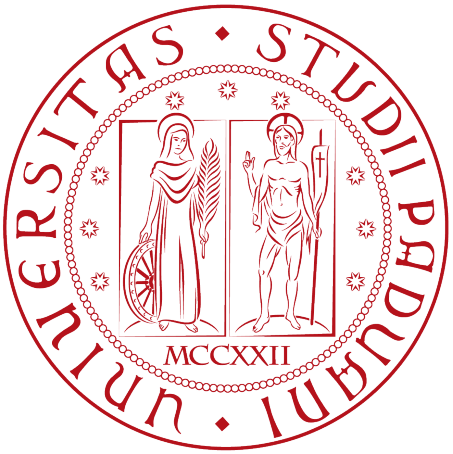
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Sviluppo di un servizio "fair queue" per l'erogazione di funzionalità in modo equo ad ogni utente

Tesi di laurea

Relatore

Prof. Ombretta Gaggi

Laureando

Alessandro Cavaliere

Matricola 1224440

Anno Accademico 2021-2022

"I computer sono incredibilmente veloci, accurati e stupidi.
Gli uomini sono incredibilmente lenti, inaccurati e intelligenti.
L'insieme dei due costituisce una forza incalcolabile."

— Albert Einstein

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine alla Professoressa Ombretta Gaggi, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.

Desidero ringraziare con affetto la mia famiglia e la mia fidanzata Sabrina per il sostegno, la vicinanza e la fiducia con cui mi hanno supportato in ogni momento durante gli anni di studio.

Infine vorrei ringraziare i miei amici, che da qui e da lassù, mi sono stati vicini e mi hanno accompagnato in questi anni, soprattutto nei momenti difficili.

Padova, Luglio 2022 Alessandro Cavaliere

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Il seguente capitolo vuole introdurre brevemente l'azienda ospitante e il progetto affrontato.

1.1 L'azienda

THRON S.p.A[1] è un'azienda italiana, nata negli anni 2000, situata a Piazzola sul Brenta. Essa opera nell'ambito dello sviluppo di *SaaS*. I servizi offerti, di *marketing* e *business intelligence*, collocano l'azienda nel settore dei *DAM (Data Asset Management)*.

L'azienda utilizza, come metodologia di lavoro, il framework *agile SCRUM*, con grande coinvolgimento degli *stakeholder*, spesso interpellati per suggerire modifiche e miglioramenti. L'area *R&D* è organizzata in due principali *team*: il *team* contenuti e il *team* prodotto. Il primo segue principalmente le tematiche legate al *DAM* e a tutte le funzionalità da lui derivate. Il secondo si occupa della gestione del *PIM (Product Information Management)* e delle funzionalità legate alle *tag* sui prodotti. Il logo dell'azienda è illustrato nella figura 1.1.

Figura 1.1: Logo THRON S.p.A.

Il prodotto di THRON è la *"Thron DAM Platform"* (da qui in poi "Thron"), una piattaforma per la gestione dei contenuti digitali. L'idea, alla base del prodotto, è la valorizzazione dei contenuti e delle informazioni sui prodotti. Da qui la necessità di separare la loro organizzazione e gestione dalla piattaforma di distribuzione finale. Il prodotto, fruibile attraverso un'applicazione web, è in grado di gestire contenuti di vario genere (documenti, video, immagini e audio).

La piattaforma si occupa della vita a 360 gradi del contenuto, mettendo a disposizione degli utenti varie funzionalità per ottimizzarne la gestione. L'obiettivo finale è quello di unificare la gestione del contenuto, evitando doppioni o perdite di dati e agevolare l'erogazione del prodotto su piattaforme di distribuzione esterne.

Una delle principali funzionalità è l'arricchimento del contenuto con *tag* e metadati: essi vengono ricavati da un motore semantico che, analizzando il contenuto, estrapola informazioni utili al cliente, al fine di rendere più semplice la gestione e l'organizzazione in categorie dei contenuti.

[1] Sito ufficiale: https://www.thron.com/

Altre funzionalità permettono di organizzare i prodotti per categorie, eseguire operazioni su di essi (anche includendo più prodotti per ogni operazione) e modificare i contenuti ad essi associati con programmi esterni. Infine, ogni contenuto viene analizzato dal reparto *marketing* aziendale, il quale fornisce consigli su come migliorarlo.

In generale l'azienda mira ad una vasta ed eterogenea clientela, per questo motivo fornisce un prodotto che si adatta alle esigenze dei clienti e non viceversa. Inizialmente il prodotto viene venduto con un solo modulo base, che offre le funzionalità standard. Attraverso un *marketplace* è possibile comprare o aggiungere moduli gratuiti che offrono diverse funzionalità ed incrementano il valore e la funzione della piattaforma.
L'azienda propone anche soluzioni personalizzate per il caso d'uso specifico del cliente. Lo stretto contatto con gli *stakeholder*, con esigenze anche molto diverse, porta ad una naturale evoluzione della piattaforma, che deve innovarsi continuamente per poterle soddisfare.

1.2 L'idea

Lo *stage* consiste nella progettazione e sviluppo di un servizio di code, volto a gestire l'erogazione delle funzionalità di Thron *DAM* Platform ai vari clienti. Il servizio deve garantire la *fairness* tra i vari *tenant*, in modo tale che tutti i clienti possano usufruire in uguale maniera delle funzionalità esposte dalla piattaforma.

Il microservizio andrà a sostituire una libreria già esistente e attualmente importata dai servizi Thron, che espone simili funzionalità a quelle che dovrebbe esporre il servizio. Tuttavia essa ha delle limitazioni, una fra tutte è la dipendenza dal linguaggio in cui è scritta: questo non permette ai servizi Thron, scritti con linguaggi non compatibili a quello della libreria, di utilizzare le funzionalità della libreria.
Il servizio dovrà esporre un'interfaccia per registrare code e per accodare, recuperare e gestire *task* via *API (Application Programming Interface)*. Questa verrà usata dai servizi Thron per gestire i processi in background.

È prevista anche l'implementazione di un'*API* che raccolga delle metriche per consentire il monitoraggio delle code definite.
Lo sviluppo deve avvenire con tecnologie *cloud-native*, tenendo in considerazione che l'esecuzione del servizio avverrà in ambienti *serverless*.

1.3 Descrizione dello *stage*

1.3.1 Introduzione al progetto

Lo *stage*, in ambito *back-end*, aveva come scopo la realizzazione di un servizio per la gestione di code. Tale servizio andrà a sostituire una libreria preesistente, utilizzata dai vari servizi di Thron. Essi, a partire dalle *API* esposte dalla libreria, definivano dei produttori e dei consumatori, che rispettivamente riempivano e svuotavano la coda definita. Nonostante il buon funzionamento, la libreria presenta delle limitazioni:

* **Dipendenza dal linguaggio**: i servizi Thron che volevano creare ed utilizzare una coda, devono essere stati codificati con un linguaggio compatibile con *Scala*, principalmente utilizzato in ambito *back-end* dall'azienda. Questo costringeva alcuni servizi a non poter utilizzare la libreria.

* **Architettura**: la libreria, per erogare le sue funzionalità, aveva bisogno di mantenere uno stato interno. Questo aspetto non si sposa bene con la filosofia *serverless* che l'azienda sta cercando di applicare alla sua piattaforma.

È principalmente per queste due ragioni che si è ritenuto utile realizzare un microservizio che la sostituisse.

Il principio fondante su cui si basa il servizio, è la garanzia della *fairness*: tutti i clienti (o *tenant*), utilizzatori di Thron, devono poter usufruire in uguale misura dei servizi di Thron. Questo aspetto è essenziale per permettere a tutti i clienti di avere accesso ai servizi della piattaforma, evitando che clienti di grandi dimensioni, con maggiori contenuti da gestire, usufruiscano in maniera totale dei servizi esposti da Thron.
Questo di fatto toglierebbe la possibilità ai clienti più piccoli, meno utilizzatori, di sfruttare le funzionalità di Thron, costretti ad aspettare la fine di lunghi processi innescati da clienti più grandi. Il microservizio da realizzare deve quindi rispettare rigorosamente il principio della *fairness*.

Per comprendere il perchè sia necessario realizzare un servizio di questo tipo, è opportuno fare un esempio pratico.
Un tipico caso d'uso della piattaforma Thron è un'operazione di tipo *batch* sui prodotti: l'operazione consiste nella selezione di tutti i prodotti con una serie di caratteristiche, per poi effettuare certe operazioni su di essi. Nel caso in cui l'operazione riguardasse molti prodotti, anche sull'ordine delle decine di migliaia, è facilmente comprensibile che se l'operazione venisse eseguita per intero, un particolare servizio sarebbe totalmente dedicato ad un singolo *tenant* per un periodo di tempo molto lungo. Si può quindi notare come l'*user experience* per tutti gli altri utilizzatori non sarebbe ottimale. Per questo motivo è necessario introdurre un meccanismo di code, che sia in grado di garantire la *fairness* in casi come questi.

Oltre a questo, il servizio deve esporre un'interfaccia per la gestione di code (registrazione, cancellazione) e *task* (accodamento, recupero, modifica) via *API*. Le code sono composte da *task*, che consistono in lavori da eseguire da parte del servizio che li produce: ogni *task* è associato ad un *client-id* e contiene un *payload* che raccoglie tutte le informazioni necessarie al servizio per processarlo. L'interfaccia verrà poi utilizzata dai vari servizi di Thron per processare i *task* in *background*, in maniera asincrona e concorrente.

Il servizio, come tutti gli altri nell'infrastruttura Thron, deve esporre delle metriche per consentire il monitoraggio delle singole code. Questo soprattutto per identificare e porre rimedio a situazioni di allarme. Le metriche devono essere esposte tramite un qualche servizio di monitoraggio, il quale dovrà fornire una *dashboard* per poterle visualizzare.

Un ultimo aspetto importante riguarda le tecnologie da utilizzare per l'implementazione del microservizio. Esso deve essere sviluppato con tecnologie *cloud-native* e *serverless*, per garantire massima scalabilità e contribuire alla migrazione aziendale su architetture *serverless*.

Per l'azienda, al momento in fase di sperimentazione su nuovi ambienti e tecnologie, questo *stage* rappresenta un modo per testare e provare nuove soluzioni, nuove tecnologie e nuovi linguaggi di programmazione. Inizialmente non si era sicuri della buona riuscita della soluzione e del soddisfacimento di tutti i requisiti. Tuttavia questa sperimentazione ha permesso all'azienda di avere *feedback* e dati reali su cui basarsi per decidere sul possibile utilizzo della tecnologie impiegate in questo progetto.

1.3.2 Obiettivi del progetto

Di seguito vengono elencati gli obiettivi di massima previsti dallo *stage*.

* Analisi progetto preesistente e inquadramento del problema;
* Redazione di uno studio di fattibilità;
* Studio del dominio e delle funzionalità del servizio (Analisi dei Requisiti);
* Sviluppo e codifica del servizio con tecnologie non largamente utilizzate dall'azienda;

* Sviluppo ed esposizione delle metriche per il monitoraggio;
* Redazione di documentazione per le *API* esposte;
* Redazione di documentazione tecnica/funzionale sulle scelte implementative fatte;
* Sviluppo e documentazione di vari scenari presi in esame per la fase di test;
* *Report* finale sui risultati ottenuti dall'esperimento.

1.3.3 Analisi preventiva dei rischi

Durante la fase di analisi iniziale, sono stati individuati dei possibili rischi a cui si poteva andare incontro durante lo svolgimento dello *stage*.
Si è quindi proceduto ad elaborare delle possibili strategie per mitigarli.

1. *Stack* tecnologico
Descrizione: le tecnologie proposte per lo sviluppo del progetto erano a me sconosciute. Questo poteva causare ritardi nella codifica e l'utilizzo di *bad practice* nel codice.
Soluzione: è stato definito un periodo iniziale di formazione personale sulle tecnologie scelte per lo sviluppo. Oltre a questo, è stata individuata una figura di un altro team, esperta nella tecnologia scelta, a cui rifarsi in caso di dubbi e perplessità.

2. Infrastruttura e servizi *AWS*
Descrizione: il progetto richiede di interfacciarsi con lo *stack* dei servizi offerti *AWS (Amazon Web Services)*, anch'esso sconosciuto. Questo potrebbe creare problemi nella configurazione dell'ambiente di sviluppo in locale prima e nel *deploy* del prodotto poi.
Soluzione: è stato effettuato il *setup* dell'ambiente di sviluppo con il tutor aziendale e l'amministratore. Sono state individuate delle figure all'interno del team a cui fare riferimento in caso vi fossero problemi.

3. Fattibilità dei requisiti di partenza
Descrizione: alcuni dei requisiti di partenza risultano essere "ambiziosi", ovvero sono stati individuati, ma non è detto che sia possibile soddisfarli in modo completo tramite il servizio.
Soluzione: si è deciso, in accordo con il tutor aziendale, di effettuare il punto della situazione settimanalmente, in modo da individuare possibili criticità ed eventualmente porvi rimedio prima che sia troppo tardi.

1.4 Organizzazione del testo

Il secondo capitolo illustra lo studio di fattibilità svolto preventivamente, in cui si vanno ad analizzare delle soluzioni al problema già presenti sul mercato, evidenziando i loro punti di forza e di debolezza.

Il terzo capitolo descrive l'analisi dei requisiti del progetto di stage, mediante la definizione dei casi d'uso e l'estrapolazione dei requisiti a partire da essi.

Il quarto capitolo approfondisce la progettazione e la codifica del progetto, esponendo successivamente le problematiche incontrate e i punti rimasti irrisolti.

Il quinto capitolo espone tutte le verifiche effettuate sul progetto durante lo sviluppo e la validazione finale, a garanzia del soddisfacimento dei requisiti inizialmente individuati.

Il sesto capitolo presenta le conclusioni tratte dallo *stage*, enunciando considerazioni, conoscenze acquisite e considerazioni personali.

Relativamente al documento, sono state adottate le seguenti convenzioni tipografiche:

* Gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune, menzionati nel documento, vengono definiti nel glossario, situato alla fine del presente documento;
* I termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Studio di fattibilità

In questo capitolo viene esposto lo studio di fattibilità, evidenziando i punti critici, vantaggi e svantaggi delle soluzioni analizzate.

2.1 Introduzione allo studio

Durante lo studio di fattibilità sono state prese in considerazione due diverse soluzioni proposte dall'azienda. La prima si concentrava sulla ricerca di soluzioni *cloud*, da usare *out-of-the-box*, così come venivano proposte. La seconda proponeva l'implementazione di un servizio completamente *custom*.

Nel primo caso, si fa riferimento a servizi di terze parti già pronti all'uso. Questo rende molto semplice l'approccio al servizio da parte dell'azienda, che deve semplicemente integrare il servizio alla propria infrastruttura, evitando la creazione e la gestione di un ulteriore servizio.

Con il secondo caso, si vogliono intendere tutte le soluzioni implementabili internamente all'azienda. Questo permette di evitare l'appoggio a servizi esterni, andando a definire una soluzione interna che, seppur più dispendiosa, permette di realizzare, personalizzare e modellare nella maniera più corretta possibile i requisiti derivanti dal problema.

Prima di proseguire con lo studio di fattibilità, è importante fissare dei metri di valutazione e di paragone, al fine di determinare la fattibilità e l'utilità di una determinata soluzione. La definizione di queste metriche è naturalmente derivata dagli obiettivi che la soluzione ideale deve avere.
Alla luce di questo, sono stati trovati dei requisiti minimi che una soluzione deve avere per essere considerata accettabile:

* ***Fairness***: il servizio deve, in qualche modo, garantire che tutti i clienti possano usufruire in modo equo dei servizi di Thron;
* **Ordinamento**: all'interno della coda deve essere possibile ordinare i *task* secondo un qualche criterio (priorità, istante di arrivo). Non è accettabile una soluzione che non preveda qualche forma di ordinamento;
* **Gestione processi lunghi**: il servizio deve essere in grado di gestire *task* molto lunghi, da suddividere in più *step* di elaborazione;
* **Gestione errori**: deve essere data la possibilità ad ogni *task* di essere riprocessato per un certo numero di volte, nel caso un processamento non andasse a buon fine;
* **Gestione code**: il servizio deve fornire la possibilità di registrare più code, almeno una per ogni servizio di Thron;
* **Monitoraggio**: il servizio deve prevedere la possibilità di registrare delle metriche che vadano a misurare lo stato di salute delle code.

2.2 Soluzioni proposte

Per ogni soluzione proposta, viene fatta una breve introduzione, seguita da considerazioni su vantaggi e svantaggi che queste porterebbero all'azienda in relazione al problema da risolvere.

