Alma Mater Studiorum - Università di Bologna

Campus di Cesena

Corso di Laurea in Ingegneria e Scienze Informatiche

Gli strumenti Serverless di AWS

Tesi di Laurea in Systems Integration

Relatore:

Prof. Vittorio Ghini

Presentata da:

Federico Mengozzi

Anno Accademico 2019/2020

Sommario

Il serverless è un trend del cloud computing che si sta diffondendo rapidamente. I cloud provider offrono tutte le risorse che servono per eseguire applicazioni. Lasciando agli sviluppatori la possibilità di concetrarsi solo sulla creazione del prodotto. In questo lavoro, dopo una breve introduzione al serverless e ai microservizi, analizzerò nel dettaglio i servizi AWS che sono alla base dello sviluppo serverless con Amazon. Offrirò due esempi, di come AWS è utilizzato in production per risolvere vecchi problemi in modi nuovi e creativi. Infine, mostrerò una semplice applicazione serverless per mostrare i vantaggi che creare architetture serverless su AWS fornisce.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Sommario

1.1 Introduzione al serverless

All'inizio degli anni 2000, per via della rapida crescita Amazon stava faticando nelle gestione della sua infrastruttura. Ben presto ci si rese conto che l'architettura monolitica utilizzata fino a quel momento non potesse scalare abbastanza velocemente. Nel 2004 Amazon si accorse che il migliore modo per gestire un sistema relativamente grande, era quelli di dividerlo in parti più piccole come meno dipendenze possibili le une dalle altre, ciò che viene definito come decoupling. Da questo primo step innovativo nacque anche il primo servizio di AWS che serviva per permettere a queste piccole parti di comunicare, il servizio era Amazon Simple Queue Service. Successivamente, nel 2006 venne rilasciato Simple Storage Service o S3, servizio di storage di Amazon che permette di memorizzare oggetti fino a 5 terabyte. Da lì a breve Amazon avrebbe introdotto il servizio IaaS di Elastic Compute Cloud EC2. Questi tre servizi costituiscono la base degli Amazon Web Services che sono oggi il pilastro di Amazon.

Capitolo 2

Serverless e Microservizi

Sommario

Amazon e altre compagnie tra cui Google e Salesforce furono le prime ad offrire infrastrutture di Cloud Computing. Oggi AWS è il primo vendor di servizi per il cloud computing.

Tra i vari trend del cloud computing, due principali componenti sono Serverless e Microservizi. Generalmente, passare ad un sistema serverless che viene eseguito nel cloud richiede di rivederne l'architettura. Molto spesso, transitando da cluster di server fisici al cloud si finisce con lo spezzare una applicazione monolitica in microservizi indipendenti.

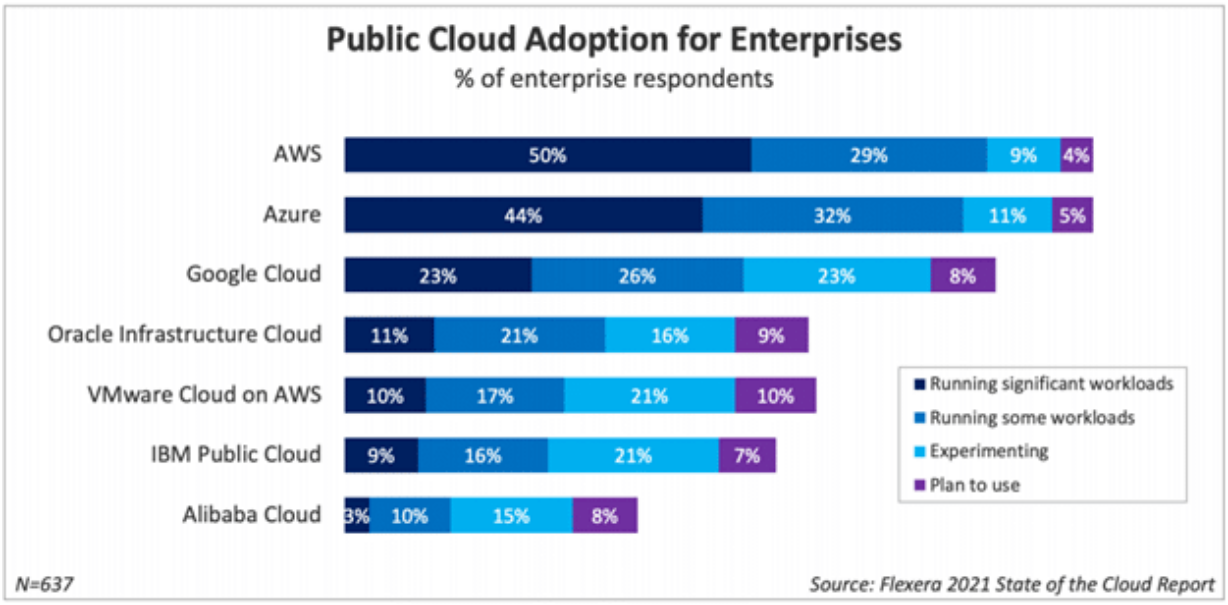

Figura 2.1: Adozione del Cloud Computing

2.1 Serverless

Le applicazioni serverless sono quelle in cui non è necessario gestire alcun server. L'intero focus passa così al codice e alla business logic permettendo agli sviluppatori di delegare le responsabilità di gestire il sistema operativo, gli access controll, la scalabilità e l'availability alla piattaforma sottostante.

Serverless offre una scalability flessibile e automatica a partire dalla configurazione delle singole unità di computazione invece che di intere macchine. Offre fault tolerance by design, implementata nella piattaforma sottostante stessa. Nel caso di AWS il network di computing machine è esteso nella maggior parte della regioni con un alto numero di repliche. Inoltre, con il serverless l'utilizzo delle risorse è ottimizzato dato che l'unità computazionale viene scalata su e giù in base al workload, eliminando così momenti di idle capacity.

Il paradigma serverless si basa sul concetto più generale del cloud computing. Nel cloud computing le risorse necessarie per la computazione vengono messe a disposizione dal provider del servizio. Queste risorse vengono offerte in base a diversi modelli. I principali sono

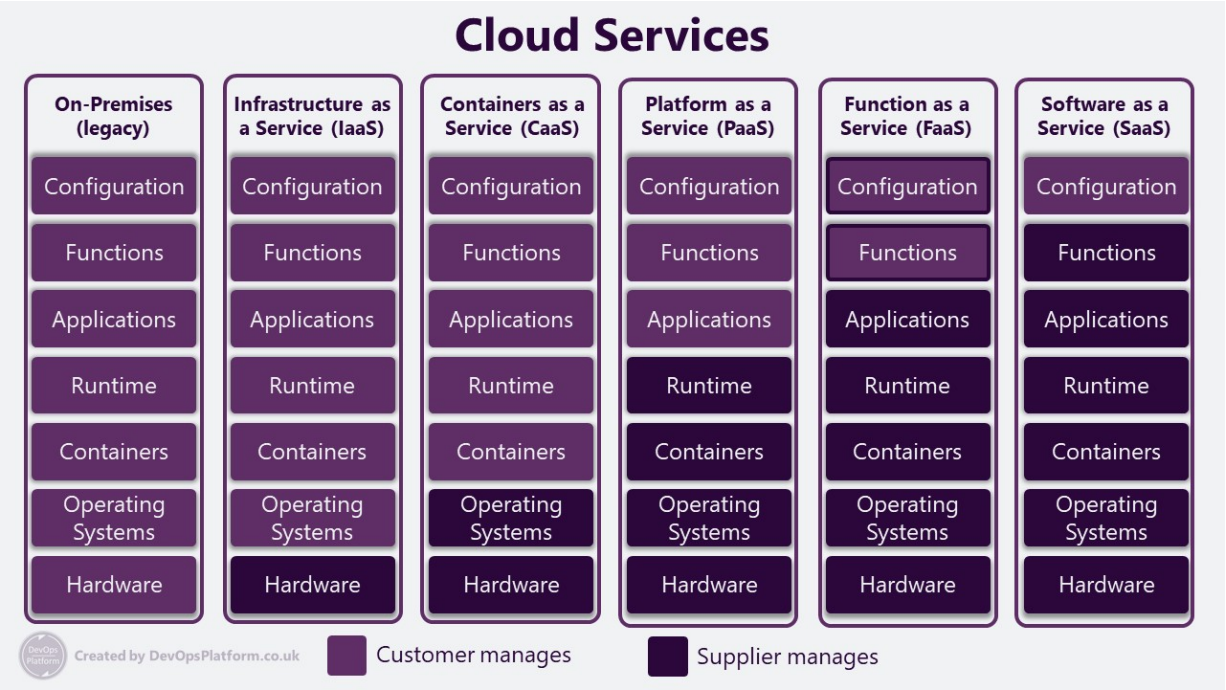

Figura 2.2: Modelli di cloud computing

2.1.1 Infrastructure as a Service (IaaS)

Infrastructure as a Service (IaaS) offre i blocchi base per l'IT cloud permettendo la gestione di network, computer (hardware virtuale o dedicato) e sistemi di storage. IaaS permette la massima flessibilità e controllo su come le risorse vengono distribuite e gestite. Allo stesso tempo la gestione di queste risorse deve essere configurata più attentamente.

