Gli script utilizzati sono simili allo script della funzione unica, rimuovendo la presenza degli altri metodi HTTP non gestiti dalla specifica funzione.

Viene mostrato come riferimento solo uno dei quattro script, nello specifico il metodo *GET /items/{id}*:

```
1  import json
2  import boto3
3
4  client = boto3.client('dynamodb')
5  dynamodb = boto3.resource("dynamodb")
6  table = dynamodb.Table('product-inventory')
7
8  def lambda_handler(event, context):
9      body = {}
10     statusCode = 200
11
12     try:
13         body = table.get_item(Key={'id': event['
               pathParameters']['id']})
14         body = body["Item"]
15         responseBody = [{'price': body['price'], 'id': body[
               'id'], 'name': body['name'], 'description': body[
               'description']}]
16         body = responseBody
17     except KeyError:
18         statusCode = 400
19         body = 'Unsupported route: ' + event['routeKey']
20
21     body = json.dumps(body)
22     res = {
23         "statusCode": statusCode,
```

```
        "headers": {
            "Content-Type": "application/json"
        },
        "body": body
    }

    return res
```

E' quindi possibile osservare come la struttura sia la medesima tra i due approcci, dividendo però la funzione unica in quattro funzioni specifiche, ognuna con una responsabilità differente.

In questo caso, i percorsi creati in *API Gateway* puntano alla funzione specifica.

6.3 Implementazione su Google Cloud Functions

Prima di iniziare, va fatta una premessa che vale per entrambe le implementazioni su *Google Cloud Functions*. Per un corretto funzionamento delle librerie Python, insieme allo script vanno specificati i suoi *requirements* all'interno di un file *.txt*. Per tutti gli script utilizzati nel caso studio, i requisiti sono i seguenti:

```
functions-framework==3.*
google-cloud-firestore
```

6.3.1 Deploy dell'Approccio Funzione Unica

Anche per l'implementazione su *Cloud Functions*, l'approccio a funzione unica si basa su una singola funzione che gestisce tutte le chiamate API presenti.

In questo caso il database utilizzato è il precedentemente citato *Firestore*. A differenza di *AWS*, *GCP* non richiede ruoli aggiuntivi per consentire alle proprie funzioni serverless di accedere al database.

Viene quindi utilizzato il seguente script:

```
import json
from google.cloud import firestore

client = firestore.Client()
collection_name = 'ProductInventory'

```

```
def function(request):
    request_json = request.get_json(silent=True)
    path = request.path
    method = request.method

    body = {}
    statusCode = 200

    try:
        if path.startswith("/items/") and method == "DELETE":
            item_id = path.split('/')[-1]
            doc_ref = client.collection(collection_name).document(item_id)
            doc_ref.delete()
            body = f'Deleted item {item_id}'
        elif path.startswith("/items/") and method == "GET":
            item_id = path.split('/')[-1]
            doc_ref = client.collection(collection_name).document(item_id)
            doc = doc_ref.get()
            if doc.exists:
                doc_dict = doc.to_dict()
                responseBody = [{'price': doc_dict['price'], 'id': doc.id, 'name': doc_dict['name'], 'description': doc_dict['description']}]
                body = responseBody
            else:
                statusCode = 404
                body = f'Item with id {item_id} not found'
        elif path == "/items" and method == "GET":
            docs = client.collection(collection_name).stream()
            responseBody = []
            for doc in docs:
                doc_dict = doc.to_dict()
                responseItems = [{'price': doc_dict['price'], 'id': doc.id, 'name': doc_dict['name'], 'description': doc_dict['description']}]
```

```
38                  responseBody.append(responseItems)
39              body = responseBody
40          elif path == "/items" and method == "PUT":
41              if not request_json:
42                  raise ValueError("No request body")
43              item_id = request_json['id']
44              doc_ref = client.collection(collection_name).
                    document(item_id)
45              doc_ref.set({
46                  'price': request_json['price'],
47                  'name': request_json['name'],
48                  'description': request_json['description']
49              })
50              body = f'Put item {item_id}'
51          else:
52              raise KeyError(f'Unsupported route: {method} {
                    path}')
53      except KeyError as e:
54          statusCode = 400
55          body = str(e)
56      except Exception as e:
57          statusCode = 500
58          body = str(e)
59
60      res = {
61          "statusCode": statusCode,
62          "headers": {
63              "Content-Type": "application/json"
64          },
65          "body": json.dumps(body)
66      }
67
68      return (res["body"], res["statusCode"], res["headers"])
```

Lo script segue la medesima struttura utilizzata per le *Lambda Functions*. Anche in questo caso si stabilisce prima di tutto la connessione al database (grazie alla libreria *firestore*).