2.2.1 Servizi *cloud out-of-the-box*

Per questa categoria si sono presi in considerazione 3 servizi offerti dallo *stack* di *AWS*: *Amazon SQS*[1], *Amazon SNS*[2] e *Amazon EventBridge*[3]. Tipicamente questi servizi vengono combinati tra di loro, per ottenere il miglior risultato possibile, in termini di gestione di una coda.
In azienda, al momento, solo *Amazon SQS* viene utilizzato in produzione, mentre non si ha molta familiarità con *Amazon SNS* e *Amazon EventBridge*.

Amazon SQS

Amazon Simple Queue Service è un servizio per l'accodamento di *task* completamente gestito, che consente di disaccoppiare e scalare microservizi, sistemi distribuiti e applicazioni *serverless*. Grazie ad esso, è possibile inviare, salvare e ricevere *task* in formato *JSON (JavaScript Object Notation)* tra componenti software.
Amazon SQS offre due tipi di code: le code standard e le code *FIFO (First In First Out)*. Le prime offrono la massima produttività, ordinamento *best effort* e *at-least-once-delivery*. Le seconde garantiscono l'elaborazione dei *task* esattamente una volta, nell'ordine in cui sono arrivati all'interno delle code.

Amazon SNS

Amazon Simple Notification Service è un servizio di notifiche del tipo *publish-subscribe*, che permette di gestire situazioni in cui si hanno più produttori e più consumatori. È possibile esporre dei *topic*, all'interno dei quali i *publishers* possono scrivere; il meccanismo si occuperà di notificare tutti i *subscribers* quando un nuovo *task* verrà inserito nel *topic* al quale essi si sono sottoscritti.
Questo servizio è facilmente integrabile con *Amazon SQS*. È possibile sottoscrivere delle code *SQS* a dei *topic*, in modo tale che vengano riempite non appena vi è un nuovo messaggio nel *topic*, e avere quindi un consumatore, che, in ascolto su una determinata coda *SQS*, consumi messaggi man mano che si presentano.

Amazon EventBridge

Amazon EventBridge è un servizio di notifiche simile ad *Amazon SNS*. Vi sono i *message bus*, simili ai *topic* di *Amazon SNS*, dove i *publisher* pubblicano degli eventi, costituiti dalla sorgente e dal *payload* dell'evento. È possibile definire più schemi per gli eventi, in modo tale da rendere più semplice il processo di decodifica da evento a oggetto in un certo linguaggio di programmazione.
Analogamente ad *Amazon SNS* vi sono dei consumatori che rimangono in ascolto in un particolare *message bus*. L'aspetto peculiare di questo servizio è che gli eventi possono essere dirottati a specifici *target*, seguendo delle regole precedentemente definite (i cosiddeti *event pattern*).

Aspetti positivi

* Servizi *out-of-the-box*, già implementati e facilmente integrabili con l'infrastruttura del prodotto aziendale;
* Scalabilità automatica *on-demand* e code potenzialmente illimitate;
* Implementazione del concetto di *dead letter queue*;
* Buon numero di metriche disponibili con il servizio, esposte attraverso *Amazon CloudWatch*[4].

[1] Amazon Simple Queue Service: https://aws.amazon.com/sqs
[2] Amazon Simple Notification Service: https://aws.amazon.com/sns/
[3] Amazon EventBridge: https://aws.amazon.com/eventbridge/
[4] Amazon CloudWatch: https://aws.amazon.com/cloudwatch/

Aspetti critici

* *Set* di operazioni disponibili sulla coda limitato, in particolare non c'è la possibilità di aggiornare il contenuto di una *task* all'interno della coda;
* Non vi è la possibilità di gestire dei *task* che contengono lavori lunghi da eseguire;
* Non si può processare più di un elemento alla volta, è necessario rimuovere un elemento per poter avere il prossimo;
* Se il *processing* di un elemento non va a buon fine, è necessario riaccodare il messaggio in fondo alla coda;
* Ordinamento secondo priorità e istante di arrivo non presente di *default*, è necessario utilizzare più code per definire più livelli di priorità;
* Nessun tipo di coda offerto permette la gestione, in modo autonomo, della *fairness* all'interno del servizio;

2.2.2 Servizio *custom*

Lo sviluppo di una soluzione completamente *custom* permetterebbe all'azienda di implementare tutti i requisiti necessari per soddisfare a pieno tutti gli aspetti del problema. Grazie a questo approccio è possibile analizzare singolarmente le necessità derivanti dal problema e quindi poi risolverle, creando un servizio completamente personalizzato e coerente con gli obiettivi di partenza.

Aspetti positivi

* Possibilità di personalizzazione totale del servizio, in quanto da implementare internamente;
* Maggiore flessibilità e possibilità di adattamento alle esigenze che si potrebbero presentare nel corso del tempo;
* Controllo completo sull'ambiente di rilascio;
* Presenza di un progetto di partenza da cui partire per comprendere a fondo il problema e realizzare un servizio migliorativo rispetto ad esso.

Aspetti critici

* Costi per la progettazione, lo sviluppo e il mantenimento del servizio;
* Necessità di adattare e riscrivere parte dei servizi già presenti per integrarli con quello nuovo.

2.3 Conclusioni studio

Sebbene lo *stage* prevedesse fin da subito la realizzazione di un nuovo servizio partendo da zero, è risultato molto utile analizzare anche altre soluzioni presenti sul mercato. Questo ha permesso di comprendere il perchè fosse necessario realizzare una soluzione *custom*, evidenziando quali fossero i limiti delle architetture già presenti sul mercato rispetto alle esigenze dell'azienda.
Lo studio è servito anche per capire a fondo i requisiti funzionali che il servizio avrebbe dovuto avere, evitando quindi di orientare lo sviluppo verso soluzioni che non avrebbero portato ai risultati desiderati.
Infine, questo *scouting* iniziale ha permesso l'introduzione nel servizio di alcune funzionalità inizialmente non preventivate, come lo sviluppo di una *dead letter queue*.

Dallo studio di fattibilità risulta evidente che la soluzione più corretta e ragionevole è quella di implementare un servizio *custom*.

Da una parte, è necessario sottolineare che le soluzioni offerte dal mercato offrono dei vantaggi che difficilmente sono raggiungibili con un prodotto "fatto in casa": un esempio sono i livelli di *performance* ed efficienza raggiungibili utilizzando i servizi di code disponibili con *AWS*.
Dall'altra parte è doveroso constatare che, da un punto di vista prettamente funzionale, non esiste soluzione sul mercato che implementi tutti i requisiti funzionali richiesti. Questo secondo aspetto pesa molto sulla scelta della soluzione da adottare: non è possibile scegliere una soluzione che non garantisca il soddisfacimento delle necessità primarie del problema di partenza.

Concludendo, tutti i servizi *out-of-the-box* analizzati non sono in grado di fornire una soluzione concreta a tutte le sfumature del problema di partenza, rendendo di fatto la loro applicazione inutile per risolverlo. Da questo si evince che l'unica strada percorribile è quella di implementare internamente un servizio che vada a gestire le code e i *task* al loro interno.

Capitolo 3

Analisi dei requisiti

In questo capitolo viene esposta l'analisi dei requisiti effettutata durante lo stage, nella quale si descrivono le funzionalità tramite casi d'uso e i requisiti identificati.

3.1 Analisi del servizio

3.1.1 Concetti base sulla gestione delle code

Questo paragrafo vuole illustrare dei semplici concetti che stanno alla base del servizio e delle logiche di accodamento e recupero dei *task*.

Task

Un *task* rappresenta un generico lavoro che un servizio Thron deve portare a termine. Tipicamente un servizio Thron definisce dei produttori e dei consumatori: i primi sono quelli che generano i *task* da eseguire, i secondi quelli che li eseguono effettivamente.
Un *task* è sempre associato ad un cliente tramite un *client-id* (*tenant*) ed è principalmente costituito da un *payload*, il quale contiene le informazioni essenziali per la sua esecuzione.
Dato che un *task* è sempre associato ad un cliente, è possibile implementare alcune logiche di prelievo in modo tale che venga garantita la *fairness* tra tutti i clienti che stanno usufruendo di uno specifico servizio Thron.

Prelievo *task* accodati

Un consumatore, definito da un servizio Thron, può prelevare dei *task* precedentemente accodati dallo stesso servizio. Quando avviene un prelievo, il *task* prelevato viene reso invisibile a tutti gli altri consumatori dello stesso servizio. Lo rimarrà per un periodo di tempo definito dal consumatore che sta effettuando il prelievo.
Quando un *task* viene prelevato, non viene rimosso dalla coda: è compito del consumatore che l'ha "consumato" rimuoverlo. In questa maniera si evita che due consumatori possano prelevare e processare più volte lo stesso *task*.
Nonostante ciò, il servizio segue la semantica *at-least-once-delivery*, il che significa che i *task* vengono consegnati almeno una volta, ma potrebbero essere prelevati più volte da un consumatore.

Gestione *task* in errore e *dead letter queue*

Ogni *task* ha più possibilità di essere processato, anche da consumatori diversi. Se un *task* non dovesse essere processato correttamente, può essere prelevato e processato nuovamente una volta scaduto il suo tempo di invisibilità.
Possono accadere delle situazioni in cui l'elaborazione dei *task* si concluda in stato di errore. Più generalmente, è possibile avere dei *task* le cui elaborazioni continuano a fallire. Qualora questo accadesse, non ha senso continuare a processare questi *task*. Per questo motivo, dopo un certo

numero di tentativi falliti, il *task* viene spostato in una *dead letter queue*, che raccoglie tutte quei *task* che non riescono ad essere processati correttamente entro un certo numero di tentativi.

3.1.2 Descrizione del servizio

Contesto

Il servizio verrà utilizzato internamente da THRON, all'interno della *Thron DAM platform*. I servizi della piattaforma possono utilizzare il sistema per gestire i loro lavori da eseguire, al fine di controllare l'erogazione delle funzionalità di Thron, garantendo equità nella fruizione dei servizi della piattaforma a tutti gli utilizzatori.

Funzionalità

Il servizio dovrà fornire tutte le funzionalità principali che un qualsiasi servizio di code avrebbe:

* Registrazione di una nuova coda;
* Accodamento di un nuovo *task* in una coda;
* Recupero di *task* precedentemente accodati in una determinata coda;
* Recupero delle informazioni di un determinato *task*;
* Rimozione di un *task* una volta che è stato completato.

Oltre a queste, sempre a livello di gestione di code e *task*, esso dovrà garantire la possibilità di:

* Mantenere e garantire la *fairness* tra tutti i clienti;
* Recuperare i *task* in base al loro livello di priorità;
* Gestire il tempo di invisibilità di un *task* all'interno della sua coda;
* Segnalare se un *task* non riesce ad essere eseguito correttamente;
* Eseguire un *task* in più *step* di elaborazione;
* Esporre delle metriche che mostrino lo stato delle code;

Il servizio dovrà fornire tutte queste funzionalità via *API REST (REpresentiational State Transfer)*.

3.1.3 Analisi della struttura funzionale

Il servizio può essere decomposto funzionalmente in 3 principali aree:

* Gestione delle code;
* Gestione di *task* su una coda;
* Gestione di *task* recuperati da un servizio;

A questa suddivisione, si aggiunge l'esposizione del corredo delle metriche per misurare lo stato di salute delle varie code.

Gestione code

Questa area funzionale si occupa della gestione delle code. Deve essere prevista la possibilità di registrare una nuova coda con un determinato nome. Una volta creata una coda, deve essere possibile, per i produttori di un servizio Thron, accodare dei *task* da eseguire in tale coda. Infine il servizio deve dare la possibilità, ai consumatori di un servizio Thron, di recuperare dei *task* precedentemente accodati, per poterli eseguire.

Gestione *task* su una coda

Quando un *task* è stato inserito in una coda, deve essere sempre possibile recuperarne le informazioni fondamentali come il *payload* e il *tenant* associato.
Inoltre deve essere garantita la possibilità di rimuovere un *task* da una coda in modo forzato, questo è utile in particolari situazioni.

Gestione *task* recuperate da un servizio

Un consumatore, istanziato da un servizio Thron, deve avere la possibilità di gestire un *task* che ha precedentemente recuperato.
Deve essere possibile modificare lo stato di un *task*, all'interno della coda, che è stato parzialmente eseguito.
Un consumatore deve poter prolungare il tempo per l'esecuzione del *task*, in modo da abilitarlo a finire il lavoro prima che esso torni visibile sulla coda.
Deve essere anche possibile segnalare che un *task* non è stato processato correttamente, in modo tale da poterlo rimuovere dalla coda qualora fallisse troppe volte nel futuro.
Infine il consumatore deve poter rimuovere dalla coda un *task* che è stato eseguito e completato correttamente.

Corredo delle metriche

Per misurare lo stato e il funzionamento di una coda, sono state individuate le seguenti metriche:

* *Task* accodati: il numero di *task* che sono stati accodati in una determinata coda;
* *Task* recuperati: il numero di *task* che sono stati recuperati da una determinata coda;
* *Task* rimossi: il numero di *task* che sono stati rimossi da una determinata coda;
* *Task* falliti: il numero di *task* che non sono stati eseguiti correttamente in una determinata coda;
* Recuperi vuoti: il numero di volte che un consumatore non ha potuto recuperare dei *task* poiché la coda era vuota;
* Tempo di permanenza: il tempo di permanenza di un *task* all'interno di una coda, dall'accodamento alla rimozione.

Le fonti da cui derivano le metriche sono:

* Servizi di code già presenti sul mercato. La principale fonte di ispirazione per questa tipologia è stato *Amazon SQS*, che espone un alto numero di metriche. Si sono prese solo le più rilevanti;
* Richieste e suggerimenti provenienti dall'interno del team.

Tutte queste metriche dovranno poi essere correttamente interpretate e valutate, anche utilizzando qualche tipo di aggregazione sui risultati. L'interpretazione dipende dal tipo di sistema di monitoraggio utilizzato.
Uno dei maggiori interessi dell'azienda è settare delle sonde che permettano di segnalare situazioni allarmanti in modo automatico, senza dover effettuare controlli manuali periodici. La definizione degli allarmi non è stata presa in considerazione durante lo sviluppo di questo progetto, tuttavia rappresenta uno dei punti da approfondire per l'eventuale utilizzo in produzione del servizio.

3.2 Casi d'uso

In questa sezione vengono elencati i casi d'uso rilevati nel corso dell'analisi dei requisiti del progetto. Per lo studio dei casi di utilizzo del prodotto sono stati creati dei diagrammi *UML (Unified Modeling*

Language), i quali forniscono la descrizione delle funzioni o dei servizi offerti dal sistema, così come percepiti e utilizzati dagli attori che interagiscono col sistema stesso.
Essendo il progetto incentrato sullo sviluppo di un servizio che dovrà essere utilizzato da altri servizi, non possono esservi interazioni da parte di un utente fisico, bensì le funzionalità individuate sono quelle che verranno utilizzate dai singoli servizi.

Ogni caso d'uso ha un codice gerarchico ed univoco che lo identifica, nella forma:

UC<CodicePadre>.<CodiceFiglio>

Il codice progressivo può includere diversi livelli di gerarchia separati da un punto.

3.2.1 Attori

Gli attori sono contenuti nella figura 3.1. Essendo il servizio utilizzato solamente dai servizi di Thron, l'attore principale sarà un generico servizio di Thron.

Figura 3.1: Attori individuati

Per evitare ambiguità nella descrizione dei casi d'uso, si utilizzeranno i seguenti identificativi:

* **Servizio Thron**: generico servizio esposto dalla *Thron DAM platform*;
* **Servizio** o **Sistema**: servizio di code.

3.2.2 Gestione code

Per la seguente area funzionale, i casi d'uso sono riassunti nel diagramma *UML* contenuto nella figura 3.2.

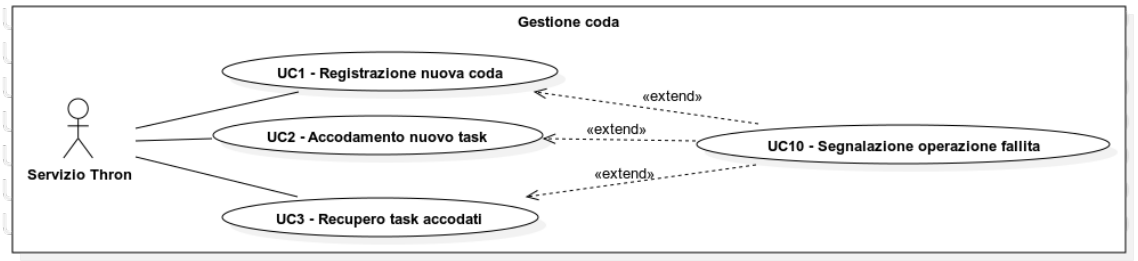


Figura 3.2: Casi d'uso - Gestione coda

UC1: Registrazione nuova coda

Il caso d'uso è raffigurato, mediante diagramma *UML*, nella figura 3.3.