2.1.2 Container as a Service (Caas)

Container as a Service (Caas) è uno dei più recenti modelli di cloud computing ad essersi diffuso, nato dopo la diffusione di servizi come docker, permette di eseguire container gestendone l'orchestrazione. Semplifica IaaS in quanto non è necessaria nessuna configurazione dell'hardware.

2.1.3 Platform as a Service (PaaS)

Platform as a Service (PaaS) semplifica il modello CaaS offrendo out of the box tutta l'infrastruttura (generalmente l'hardware e il sistema operativo) per eseguire applicazioni e lascia la libertà di potersi concentrare solamente sullo sviluppo e deployment dell'applicazione.

2.1.4 Function as a Service (FaaS)

Function as a service (FaaS) è un modello che permette l'esecuzione di funzioni ospitate nel cloud che sfruttando i vantaggi del serverless. Permette di non dover gestire problemi di scalabilità e availability. È possibile fare il deployment di singole funzioni che possono poi essere combinate per creare soluzioni complesse.

2.1.5 Software as a Service (SaaS)

Software as a Service (SaaS), astraea un livello ulteriore, offrendo software che esegue un compito preciso. Il prodotto può essere utilizzato direttamente per compiere un compito desiderato. Un esempio di SaaS è la web-based email, che può essere utilizzata senza dover sviluppare un intero servizio di mailing da zero. In questo modo è possibile combinare diversi software per creare un'applicazione complessa, senza preoccuparsi di mantenere né il codice, né la piattaforma e nemmeno l'infrastruttura su cui l'applicazione viene eseguita.

Ogni modello costruisce sul precedente per semplificare la risoluzione di un problema. Partendo da IaaS fino ad arrivare a SaaS, i tempi di configurazione e la complessità di gestione del servizio diminuiscono al costo di delegare il controllo specifico delle risorse.

2.2 Microservizi

Con il termine microservizi si riferisce al pattern architetturale secondo il quale le applicazioni vengono divise in servizi più piccoli, il più indipendenti possibile. Alcune caratteristiche che sono tipiche di un micro servizio sono

- Single Responsibility Principle, con boundaries in linea con i business boundaries di una particolare funzionalità.
- Autonomo, ogni microservizio è un'entità separata che può essere deployata indipendentemente dal resto. La comunicazione tra servizi deve avvenire tramite una API di interfaccia.
- Resilienza, quando un microservizio fallisce altri componenti del sistema possono continuare a funzionare senza che si presentino fallimenti a catena.
- Scalabilità, in base al carico di lavoro un microservizio può essere replicato lasciando invariati gli altri. Ciò permette un più efficace utilizzo delle risorse.

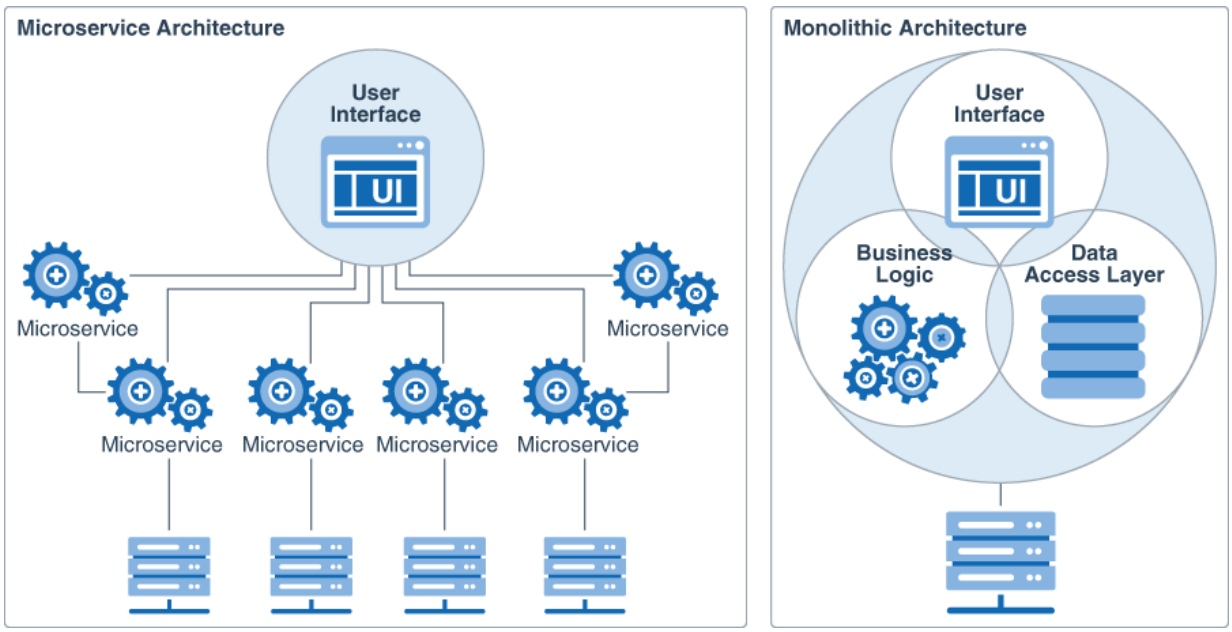

Figura 2.3: Microservizi vs architettura monolitica

- Composizione, la natura dei microservizi gli rende agnostici rispetto all'applicazione in cui vengono usati e possono venire riutilizzati in diverse applicazioni senza bisogno di complicate integrazioni.

2.2.1 Patterns

Essendo quello dei Microservizi un nuovo modo per svillupare applicazioni rispetto al classico approccio monolitico, vengono utilizzate anche procedure diverse per pianificare e implementare l'applicazione. Tra gli approcci piu' utilizzati per sviluppare microservizi, di seguito sono elencati i più diffusi

Decomposition

Uno degli step più importanti durante lo sviluppo di microservizi è definire i confini di ciascun servizio. Non sempre la linea che separa un servizio da un altro è netta, in certe situazioni conviene utilizzare alcuni principi, in altre altri.

Decompose by subdomain In questo scenario, ogni servizio rappresenta un parte specifica di business logic. Ogni business svolge alcuni funzioni che servono per offrire un determinato servizio. Alcune di queste funzioni sono core functions, indispensabili al business, altre sono di supporto e altre ancora generiche. Ognuno di questi subdomain può

essere considerato un'unità a sè stante, rappresenta quindi una buona astrazione con cui separare i servizi.

Service per team La legge di Conway dice che un'organizzazione che costruisce un sistema, produrrà un design che assomiglia alla struttura di comunicazione dell'organizzazione. Similmente a questo principio, il pattern rappresenta una definizione di microservio specifica per team. Così facendo ogni team è responsabile dell'intera gestione del servizio. Una volta definito il contratto con gli altri servizi, gli sviluppatori sono liberi di procedere senza doversi interfacciare con altri teams.

Refactoring

Dato l'altissimo livello di decoupling, muoversi da una applicazione monolitica ad una a microservizi può dover richiedere il refactoring dell'intera applicazione. Per evitare di spenderi mesi solamente per riscrivere codice senza far avanzare l'applicazione è opportuno seguire alcuni pratiche.

Strangler pattern Lo strangler pattern prevede di aggiungere nuove funzionalità come microservizi e riscrivere le vecchie porzioni dell'app monolitica solo quando è necessario rilasciarne una nuova versione. Con il tempo, la vecchia applicazione monolitica verrà "strangolate" e completamente rimpiazzata dai microservizi.

Deployment

Multiple service instances per host Eseguendo molteplici servizi su un solo host, è possibile fare il deploy dell'intera applicazione senza dover gestire un numero troppo elevato di macchine.

Single service instance per host Eseguendo un solo servizio per host, ciascun servizio ha a disposizione tutte le risorse della macchina, questo può essere un requisito indispendabile per servizi che che devono supportare alte performance. Richiede però una alto numeri di host, più difficile da scalare.

Single service instance per Container Con questo stile il deployment risulta notevolmente velocizzato, in particolare utilizzando servizi di orchestration per container, la

scalabilità dell'applicazione può essere gestita automaticamente. Dal punto di vista di packaging del servizio, creare ed eseguire un container è decisamente più efficente che creare ed eseguire una VM.

Communication

Gestendo microservizi molte delle assunzioni fatte con le app monolitiche non sono valide. Non si può assumere che i servizi vengano eseguiti sullo stesso sistema operativo, sullo stesso file system o sulla stessa rete. Per questo la comunicazione deve essere astratta il più possibile.