Il codice gestisce tutte le richieste ricevute, controllando a quale metodo HTTP appartiene e rispondendo di conseguenza. Anche in questo caso, davanti alla presenza di errori, viene restituito un messaggio di errore.

GCP non richiede di configurare alcun tipo di *API Gateway*, esponendo un *endpoint* automaticamente alla creazione della *Cloud Function*.

6.3.2 Deploy dell'Approccio Funzione per Ogni Chiamata

Per l'approccio con singola funzione per chiamata, anche in questo caso viene creata una *Cloud Function* per ogni metodo HTTP da esporre.

Per il database vale il discorso del precedente approccio.

Gli script utilizzati tendono ad essere simili allo script della funzione unica, rimuovendo tutto il codice inerente alla gestione degli altri metodi HTTP.

Viene mostrato come riferimento solo uno dei quattro script, nello specifico il metodo *PUT /items*:

```
import json
from google.cloud import firestore

client = firestore.Client()
collection_name = 'ProductInventory'

def function(request):
    request_json = request.get_json(silent=True)
    path = request.path
    method = request.method

    body = {}
    statusCode = 200

    try:
        if path == "/items" and method == "PUT":
            if not request_json:
                raise ValueError("No request body")
            item_id = request_json['id']
            doc_ref = client.collection(collection_name).
               document(item_id)
            doc_ref.set({
                'price': request_json['price'],
                'name': request_json['name'],
                'description': request_json['description']
            })
```

```
26              body = f'Put item {item_id}'
27          else:
28              raise KeyError(f'Unsupported route: {method} {
                    path}')
29      except KeyError as e:
30          statusCode = 400
31          body = str(e)
32      except Exception as e:
33          statusCode = 500
34          body = str(e)
35
36      res = {
37          "statusCode": statusCode,
38          "headers": {
39              "Content-Type": "application/json"
40          },
41          "body": json.dumps(body)
42      }
43
44      return (res["body"], res["statusCode"], res["headers"])
```

Ogni funzione avrà quindi una singola responsabilità: gestire un solo metodo HTTP.

6.4 Risultati ottenuti

Il *processo di testing* delle due piattaforme si basa su tre fattori principali: *Performance*, *Costi* e *Usabilità-Facilità di Deploy*.

Per tutti e tre i fattori si andranno a spiegare i criteri di confronto ed i risultati ottenuti. Nel *Capitolo 7*, invece, si andranno a discutere e confrontare i risultati.

6.4.1 Performance

Per le *performance* si è deciso di testare la *latenza* dell'API.

Per *latenza* si intende il tempo che un'API impiega per elaborare una richiesta ed inviare una risposta, compresi eventuali ritardi di rete o di elaborazione. Può essere influenzata da vari fattori, come la velocità della connessione di rete, il tempo di elaborazione dell'API e la quantità di dati trasferiti.

Nel nostro caso andremo a testare la latenza ottenuta in caso di *cold startup* e non, osservando le possibili differenze anche tra i due approcci di API.

La fase di testing consiste nel misurare la latenza effettuando una media di 10 chiamate, escludendo i due valori estremi.
In entrambi i casi viene testato il metodo *GET /items*.

I risultati ottenuti per *Lambda* sono:

	Funzione Unica	**Funzione Singola**
Cold Startup	585 ms	556 ms
Normale	84 ms	86 ms

I risultati ottenuti per *Cloud Functions* sono:

	Funzione Unica	**Funzione Singola**
Cold Startup	857 ms	802 ms
Normale	113 ms	106 ms

6.4.2 Costi

Per i *costi* si è deciso di osservare il *costo per singola richiesta* alla funzione. I costi vengono dichiarati dai due provider, dove entrambi offrono un numero di richieste gratuite al mese.