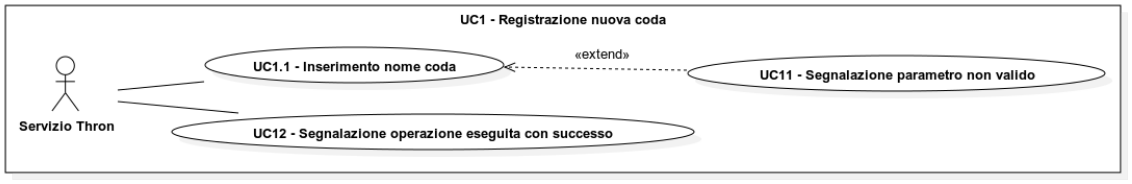


Figura 3.3: UC1 - Registrazione nuova coda

Attori principali: Servizio Thron.
Descrizione: Il servizio Thron vuole creare una nuova coda, a lui dedicata, all'interno del sistema.

Pre-condizione: Il servizio Thron deve avere accesso al sistema di code.
Post-condizione: Una nuova coda è stata creata all'interno del sistema.
Scenario principale:

1. Il servizio Thron specifica il nome da assegnare alla nuova coda (UC1.1);
2. Il servizio Thron invia la richiesta per la creazione di una nuova coda al sistema;
3. Il sistema segnala che l'operazione è stata eseguita con successo (UC12).

Scenario alternativo:

* Il sistema segnala che l'operazione è fallita (UC10).

UC1.1: Inserimento nome coda

Attori principali: Servizio Thron.
Descrizione: Il servizio Thron specifica il nome della nuova coda che vuole creare.
Pre-condizione: Il servizio Thron deve avere accesso al sistema di code e sta richiedendo la creazione di una nuova coda.
Post-condizione: Il nome della nuova coda è stato specificato.
Scenario principale:

1. Il servizio Thron specifica, nella richiesta, il nome da assegnare alla nuova coda.

Scenario alternativo:

* Il sistema segnala che il nome assegnato alla coda non è valido (UC11).

UC2: Accodamento nuovo *task*

Il caso d'uso è raffigurato, mediante diagramma *UML*, nella figura 3.4.

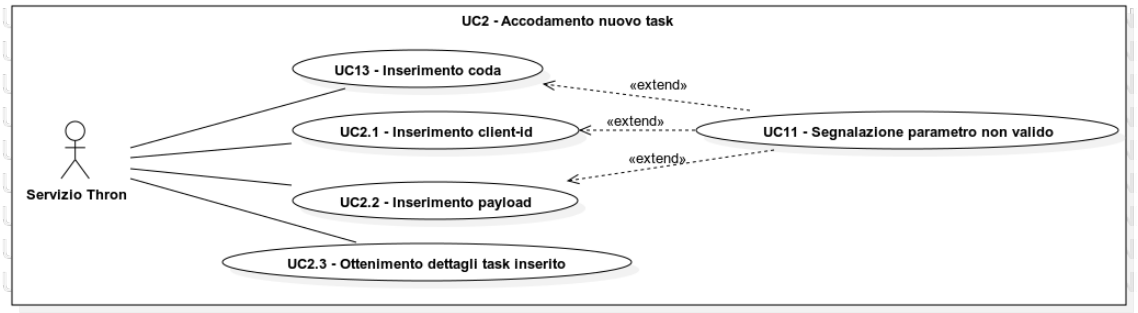


Figura 3.4: UC2 - Accodamento nuovo *task*

Attori principali: Servizio Thron.
Descrizione: Il servizio Thron vuole inserire un nuovo *task* all'interno di una coda.
Pre-condizione: Il servizio Thron deve aver precedentemente creato una coda a lui dedicata.
Post-condizione: Il *task* è stato correttamente accodato all'interno della coda specificata.
Scenario principale:

1. Il servizio Thron specifica la coda in cui vuole accodare il *task* (UC13);
2. Il servizio Thron specifica il *client-id* associato al *task* da accodare (UC2.1);
3. Il servizio Thron specifica il *payload* del *task* da accodare (UC2.2);
4. Il servizio Thron invia la richiesta di accodamento del *task* al sistema;
5. Il sistema segnala che l'operazione è stata conclusa con successo e restituisce i dettagli del *task* appena accodato (UC2.3).

Scenario alternativo:

* Il sistema segnala che l'operazione è fallita (UC10).

UC2.1: Inserimento *client-id*

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron specifica il nome del cliente a cui è associato il *task* che vuole accodare.

Pre-condizione: Il servizio Thron deve aver precedentemente creato una coda a lui dedicata e sta richiedendo l'accodamento di un nuovo *task*.

Post-condizione: Il *client-id* della coda in cui inserire il *task* è stato specificato.

Scenario principale:

1. Il servizio Thron specifica, nella richiesta, il nome del cliente associato al *task* che vuole accodare.

Scenario alternativo:

* Il sistema segnala che il *client-id* specificato non è valido (UC11).

UC2.2: Inserimento *payload*

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron specifica il contenuto del *task* che vuole accodare.

Pre-condizione: Il servizio Thron deve aver precedentemente creato una coda a lui dedicata e sta richiedendo l'accodamento di un nuovo *task*.

Post-condizione: È stato definito il contenuto del *task* da accodare.

Scenario principale:

1. Il servizio Thron specifica, nella richiesta, il contenuto associato al *task* da accodare.

Scenario alternativo:

* Il sistema segnala che il contenuto del *task* non è valido (UC11).

UC2.3: Ottenimento dettagli *task* inserito

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron riceve una risposta di conferma di accodamento, con il contenuto del *task* appena accodato.

Pre-condizione: Il servizio Thron deve aver inviato al sistema una richiesta per l'accodamento di un nuovo *task*.

Post-condizione: È stato segnalato al servizio Thron che il *task* è stato accodato con successo, fornendone i dettagli.

Scenario principale:

1. Il sistema risponde al servizio Thron indicando i dettagli (contenuto) del *task* appena accodato.

UC3: Recupero *task* accodati

Il caso d'uso è raffigurato, mediante diagramma *UML*, nella figura 3.5.

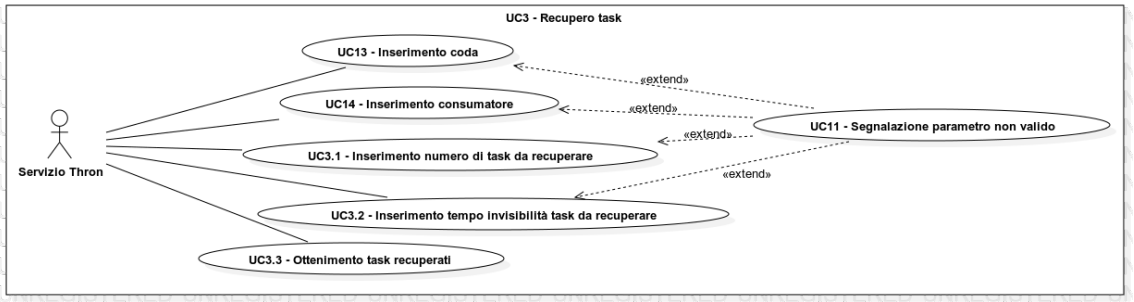


Figura 3.5: UC3 - Recupero *task* accodati

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron vuole recuperare da una coda dei *task* precedentemente accodati.

Pre-condizione: Il servizio Thron deve aver precedentemente accodato dei *task* in una coda.

Post-condizione: Sono stati recuperati dei *task* da eseguire dalla coda specificata.

Scenario principale:

1. Il servizio Thron specifica la coda da cui vuole recuperare i *task* da eseguire (UC13);
2. Il servizio Thron specifica il nome del consumatore che andrà ad eseguire i *task* che verranno prelevati (UC14);
3. Il servizio Thron specifica il numero di *task* che vuole recuperare dalla coda (UC3.1);
4. Il servizio Thron specifica il tempo per il quale i *task* saranno invisibili ad altri consumatori della coda (UC3.2);
5. Il servizio Thron invia la richiesta di recupero dei *task* al sistema;
6. Il sistema segnala che l'operazione è stata conclusa con successo e restituisce l'elenco dei *task* prelevati (UC3.3).

Scenario alternativo:

* Il sistema segnala che l'operazione è fallita (UC10).

UC3.1: Inserimento numero di *task* da recuperare

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron specifica il numero di *task* che vuole recuperare dalla coda.

Pre-condizione: Il servizio Thron deve aver precedentemente accodato dei *task* in una coda e sta richiedendo di recuperarli.

Post-condizione: Il numero di *task* da prelevare è stato specificato.

Scenario principale:

1. Il servizio Thron specifica, nella richiesta, il numero di *task* da recuperare.

Scenario alternativo:

* Il sistema segnala che il numero specificato non è valido (UC11).

UC3.2: Inserimento tempo invisibilità *task* da recuperare

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron specifica il tempo per il quale i *task* prelevati saranno invisibili agli altri consumatori della coda.

Pre-condizione: Il servizio Thron deve aver precedentemente accodato dei *task* in una coda e sta richiedendo di recuperarli.

Post-condizione: È stato definito il tempo di invisibilità per i *task* da prelevare.

Scenario principale:

1. Il servizio Thron specifica, nella richiesta, il tempo di invisibilità per i *task* da prelevare.

Scenario alternativo:

* Il sistema segnala che il tempo specificato non è valido (UC11).

UC3.3: Ottenimento *task* recuperati

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron riceve una risposta di conferma di recupero dei *task*, assieme ad un elenco contenente i singoli *task* estratti.

Pre-condizione: Il servizio Thron deve aver inviato al sistema una richiesta per il recupero dei *task* precedentemente accodati.

Post-condizione: È stato segnalato al servizio Thron che sono stati recuperati dei *task*, fornendogli l'elenco di questi.

Scenario principale:

1. Il sistema risponde al servizio Thron con un elenco contenente i *task* appena recuperati.

3.2.3 Gestione *task*

Per la seguente area funzionale, i casi d'uso sono riassunti nel diagramma *UML* contenuto nella figura 3.6.

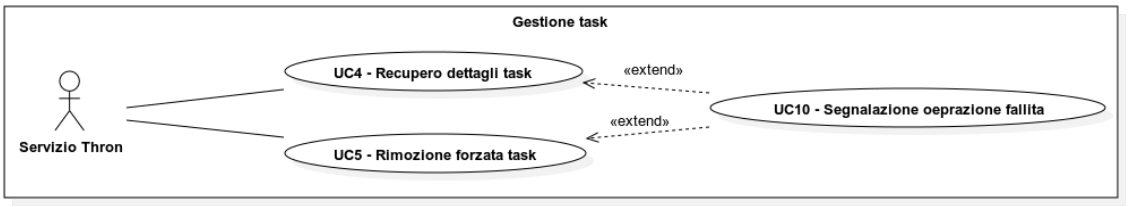


Figura 3.6: Casi d'uso - Gestione *task*

UC4: Recupero dettagli *task*

Il caso d'uso è raffigurato, mediante diagramma *UML*, nella figura 3.7.

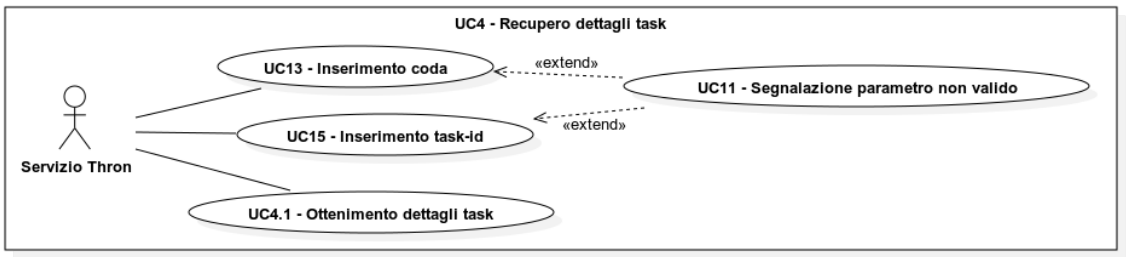


Figura 3.7: UC4 - Recupero dettagli *task*

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron vuole recuperare le informazioni di un *task* precedentemente accodato.

Pre-condizione: Il servizio Thron deve aver precedentemente accodato il *task* di cui è interessato a conoscere i dettagli.

Post-condizione: Sono state recuperate le informazioni sul *task* specificato.

Scenario principale:

1. Il servizio Thron specifica la coda in cui si trova il *task* di cui vuole ricevere i dettagli (UC13);
2. Il servizio Thron specifica l'*id* del *task* di cui vuole recuperare i dettagli (UC15);
3. Il servizio Thron invia la richiesta di recupero delle informazioni sul *task* al sistema;
4. Il sistema segnala che l'operazione è stata conclusa con successo e restituisce i dettagli recuperati (UC4.1).

Scenario alternativo:

∗ Il sistema segnala che l'operazione è fallita (UC10).

UC4.1: Ottenimento dettagli *task*

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron riceve una risposta di conferma sulla corretta esecuzione dell'operazione, assieme alle informazioni recuperate sul *task* specificato.

Pre-condizione: Il servizio Thron deve aver inviato al sistema una richiesta per il recupero delle informazioni sul *task* precedentemente accodato.

Post-condizione: I dettagli del *task* specificato sono stati forniti al servizio Thron.

Scenario principale:

1. Il sistema risponde al servizio Thron con le informazioni appena recuperate sul *task*.

UC5: Rimozione forzata *task*

Il caso d'uso è raffigurato, mediante diagramma *UML*, nella figura 3.8.

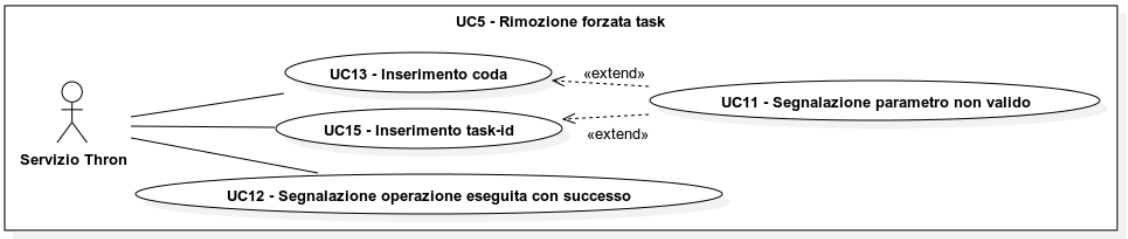


Figura 3.8: UC5 - Rimozione forzata *task*

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron vuole eliminare un *task* precedentemente accodato.

Pre-condizione: Il servizio Thron deve aver precedentemente accodato il *task* che vuole eliminare.

Post-condizione: Il *task* specificato è stato correttamente eliminato dalla coda.

Scenario principale:

1. Il servizio Thron specifica la coda in cui si trova il *task* che vuole eliminare (UC13);
2. Il servizio Thron specifica l'*id* del *task* che vuole eliminare (UC15);
3. Il servizio Thron invia la richiesta di rimozione del *task* al sistema;
4. Il sistema segnala che l'operazione è stata conclusa con successo e il *task* è stato eliminato (UC12).

Scenario alternativo:

∗ Il sistema segnala che l'operazione è fallita (UC10).

3.2.4 Gestione *task* recuperati

Per la seguente area funzionale, i casi d'uso sono riassunti nel diagramma *UML* contenuto nella figura 3.9.

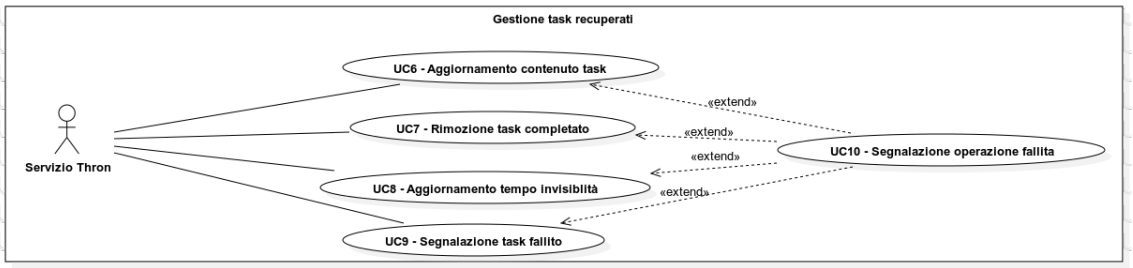


Figura 3.9: Casi d'uso - Gestione *task* recuperati

UC6: Aggiornamento contenuto *task*

Il caso d'uso è raffigurato, mediante diagramma *UML*, nella figura 3.10.