API Gateway L'API Gateway rappresenta un unico punto di ingresso per tutti i client (microservizi stessi) che si occupa di gestire tutte le richieste. Alcune richieste possono essere indirizzate al servizio appropriato, mentre altre richiedono di essere spacchettate e inoltrare a diversi servizi.

Backends for Frontends Una variante dell'API Gateway è il pattern Backend for frontend. Con questa soluzione, ogni tipo di client (mobile, console, PC) manda richieste ad un diverso API Gateway. Questo permette di effettuare ottimizzazioni specifiche per ogni hardware, specialmente per quanto riguarda come i dati vengono mostrati all'utente piuttosto che per il tipo di dati stesso. Netflix fa largo uso di questa pratica, in quanto supportano client molto differenti tra loro con diverse risorse disponibili.

Observability

Avere servizi indipendenti distribuiti nell'internet può portare a serie difficoltà quando si tratta di gestire log e fare debugging. Per questo è importante ottimizzare la raccolta di log e la raccolta di metadata sullo stato di ciascuna operazione.

Distributed tracing Con il distributed tracing, non appena una richiesta arriva all'applicazione, si genera un id che sarà assegnato alla richiesta in tutte le sue fasi. In questo modo ogni servizio potrà loggare i metadati della richiesta e sarà possibile analizzare ogni suo step tramite l'identificatore comune. Amazon fornisce il servizio AWX X-Ray che implementa questa funzionalità.

Log aggregation Gestire una grande quantità di logs può diventare molto complicato se ogni log viene salvato in una location indipendente. Per questo è importante avere un servizio di log centrallizzato che raccoglie i log da ogni servizio. Amazon offre questo servizio tramite AWS Cloudwatch.

AWS offre tutti gli strumenti per sviluppare microservizi. In particolare i due approcci più diffusi nello sviluppo di microservizi sono l'utilizzo di funzioni Lambda o container con Fargate. Tanti altri servizi assistono questi due compute service per operazioni di logging e integrazione.

Capitolo 3

AWS

Sommario

Di seguito sono illustrati i principali servizi serverless che Amazon offre. Sono tra i più diffusi e maggiormete utilizzati.

3.1 Computing

Essendo AWS un provider che si occupa di gestire tutta l'infrastruttura, quello che interessa ai customer di AWS non è solamente la scalability e l'aviability. Parametri che potrebbere essere garantiti semplicemente riservando un quantitativo superfluo di risorse. Ciò che porta molti clienti a scegliere AWS è l'elasticity, la capacità di un sistema di scalare up quando il workload aumenta e scalare down quando le risorse non sono più utilizzate.

3.1.1 Lambda

Lambda è un servizio di computazione event-driven basato su funzioni. Costituisce il layer di cloud logic dell'applicazione, permettendo l'esecuzione di codice arbitrario (supporta Go, Java, Nodejs, Python, Ruby, C). Una funzione Lambda può essere innescata da diversi tipi di eventi AWS o da servizi esterni. Quando gli eventi di innesco sono molti e simultanei Lambda esegue più copie della stessa funzione in parallelo, scalando linearmente in base alla quantità di lavoro richiesto. Questo garantisce una miglior efficienza delle risorse evitando di avere servizi o container idle.

Lambda rappresenta il tipo di cloud computing definito Function-as-a-Service (Faas), dove la funzione costituisce l'unità base del deployment e dell'esecuzione. Non serve configurare nessun server o container, in quanto vengono gestiti internamente da Lambda stessa.

Le funzioni Lambda hanno la particolarità di essere stateless, non devono essere quindi fatte assunzioni sullo stato della funzione se non che ad ogni esecuzioni esso viene ripristinato a valori originali. È abbastanza preciso pensare all'esecuzione di una Lambda come un processo che crea un container contenente il codice da eseguire e il relativo framework, lo esegue e poi lo distrugge.

AWS è però in grado di riutilizzare un container esistente nel caso l'esecuzione delle Lambda avvenga entro alcuni minuti di distanza. In questo caso si descrive l'esecuzione

come hot start. Quando invece il container contenente la Lambda deve essere creato da zero si parla di cold start. Può essere utile tenere a mente questo concetto per eseguire operazioni in batch, ma non è strettamente essenziale per il corretto funzionamento dell'applicazione.

Lambda introduce inoltre il concetto di livelli. I livelli Lambda sono parti di codice, come librerie o funzioni di setup che possono essere condivisi tra diverse funzioni Lambda per riutilizzare porzioni di codice che richiedono un importante quantità di risorse per essere inizializzate.

Gli eventi che possono invocare una Lambda possono essere di tipo Push e Pull. Gli eventi di tipo push invocano la Lambda non appena un evento si verifica. Per quanto riguarda gli eventi di tipo pull invece, Lambda fa il polling di un data source e invoca la funzione solamente quando arrivano nuovi dati.

Ogni funzione Lambda può essere configurata con CPU, memoria e timeout (fino a 300s dedicati).

3.1.2 Fargate

Fargate è un servizio AWS relativamente recente, rilasciato nel 2018. È un engine di computazione per Amazon ECS che permette di eseguire container senza dover gestire server o interi clusters. Questo rimuove la necessità di dover scegliere il tipo di server, i parametri di scalabilità, di gestire sistemi operativi, aggiornando e patchando ogni versione e altre configurazioni di ottimizzazione. Con Fargate è sufficiente creare un container con l'applicazione da eseguire, specificare requisiti di CPU e memoria e lanciare l'applicazione.

Fargate rappresenta il tipo di cloud computing definito Container-as-a-Service (Caas), supporta un esecuzione on demand, generalmente eseguita manualmente o tramite una Lambda, ma anche esecuzione programmata o innescata da alcuni eventi specifici.

Per definire un task Fargate è sufficiente definire la dimensione in CPU e memoria del task e i container che devono essere eseguiti (nome e repository da cui recuperare l'immagine). È possibile specificare variabili di sistema e volumi da collegare ai container per salvarnelo stato qualora non fosse necessario utilizzare sistemi di storage specifici. Ad ogni lancio di un task, è inoltre possibile fare l'ovveride delle variabili di sistema, fare mapping di porte e collegare volumi, esattamente come avviene per il lancio di un container. Possono anche essere usate configurazioni di network e di sicurezza specifiche. Una volta creata la `Task Definition`, un container che esegue una definizione prende il nome di

`Fargate Task`.

AWS offre anche il servizio Amazon Elastic Container Service, un servizio dedicato per orchestrare cointainer basato su Kubernetes.

3.2 Integration

3.2.1 API Gateway

API Gateway è un servizio che permette di sviluppare API a qualsiasi scala. Funge da front door tra l'applicazione e la business logic. È in grado di processare automaticamente migliaia di chiamate API concorrenti, gestisce l'autorizzazione e l'access control alle risorse, il monitoring e il versioning dell'API.

3.2.2 Event Bridge

La maggior parte dei servizi AWS permette di utilizzare event per richiamarne altri. Qualora non fosse possibile collegare servizi tramite determinati eventi o si vogliano utilizzare eventi generati da servizi esterni a AWS, si può ricorrere ad Event Bridge.

Event Bridge è un bus di eventi serverless che permette di definire regole con filtri su ogni tipo di evento. Quando un incoming event viene matchato con una regola definita, l'event viene passato al servizio target specificato nella regola.

3.2.3 Step Functions

Step Function permette di coordinare facilmente complesse sequenze di task, gestendo errori e logiche di retry. Questo permette di separare la business logic dalla workflow logic dell'applicazione. I workflow vengono definiti collegando step diversi tra loro. Una Step Function è quindi la rappresentazione logica del workflow di una certa parte dell'applicazione. Non esegue effettivamente alcun lavoro, ma definisce la coordinazione di più parti o step all'interno dell'applicazione esattamente come le state machines vengono utilizzate per modellare e definire algoritmi.

Ogni step rappresenta un task che può essere svolto da diversi servizi di computazione (Lambda e EC2 tra i più utilizzati). Ciascuno step comunica un output allo step successivo. È inoltre possibile avere pieno controllo sulla gestione degli errori. Ogni errore definito da

un utente può essere gestito da un branch diverso della step function, mentre errori generali possono essere utilizzati per garantire una completa gestione delle eccezioni.

Ogni step function viene rappresentata da una state machine che oltre ad offrire una intuizione visiva sul comportamento della funzione, permette di osservarne in tempo reale il workflow.

Definizione State Machine

La state macine di una Step Function viene definita utilizzando il linguaggio ASL (Amazon State Language) basato su JSON.

La definizione di ogni Step Function deve avere un top level field `StartAt` che definisce lo stato di inizio e un campo `States` che definisce tutti gli stati che possono essere raggiunti.