Lambda include *1 milione* di richieste gratuite al mese, mentre *Cloud Functions* ne offre *2 milioni*.

Superata la soglia di richieste gratuite, *Lambda* ha un costo di *0,23$ per 1 milione di richieste*, mentre *Cloud Functions* ha un costo di *0,40$ per 1 milione di richieste*.

6.4.3 Usabilità e Facilità di Deploy

Come ultimo fattore di confronto si è deciso di parlare dell'*usabilità della piattaforma* e della *facilità di deploy*.

Per quanto riguarda l'*usabilità*, entrambe le piattaforme presentano interfacce intuitive, curate e dettagliate. *AWS* fornisce di gran lunga una miglior documentazione per lo sviluppatore, fornendo esempi di piccoli progetti. *GCP* risulta più carente di documentazione ufficiale.

Per il discorso *facilità di deploy*, le *Cloud Functions* richiedono un minor numero di step per avere una funzione pronta all'uso. Non c'è bisogno di creare alcun tipo di API Gateway o ruoli per accedere al database, a differenza di *Lambda*.

Capitolo 7

Discussione dei Risultati

Osservando i risultati ottenuti dopo il *processo di testing* e di *analisi* è possibile arrivare ad alcune conclusioni.

Premettendo che tutti i dati ottenuti non sono in alcun modo perfetti, e di conseguenza le conclusioni non vogliono affermare una verità assoluta, si può procedere ad un confronto tra le due piattaforme.

7.1 Performance

Per quanto riguarda le *performance*, le *Lambda Functions* tendono ad avere una *latenza minore*, sia in casi di *cold startup* che non, rispetto alle *Cloud Functions*.

E' possibile osservare inoltre come i due approcci di scrittura delle API non portino quasi alcun tipo di differenza sulla latenza nel nostro caso studio, potendo quindi affermare che in casi con funzioni non molto lunghe è indifferente l'approccio scelto. Ovviamente, all'aumentare della lunghezza della funzione, la funzione unica arriverà ad avere latenze maggiori.

Durante la fase di testing è stato possibile notare anche un altro piccolo dettaglio. *Lambda* esegue *cold startup* molto più frequentemente di *Cloud Functions*, dove la funzione tende a reagire in maniera più reattiva anche dopo un periodo di inattività più lungo.

7.2 Costi

Per quanto riguarda i *costi*, *AWS* offre di gran lunga un costo più vantaggioso, inferiore a *GCP* di *0,17$* per milione di richieste.

Va però ribadito il numero di richieste gratuite al mese offerto da entrambe le piattaforme, dove *Google* offre il doppio delle richieste rispetto ad *Amazon*. In progetti di dimensioni ridotte, o con un numero di richieste mensili ridotto, la possibilità di non pagare alcuna cifra al di sotto delle 2 milioni di richieste potrebbe portare piccoli team di sviluppo a scegliere la piattaforma offerta da *Google*.

7.3 Usabilità e Facilità di Deploy

Per concludere, la *facilità di deploy* offerta da entrambe le piattaforme è più che soddisfacente, con *Cloud Functions* che risulta essere più immediata da configurare e subito pronta all'uso.

Un aspetto dove *Amazon* vince con un netto distacco è la presenza di documentazione ufficiale e di progetti d'esempio ben fatti, i quali consentono agli sviluppatori alle prime armi nel mondo *serverless* di avere esempi di qualità come riferimento.

7.4 Risultati finali

Dovendo proclamare una sorta di "vincitore" - pur sempre da un punto di vista soggettivo - tra le due piattaforme, mi sento di affermare che *AWS Lambda* sia leggermente migliore di *GCP Cloud Functions*, con costi per richiesta inferiori e miglior documentazione, nonostante una più lunga fase di deploy.

Questa conclusione è in linea con l'utilizzo generale delle piattaforme nel mondo serverless, dove *AWS* ha una percentuale di utlizzo nettamente maggiore rispetto ai competitors, coprendo circa l'*80% del mercato*[11].

Capitolo 8

Conclusioni

Questa tesi ha esaminato due delle principali piattaforme per l'esecuzione di funzioni serverless oggi disponibili, *AWS Lambda* e *Google Cloud Functions*, con l'obiettivo di confrontarne le *prestazioni*, i *costi* e la *facilità di deploy*.