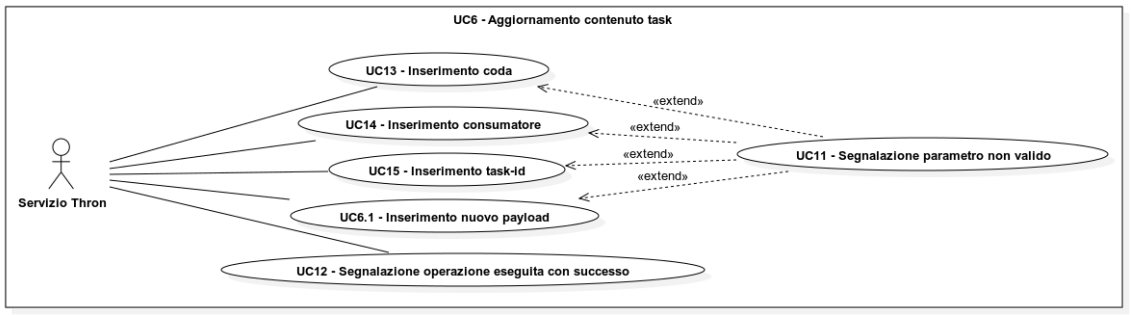


Figura 3.10: UC6 - Aggiornamento contenuto *task*

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron vuole aggiornare il contenuto di un *task* precedentemente recuperato.

Pre-condizione: Il servizio Thron deve aver precedentemente prelevato il *task* di cui vuole aggiornare il contenuto.

Post-condizione: Il contenuto del *task* specificato è stato aggiornato.

Scenario principale:

1. Il servizio Thron specifica la coda in cui si trova il *task* di cui vuole aggiornare il contenuto (UC13);

2. Il servizio Thron specifica l'*id* del consumatore che ha precedentemente prelevato il *task* (UC14);
3. Il servizio Thron specifica l'*id* del *task* di cui vuole aggiornare il contenuto (UC15);
4. Il servizio Thron specifica il nuovo *payload* da assegnare al *task* (UC6.1);
5. Il servizio Thron invia la richiesta di aggiornamento del contenuto del *task* al sistema;
6. Il sistema segnala che l'operazione è stata conclusa con successo e il contenuto del *task* è stato aggiornato (UC12).

Scenario alternativo:

* Il sistema segnala che l'operazione è fallita (UC10).

UC6.1: Inserimento nuovo *payload*

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron specifica il nuovo contenuto da assegnare al *task* da aggiornare.

Pre-condizione: Il servizio Thron deve richiede di aggiornare il contenuto di uno specifico *task*.

Post-condizione: Il nuovo *payload* del *task* specificato è stato definito.

Scenario principale:

1. Il servizio Thron specifica, nella richiesta, il nuovo contenuto da assegnare al *task* da aggiornare.

Scenario alternativo:

* Il sistema segnala che il contenuto specificato non è valido (UC11).

UC7: Rimozione *task* completato

Il caso d'uso è raffigurato, mediante diagramma *UML*, nella figura 3.11.

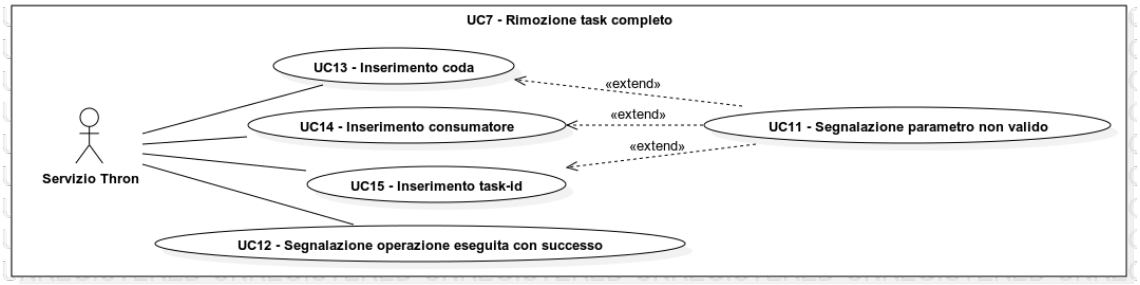


Figura 3.11: UC7 - Rimozione *task* completato

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron vuole rimuovere, da una specifica coda, un *task* precedentemente recuperato ed eseguito con successo.

Pre-condizione: Il servizio Thron deve aver precedentemente prelevato ed eseguito il *task* che vuole rimuovere.

Post-condizione: Il *task* è stato rimosso dalla coda con successo.

Scenario principale:

1. Il servizio Thron specifica la coda in cui si trova il *task* da rimuovere (UC13);
2. Il servizio Thron specifica l'*id* del consumatore che ha precedentemente prelevato ed eseguito il *task* (UC14);

3. Il servizio Thron specifica l'*id* del *task* da rimuovere (UC15);
4. Il servizio Thron invia la richiesta di rimozione del *task* al sistema;
5. Il sistema segnala che l'operazione è stata conclusa con successo e il *task* è stato eliminato (UC12).

Scenario alternativo:

* Il sistema segnala che l'operazione è fallita (UC10).

UC8: Aggiornamento tempo invisibilità

Il caso d'uso è raffigurato, mediante diagramma *UML*, nella figura 3.12.

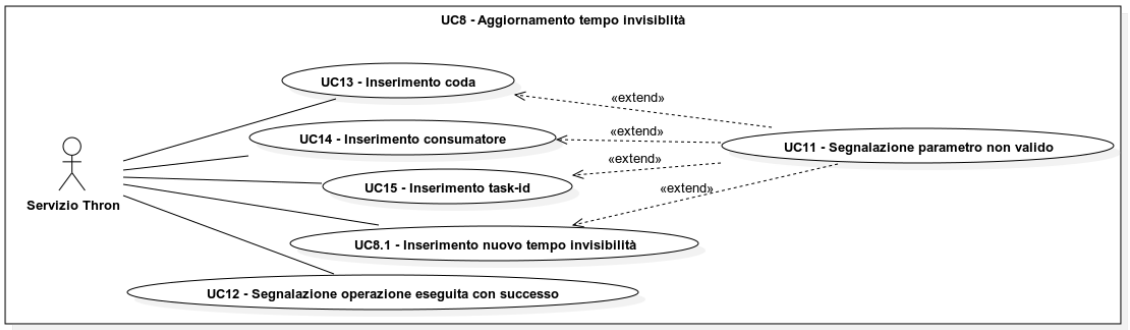


Figura 3.12: UC8 - Aggiornamento tempo invisibilità

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron vuole aggiornare il tempo di invisibilità di un *task* precedentemente recuperato.

Pre-condizione: Il servizio Thron deve aver precedentemente prelevato il *task* di cui vuole aggiornare il tempo di invisibilità.

Post-condizione: Il tempo di invisibilità del *task* specificato è stato aggiornato.

Scenario principale:

1. Il servizio Thron specifica la coda in cui si trova il *task* di cui vuole aggiornare il tempo di invisibilità (UC13);
2. Il servizio Thron specifica l'*id* del consumatore che ha precedentemente prelevato il *task* (UC14);
3. Il servizio Thron specifica l'*id* del *task* di cui vuole aggiornare il tempo di invisibilità (UC15);
4. Il servizio Thron specifica il nuovo tempo di invisibilità da assegnare al *task* (UC8.1);
5. Il servizio Thron invia la richiesta di aggiornamento del tempo di invisibilità del *task* al sistema;
6. Il sistema segnala che l'operazione è stata conclusa con successo e il tempo di invisibilità del *task* è stato aggiornato (UC12).

Scenario alternativo:

* Il sistema segnala che l'operazione è fallita (UC10).

UC8.1: Inserimento nuovo tempo di invisibilità

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron specifica il nuovo tempo di invisibilità da assegnare al *task*.

Pre-condizione: Il servizio Thron deve richiede di aggiornare il tempo di invisibilità di uno specifico *task*.

Post-condizione: Il nuovo tempo di invisibilità del *task* specificato è stato definito.

Scenario principale:

1. Il servizio Thron specifica, nella richiesta, il nuovo tempo di invisibilità da assegnare al *task*.

Scenario alternativo:

∗ Il sistema segnala che il tempo specificato non è valido (UC11).

UC9: Segnalazione *task* fallito

Il caso d'uso è raffigurato, mediante diagramma *UML*, nella figura 3.13.

Figura 3.13: UC9 - Segnalazione *task* fallito

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron vuole segnalare che un *task*, precedentemente recuperato, non è stato eseguito correttamente.

Pre-condizione: Il servizio Thron deve aver precedentemente prelevato il *task* di cui vuole segnalare il fallimento.

Post-condizione: È stato segnalato al sistema il fallimento del *task* specificato.

Scenario principale:

1. Il servizio Thron specifica la coda in cui si trova il *task* che non è riuscito ad eseguire correttamente (UC13);
2. Il servizio Thron specifica l'*id* del consumatore che ha precedentemente prelevato il *task* (UC14);
3. Il servizio Thron specifica l'*id* del *task* che non è riuscito ad eseguire correttamente (UC15);
4. Il servizio Thron invia la segnalazione di fallimento dell'esecuzione del *task* al sistema;
5. Il sistema segnala che l'operazione è stata conclusa con successo ed è stato incrementato il numero di tentativi falliti per il *task* specificato (UC12).

Scenario alternativo:

∗ Il sistema segnala che l'operazione è fallita (UC10).

3.2.5 Operazioni standard

Vengono ora elencati i casi d'uso che modellano le più frequenti operazioni utilizzate anche dalle altre funzionalità.

UC10: Segnalazione operazione fallita

Attori principali: Servizio Thron.

Descrizione: Il sistema segnala al servizio Thron che l'operazione da lui richiesta non è stata eseguita correttamente.

Pre-condizione: Il servizio Thron deve aver effettuato una richiesta al sistema per eseguire un'operazione.

Post-condizione: È stato segnalato al servizio Thron il fallimento dell'operazione richiesta.

Scenario principale:

1. Il sistema segnala che l'operazione è fallita, allegando nella risposta un messaggio di errore.

UC11: Segnalazione parametro non valido

Attori principali: Servizio Thron.

Descrizione: Il sistema segnala al servizio Thron che il parametro inserito in una richiesta non è valido.

Pre-condizione: Il servizio Thron deve aver effettuato una richiesta al sistema per eseguire un'operazione.

Post-condizione: È stato segnalato al servizio Thron l'invalidità di un parametro inserito.

Scenario principale:

1. Il sistema segnala che il parametro inserito non è valido, allegando una motivazione.

UC12: Segnalazione operazione eseguita con successo

Attori principali: Servizio Thron.

Descrizione: Il sistema segnala al servizio Thron che l'operazione da lui richiesta è stata eseguita correttamente.

Pre-condizione: Il servizio Thron deve aver effettuato una richiesta al sistema per eseguire un'operazione.

Post-condizione: È stato segnalato al servizio Thron che l'operazione richiesta è stata eseguita con successo.

Scenario principale:

1. Il sistema segnala che l'operazione richiesta è stata eseguita con successo.

UC13: Inserimento coda

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron specifica l'*id* della coda in cui vuole effettuare una particolare operazione.

Pre-condizione: Il servizio Thron deve aver precedentemente richiesto di effettuare un'operazione su una coda.

Post-condizione: Il nome della coda, in cui il servizio Thron vuole effettuare l'operazione, è stata specificata.

Scenario principale:

1. Il servizio Thron specifica, nella richiesta, l'*id* della coda in cui vuole effettuare l'operazione.

Scenario alternativo:

* Il sistema segnala che l'*id* specificato non è valido (UC11).

UC14: Inserimento consumatore

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron specifica l'*id* del consumatore che vuole effettuare un'operazione su un *task* precedentemente estratto.

Pre-condizione: Il servizio Thron deve aver precedentemente richiesto di effettuare un'operazione su uno specifico *task*.

Post-condizione: Il nome del consumatore, che in precedenza aveva prelevato il *task*, è stato specificato.

Scenario principale:

1. Il servizio Thron specifica, nella richiesta, l'*id* del consumatore che vuole effettuare l'operazione su un *task* precedentemente prelevato.

Scenario alternativo:

* Il sistema segnala che l'*id* specificato non è valido (UC11).

UC15: Inserimento *task-id*

Attori principali: Servizio Thron.

Descrizione: Il servizio Thron specifica l'*id* del *task* su cui vuole effettuare l'operazione.

Pre-condizione: Il servizio Thron deve aver precedentemente richiesto di effettuare un'operazione su un *task*.

Post-condizione: L'*id* del *task* è stato specificato.

Scenario principale:

1. Il servizio Thron specifica, nella richiesta, l'*id* del *task* su cui vuole effettuare l'operazione.

Scenario alternativo:

* Il sistema segnala che l'*id* specificato non è valido (UC11).

3.3 Tracciamento dei requisiti

In questa sezione, vengono riportati i requisiti del progetto, classificati per tipo ed obbligatorietà. Ciascun requisito possiede un codice identificativo, il cui formalismo viene riportato di seguito:

R<NumeroRequisito>-<Tipo>-<Classificazione>

Il **Tipo** specifica la natura del requisito e segue il seguente formalismo:

* **F**: funzionale;
* **Q**: qualitativo;
* **V**: vincolo;

La **Classificazione** indica l'obbligatorietà del requisito e segue il seguente formalismo:

- **O**: obbligatorio;
- **D**: desiderabile;

Nelle tabelle 3.1, 3.2 e 3.3 sono riassunti i requisiti e il loro eventuale tracciamento con i casi d'uso individuati nella sezione precedente.

Tabella 3.1: Tabella del tracciamento dei requisiti funzionali

Requisito	Descrizione	Use Case
R1-F-O	Un servizio Thron deve poter creare una coda a lui dedicata	UC1
R2-F-O	Un servizio Thron deve poter specificare il nome della nuova coda che vuole creare	UC.1.1
R3-F-O	Un servizio Thron deve poter accodare dei *task* in una coda precedentemente creata	UC2
R4-F-O	Un servizio Thron deve poter specificare la coda in cui vuole accodare un nuovo *task*	UC13
R5-F-O	Un servizio Thron deve poter associare un nuovo *task* da accodare ad un *client-id*	UC2.1
R6-F-O	Un servizio Thron deve poter definire il contenuto del nuovo *task* da accodare	UC2.2
R7-F-O	Un servizio Thron deve poter ricevere i dettagli di un *task* appena accodato	UC2.3
R8-F-O	Un servizio Thron deve poter recuperare dei *task* precedentemente accodati	UC3
R9-F-O	Un servizio Thron deve poter specificare la coda da cui vuole recuperare dei *task*	UC13
R10-F-O	Un servizio Thron deve poter inserire l'*id* del consumatore che eseguirà i *task* prelevati	UC14
R11-F-O	Un servizio Thron deve poter specificare il numero di *task* che vuole prelevare	UC3.1
R12-F-O	Un servizio Thron deve poter specificare il tempo di invisibilità per i *task* da prelevare	UC3.2
R13-F-O	Un servizio Thron deve poter ricevere le informazioni dei *task* appena prelevati	UC3.3
R14-F-O	Un servizio Thron deve poter recuperare le informazioni di un *task* precedentemente accodato	UC4
R15-F-O	Un servizio Thron deve poter specificare la coda in cui si trova il *task* di cui vuole recuperare i dettagli	UC13
R16-F-O	Un servizio Thron deve poter specificare l'*id* del *task* di cui recuperare le informazioni	UC15
R17-F-O	Un servizio Thron deve poter ricevere le informazioni recuperate dal *task* richiesto	UC4.1
R18-F-O	Un servizio Thron deve poter rimuovere un *task* precedentemente accodato	UC5
R19-F-O	Un servizio Thron deve poter specificare la coda in cui si trova il *task* da rimuovere	UC13
R20-F-O	Un servizio Thron deve poter specificare l'*id* del *task* da rimuovere	UC15
R21-F-O	Un servizio Thron deve poter aggiornare il contenuto di un *task* precedentemente accodato	UC6
R22-F-O	Un servizio Thron deve poter specificare la coda in cui si trova il *task* da aggiornare	UC13