State definition Ogni stato deve specificare il tipo di esecuzione, che può assumere uno dei seguenti valori

- `Task` - Lo stato esegue del lavoro vero e proprio, generalmente una lamba
- `Choice` - In base allo stato precedente, l'esecuzione viene indirizzata verso uno specifico branch
- `Fail` o `Succeed` - L'esecuzione viene interrotta con un fallimento o successo
- `Pass` - L'output di uno stato precedente viene passato in input al successivo, aggiungendo eventualmente dati
- `Wait` - Pausa l'esecuzione per un periodo di tempo specificato
- `Parallel` - Esegui parallelamente i branch collegati allo stato
- `Map` - Definisce uno step di iterazione che verra eseguito ripetutamente su i dati in input

Ciascuno stato deve definire anche la sua terminazione, questo avviene definendo il campo `Next` che passa il controllo ad uno stato successivo (lo stato `Choice` permette multipli campi `Next`) oppure definendo il campo `End` settato a `true` che definisce la terminazione di un certo branch.

Altri stati opzionali come `InputPath` e `OutputPath` servono per selezionare la porzione di dati specifica da prendere in input dallo stato precedente e da passare in output allo stato successivo. Generalmente i dati vengono passato come JSON.

Per specificare quale parte di dati selezionare si specifica il path del JSON partendo dall'operatore `$` che rappresenta la root del JSON di dati (ad esempio `$.data.name`).

Error handling Le Step Functions permettono di gestire gli errori con molto flessibilità, per farlo è sufficente specificare il campo `Retry` in cui si gestisce un array di error handler. Ogni error handler definisce il nome dell'errore con il campo `ErrorEquals`, un `IntervalSeconds` da attendere prima di tentare nuovamente l'esecuzione, una `BackoffRate` e numero di `MaxAttempts`.

Inoltre è possibile definire un fallback se lo step continua a terminare con errori. In questo caso il comportamento dell'erorr handler viene definitito in un campo `Catch` in cui è richiesto il nome dell'errore e lo stato successivo a cui passare il controllo (nel campo `Next`).

AWS metta a disposizione alcuni error codes di default con cui è possibile gestire diversi scenari, i piu utilizzati sono

- `States.Timeout` - Quando uno step non termina entro il timeout definito nello stato
- `States.TaskFailed` - Lo stato è fallito durante l'esecuzione
- `States.Permissions` - Uno stato non ha sufficenti permessi per eseguire le operazioni definite
- `States.BranchFailed` - Il branch di uno step parallelo è fallito
- `States.NoChoiceMatched` - Non è stato trovato alcun branch in uno stato di choice
- `States.ALL` - Wildcard che matcha ogni stato

3.2.4 Simple Queue Service

Simple Queue Service (Amazon SQS) è una coda di messaggistica distribuita che permette di scollegare e scalare liberamente microservizi, sistemi distribuiti e applicazioni

serverless. Con SQS è possibile inviare, memorizzare e ricevere messaggi tra diversi servizi ad alto volume.

SQS è costituita da alcuni componenti: servizi producer che inseriscono messaggi nella coda, servizi consumer che estraggono messaggi e infine la coda stessa che memorizza i messaggi con ridondanza su molteplici server SQS. SQS supporta la maggior parte dei servizi AWS come producer e consumer. Quando un consumer è pronto a processare un messaggio, il messaggio viene nascosto agli altri consumer per un tempo pari al `VisibilityTimeout` e viene poi rimosso dalla coda.

Il messaggio da inserire nella coda supporta un message body, in cui possono essere memorizzati fino a 265 kb e message attributes per definire metadati per il messaggio.

SQS offre code di tipo standard, le quali non danno garanzie sull'ordine di distribuzione dei messaggi ma che garantiscono massimo throughput, e code FIFO che invece garantiscono che ogni messaggio venga processato esattamente una volta, nell'ordine in cui è stato inserito nella coda.

Nel caso di code FIFO, i messaggi possono definire anche un message group, questo permette a AWS di processare i messaggi di uno stesso message group in order. Mentre l'ordine non viene strettamente rispettato quando i messaggi appartengono a diversi message groups.

SQS inoltre supporta un servizio chiamato `dead-queue-letter`. Una dead-queue-letter è un'altra coda dello stesso tipo della principale che serve per gestire i messaggi che non sono stati processati correttamente. La code supportando automaticamente un retry nella distribuzione dei messaggi, con un valore di default per il retry pari a 3. Se un messaggio fallisce nell'essere processato per più di tre volte viene inserito nella dead-queue-letter, questo è utile per operazioni di debug.

3.3 Data Store

3.3.1 Aurora

Aurora è un database engine relazionale compatibile con MySQL e PostgreSQL, fino a cinque volte più veloce di database MySQL standard. Utilizza S3 internamente per diverse operazioni di fault-tolerance, self-healing e auto-scaling fino a 128TB per istanza.

3.3.2 S3

Simple Storage Service (S3) è un servizio di storage di oggetti che offre un'elevata scalabilità, data availability e sicurezza. È possibile configurare automaticamente operazioni di backup, restore, archive e versioning del contenuto. Offre un durabilita del 99.999999999%.

3.3.3 DynamoDB

DynamoDB è un database NoSQL key-value in grado di garantire tempi di risposta nell'ordine delle unità di millisecondo a qualsiasi scala. Supportando fino a 20 millioni richieste al secondo. Nasce come database interno, creato per gestire gli elevati carichi di Amazon.

Per renderlo cosi performante, sono state introdotte ottimazioni a livello del sistema operativo e sono state implementati miglioramenti derivanti dalla ricerca in sistemi distribuiti. DynamoDB è il primo prodotto ad implemetare in un solo prodotto

- Consistent hashing - Per garantire una scalabilità incrementale
- Versioning - Utilizzando vector clocks
- "Sloppy quorum" - Algoritmo di consenso ad hoc per massimizzare l'aviability anche con alcune repliche offline
- Anti-entropy based recovery - Utilizza Merkle trees per sincronizzare repliche divergenti

Dynamo offre un'eventual consistency di default o strong consistency se configurato. Inoltre, garantisce il modello ACID per tutte le transazioni. Essendo un database di tipo SQL le query sono implemetate su una API di tipo PUT e GET.

3.3.4 DocumentDB

DocumentDB è database orientato ai documenti compatibile con MongoDB, replica i dati su sei diverse istanze in tre diverse zone AWS (un indice primario e cinque repliche). Le transazioni sono di tipo atomico sul singolo documento, riuscendo cosi a garantire una strong consistency. A differenza di DynamoDB richiede la configurazione manualmente di un cluster.

3.4 Sviluppo con AWS

3.4.1 IAM

IAM (Identity Access and Managment) è il servizio AWS utilizzato per assegnare granular permissions alle risorse. Alcuni entita di IAM importanti sono

- `Users` - Ogni utente che si collega ad AWS deve avere un'account personale
- `Group` - Gruppi di utenti possono essere raggruppati per assegnare permessi in comune
- `Polices` - Definite in formato JSON, permettono di specificare i permessi sulle risorse.
- `Role` - Viene assegnato ad un servizio AWS, in base alle policies associate al role, il servizio può interagire o meno con altri servizi. AWS consiglia di assegnare un ruolo ad un solo servizio.

Con le policies è possibile configurare le modalità di accesso ad una risorsa. Per definire una policy AWS offree tre tabelle: policies summary, services summary e actions summary. Si seleziona una servizio dalla policies table, questo sarà il servizio a cui la policy verrà applicata. Dalla tabella dei servizi si seleziona una lista di azioni da associare alla policy. Infine, per ogni action, si seleziona a quale risorsa applicare l'azione.

3.4.2 VPC

Uno strumento per offrire ulteriore sicurezza ai servizi AWS sono i VPC (Virtual Private Cloud) che permettono di eseguire applicazione all'interno di rete virtuali definite dall'utente. Sono astrazioni virtuali di una rete tradizionale, con tutti i benefici di elasticità che AWS generalmente offre.

Per ogni VPC è possibile configurare alcuni parametri

- `Subnet` - Range di IPs all'interno del VPC
- `Route table` - Regole di routing per configurare il traffico
- `Internet gateway` - Gateway per permettere al VPC di collegarsi ad internet

Servizi nello stesso VPC possono vedersi l'un l'altro, mentre per servizi in diversi VPC è necessario configurare le opzioni di routing in maniera opportuna.

Capitolo 4

Case studies

Sommario

4.1 Coinbase

4.1.1 Step Functions deployment

Coinbase è una piattaforma che permette di vendere e comprare cryptocurrencies. Ho deciso di analizzare uno dei loro tool open source Odin, per verificare come gli strumenti di AWS, in questo caso Lambda e Step Functions possano essere utilizzati per semplificare la pipeline di deployment.