Attraverso un caso studio che prevedeva l'integrazione con database serverless come *Amazon DynamoDB* e *Google Firestore*, sono stati messi alla prova due approcci architetturali differenti: la *funzione unica per tutte le API* e la *funzione dedicata per ciascuna chiamata API*.

8.1 Riassunto dei Risultati

I diversi test delle *performance* hanno rivelato che *AWS Lambda* offre una *latenza inferiore* rispetto a *Google Cloud Functions*, sia in condizioni di *cold start* che in *esecuzioni normali*.

Inoltre, dal punto di vista economico, *AWS Lambda* si è dimostrata **più conveniente**, con un **costo per richiesta inferiore** rispetto a *Google*, sebbene Google offra un maggior numero di richieste gratuite mensili.

Per quanto riguarda la facilità d'uso, entrambe le piattaforme presentano interfacce intuitive, ma *Google Cloud Functions* si distingue per la semplificazione nel processo di deploy, eliminando la necessità di configurare ruoli aggiuntivi o Gateway API.

8.2 Implicazioni dello studio

I risultati di questa tesi possono essere rilevanti per sviluppatori e aziende che vogliono adottare soluzioni serverless. Le prestazioni migliori e i costi ridotti di *AWS*

Lambda lo rendono una scelta preferibile per progetti su larga scala o che richiedono alta efficienza, mentre la maggiore semplicità di deploy di *Google Cloud Functions* potrebbe attrarre piccoli team o startup con necessità immediate e minori risorse tecniche.

8.3 Limitazioni dello studio

Lo studio si è concentrato su un caso studio *limitato*, con un set ristretto di funzioni e integrazioni. Le conclusioni *potrebbero variare* in contesti differenti, come progetti con requisiti di *sicurezza più stringenti* o in applicazioni che coinvolgono *carichi di lavoro estremamente variabili*.

Inoltre, la scelta di limitare il confronto a due soli fornitori potrebbe non riflettere la varietà di soluzioni serverless disponibili sul mercato.

8.4 Sviluppi futuri

Per approfondire l'analisi, sarebbe interessante espandere il confronto ad altre piattaforme per l'esecuzione di funzioni serverless come *Microsoft Azure Functions* o *IBM OpenWhisk*, per offrire una panoramica più ampia delle opzioni disponibili sul mercato.

Inoltre, testare applicazioni più complesse e includere altri fattori, come la *gestione della sicurezza* e il *supporto per tecnologie emergenti* come l'*intelligenza artificiale* e il *machine learning*, potrebbe fornire ulteriori spunti di riflessione su quale piattaforma sia più adatta a specifici scenari.

In conclusione, sebbene *AWS Lambda* si sia dimostrata leggermente superiore in termini di performance e costi, la scelta finale della piattaforma dipenderà dalle specifiche esigenze del progetto e dal contesto di utilizzo.

Riferimenti bibliografici

[1] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 405–410, 2017.

[2] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surv.*, vol. 54, sep 2022.

[3] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[4] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: a survey of opportunities, challenges, and applications," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–32, 2022.

[5] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions," *Future Generation Computer Systems*, vol. 110, pp. 502–514, 2020.

[6] V. Giménez-Alventosa, G. Moltó, and M. Caballer, "A framework and a performance assessment for serverless mapreduce on aws lambda," *Future Generation Computer Systems*, vol. 97, pp. 259–274, 2019.

[7] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 162–169, IEEE, 2017.

[8] T. Naumenko and A. Petrenko, "Analysis of problems of storage and processing of data in serverless technologies," *Technology audit and production reserves*, vol. 2, no. 2, p. 58, 2021.

[9] M. Elhemali, N. Gallagher, B. Tang, N. Gordon, H. Huang, H. Chen, J. Idziorek, M. Wang, R. Krog, Z. Zhu, *et al.*, "Amazon {DynamoDB}: A scalable, predictably performant, and fully managed {NoSQL} database service," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 1037–1048, 2022.

[10] R. Kesavan, D. Gay, D. Thevessen, J. Shah, and C. Mohan, "Firestore: The nosql serverless database for the application developer," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pp. 3376–3388, IEEE, 2023.

[11] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4152–4166, 2021.