R23-F-O	Un servizio Thron deve poter specificare l'*id* del consumatore che ha prelevato il *task*	UC14
R24-F-O	Un servizio Thron deve poter specificare l'*id* del *task* da aggiornare	UC15
R25-F-O	Un servizio Thron deve poter specificare il nuovo contenuto da assegnare al *task*	UC6.1
R26-F-O	Un servizio Thron deve poter rimuovere dalla coda un *task* che è stato eseguito correttamente	UC7
R27-F-O	Un servizio Thron deve poter specificare la coda in cui si trova il *task* eseguito correttamente e quindi da rimuovere	UC13
R28-F-O	Un servizio Thron deve poter specificare l'*id* del consumatore che ha prelevato ed eseguito correttamente il *task*	UC14
R29-F-O	Un servizio Thron deve poter specificare l'*id* del *task* eseguito correttamente da rimuovere	UC15
R30-F-O	Un servizio Thron deve poter aggiornare il tempo di invisibilità di un *task* che sta eseguendo	UC8
R31-F-O	Un servizio Thron deve poter specificare la coda in cui si trova il *task* di cui vuole aggiornare il tempo di invisibilità	UC13
R32-F-O	Un servizio Thron deve poter specificare l'*id* del consumatore che ha prelevato e che sta eseguendo il *task*	UC14
R33-F-O	Un servizio Thron deve poter specificare l'*id* del *task* di cui vuole aggiornare il tempo di invisibilità	UC15
R34-F-O	Un servizio Thron deve poter specificare il nuovo tempo di invisibilità da assegnare al *task* che sta eseguendo	UC8.1
R35-F-O	Un servizio Thron deve poter segnalare che un *task* non è stato eseguito correttamente	UC9
R36-F-O	Un servizio Thron deve poter specificare la coda in cui si trova il *task* che non è stato eseguito correttamente	UC13
R37-F-O	Un servizio Thron deve poter specificare l'*id* del consumatore che ha eseguito con esito fallimentare il *task*	UC14
R38-F-O	Un servizio Thron deve poter specificare l'*id* del *task* che non è stato eseguito correttamente	UC15
R39-F-O	Il sistema deve segnalare al servizio Thron se l'operazione richiesta è fallita	UC10
R40-F-O	Il sistema deve segnalare al servizio Thron se la richiesta contiene uno o più parametri non validi	UC11
R41-F-O	Il sistema deve segnalare al servizio Thron se l'operazione richiesta è stata eseguita con successo	UC12
R42-F-O	Il sistema deve garantire il prelievo dei *task* in modo equo rispetto ai *client-id* all'interno della stessa coda	-
R43-F-O	Il sistema deve esporre delle metriche funzionali che permettano di misurare lo stato di salute delle code	-
R44-F-D	Il sistema deve poter permettere l'accodamento e il prelievo di *task* con priorità diversa	-
R45-F-D	ll sistema deve poter spostare dei *task* che sono falliti molte volte in una *dead letter queue*	-

Tabella 3.2: Tabella del tracciamento dei requisiti qualitativi

Requisito	Descrizione	Use Case
R46-Q-O	Deve essere redatto un documento che raccolga la descrizione tecnica del microservizio	-
R47-Q-O	Deve essere redatto un documento che raccolga la descrizione delle funzionalità implementate nel microservizio	-
R48-Q-O	Il codice deve essere documentato tramite commenti	-
R49-Q-O	Deve essere redatto un *report* sugli scenari presi in considerazione nelle fasi di test e sull'esito degli stessi	-

Tabella 3.3: Tabella del tracciamento dei requisiti di vincolo

Requisito	Descrizione	Use Case
R50-V-D	Il microservizio deve avere architettura *serverless*	-
R51-V-O	Il microservizio dovrà utilizzare *Amazon DynamoDB*[1] come sistema di persistenza	-
R52-V-O	Le funzionalità del servizio dovranno essere esposte tramite *API* utilizzando il paradigma *REST* o *RPC (Remote Procedure Call)*	-
R53-V-O	Il sistema di monitoraggio deve essere implementato utilizzando *Prometheus*[2] o *Amazon CloudWatch*[3]	-
R54-V-O	Il servizio dovrà essere codifica utilizzando *Go* o *Java*	-

[1] Amazon DynamoDB: https://aws.amazon.com/dynamodb/
[2] Monitoraggio con Prometheus: https://prometheus.io/
[3] Amazon CloudWatch: https://aws.amazon.com/cloudwatch/

Capitolo 4

Progettazione e codifica

In questo capitolo vengono esposte le attività di progettazione e codifica del microservizio.

4.1 Progettazione

In questa sezione verrà analizzata l'architettura progettata per la realizzazione del servizio di code.

4.1.1 Architettura del servizio

L'architettura del servizio è riassunta nel diagramma raffigurato in figura 4.1.
Non sono state aggiunte tutte le funzionalità, in quanto il diagramma sarebbe risultato più pesante e non avrebbe dato alcuna informazione aggiuntiva.

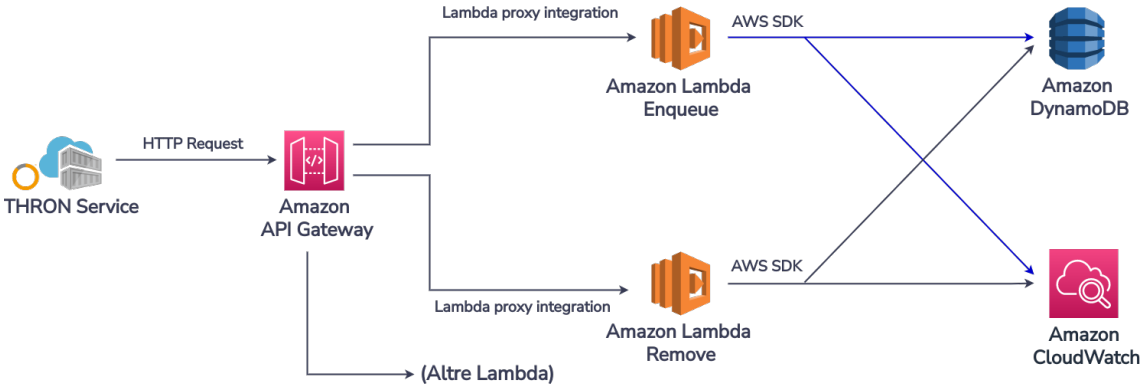


Figura 4.1: Architettura generale del microservizio

Come si evince dal diagramma, l'infrastruttura del servizio è una combinazione di servizi *AWS*, collegati tra di loro attraverso paradigmi di comunicazione differenti.
L'interazione tra questi servizi consente di esporre e implementare le funzionalità individuate durante l'analisi dei requisiti, oltre che a garantire alcuni aspetti peculiari come la scalabilità e la predisposizione del servizio ad eseguire in ambienti *serverless*.

Il servizio, in seguito decomposto più nel dettaglio, presenta le seguenti macro componenti:

* ***Amazon API Gateway***: espone gli *endpoint* attraverso i quali servizi Thron possono accedere alle funzionalità esposte dal servizio;
* ***Amazon Lambda***: ogni istanza implementa una particolare funzionalità del sistema. Contiene quindi le validazioni di *business* e le logiche delle singole operazioni;

* ***Amazon DynamoDB***: sistema di persistenza al quale il servizio si appoggia per l'erogazione delle funzionalità e l'esecuzione delle operazioni richieste dai servizi Thron;
* ***Amazon CloudWatch***: sistema di monitoraggio attraverso il quale vengono raccolte le metriche relative alle varie code definite dai servizi Thron all'interno del servizio.

4.1.2 Funzionamento di base

In questa sezione viene descritta una classica interazione tra un generico servizio Thron e il servizio di code.
Nella figura 4.2 si può trovare un diagramma delle attività che riassume il flusso successivamente descritto.

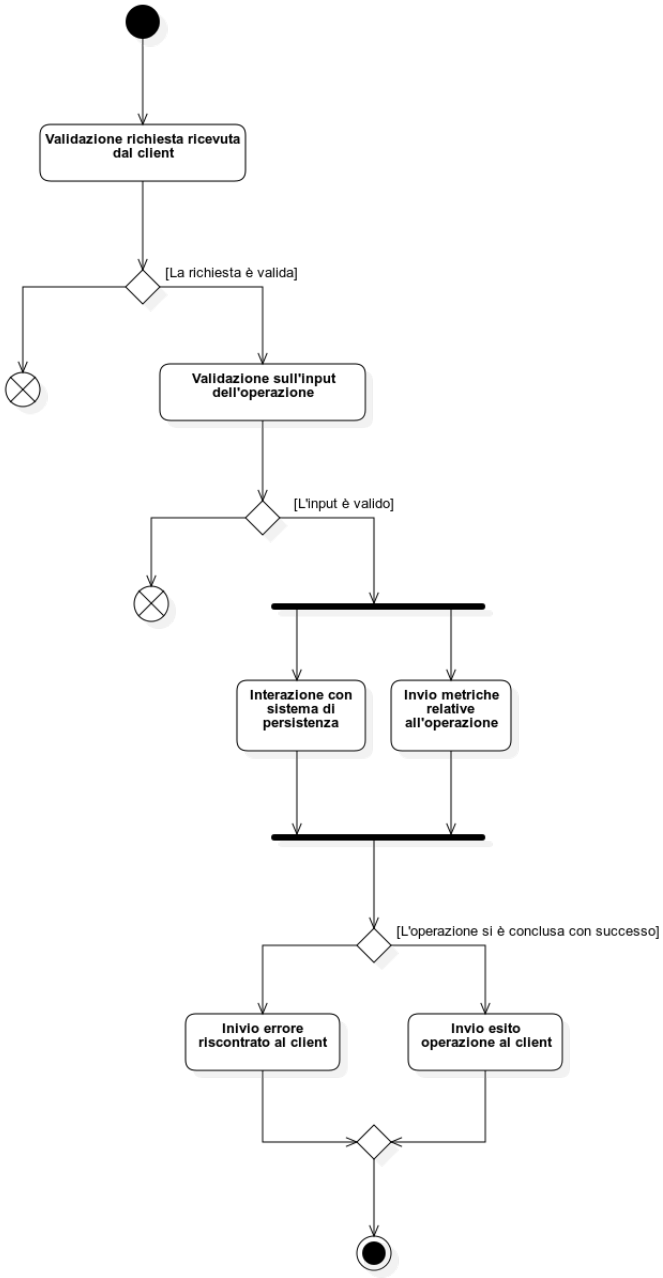


Figura 4.2: Diagramma di attività del flusso di interazione tra un servizio Thron e il servizio di code

Come si evince dai diagrammi 4.1 e 4.2, è il servizio Thron, utilizzatore del servizio di code, che innesca il flusso di interazione per l'esecuzione di una particolare operazione. Il servizio Thron

effettua una chiamata *HTTP (HyperText Transfer Protocol)* all'*endpoint* corrispondente al tipo di operazione che vuole svolgere. Nella richiesta verranno allegati una serie di parametri, necessari per eseguire l'operazione.

Tutti gli *endpoint* vengono esposti dall'*API Gateway*, che si occupa quindi di ricevere e servire le richieste dei vari *client*.
A questo livello viene fatta una prima validazione sulla richiesta: l'*API Gateway* controlla siano presenti al suo interno tutti i parametri richiesti per eseguire l'operazione che il servizio sta richiedendo, verificandone la presenza e la validità rispetto all'*input* dell'operazione richiesta.
Se questo primo controllo fallisse, l'*API Gateway* rispedisce la richiesta al mittente, accompagnata da un messaggio di errore che segnala quali parametri sono mancanti o invalidi.

Nel caso tutto andasse a buon fine, la richiesta viene impacchettata automaticamente dall'*API Gateway* e inoltrata alla *Lambda* associata all'*endpoint* su cui è stata fatta la richiesta. La comunicazione avviene tramite il meccanismo di *Lambda Proxy Integration*, fornito da *AWS*.
Una volta che la *Lambda* ha eseguito l'operazione e ritornato una risposta, l'*API Gateway* si occupa, autonomamente, di spedire la risposta al servizio che aveva precedentemente fatto la richiesta.

La *Lambda* riceve la richiesta e si occupa principalmente di validarla semanticamente e di processarla. Prima di processare la richiesta, effettua delle validazioni di *business* sui parametri in *input*, controllandone la validità anche dal punto di vista semantico.
Se qualche parametro fosse invalido, la *Lambda* ritorna all'*API Gateway* un messaggio di errore, che contiene la ragione dell'invalidità dei parametri invalidi.
Se la validazione va a buon fine, la *Lambda* esegue l'operazione, interagendo con il sistema di persistenza e con quello di monitoraggio. Nel primo caso va ad effettuare l'operazione richiesta nel *database*: inserisce, modifica o elimina *item* dalle tabelle. Nel secondo caso va a registrare una misurazione associata all'operazione e la spedisce al sistema di monitoraggio. Le comunicazioni con i due sistemi, essendo servizi *AWS*, avvengono tramite l'utilizzo dell'*AWS SDK (Software Development Kit)*.
Conclusa l'operazione, la *Lambda* ritorna l'esito dell'operazione, indipendente che sia positivo o negativo, all'*API Gateway*.

4.1.3 Definizione dell'infrastruttura

Prima di approfondire come sono stati utilizzati i singoli servizi, è opportuno specificare il modo in cui questi vengono legati e come comunicano tra di loro. L'infrastruttura del microservizio, composta da una combinazione di servizi *AWS* è stata definita utilizzando *Amazon SAM*[1], un *framework* per definire la struttura di applicazioni basate su servizi *AWS* e facilitare la fase di *deploy*.

Utilizzando questo *tool* è stato possibile definire l'infrastruttura come risorse di servizi *AWS*. Tramite un *template file* sono state definite le istanze necessarie, il modo in cui queste comunicano tra loro e i loro permessi di esecuzione.
Avere un *file* contenente la struttura del servizio, ha permesso il suo versionamento e ha praticamente azzerato il tempo necessario per la fase di *deploy*.

4.1.4 *Amazon API Gateway*

Introduzione

Amazon API Gateway[2] è un servizio di *AWS* che permette di creare, pubblicare, mantenere e monitorare *API* su qualsiasi scala. Offre la possibilità di creare, tra le altre, *API REST* ed è in grado di gestire migliaia di richieste concorrenti.
Il pagamento del servizio avviene in base al numero di richieste che l'*API Gateway* riceve.

[1] Amazon Serverless Application Model: https://aws.amazon.com/serverless/sam/
[2] Amazon API Gateway: https://aws.amazon.com/api-gateway/

Utilizzo del servizio

Per la definizione delle *API REST* che l'*API Gateway* deve esporre, si è sfruttata una peculiarità del servizio. Essa permette di importare la definizione di *API REST* da un *file* esterno che rispetti il formato *OpenAPI*. A questo scopo è stata definita la specifica *OpenAPI*, che, oltre a rappresentare documentazione funzionale, è stata utilizzata per velocizzare il processo di definizione delle *API REST*. Gli *endpoint* esposti vengono trattati nella sezione 4.1.8.

Funzionalità del servizio

Come precedentemente descritto, il servizio ha tre principali compiti:

* Ricevere le richieste dai *client* e fornire loro una risposta;
* Controllare la presenza e la validità dei parametri della richiesta;
* Inoltrare una richiesta valida alla *Lambda* associata all'*endpoint*.

Il primo compito è la natura stessa del servizio.

Per gli altri due compiti il servizio deve essere configurato, in modo tale da poter capire se una richiesta è valida o meno. In base all'esito può rifiutare la richiesta ed evitare di chiamare inutilmente la *Lambda* associata.

Validazione

Nella specifica *OpenAPI* vengono definiti i *path params*, i *query params* e il *body* che la richiesta deve avere per essere considerata valida.
Per poter definire come il servizio debba validare i parametri, viene utilizzato un oggetto chiamato *x-amazon-apigateway-request-validator* all'interno della specifica *OpenAPI*. Esso permette di specificare come un *API Gateway* debba validare una richiesta su un determinato *endpoint*. Le opzioni disponibili permettono di scegliere quali parti della richiesta validare: solo i *path* e *query params*, solo il *body* o entrambi. È stato scelto di validare sempre tutte le parti possibili.

La validazione fatta dall'*API Gateway* controlla se:

* Tutti gli *item*, definiti obbligatori nella specifica *OpenAPI*, sono presenti nella richiesta;
* Tutti gli *item* sono dello stesso tipo rispetto a quanto definito nella specifica *OpenAPI*;
* Per gli *item* di tipo numerico, se sono nel *range* definito nella specifica *OpenAPI*.

Al momento la validazione non supporta il *matching* con le espressioni regolari. Questo costringe a spostare parte dei controlli, soprattutto su stringhe, sulle *Lambda*, rallentando un po' l'esecuzione delle stesse.