Odine è stato rilasciato su github nel 2018, è il tool utlizzo per il deployment su AWS. Il processo di deployment di nuove realese segue generalmente gli stessi steps. Quando una nuova versione è pronta per essere rilasciata vengono eseguiti i tests, il codice/container vengono rimpiazzati con i nuovi e se la versione non presenta bug nell'immedato si considera il deployment un successo. Se c'è qualche problema con la nuova versione occorre effettuare un rollback.

Questo è quello che Odin svolge, come si può vedere dalla definizione della state machine.

In particolare gli step che esegue sono

- `Validate` - Controlla che la release sia corretta
- `Lock` - Ottiene un lock per evitare deployment concorrenti dello stesso progetto
- `ValidateResources` - Controlla che le risorse per il progetto siano valide
- `Deploy` - Crea cluster EC2 e altri servizi AWS
- `CheckHealthy` - Dopo il deploy, alcuni stati di wait attendono che la release finisca in produzione. Lo step successivo controlla che la release funzioni correttamente.
- `CleanUpSuccess` - Se la release funziona, vengono eliminate le vecchie istanze
- `CleanUpFailure` - Se la release non funziona, vengono eliminate le istanze nuove

In questo modo, essendo tutti gli step della pipeline gestiti automaticamente, non è necessario l'intervento umano per lanciare il deployment ed eventualmente gestire gli errori.

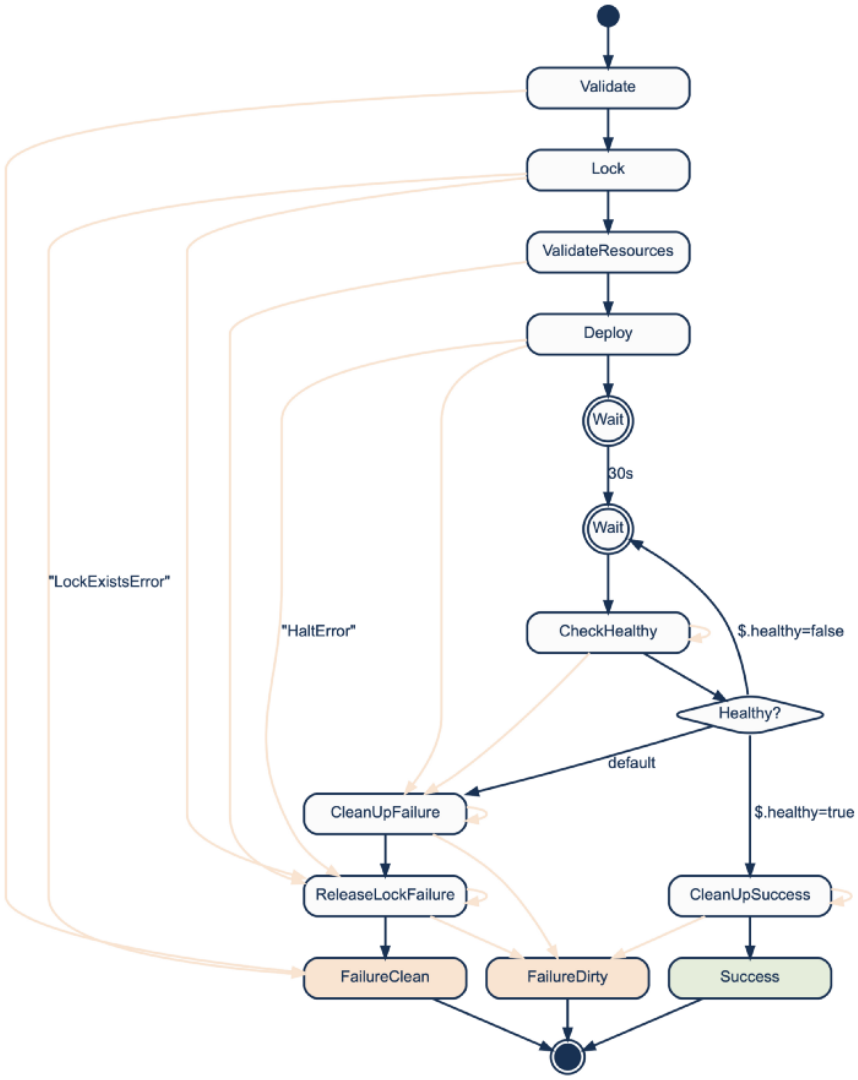

Figura 4.1: Odin State Machine

4.2 Reuters

Reuters è il più grande provider di news multimediali, offre servizi a miliardi di utenti ogni giorno tra cui BBC, CNN, The New York Times e The Washington Post. Di seguito una breve descrizione di come Reuters utilizza AWS SQS per semplificare la loro piattaforma di distribuzione eventi.

4.2.1 SQS rimpiazza WebSockets

Generalemente, quando devono essere inviati molteplici eventi generati da diverse sorgenti ad un client, diverse techinche possono essere utilizzate, ad esempio polling e websoc-

kets. Le web sockets vengono utilizzate per garantire un interazione real time senza troppe complicazioni aggiuntive, evitano ad esempio il sovraccarico del server che un implementazione di tipo polling porterebbe. È però comunque necessario gestire la connessione socket per ogni client e i server che offrono la connessione.

Per rimuovere queste operazioni di gestione, Reuters ha deciso di utilizzare AWS SQS e AWS Cognito (servizio che gestisce operazioni di authorization e authentication).

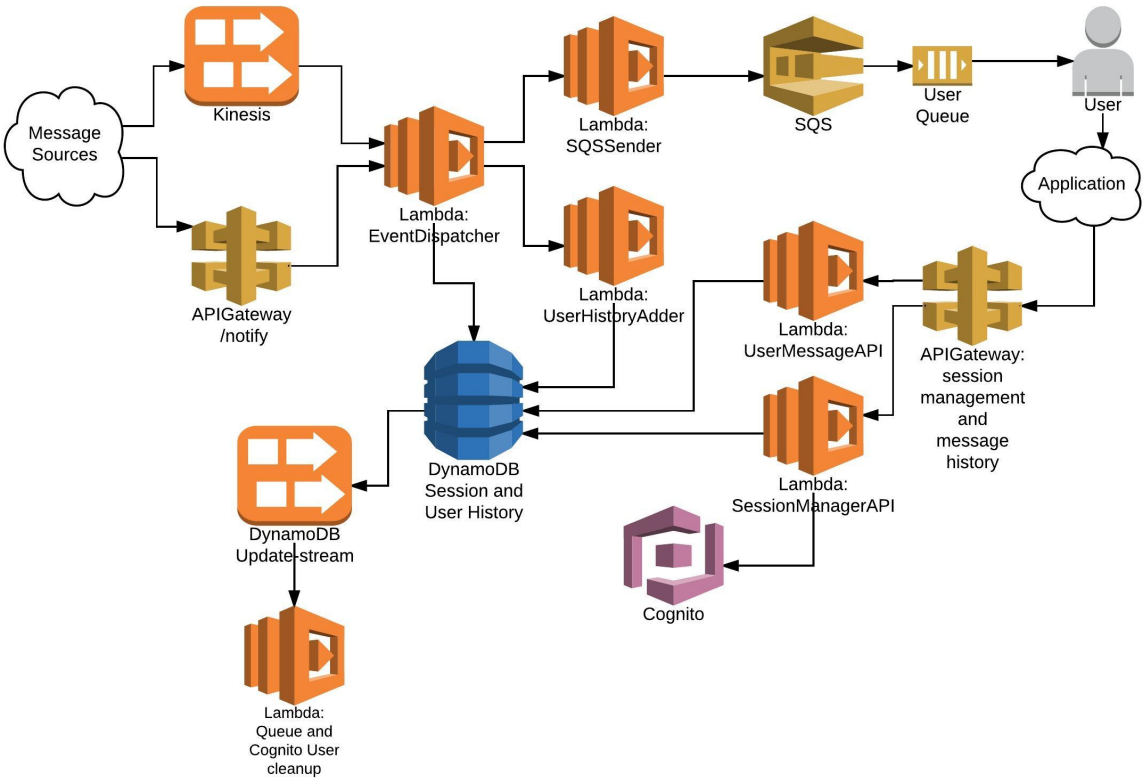

Figura 4.2: Architettura AWS di Reuters

Per fare il push di un evento ad un browser, viene associato una coda SQS ad ogni sessione di un client. Subito dopo l'autenticazione, Cognito assegna i permessi al client per accedere alla coda. Le code di AWS supportano long polling, con un intervalli di timeout pari a 20 secondi. In questo modo il client puo ricevere i messaggi non appena sono stati prodotti dalle sorgenti, senza appesantire il server con polling diretto. La scalabilità e l'elasticitàdelle SQS di AWS permette a Reuters di suportare il broadcast di eventi a diverse centinaia di migliaia di utenti per broadcasting topic.

Capitolo 5

Demo

Sommario

5.1 Demo

Dopo aver studiato nel dettaglio gli strumenti serverless di AWS ho deciso di modificare un progetto a cui avevo lavorato qualche tempo fa e renderlo serverless. Il progetto in questione consiste nello scaricare una playlist Spotify cercando file mp3 tra diversi servizi online che offrono di scaricare musica.

Inizialmente l'applicazione era costituita da un frontend in Reactjs e un backend in Nodejs. Ho potuto riutilizzare il frontend, mentre ho creato un nuovo backend utilizzando una architettura serverless.

Il workflow è il seguente: dal frontend vengono inviate un insieme di titoli da scaricare, il backend crea un job manager per cercare e scaricare i file. Una volta scaricati i file crea un archivio zip e lo restituisce al client.

Il problema che ho riscontrato utilizzando una architettura monolitica riguarda la gestione degli archivi di file. Il framework che ho utilizzato, Nodejs, è basato su l'event loop, è quindi in grado di gestire tante richieste semplici come possono essere chiamate API, mentre non è ideale per svolgere task che richiedono un intensivo uso di risorse.

Di seguito spiego come ho risolto il problema utilizzando una architettura serverless risolvendo il problema delle performance.

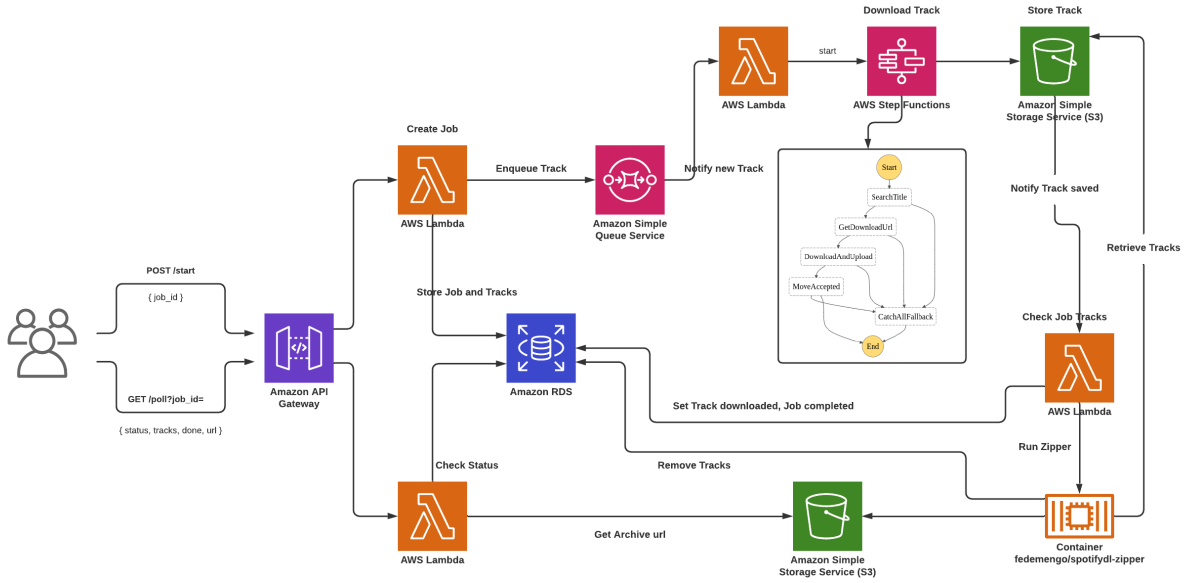

Figura 5.1: Rappresentazione dell'architettura serverless

Il client si interfaccia ad API Gateway, in cui sono definiti due endpoint

- `/start`
- `/poll`

L'endpoint start chiama una prima funzione Lambda che crea una entry nel database Amazon Aurora e associa la lista di titoli ad un jobId. Una volta creata la entry, inserisce in una coda SQS i titoli da ricercare uno ad uno.

```
handle(event: API_Gateway)
    titles = event.getTitles()
    jobId = new JobId()
    db.newJob(jobId, titles)

    foreach title in titles
        sqs.enqueue({title, jobId})
    end
end
```

Un'altra Lambda è configurata in modo da essere lanciata quando un nuovo messaggio viene inserito nella coda. In questo modo, non appena un titolo viene inserito nella coda, la funzione estra i dati del titoli e li passa allo step successivo.

```
handle(event: SQS_Message)
    title = event.getTitle()
    job_id = event.getJobId()

    if db.downloaded(title)
        db.setDownloaded(title, job_id)
        return
    end

    stepFunction.execute(title)
end
```

Il passo successivo è costituito da una Step Function che esegue la ricerca, il download e l'upload del file mp3 del titolo.

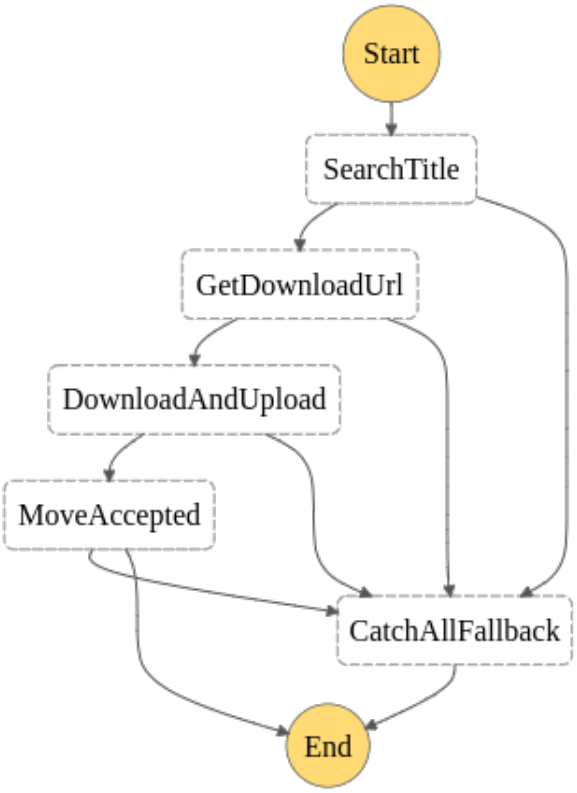

Figura 5.2: State machine della Step Function

Utilizzando le Step Functions ho potuto gestire automaticamente diversi errori e avere un workflow definito in ogni punto. Se la ricerca fallisce e nessun risultato viene ritornato la Step Function riproverà altre due volte a ricercare il file. Per ogni step di retry la Step Function attende un intervallo di 1 secondo con un BackoffRate pari a 2. Se al terzo tentativo la funzione termina con un errore, il controllo viene passato ad uno step di graceful termination in cui vengono eliminate le risorse utilizzate.

```
step1_handle(event: Lambda_Trigger, attempt: Retry)
    if attempt.Count = 3
        throw Error
    end

    title = event.getTitle()
    url_data = service1.search(title)
    if url_data is Null
        throw NotFound
    end

    return url_data
end
```

Se la ricerca ha successo, le informazioni trovate vengono passate allo step successivo che si occupa di recuperare un download url per scaricare il file. Questa Lambda prende un link ad una pagina web, riempie alcuni form ed estrae il link da cui è possibile accedere al file. Se l'url non viene trovato o si verificano altri errori, il meccanismo di retry tenterà per altre due volte ad eseguire la funzione. Se l'url viene recuperato correttamente, viene passato allo step successivo. In caso di errore, ci si sposta allo stato di graceful termination.

```
step2_handle(event: Lambda_Trigger, attempt: Retry)
    if attempt.Count = 3
        throw Error
    end

    url_data = event.getUrlData()
    url = service1.getDownloadUrl(url_data)
    if url is Null
        throw ErrorCreatingDownloadUrl
    end

    return url
end
```

Ho deciso di dividere in più step le operazioni di ricerca e di recupero dellurl di download per avere maggiore flessibilità nel caso fosse necessario sostituire il servizio.