Se una richiesta, inviata da un servizio Thron, non passasse la prima fase di validazione, l'*API Gateway* restituirà automaticamente ad esso una risposta con *status code 400 (Bad Request)*, dove nel *body* viene specificato cosa ha fatto fallire la validazione.

Integrazione con la *Lambda*

L'integrazione tra un *endpoint* esposto dall'*API Gateway* e la relativa *Lambda* avviene attraverso un meccanismo chiamato *Lambda Proxy Integration*[3]. Questo meccanismo consente di associare un'*API* ad una *Lambda* da eseguire ogni volta che arriva una richiesta all'*API*.
Il meccanismo si occupa anche di impacchettare le richieste ricevute dall'*API Gateway*, in modo

[3]Lambda Proxy Integration: https://docs.aws.amazon.com/apigateway/latest/developerguide/set-up-lambda-proxy-integrations.html

tale che possano essere ricevute in un formato corretto dalle *Lambda*.

Per poter definire quale *Lambda* associare ad un determinato *endpoint*, all'interno della specifica *OpenAPI* viene utilizzano un oggetto chiamato *x-amazon-apigateway-integration*. Esso permette di definire univocamente la risorsa *AWS* da invocare quando arriva una richiesta all'*endpoint* dove l'oggetto è situato. In questo caso la risorsa invocata sarà sempre una *Lambda*.

4.1.5 *Amazon Lambda*

Introduzione

Amazon Lambda[4] è un servizio di *AWS* che permette l'esecuzione di codice senza la necessità di fornire o gestire l'infrastruttura dove il codice viene eseguito. Il codice di una particolare *Lambda* viene eseguito automaticamente ogni volta che si verifica un particolare evento ad essa associato, definito come *trigger* della *Lambda*.
Il servizio offre la possibilità di eseguire codice di diversi linguaggi e di configurare la memoria necessarie per l'esecuzione.
Il pagamento del servizio avviene in base al numero di millisecondi che la *Lambda* ha impiegato per eseguire, più una quota fissa ad ogni richiesta di esecuzione.

Utilizzo del servizio

Ad ogni funzionalità del servizio corrisponde una *Lambda*, che può essere invocata da una richiesta sull'*endpoint* associato all'operazione da eseguire. Il *binding* tra un *endpoint* e la *Lambda* avviene attraverso il meccanismo di *Lambda Proxy Integration* precedentemente descritto.

Flusso di esecuzione

La funzione riceve in *input* la richiesta del *client*, opportunamente impacchettata. Prima di eseguire l'operazione, esegue i seguenti passi:

1. Recupera i *path* e *query params* della richiesta;
2. Decodifica l'eventuale *body* della richiesta in un oggetto del linguaggio, utilizzando i meccanismi di *unmarshalling* forniti dal linguaggio;
3. Valida semanticamente gli *input* decodificati.

Nel caso la fase di validazione fallisse, la *Lambda* fornisce al *client* (in questo caso l'*API Gateway*) una risposta con *status code 400 (Bad Request)*, dove nel *body* viene specificato il motivo dell'errore.

Se la fase di validazione viene superata, la *Lambda* si occupa di interagire con il sistema di persistenza, per eseguire effettivamente l'operazione richiesta sul *datastore*. Le operazioni che vengono eseguite sul sistema di persistenza corrispondo alle classiche operazioni *CRUD (Create-Read-Update-Delete)*.

Quando l'interazione con il sistema di persistenza si è conclusa con successo, la *Lambda*, nel caso l'operazione lo richiedesse, si occupa dell'interazione con il sistema di monitoraggio con due diverse azioni:

1. Registra una metrica contenente un particolare valore relativo all'operazione svolta;
2. Invia la metrica registrata al sistema di monitoraggio.

Quando anche questa interazione si è conclusa, la *Lambda* ritorna al *client* una risposta con *status code 2xx*, che può variare in base al tipo di operazione richiesta. Il *body* della risposta può contenere un messaggio di conferma o altri tipi di oggetti. È importante sottolineare che l'operazione di

[4] Amazon Lambda: https://aws.amazon.com/lambda/

marshalling del *body* viene effettuata con i meccanismi forniti dal linguaggio.

Nel caso in cui l'operazione fallisse per via di un errore semantico o di un errore di interazione con il sistema di persistenza, la *Lambda* non va mai in errore. È stata fatta questa scelta poichè se la funzione ritornasse un errore (un oggetto di tipo `error` in *Go*), questo genererebbe una risposta automatica al *client* con *status code 502 (Bad Gateway)*, senza nessuna informazione aggiuntiva sull'errore.
L'approccio di gestione degli errori utilizzato prevede che, in caso si verificasse un errore, venga generata una risposta con *status code 500 (Internal Server Error)*, dove nel *body* è contenuta una spiegazione sul motivo del fallimento dell'operazione.

Le interazioni con entrambi i sistemi avvengono attraverso l'utilizzo dell'*AWS SDK*, fornita direttamente *AWS* per vari linguaggi di programmazione, tra cui *Go*, il linguaggio scelto.

All'interno dell'infrastruttura del servizio, le uniche componenti che contengono del codice in senso stretto sono le *Lambda*.

4.1.6 *Amazon DynamoDB*

Introduzione

Amazon DynamoDB[5] è un servizio di *AWS* che offre la possibilità usufruire di un *database NoSQL (Structured Query Language)*, completamente gestito e che garantisce *performance* elevate.
Il suo utilizzo è ideale quando si conoscono i *pattern* di accesso ai dati: in questo modo è possibile fare *query* semplici solo sulle chiavi o su indici, in modo da renderle veloci ed efficienti.
Il costo del servizio può essere addebitato *on-demand* o secondo un livello di carico di letture e scritture prestabilito. Nel caso del servizio di code, si utilizza la prima modalità, che addebita il costo in base al numero e alla dimensione delle letture e scritture sul *database*.

Concetti chiave

Vengono ora illustrati alcuni concetti chiave di *DynamoDB*.

Tabelle Sono delle collezioni di *item*, dove ognuno è composto da un insieme di attributi (coppie *key: value*) senza avere uno schema rigido. I principali tipi per gli attributi sono `String`, `Number` e `Binary`.
Una tabella è specifica di una regione geografica, è quindi possibile avere più tabelle con lo stesso nome su regioni diverse.

Partizione I dati di una tabella vengono partizionati su macchine fisiche o virtuali. L'effetto di questo approccio è il bilanciamento del carico delle tabelle, al fine mantenere le operazioni su di esse efficienti al crescere del numero di *item* presenti.

Chiave primaria Una *primary key* è composta da un attributo chiamato *partition key*, sul quale si applicano algoritmi di *hash* per recuperare e dividere i dati tra più *host* per ragioni di scalabilità e disponibilità.
La chiave può eventualmente contenere un altro attributo, chiamato *sorting key*, usata per ordinare e cercare gli *item* di una tabella all'interno di una partizione.
La chiave primaria, sia essa composta da *partition key* o *partition key* e *sorting key*, deve sempre poter identificare univocamente un *item* di una tabella.

Global Secondary Index (GSI) Permette di definire un indice su alcuni attributi di una tabella, in modo tale da velocizzare l'accesso e le *query* sugli attributi. Il loro principale scopo, infatti, è quello di rendere efficiente e rapido l'accesso ad un *subset* degli attributi di un *item*.

[5] Amazon DynamoDB: https://aws.amazon.com/dynamodb/

Un indice di questo tipo deve avere una *partition key* ed eventualmente una *sorting key*; questi due attributi fungono da chiave primaria per l'accesso all'indice.
È possibile definire anche quali attributi, oltre a quelli della chiave primaria, fanno parte della proiezione di una *query* sull'indice.

Query L'operazione di scansione lineare di una tabella risulta essere molto inefficiente. Per questo, quando vi è la necessità di recuperare dei dati dalle tabelle, è preferibile utilizzare le *query*. Una *query* può essere eseguita su una tabella o su indice definito a partire da una tabella. In entrambi i casi bisogna sempre definire una condizione sulla chiave primaria. È possibile definire delle condizioni sugli attributi ed eventualmente indicare quali attributi ritornare.

Tabella *messages*

La tabella principale contiene tutti i *task* che sono stati accodati dai servizi Thron e che non sono ancora stati processati correttamente e completamente. Lo schema della tabella *messages* è visibile nella tabella 4.1.

Tabella 4.1: Schema della tabella *messages*

Nome attributo	Descrizione	Tipo
`MessageId`	*UUID (Universally Unique IDentifier)* generato dal servizio	`String`
`QueueAndShardId`	Concatenazione di `queue-id` e `client-id`	`String`
`VisibleAfter`	Data e ora in cui il *task* ritornerà visibile a tutti i consumatori della coda	`String` (*RFC3339*)
`EnqueuedDate`	Data e ora in cui il *task* è stato accodato	`String` (*RFC3339*)
`Payload`	Contenuto del *task* (oggetto *JSON*)	`Binary` (*base64*)
`ConsumerId`	*Id* del consumatore che sta eseguendo il *task*	`String`
`Attempts`	Numero di tentativi di esecuzione falliti	`Number`

La chiave primaria di questa tabella è composta solamente da una *partition key* (`MessageId`). Questa è sufficiente poichè permette di identificare univocamente un *task* all'interno della tabella. Non è necessario avere una *sorting key*, in quanto non serve ordinare i *task* secondo qualche criterio direttamente nella tabella principale.

L'attributo `QueueAndShardId` permette di implementare, in parte, alcuni requisiti funzionali essenziali. La combinazione del `QueueId` e dello `ShardId` consente di avere più code per ogni servizio Thron (basta definirle con un nome diverso) e, a livello astratto, all'interno di ogni coda, più code per ogni *client-id*.
In questa maniera è possibile avere più code per ogni servizio e implementare in parte il concetto di *fairness*, poichè è possibile prelevare da ogni coda un certo numero di *task* associati ad un particolare *tenant*.

`VisibleAfter` modella il concetto di invisibilità di un *task* all'interno della coda. Dato che il formato orario *RFC3339* fornisce un ordinamento lessicografico, è possibile recuperare tutti i *task* visibili con una condizione sulla *query*, creando un *global secondary index*.

Il `Payload` contiene il contenuto del *task*, ovvero tutte le informazioni necessarie al consumatore per eseguire il lavoro. Nell'*API* di accodamento questo viene preso in *input* come *JSON object*, il servizio si occupa di tradurlo in *bytes* e salvarne il contenuto su *DynamoDB*. La codifica, in *base64*, viene fatta automaticamente dall'*AWS SDK* in fase di *marshalling*.

Indice su *messages*

Nella tabella `messages` vengono fatte operazioni di inserimento, modifica e cancellazione, ma non di selezione. Per agevolare l'operazione di recupero dei *task* da eseguire (selezione), viene definito un indice secondario globale. L'indice si chiama `QueueShard-VisibleAfter-Idx` ed ha lo schema riportato nella tabella 4.2.

Tabella 4.2: Schema del *GSI QueueShard-VisibleAfter-Idx*

Nome attributo	Descrizione	Tipo
`MessageId`	*UUID* generato dal servizio	`String`
`QueueAndShardId`	Concatenazione di `queue-id` e `client-id`	`String`
`VisibleAfter`	Data e ora in cui il *task* ritornerà visibile a tutti consumatori della coda	`String` (*RFC3339*)

La chiave primaria di questo indice è composta da una *partition key* (`QueueAndShardId`) e una *range key* (`VisibleAfter`).
Ordinare gli *item* di questo indice permette di ottenere elevata efficienza nella ricerca di *task* visibili in una determinata coda associate ad un particolare *client-id*.
Questo aspetto rende la prima parte dell'operazione di prelievo molto efficiente e poco costosa, in quanto si andranno sempre a recuperare dei *task* visibili in una determinata coda.

Tabella *Queue-Shard*

`Queue-Shard` è la seconda tabella utilizzata. Questa è una tabella di utilità, necessaria per la realizzazione della garanzia della *fairness*. Lo schema è riassunto nella tabella 4.3.

Tabella 4.3: Schema della tabella *Queue-Shard*

Nome attributo	Descrizione	Tipo
`QueueId`	*Id* della coda	`String`
`ShardId`	*Id* del *tenant*	`String`

La chiave primaria della tabella è composta da una *partition key* (`QueueId`) e una *range key* (`ShardId`). L'ordinamento alfabetico degli *shard* è fondamentale per andare ad implementare la logica di *fairness* descritta nella sezione 4.1.9.

Ogni volta che un servizio Thron accoda un *task* da eseguire, in questa tabella viene inserito un *item*. In questo modo è possibile registrare tutti gli *shard-id* che sono passati in una determinata coda, un aspetto chiave per garantire la *fairness* tra i vari clienti.

Tabella *Dead-Letter-Queue*

L'ultima tabella definita nel *database* è la `Dead-Letter-Queue`, che implementa il concetto di *dead letter queue*. Lo schema è riassunto nella tabella 4.4.

Tabella 4.4: Schema della tabella *Dead-Letter-Queue*

Nome attributo	Descrizione	Tipo

`MessageId`	*UUID* generato dal servizio	`String`
`QueueAndShardId`	Concatenazione di `queue-id` e `client-id`	`String`
`EnqueuedDate`	Data e ora in cui il *task* è stato accodato	`String` (*RFC3339*)
`Payload`	Contenuto del *task* (oggetto *JSON*)	`Binary` (*base64*)
`ConsumerId`	*Id* del consumatore che sta eseguendo il *task*	`String`

La tabella presenta una *primary key* formata da una *partion key* (`MessageId`).
Lo schema suggerisce che la tabella sia molto simile alla tabella `messages`, questo perchè il suo scopo è quello di tenere traccia di tutti i *task* che non sono stati processati correttamente per un certo numero di volte.

4.1.7 *Amazon CloudWatch*

Introduzione

Amazon CloudWatch[6] è un servizio di *AWS* che offre un sistema di monitoraggio per le applicazioni. Esso fornisce la possibilità di definire e collezionare, con diverse risoluzioni temporali, metriche per monitorare lo stato di un'applicazione o lo stato di qualche componente dell'applicazione.
Il servizio mette a disposizione una *dashboard* personalizzabile, attraverso la quale è possibile visualizzare lo stato delle metriche ed eseguire operazioni. È possibile anche definire delle sonde a partire dalle metriche.
Il costo del servizio viene calcolato in base al numero di misurazioni inviate al sistema.

Concetti chiave

Metrica Una metrica rappresenta un dato sulle prestazioni di una componente dell'applicazione. Le sue misurazioni sono delle tuple composte dall'istante in cui avviene la misurazione e dal valore della misurazione.

Dimensione Una dimensione è coppia *nome-valore* che è parte dell'identità di una metrica. Poiché è parte integrante dell'*id* di una metrica, ogni volta che se ne aggiunge una, si aggiunge una nuova variazione della metrica. Vengono utilizzate per specificare più nel dettaglio cosa la metrica sta andando a misurare.

Aggregazione Per visualizzare le misurazioni di una metrica, è possibile utilizzare delle aggregazioni statistiche. Vi sono vari tipi di aggregazione, che permettono di combinare le misurazioni con diverse operazioni matematiche. Alcuni esempi sono la somma, la media, il massimo e il minimo. Per ogni tipo di aggregazione deve essere definito anche l'intervallo temporale su cui verrà effettuata l'aggregazione.

Metriche

Vengono elencate nel dettaglio le metriche individuate per il monitoraggio delle code definite nel servizio.
Per tutte le metriche valgono le seguenti proprietà:

* **Salvataggio in alta risoluzione**: vengono salvate con granularità al secondo. Di *default* *CloudWatch* aggrega tutti i valori misurati all'interno del minuto in cui esse avvengono, ma questo potrebbe non rendere visibile dei valori anomali e interessanti da monitorare;
* **Dimensioni**: su ogni metrica sono state definite due dimensioni *custom*: l'*id* della coda e l'*id* del *tenant*. In questo modo è possibile avere metriche molto specifiche, che monitorano lo stato delle code ma anche lo stato dei lavori creati dai vari *tenant* all'interno delle code.

[6] Amazon CloudWatch: https://aws.amazon.com/cloudwatch/

Le metriche selezionat3 sono elencate nella tabella 4.5.