Lo step successivo si occupa di scaricare il file e caricarlo su un bucket S3 per poter poi essere processato, in questo step vengono gestiti errori di download e upload e in più vengono effettuati controlli sulla dimensione del file. Se il download o l'upload falliscono, la funzione viene eseguita altre tre volte. Nel caso si verifichi un errore in tutti i tentativi, la funzione si sposta allo stato di graceful termination. Ho configurato inoltre un soft limit sulla dimensione del file. Nel caso in cui il file non abbia una dimensione abbastanza grande, si tenta il download con un url diverso.

```
step3_handle(event: Lambda_Trigger, attempt: Retry)
    if attempt.Count = 3 and attempt.Error is ErrorDownloadingFile
        throw Error
    end

    url = event.getDownloadUrl()
    file = service1.download(url)
    if file is Null
        throw ErrorDownloadingFile
    end

    metadata = S3.upload("/files", file)
    if metadata.Succeded is True and metadata.size < MIN_ACCEPTED_SIZE
        throw SmallSize
    end

    return {True, metadata.Key}
end
```

Al termine dei tre retry, se la dimensione non è maggiore del soft limit, si continua con il file trovato e lo si salva in S3.

```
step4_handle(event: Lambda_Trigger, attempt: Retry)
    if attempt.Count = 3
        throw Error
    end

    if event.Succeded is True
        metadata = S3.move("/files", event.Key, "/jobs")
        if metadata.Succeded is False
            throw ErrorStoringFile
        end
    end
end
```

Un'altra lambda viene lanciata non appena un file viene caricato sul bucket `/files` di S3. La lambda scrive sul database di Amazon Aurora che il titolo è stato scaricato e controlla se il file completi qualsiasi job in corso. Se un job è stato completato, la lamba recupera l'id del job e invoca un Fargate task passando il jobId.

```
handle(event: S3_Upload)
    title = event.getTitle()
    db.downloaded(title, True)
    completed_jobs = db.getCompletedJob()

    foreach job_id in completed_jobs
        Fargate.run(Archiver, job_id)
    end
end
```

Il Fargate task si occupa di recuperare tutti i file per il job assegnato e crea l'archivio zip da restituire al client. Per questo task ho deciso di utilizzare Go per assicurare ottime performance durante la gestione di files. Il task scarica tutti i file associati ad un job, create un archivio zip, carica il file in un altro bucket `/jobs` su S3 e scrive sul database

Aurora che il job è stato completato inserendo l'url dellarchivio.

Questo completa il workflow iniziato dalla chiama API a `/start`.

Una volta che il client effettua la prima chiama a `/start` passando la lista di titoli da scaricare, la Lambda risponde con il jobId appena creato. In questo modo il client può mettersi in polling sull'endpoint `/poll` passando il jobId. A questo endpoint viene associata, tramite API Gateway, un'ulteriore Lambda che invia feedback al client sullo stato del job. Una volta che il job è terminato, la lambda restituisce l'url del file zip da S3 e il polling termina.

```
handle(event: API_Gateway)
    job_id = event.getJobId()
    titles = db.findDownloaded(job_id)
    total = db.getTotalTitles(job_id)
    url = db.getUrl(job_id)
    response = {}
    response.titles = titles
    response.status = "processing"

    if size(titles) = total
        response.status = "finalizing"
    end

    if url not Null
        response.status = "done"
        response.url = url
    end

    return response
end
```

Capitolo 6

Conclusioni

Con questo lavoro ho voluto analizzare a grande linee il trend del cloud computing, in particolare il modello serverless e come si relaziona al nuovo metodo architetturale dei microservizi. In particolare, i sistemi serverless vantano un efficente utilizzo delle risorse data la loro elasticità, possono scalare automaticamente senza il bisogno di allocare risorse hardware explicitamente ed inoltre eliminano, per il cliente, la necessità di dover gestire l'infrastruttura su cui questi sistemi vengono costruiti.

Ho poi analizzato nello specifico gli strumenti AWS che possono essere utilizzati per sviluppare applicazioni serveless. AWS offre un'ampia scelta di servizi utilizzabili per gestire le diverse parti di un applicazione.

Lambda e Fargate sono i principali servizi di computazione, i quali sono utilizzabili in contesti diversi.

Numerosi sono i servizi dedicati all'integrazione, API Gateway, Event Bridge, Step Functions e Simple Queue Service servono per permette a servizi diversi di comunicare tra loro, utilizzando eventi, trigger, messaggi o anche solo gestendo automaticamente il passaggio di input e output tra un servizio e l'altro.

I sistemi serverless richiedono servizi di storage con diverse caratteristiche rispetto alle architetture monolitiche, Aurora è il servizio storage SQL mentre S3 viene sfruttato per memorizzare qualsiasi tipo di file. DynamoDB è la soluzione NoSQL di Amazon più efficente.

Altri servizi come IAM e VPC sono necessari per svlippare un sistema sicuro, com-

partimentato e per ridurre al minimo il rischio di esporre risorse di sistema ad utenti esterni.

Nonostante molti di questi servizi siano abbastanza recenti ed in continuo aggiornamento, molto compagnie e startup consolidate utilizzano questi servizi in produzione riscontrando notevole miglioramente in termini di prestazione e velocità di sviluppo.

Capitolo 7

Appendice

Sommario

7.1 Codice

7.1.1 Lambda - Create Job

Lambda che inserisce i titoli in una coda SQS

```
const sqlHelper = require('sql-helper')
const queryHelper = require('./query.js')
const { v4: uuidv4 } = require('uuid');

const AWS = require('aws-sdk');
AWS.config.update({region: 'us-east-1'});

const sqs = new AWS.SQS({apiVersion: '2012-11-05'});
const QUEUE_URL = process.env.QUEUE_URL;

exports.handler = async (data) => {
    const tracks = data.tracks
    const tracksValues = tracks.map(track => `("${jobId}", "${track}", 0)`).join(',')
    const jobId = uuidv4()

    await sqlHelper.query(queryHelper.CREATE_TRACKS_TABLE_STMT)
    await sqlHelper.query(queryHelper.CREATE_JOB_TABLE_STMT)
    await sqlHelper.query(queryHelper.INSERT_JOB_STMT + `("${jobId}", NULL, 0)`)
    await sqlHelper.query(queryHelper.INSERT_TRACKS_STMT + tracksValues)

    const sentMessages = []
    for(let track of tracks) {
        const messageParams = {
            QueueUrl: QUEUE_URL,
            MessageBody: track,
        }
        try {   // Add tracks to queue
            const sendSqsMessage = await sqs.sendMessage(messageParams).promise()
            sentMessages.push(sendSqsMessage)
        } catch {}
    }

    return {
        id: jobId,
        tracks_count: tracks.length,
        messages: sentMessages
    };
};
```

7.1.2 Lambda - Start Download

Lambda usata per eseguire la Step Function

```
const AWS = require('aws-sdk');
AWS.config.update({region: 'us-east-1'});

const STEP_FUNC_ARN = process.env.STEP_FUNC_ARN

const execute = async params => new Promise((res, rej) => {
    const stepfunctions = new AWS.StepFunctions()

    stepfunctions.startExecution(params, (err, data) => {
        if (err) return rej(err)
        return res(data)
    })
})

exports.handler = async (event) => {
    const message = event.Records[0]
    const track = message.body
    console.log("Downloading", track)

    const params = {
        stateMachineArn: STEP_FUNC_ARN,
        input: {title: track}
    }

    await execute(params)

    return {
        statusCode: 200,
        body: {
            track: track,
        },
    };
};
```

7.1.3 Step Function - Download and Upload

Definizione della State Machine per la Step Function

```
{
  "Comment": "Description of the Step Function",
  "StartAt": "SearchTitle",
  "States": {
    "SearchTitle": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:SearchTitle",
      "Next": "GetDownloadUrl",
      "Retry": [{
          "ErrorEquals": [ "EmptyListError" ],
          "IntervalSeconds": 1,
          "BackoffRate": 2.0,
          "MaxAttempts": 2
       }],
       "Catch": [{
          "ErrorEquals": [ "States.ALL" ],
          "Next": "CatchAllFallback"
       }]
    },
    "GetDownloadUrl": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:GetDownloadUrl",
      "Next": "DownloadUpload",
      "Retry": [{
          "ErrorEquals": [ "DownloadUrlNotFound" ],
          "IntervalSeconds": 1,
          "BackoffRate": 2.0,
          "MaxAttempts": 2
       }],
       "Catch": [{
          "ErrorEquals": [ "States.ALL" ],
          "Next": "CatchAllFallback"
       }]
    },
    "DownloadUpload": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:DownloadUpload",
      "Next": "MoveAccepted",
      "Retry": [{
```

```
            "ErrorEquals": [ "DownloadError" ],
            "IntervalSeconds": 1,
            "BackoffRate": 2.0,
            "MaxAttempts": 2
        }, {
            "ErrorEquals": [ "SmallSize" ],
            "IntervalSeconds": 1,
            "BackoffRate": 2.0,
            "MaxAttempts": 2
        }],
        "Catch": [{
            "ErrorEquals": [ "SmallSize" ],
            "Next": "MoveAccepted"
        }, {
            "ErrorEquals": [ "States.ALL" ],
            "Next": "CatchAllFallback"
        }]
    },
    "MoveAccepted": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:MoveAccepted",
      "End": true,
      "Retry": [{
            "ErrorEquals": [ "StoringError" ],
            "IntervalSeconds": 1,
            "BackoffRate": 2.0,
            "MaxAttempts": 3
        }],
        "Catch": [{
            "ErrorEquals": [ "States.ALL" ],
            "Next": "CatchAllFallback"
        }]
    },
    "CatchAllFallback": {
      "Type": "Pass",
      "Result": "This is a fallback from any error code",
      "End": true
    }
  }
}
```

7.1.4 Lambda - Search Title

Servizio ad hoc per il download di file mp3

```
const cheerio = require('cheerio');
const fetch = require('node-fetch');
const { mapKeys } = require('lodash');

const SEARCH_ENDPOINT = process.env.SEARCH_ENDPOINT

const extractList = rawBodyHTML => {
    const keysMapping = {
        video_id: 'vid',
        title: 'fn',
        is_playlist: 'pl'
    };

    const $ = cheerio.load(rawBodyHTML);
    try {
        // hopefully they won't change the html format
        return $('form[class=result-form]')
            .get()
            .map(form =>
                form.children
                    .filter(x => x.type == 'tag' && x.name == 'input')
                    .map(x => x.attribs)
                    .map(a => ({ [a.name]: a.value }))
            )
            .map(attrArray =>
                attrArray.reduce((base, nextAttr) => ({
                    ...base,
                    ...nextAttr
                }))
            )
            .filter(song => song.is_playlist === 'False')
            .map(song => mapKeys(song, (v, k) => keysMapping[k]));
    } catch (e) {
        console.log('[extractList] An error occurred ' + e);
        return [];
    }
};

const search = async title => {
```

```
40      const encodedTitle = encodeURI(title);
41      console.log("Search", encodedTitle)
42      try {
43          const rawResponse = await fetch(
44              `${SEARCH_ENDPOINT}?search_query=${encodedTitle}`, {
45              method: 'GET'
46          });
47
48          const htmlResponse = await rawResponse.text();
49          const data = extractList(htmlResponse);
50
51          console.log(`[searchSong] received DATA for ${title}`);
52          return data;
53      } catch (e) {
54          console.log('[searchSong] An error occurred ' + e);
55          return [];
56      }
57  };
58
59  exports.handler = async (event, context, callback) => {
60      const title = event.title
61      console.log("Title", title)
62
63      let songsList = []
64      try {
65          songsList = await search(title)
66      } catch {}
67
68      if(songsList.length == 0) {
69          throw new GenericError('EmptyListError', 'no tracks found');
70      }
71
72      return {
73          statusCode: 200,
74          list: songsList,
75          title: title,
76      };
77  };
```

7.1.5 Lambda - Get Download Url

```
const fetch = require('node-fetch');
const FormData = require('form-data');

const DOWNLOAD_ENDPOINT = process.env.DOWNLOAD_ENDPOINT
const GET_FILE_ENDPOINT = process.env.GET_FILE_ENDPOINT

const getDownloadUrl = async song => {
    const formData = new FormData();
    formData.append('title', song['fn']);
    formData.append('is_playlist', 'False');
    formData.append('video_id', song['vid']);
    formData.append('type', 'MP3');

    try {
        await fetch(DOWNLOAD_ENDPOINT, {
            method: 'POST',
            redirect: 'manual',
            headers: { 'Access-Control-Expose-Headers': 'Location' },
            body: formData
        });
    } catch (e) {
        console.error(e);
        throw e;
    }

    const queryString = new URLSearchParams();
    queryString.set('fn', song['fn']);
    queryString.set('pl', song['pl']);
    queryString.set('vid', song['vid']);
    queryString.set('dt', 'MP3');

    return `${GET_FILE_ENDPOINT}?${queryString.toString()}`;
}

exports.handler = async (event) => {
    const title = event.title
    const list = event.list
    const track = list[0]

    let url
    try {
```

```
42          url = await getDownloadUrl(track)
43      } catch {}
44
45      if(!url) {
46          throw new GenericError('DownloadUrlNotFound', 'no download url found');
47      }
48
49      return {
50          statusCode: 200,
51          title: title,
52          url: url,
53      };
54  };
```

7.1.6 Lambda - Download Upload

```
const fetch = require('node-fetch');
const AWS = require('aws-sdk');
AWS.config.update({region: 'us-east-1'});
const s3 = new AWS.S3()

const BUCKET_NAME = process.env.BUCKET_NAME
const MB = 1024 * 1024

const download = async (title, url) => {
    const { body } = await fetch(url, { method: 'GET'});
    return { data: body, name: title };
}

const s3upload = params => new Promise((res, rej) => {
    s3.upload(params, function(error, data) {
            if (error) return rej(error)
            return res(data)
    });
})

const upload = async params => {
    try {
        const result = await s3upload(params)
        const url = result.Location
        const data = await fetch(url, { method: 'HEAD' });
        const size = data.headers.get('content-length')
        return { size, url }
    } catch {
        return { size: -1, url: null }
    }
}

exports.handler = async (event) => {
    const title = event.title
    const songUrl = event.url

    let name, data
    try {
        { name, data } = await download(title, songUrl);
    } catch(e) {
        throw new GenericError('DownloadError', 'download error');
```

```
42      }
43
44      const params = {
45              Bucket: BUCKET_NAME,
46              Key: name + '.mp3',
47              Body: data,
48              ContentType: 'audio/mp3',
49              ACL: "public-read",
50      };
51
52      let size, url
53      try {
54          {size, url} = await upload(params)
55          size = result.size
56          url = result.url
57      } catch(e) {}
58
59      if(size < 3 * MB) {
60          throw new GenericError('SmallSize', `size of ${size / MB} MB is too small`);
61      }
62
63      return {
64          statusCode: 200,
65          size: size,
66          url: url,
67          source: `/${params.Bucket}/${params.Key}`,
68          key: params.Key,
69          track: title,
70          downloaded_successfully: size > 0
71      };
72  };
```

7.1.7 Lambda - Move Accepted

```
const AWS = require('aws-sdk');
AWS.config.update({region: 'us-east-1'});
const s3 = new AWS.S3()

const BUCKET_NAME = process.env.BUCKET_NAME

const s3copy = async params => new Promise((res, rej) => {
    s3.copyObject(params, (err, data) => {
        if (err) return rej(err)
        return res(data)
    });
})

exports.handler = async (event) => {
    const s3key = event.key
    const s3source = event.source

    const params = {
        Bucket: BUCKET_NAME,
        CopySource: s3source,
        Key: s3key
    };

    try {
        await s3copy(params)
    } catch(e) {
        throw new GenericError('StoringError', 'error storing file permanently');
    }

    return {
        statusCode: 200,
        ok: true,
    };
};
```

7.1.8 Lambda - Check Job Tracks

La funzione che lancia il Fargate task di zipping per tutti i jobs i cui file sono stati recuperati

```
const sqlHelper = require('sql-helper')
const queryHelper = require('./query.js')
const AWS = require('aws-sdk');
AWS.config.update({region: 'us-east-1'});

const CLUSTER = process.env.CLUSTER
const TASK = process.env.TASK
const CONTAINER_NAME = process.env.CONTAINER_NAME
const SUBNET = process.env.SUBNET
const ecs = new AWS.ECS();

const runTask = async (job_id) => {
    try {
        const runParams = {
            cluster: CLUSTER,
            taskDefinition: TASK,
            launchType: 'FARGATE',
            networkConfiguration: {
                awsvpcConfiguration: {
                    assignPublicIp: 'ENABLED',
                    subnets: [SUBNET],
                },
            },
            overrides: {
                containerOverrides: [{
                    environment: {
                        name: 'AWS_JOB',
                        value: job_id,
                    },
                    name: CONTAINER_NAME,
                }],
            },
        };
        await ecs.runTask(runParams).promise();

        return {job_id, error: false}
    } catch (e) {

```

```
        return {job_id, error: true}
    }
}

exports.handler = async (event) => {

    const object = decodeURIComponent(event.Records[0].s3.object.key)
    const name = object.replace('.mp3', '')
    const trackName = name.replace(/\+/g, ' ')

    await sqlHelper.query(
        `UPDATE tracks SET downloaded = 1 WHERE title = "${trackName}"`
    )
    await sqlHelper.query(
        `SELECT DISTINCT job_id FROM tracks WHERE title = "${trackName}"`
    )
    let queryResult = await sqlHelper.query(queryHelper.SELECT_COMPLETED_JOBS)

    const results = await Promise.all(queryResult.map(({job_id}) => runTask(job_id)))

    return {
        results: results,
    };
};
```

7.1.9 Lambda - Check Status

```
const sqlHelper = require('sql-helper')

exports.handler = async (event) => {
    const job_id = event.job_id

    const jobData = await sqlHelper.query(
        `SELECT * FROM tracks WHERE job_id = "${job_id}"`
    )
    const tracks = jobData.filter(({downloaded}) =>
        downloaded).map(({title}) => title
    )
    const total = jobData.length
    let url, status = 'start'

    if(tracks.length > 0)
        status = 'processing'

    if(tracks.length == total)
        status = 'finalizing'

    const zipUrlData = await sqlHelper.query(
        `SELECT url FROM jobs WHERE job_id = "${job_id}" AND ready = 1`
    )
    if(zipUrlData.length > 0) {
        status = 'done'
        url = zipUrlData[0].url
    }

    return {
        status: total ? status : "not_found" ,
        tracks,
        total,
        done: tracks.length,
        url
    };
};
```