Tabella 4.5: Metriche esposte dal servizio

Nome metrica	Descrizione	Aggregazione
`MessagesEnqueued`	Numero di *task* accodati in una coda, per un determinato *tenant*	Somma
`MessagesDequeued`	Numero di *task* recuperati da una coda, per un determinato *tenant*	Somma
`EmptyDequeues`	Numero di recuperi andati a vuoto in una coda, per un determinato *tenant*	Somma
`MessagesRemoved`	Numero di *task* rimossi da una coda, per un determinato *tenant*	Somma
`FailedAttempts`	Numero di tentativi di esecuzione falliti in una coda, per un determinato *tenant*	Somma
`DurationInQueue`	Tempo di permanenza di un *task* all'interno da una coda, per un determinato *tenant*	Media

Per tutte le metriche, ad eccezione dell'ultima, è ragionevole una visualizzazione utilizzando la somma come aggregazione statistica. Tutti i valori di queste metriche sono valori unitari, per cui, al fine di vedere l'andamento nel tempo, è necessario sommare tutte le singole unità. In questo modo è possibile vedere delle variazioni sensate in base al periodo di aggregazione scelto.
Per l'ultima metrica ha più senso utilizzare la media come aggregazione statistica. Questo perché è una misurazione indica il numero di secondi che un *task* ha passato all'interno di una coda, e rende possibile visualizzare la variazione dei tempi di esecuzione nel corso del tempo.

4.1.8 Esposizione funzionalità

In questa sezione vengono esposte le funzionalità per la gestione delle code e dei *task* all'interno di esse. Per ogni funzionalità si illustreranno i seguenti aspetti:

* Scopo e funzionamento generale;
* *API REST endpoint*;
* Interazione con *DynamoDB*;
* Eventuale interazione con *CloudWatch*.

Enqueue

Scopo L'operazione permette ad un servizio Thron di accodare dei *task* all'interno di una coda. È possibile anche definire automaticamente delle nuove code, andando ad accodare un primo *task* all'interno della nuova coda che si vuole creare.

API REST

***Endpoint*:** `/queues/{queue}/messages`
Metodo: `POST`
Headers:

Tipo	Valore
`Content-Type`	`application/json`

Parametri:

Nome	Posizione	Tipo	Descrizione
`queue`	*path*	`string`	Nome della coda dove accodare il *task*
`shard`	*boby*	`string`	Nome del *tenant* da associare al *task*
`payload`	*body*	`object`	Contenuto del *task* da eseguire

Risposta:

Status code	*Body*	Descrizione

201	`{`
	`"id": string,`
	`"queueId": string,`
	`"shardId": string,`
	`"visibleAfter": string (RFC3339),`
	`"payload": object`
	`}`
		Il *task* è stato accodato con successo

400	`{`
	`"error": string`
	`}`
		La richiesta contiene dei parametri non validi

500	`{`
	`"error": string`
	`}`
		Il sistema non è riuscito ad accodare il *task*

Interazione con *DynamoDB* Per eseguire questa operazione, la *Lambda* si interfaccia con *DynamoDB* in due occasioni.
La prima interazione inserisce un *item*, che ricalca il *task*, nella tabella `messages` attraverso un'operazione chiamata `PutItem`, fornita da *DynamoDB*.
La seconda inserisce, con la medesima funzionalità di *DynamoDB*, un *item* corrispondente alla tupla *queue-shard* nella tabella `Queue-Shard`. Grazie ad un peculiarità di *DynamoDB*, qualora la tupla fosse già presente, verrà sovrascritta, in modo da garantire l'assenza di duplicati.

Interazione con *CloudWatch* L'operazione di accodamento registrerà una misurazione sulla metrica `MessagesEnqueued`, per segnalare che in quell'istante è stato accodato un nuovo *task*. La misurazione avrà il valore 1, in quanto è stato accodato un singolo *task*.

Dequeue

Scopo L'operazione permette ad un servizio Thron di recuperare dei *task* che sono state precedentemente accodati all'interno di una coda. L'operazione si svolge in due fasi differenti: nella prima fase vengono raccolti un certo numero di *task* visibili e nella seconda questi vengono resi invisibili all'interno della coda, essendo ora detenuti da un consumatore del servizio Thron che possiede la coda.

API REST

***Endpoint*: `/queues/{queue}/messages`**
Metodo: `GET`
Headers:

Tipo	Valore
`Content-Type`	`application/json`

Parametri:

Nome	Posizione	Tipo	Descrizione
`queue`	*path*	`string`	Nome della coda da dove recuperare i *task*
`consumer`	*query*	`string`	Nome del consumatore che vuole recuperare i *task*

`items`	*query*	`integer`	Numero di *task* da prelevare
`timeout`	*query*	`integer`	Numero di secondi per cui i *task* prelevati saranno invisibili
`lastShard`	*query*	`string`	Ultimo *tenant* interrogato. Più informazioni nella sezione 4.1.9

Risposta:

Status code	***Body***	**Descrizione**
200	`{ "lastRetrievedShard": string, "messages": array[object] }`	Sono stati recuperati con successo dei *task* da eseguire
400	`{ "error": string }`	La richiesta contiene dei parametri non validi
500	`{ "error": string }`	Il sistema non è riuscito a recuperare dei *task* da eseguire

Interazione con *DynamoDB* L'operazione di *dequeue* è quella più onerosa dal punto di vista dell'interazione con il sistema di persistenza, dovendo, tra le altre cose, garantire la *fairness*.
Una prima interazione recupera tutti i *tenant* registrati su una determinata coda, attraverso una *query* sulla tabella `Queue-Shard`.
Una volta recuperati viene utilizzato il parametro `lastShard` per riprendere a recuperare *task* a partire dal *tenant* corretto dal punto di vista della *fairness*. Vengono eseguite più *query* sull'indice `QueueShard-VisibleAfter-Idx`, le quali recuperano un certo numero di *task* visibili per ogni *tenant* interrogato.
Quando sono stati prelevati i `MessageId` dei *task* da recuperare, su ognuno di essi viene eseguito un'*update*, per renderli invisibili agli altri consumatori della coda.

Interazione con *CloudWatch* L'operazione di recupero registrerà una misurazione sulla metrica `MessagesDequeued`, per segnalare che in quell'istante sono stati prelevati dei *task* dalla coda; il valore della misurazione corrisponderà al numero di *task* effettivamente prelevati.
Nel caso non ci fossero *task* da prelevare, l'operazione registrerà una misurazione sulla metrica `EmptyDequeues` con valore 1.

Details

Scopo L'operazione permette ad un servizio Thron di recuperare le informazioni su un *task* precedentemente accodato all'interno di una coda.

API REST

***Endpoint*: `/queues/{queue}/messages/{message}`**
Metodo: `GET`
Headers:

Tipo	**Valore**
`Content-Type`	`application/json`

Parametri:

Nome	**Posizione**	**Tipo**	**Descrizione**
`queue`	*path*	`string`	Nome della coda da dove recuperare i dettagli del *task*
`message`	*path*	`string`	*Id* del *task* di cui recuperare i dettagli

Risposta:

Status code	*Body*	Descrizione

200	{
	"id": string,
	"queueId": string,
	"shardId": string,
	"visibleAfter": string (RFC3339),
	"payload": object
	}
		Le informazioni sul *task* sono state recuperate con successo

400	{
	"error": string
	}
		La richiesta contiene dei parametri non validi

500	{
	"error": string
	}
		Il sistema non è riuscito a recuperare le informazioni sul *task* richiesto

Interazione con *DynamoDB* Questa operazione si interfaccia con *DynamoDB* per recuperare un *item* della tabella `messages`. Per farlo utilizza un'operazione di *DynamoDB* chiamata `GetItem`, che, fornita una chiave primaria, permette di recuperare tutti gli attributi dell'*item* con quella chiave.

Interazione con *CloudWatch* Questa operazione non prevede la registrazione di metriche.

Forced remove

Scopo L'operazione permette ad un servizio Thron di rimuovere in modo forzato un *task* da una coda. *"Rimuovere in modo forzato"* significa che non vengono fatti controlli di sicurezza prima di eseguire l'operazione. Questo può voler dire che potrebbe essere eliminato un *task* già prelevato da un consumatore.
Questa operazione risulta molto utile nel caso in cui si dovesse annullare un'operazione richiesta a livello di servizio Thron.

API REST

***Endpoint*:** `/queues/{queue}/messages/{message}`
Metodo: `DELETE`
Headers:

Tipo	Valore
`Content-Type`	`application/json`

Parametri:

Nome	Posizione	Tipo	Descrizione
`queue`	*path*	`string`	Nome della coda da dove risiede il *task* da eliminare
`message`	*path*	`string`	*Id* del *task* da eliminare

Risposta:

Status code	*Body*	Descrizione

200	{
	"message": string
	}
		Il *task* è stato rimosso correttamente dalla coda

400	{
	"error": string
	}
		La richiesta contiene dei parametri non validi

500	{
	"error": string
	}
		Il sistema non è riuscito ad eliminare il *task* specificato

Interazione con *DynamoDB* Per eliminare il *task*, la *Lambda* effettua una `DeleteItem` sulla tabella `messages`. Questa operazione di *DynamoDB* consente di eliminare un *item* avente una determinata *primary key*, fornita da chi richiede l'operazione.

Interazione con *CloudWatch* Questa operazione registra che un *task* è stato rimosso da una determinata coda. Ciò corrisponderà all'invio di una misurazione con valore 1 sulla metrica `MessagesRemoved`.

Update

Scopo L'operazione permette ad un consumatore di aggiornare il contenuto di un *task* precedentemente prelevato. In questo modo è possibile elaborare un *task*, contenete un lavoro lungo, in più *step*.

API REST

***Endpoint*:** `/queues/{queue}/consumers/{consumer}/messages/{message}`
Metodo: `PATCH`
Headers:

Tipo	Valore
`Content-Type`	`application/json`

Parametri:

Nome	Posizione	Tipo	Descrizione
`queue`	*path*	`string`	Nome della coda dove risiede il *task* da aggiornare
`consumer`	*path*	`string`	*Id* del consumatore che sta richiedendo l'aggiornamento
`message`	*path*	`string`	*Id* del *task* da aggiornare
`newPayload`	*body*	`object`	Nuovo contenuto da assegnare al *task*

Risposta:

Status code	*Body*	Descrizione

200	{
	"message": string
	}
		Il *task* è stato aggiornato correttamente

400	{
	"error": string
	}
		La richiesta contiene dei parametri non validi

500	{
	"error": string
	}
		Il sistema non è riuscito ad aggiornare il *task* specificato

Interazione con *DynamoDB* L'operazione richiede di effettuare l'aggiornamento di un *item* all'interno della tabella `messages`. Tramite l'operazione `UpdateItem`, fornita da *DynamoDB*, è possibile aggiornare alcuni attributi di un *item* di cui viene fornita la chiave primaria.
Questa operazione permette anche di definire delle condizioni prima di poter eseguire l'*update*. Questo meccanismo viene utilizzato per controllare due aspetti:

* **Appartenenza del *task***: il *task* deve essere detenuto dal consumatore che ne sta richiedendo l'aggiornamento;

* **Invisibilità**: il *task* deve essere ancora invisibile agli altri consumatori, altrimenti questo potrebbe generare delle *race conditions* nel caso in cui più consumatori stessero processando lo stesso *task*.

Interazione con *CloudWatch* Questa operazione non prevede la registrazione di metriche.

Change visibility

Scopo L'operazione permette ad un consumatore, di un qualche servizio Thron, di prolungare il tempo di invisibilità di un *task* precedentemente prelevato. Questa operazione risulta essere molto utile ai consumatori che necessitano di più tempo, rispetto a quanto preventivato, per portare a termine l'elaborazione del *task*.

API REST

***Endpoint*:** `/queues/{queue}/consumers/{consumer}/messages/{message}/visibility`
Metodo: `PATCH`
Headers:

Tipo	Valore
`Content-Type`	`application/json`

Parametri:

Nome	Posizione	Tipo	Descrizione
`queue`	*path*	`string`	Nome della coda dove risiede il *task* da aggiornare
`consumer`	*path*	`string`	*Id* del consumatore che sta richiedendo l'aggiornamento
`message`	*path*	`string`	*Id* del *task* da aggiornare
`timeout`	*body*	`integer`	Numero di secondi, a partire dall'istante della richiesta, per cui il *task* resterà ancora invisibile

Risposta:

Status code	*Body*	Descrizione
200	`{ "message": string }`	Il tempo di invisibilità del *task* è stato prorogato con successo
400	`{ "error": string }`	La richiesta contiene dei parametri non validi
500	`{ "error": string }`	Il sistema non è riuscito a prorogare il tempo di invisibilità per il *task* specificato

Interazione con *DynamoDB* L'operazione richiede di effettuare l'aggiornamento di un *item* all'interno della tabella `messages`. Tramite l'operazione `UpdateItem`, fornita da *DynamoDB*, è possibile aggiornare alcuni attributi di un *item* di cui viene fornita la chiave primaria.
Questa operazione permette anche di definire delle condizioni prima di poter eseguire l'*update*. Questo meccanismo viene utilizzato per controllare due aspetti:

* **Appartenenza del *task*** : il *task* deve essere detenuto dal consumatore che sta richiedendo di prorogarne il tempo di invisibilità;
* **Invisibilità**: il *task* deve essere ancora invisibile agli altri consumatori.

Interazione con *CloudWatch* Questa operazione non prevede la registrazione di metriche.

Increment attempts

Scopo L'operazione permette ad un consumatore, di un qualche servizio Thron, di segnalare che un *task*, precedentemente prelevato ed eseguito, non è stato portato a termine con successo. Le ragioni per cui questo accade possono essere molteplici: il *task* contiene un lavoro che si conclude costantemente con un errore o il lavoro viene interrotto e non è più possibile proseguire.
Dopo un certo numero di tentativi falliti per un *task*, il servizio di code può decidere di rimuoverlo dalla coda principale e spostarlo nella *dead letter queue*.

API REST

***Endpoint*:** `/queues/{queue}/consumers/{consumer}/messages/{message}/attempts`
Metodo: `PATCH`
Headers:

Tipo	Valore
`Content-Type`	`application/json`

Parametri:

Nome	Posizione	Tipo	Descrizione
`queue`	*path*	`string`	Nome della coda dove risiede il *task* da segnalare
`consumer`	*path*	`string`	*Id* del consumatore che sta richiedendo la segnalazione
`message`	*path*	`string`	*Id* del *task* da segnalare

Risposta:

Status code	***Body***	**Descrizione**
200	`{ "message": string }`	La segnalazione di fallimento del *task* è stata inviata con successo
400	`{ "error": string }`	La richiesta contiene dei parametri non validi
500	`{ "error": string }`	Il sistema non è riuscito ad inviare la segnalazione sul *task* specificato

Interazione con *DynamoDB* L'operazione richiede di effettuare l'aggiornamento di un *item* all'interno della tabella `messages`. Tramite l'operazione `UpdateItem`, fornita da *DynamoDB*, è possibile aggiornare alcuni attributi di un *item* di cui viene fornita la chiave primaria. In questo caso viene incrementato di una unità l'attributo `Attempts`.
Questa operazione permette anche di definire delle condizioni prima di poter eseguire l'*update*. Questo meccanismo viene utilizzato per controllare due aspetti:

* **Appartenenza del *task*** : il *task* deve essere detenuto dal consumatore che sta richiedendo l'operazione;
* **Invisibilità**: il *task* deve essere ancora invisibile agli altri consumatori.

Nel caso in si fosse superato il numero di tentativi falliti massimo, il servizio di code provvederà a rimuovere il *task* dalla tabella `messages` utilizzando una `DeleteItem`. Successivamente, attraverso un'operazione `PutItem`, andrà ad inserire il medesimo *item* nella tabella `Dead-Letter-Queue`.

Interazione con *CloudWatch* Questa operazione registra che un *task* non è stato eseguito con successo. Questo corrisponderà all'invio di una misurazione con valore 1 sulla metrica `FailedAttempts`.

Safe remove

Scopo L'operazione permette ad un consumatore di rimuovere dalla coda di appartenenza un *task* che è stato portato a termine con con successo.

API REST

***Endpoint*:** `/queues/{queue}/consumers/{consumer}/messages/{message}`
Metodo: `DELETE`
Headers:

Tipo	Valore
`Content-Type`	`application/json`

Parametri:

Nome	Posizione	Tipo	Descrizione
`queue`	*path*	`string`	Nome della coda dove risiede il *task* da rimuovere
`consumer`	*path*	`string`	*Id* del consumatore che sta richiedendo la rimozione
`message`	*path*	`string`	*Id* del *task* da rimuovere

Risposta:

Status code	*Body*	Descrizione
200	`{ "message": string }`	Il *task* è stato rimosso con successo
400	`{ "error": string }`	La richiesta contiene dei parametri non validi
500	`{ "error": string }`	Il sistema non è riuscito a rimuovere il *task* specificato

Interazione con *DynamoDB* L'operazione prevede la rimozione di un *item* dalla tabella `messages`. Questa può essere effettuata con una `DeleteItem`, nella quale si va a specificare la *primary key* dell'*item* da rimuovere.
A differenza dell'operazione di *forced remove*, questo tipo di rimozione è *safe*. Vengono poste due condizioni sull'operazione di *delete*, che controllano:

* **Appartenenza del *task*** : il *task* deve essere detenuto dal consumatore che ne sta richiedendo la rimozione;
* **Invisibilità**: il *task* deve essere ancora invisibile agli altri consumatori.

Interazione con *CloudWatch* Questa operazione registra che un *task* è stato eseguito con successo e quindi rimosso dalla coda di appartenenza. Questo corrisponderà all'invio di una misurazione con valore 1 sulla metrica `MessagesRemoved`.
Contestualmente alla rimozione, il servizio invierà anche il tempo di permanenza del *task* all'interno della coda. Verrà effettuato un *push* di una misurazione sulla metrica `DurationInQueue`, il cui valore sarà uguale al numero di secondi che sono trascorsi tra l'operazione di *enqueue* e *safe remove*.

4.1.9 Gestione della *fairness*

La garanzia della *fairness* è un aspetto focale su cui è incentrato questo progetto di *stage*.
Come detto in precedenza, per garanzia della *fairness* si intende che i *task* di ogni cliente (o *tenant*) debbano essere processate in modo equo, indipendentemente dal numero di *task* presenti per cliente

e dal tipo di cliente.

Premesse

Il contesto d'uso della libreria di code già esistente è completamente diverso rispetto a questo servizio di code. Essa infatti definiva un *container*, sulla quale vivevano le istanze delle code generate dai servizi Thron. Questo tipo di architettura è *stateful*, ovvero è in grado di mantenere uno stato interno in cui può salvare ed utilizzare informazioni utili per la sua esecuzione.

La *fairness*, sulla libreria, è stata implementata utilizzando un classico meccanismo di *round robin*. In questo caso l'algoritmo veniva applicato sui vari *tenant*, che a turno venivano interrogati per recuperare dei *task* da eseguire a loro associati. Nella libreria era stato implementato anche un meccanismo di sincronizzazione, che allo scadere di un intervallo temporale, andava ad aggiornare la lista dei *tenant* su cui applicare il *round robin*. Questo approccio era possibile grazie alla persistenza di uno stato sull'istanza della coda.

Il servizio di code, nato con architettura orientata all'esecuzione su ambienti *serverless*, non ha la possibilità di implementare un meccanismo di *fairness* come quello presente nella libreria.

Implementazione

Dopo una serie di ragionamenti, evidenziati nel dettaglio nella sezione 4.2.4, si è giunti alla conclusione che la migliore soluzione per implementare la *fairness* nel senso stretto del termine, fosse realizzare un meccanismo di *round robin* senza stato.

Data la natura *serverless* del servizio, ci si è dovuti in parte affidare ai servizi utilizzatori per poter soddisfare questo requisiti funzionale essenziale.
Il servizio di code, grazie alla tabella di supporto `Queue-Shard` definita su *DynamoDB*, è in grado di recuperare, in ordine alfabetico l'elenco dei *tenant* che sono stati registrati in una determinata coda. Il fatto che sia possibile recuperarli in ordine alfabetico rende possibile l'implementazione di un meccanismo di *round robin*, dove viene comunicato dall'esterno il punto in cui si è arrivati nello scorrere la lista dei *tenant*.

Come illustrato precedentemente, nell'operazione di *dequeue* viene ritornato al consumatore del servizio Thron anche l'ultimo *tenant* ispezionato per la ricerca di *task* da eseguire. Un parametro analogo viene preso in *input* sulla medesima operazione, e questo consente al servizio di capire a che punto della lista è arrivato.
Il consumatore passa in *input* lo stesso parametro ricevuto in *output* nella *dequeue* precedente. Questo permette quindi la "sincronizzazione" del servizio con la lista dei *tenant* da scorrere, facendogli capire a quale punto della lista era arrivato.

A questo punto il servizio ricomincerà a scorrere la lista dei *tenant* dall'elemento successivo a quello ricevuto in *input*. In un funzionamento a regime è quindi garantita la *fairness* in senso stretto: tutti i *tenant* verranno prima o poi interrogati, e un *tenant* non potrà essere interrogato nuovamente se prima non sono stati interrogati anche tutti gli altri.

È semplice notare che un approccio di questo tipo presenta dei rischi importanti, il cui più rilevante è il fatto di affidare una grande responsabilità sugli utilizzatori del servizio di code.
Se il consumatore di un servizio Thron non include nell'*input* di un'operazione di prelievo il *tenant* che gli ha precedentemente comunicato il servizio di code, allora il meccanismo decade e il servizio di code ricomincerà a scorrere la lista dall'inizio, non garantendo più *fairness*.
Questo rischio è comunque accettabile, in quanto il risultato ottenibile è di gran lunga più importante. Un altro aspetto rassicurante è che gli utilizzatori del servizio di code sono tutti servizi Thron definiti e codificati dall'azienda. Un'attenta integrazione con il servizio di code dovrebbe rendere molto rari, se non del tutto improbabili, i casi in cui i consumatori non si comportano come dovrebbero.

4.1.10 Tecnologie e strumenti

Durante la fase di sviluppo son stati necessari strumenti per la verifica e lo sviluppo dei vari componenti. I principali sono elencati in seguito.

Tecnologie

`Go` Linguaggio di programmazione con il quale si è sviluppato il progetto, in particolare tutto il codice che viene eseguito dalle *Lambda*.

`OpenAPI` Specifica utilizzata per la definizione delle *API REST*.

`JSON` *Markup* usato per lo scambio dei dati via *API*, scelto soprattutto per le capacità di *Go* di effettuare *marshalling* e *unmarshalling* in modo automatico da *JSON* a oggetto del linguaggio e viceversa.

`AWS SDK` Libreria utilizzata per interfacciarsi con il sistema di persistenza *DynamoDB* e con il sistema di monitoraggio *CloudWatch*. È *open source*, presente su *GitHub*.

`Amazon API Gateway` Servizio di *AWS* che ha permesso l'esposizione dei vari *endpoint* a partire da *API REST*.

`Amazon Lambda` Servizio di *AWS* utilizzato per rendere l'infrastruttura del servizio di code *serverless*.

`Amazon DynamoDB` Servizio di *AWS* che ha fornito il *database* su cui si basa il sistema di persistenza del servizio di code.

`Amazon CloudWatch` Servizio di *AWS* che ha permesso l'esposizione delle metriche per il monitoraggio delle code definite all'interno del servizio.

`Amazon CloudFormation` Servizio di *AWS* utilizzato per la definizione dell'infrastruttura del servizio di code basata su altri servizi *AWS*.

`Testify` Libreria di *Go* per lo sviluppo dei test con la possibilità di utilizzare le asserzioni, funzionalità non presente nativamente nel linguaggio.

`Git` Sistema di versionamento del codice utilizzato per il tracciamento del codice realizzato nel corso dello sviluppo del progetto.

`Markdown` Linguaggio di *markup* per creare testo formattato, utilizzato per redigere i documenti descrittivi del servizio.

Strumenti

`GoLand` *IDE* sviluppato da *JetBrains* utilizzato per la codifica del servizio e per parte delle attività di test.

`AWS CodeCommit` Servizio *hosting online* per *repository* che supporta `Git`, di proprietà di *AWS*.

`AWS SAM` *Tool* di *AWS* utilizzato per fare il *deploy* del servizio in un ambiente dedicato, definendo le risorse a partire da un *template file* compatibile con *CloudFormation*.

`Postman` Applicazione per l'interrogazione e il *testing* di *API REST*, utilizzato soprattutto durante le fasi di test.

`Docker` Prodotto utilizzato per istanziare delle istanze locali di servizi *AWS*, utile per il *testing* iniziale delle funzionalità del servizio.

4.2 Codifica

In questa sezione verranno trattate la parti principali e più interessanti dell'attività di codifica e realizzazione del servizio.

4.2.1 Ordine di sviluppo

Il servizio e le sue funzionalità sono state sviluppate in un ordine preciso. Si è cercato di implementare il prodotto con un approccio incrementale, in modo tale da avere sempre un prodotto funzionante dove venissero via via implementati tutti i requisiti individuati.
In particolare, lo sviluppo è stato svolto nel seguente ordine:

1. Definizione dell'infrastruttura del microservizio;
2. Definizione delle strutture e delle funzioni di utilità per l'interazione con il sistema di persistenza e monitoraggio;
3. Implementazione del meccanismo per garantire la *fairness*;
4. Sviluppo delle funzionalità di *enqueue* e *dequeue*;
5. Sviluppo delle funzionalità di *forced remove* e *details*;
6. Sviluppo delle funzionalità di *update*, *change visibility*, *increment attempts* e *safe remove*;
7. Definizione e implementazione di *test* di unità, *test* di integrazione e *test* di sistema;
8. Definizione e implementazione di *test* per la simulazione di casi d'uso reali.

Seguendo questo ordine è risultato più facile implementare i requisiti funzionali, in quanto ogni incremento ha permesso di concentrarsi sui suoi obiettivi specifici.

Ad ogni *milestone* raggiunta, è stato fatto un incontro con il *tutor* aziendale, per verificare la correttezza e la coerenza di quanto sviluppato con le aspettative.
Inoltre si sono tenuti, con cadenza minore, incontri con dei *senior developers* del *team*, all'interno dei quali venivano discussi i punti realizzati. L'*output* di queste riunioni, spesso consigli e critiche costruttive su quanto illustrato, è stato utilizzato come *input* per la fase di sviluppo successiva. Questo ha portato ad un miglioramento costante del prodotto sia dal punto di vista funzionale che dal punto di vista tecnico e implementativo.

4.2.2 Soddisfacimento requisiti

Il prodotto sviluppato implementa tutti i requisiti funzionali obbligatori che sono stati individuati nell'attività di analisi dei requisiti. Per quanto riguarda la realizzazione dei requisiti desiderabili, non è stato possibile soddisfarli tutti quanti.

In particolare il requisito R44-F-D, seppur inizialmente implementato, è stato successivamente scartato, in quanto non consentiva di avere una soluzione elegante che lo implementasse. Maggiori dettagli su questo aspetto saranno discussi nella sezione 4.2.4.

Alla luce dell'esclusione del requisito sopracitato, si è deciso di aggiungere un altro requisito funzionale, che prevedeva le realizzazione di una *dead letter queue*.

Oltre a questo, non ci sono stati altri scostamenti rispetto ai requisiti da implementare. Una volta che la codifica del servizio è stata completata, tutti i requisiti individuati nel corso dell'analisi dei requisiti sono stati correttamente soddisfatti.

4.2.3 Funzionalità aggiuntive

Dead letter queue

L'implementazione di una *dead letter queue* non era prevista nella pianificazione iniziale, è stata aggiunta in un secondo momento, in sostituzione del requisito funzionale riguardante la gestione di *task* con diverse priorità.
Come già descritto, sono stati coinvolti tutti i servizi *AWS* utilizzati per l'implementazione della funzionalità.
È stata definita una tabella su *DynamoDB*, chiamata `Dead-Letter-Queue`. Il fine della tabella è quello di salvare tutti i *task* che non sono stati eseguiti con successo per un certo numero di volte consecutive. Questi sono stati rimossi dalle loro code e spostati in questa coda speciale.

La funzionalità è stata esposta tramite l'*API Gateway* in un apposito *endpoint*.
La logica di segnalazione di fallimento di un *task* e la logica di spostamento dei *task* tra le code è stata implementata all'interno di una *Lambda*, successivamente collegata all'*endpoint* definito.
Associata alla funzionalità, è stata definita una metrica, chiamata `FailedAttempts`, che va a monitorare il numero di *task* falliti in una determinata coda.

L'implementazione di questa funzionalità apporta benefici sia all'interno che all'esterno del servizio.

Internamente al servizio, questa permette di non intasare una coda con una serie di *task* non eseguibili. Se tutti i *task*, compresi quelle non andati a buon fine, venissero lasciati sulla coda, si causerebbe sovraffollamento e inefficienza all'interno delle coda. L'inefficienza sta nel fatto che anche i *task* falliti molte volte continuerebbero ad essere prelevati ed eseguiti dai consumatori, ottenendo il medesimo risultato negativo.
Toglierli dalla coda consente di avere sempre *task* sani e pronti da eseguire, con ripercussioni notevoli sull'efficacia e l'efficienza della logica di *dequeue* e del mantenimento della *fairness*.

Esternamente al servizio, avere una *dead letter queue* consente di salvare tutti i *task* non andati a buon fine. Questo può risultare molto utile in fase di *debug* anche all'esterno del servizio di code: è infatti possibile recuperare il *payload* e verificare se un servizio utilizzatore nasconde qualche *bug* nella sua implementazione.

Stimatore di costi

È importante per l'azienda avere un prospetto delle spese per tutti i servizi che vengono utilizzati dalla piattaforma. Alla luce di questo, si è ritenuto utile avere uno *script* che calcoli la stima mensile dei costi basata su un carico di lavoro variabile, inserito dall'utente.

Sono stati individuati una serie di *pattern* per l'esecuzione dei *task*:

* ***Task* correttamente eseguiti**: *task* che vengono eseguiti correttamente, senza intoppi. Le operazioni, per un singolo *task*, sono: *enqueue*, *dequeue* e *safe remove*;
* ***Task* contenenti lavori lunghi**: *task* che vengono eseguiti correttamente in più *step* di elaborazione. Le operazioni, per un singolo *task*, sono: *enqueue*, *dequeue* 10 volte, *update* 10 volte e *safe remove*;
* ***Task* con più tempo per elaborazione**: *task* che vengono eseguiti correttamente, ma richiedono più tempo per essere conclusi. Le operazioni, per un singolo *task*, sono: *enqueue*, *dequeue*, *change visibility* e *safe remove*;
* ***Task* falliti**: *task* che non vengono elaborati correttamente. Le operazioni, per un singolo *task*, sono: *enqueue*, *dequeue* 5 volte, *increment attempts* 5 volte e *forced remove*;
* ***Task* rimossi**: *task* che vengono rimossi senza essere processati. Le operazioni, per un singolo *task*, sono: *enqueue* e *forced remove*.

Per ogni *pattern* viene chiesto all'utente il numero di *task* che lo seguiranno. Infine lo *script* chiede all'utente di inserire la dimensione media del *payload* in KB. Sulla base dei livello di carico inseriti, viene calcolato il costo per *pattern* e poi il costo complessivo stimato.

Per stimare i costi unitari delle singole operazioni sui servizi *AWS* utilizzati, sono state eseguite delle misurazioni statistiche. Di tali campioni è stato preso in considerazione il caso medio. Di seguito viene elencata una lista dei costi presi in considerazione per singolo servizio *AWS*:

* ***Amazon DynamoDB***: il costo varia in base alle *Read Request Units* e *Write Request Units*. La prima consente di leggere 4KB per unità, la seconda consente di scrivere 1KB per unità. Per cui i costi per questo servizio variano in base alla dimensione del *task* e alla particolare operazione eseguita sul *database*. Sono stati considerati anche i costi di *back-up* e archiviazione;
* ***Amazon Lambda***: il costo varia al variare del numero di millisecondi impiegati per eseguire la funzione. A questo si aggiunge un costo fisso per la chiamata a funzione;
* ***Amazon API Gateway***: il costo di questo servizio varia in base al numero di richieste che arrivano ai vari *endpoint* definiti dal servizio. Viene conteggiata qualsiasi richiesta, indipendentemente che sia ben formata o meno;
* ***Amazon CloudWatch***: il costo del servizio cresce al crescere del numero di misurazioni che vengono registrate nelle varie metriche.

Ogni servizio ha dei casi speciali in cui i costi possono variare ulteriormente. In linea generale sono state fatte delle assunzioni realistiche, in modo tale da poter ridurre il più possibile i casi particolari nel calcolo della stima.

Una volta visualizzato il totale, lo *script* dà la possibilità all'utente di visualizzare il dettaglio dei costi secondo due diverse modalità:

* **Dettaglio costi per servizio *AWS***: per ogni *pattern* viene elencato il costo dei singoli servizi;
* **Dettaglio costi per operazione**: per ogni *pattern* vengono elencati i costi di ogni operazione del servizio di code utilizzata.

4.2.4 Problematiche riscontrate

L'attività di codifica si è conclusa con l'adempimento di tutti i requisiti individuati durante la fase di analisi dei requisiti. Di seguito vengono esposte le problematiche principali riscontrate durante lo sviluppo del progetto e le relative soluzione adottate.

Gestione della *fairness*

La *fairness* è stata un argomento molto caldo per tutta la durata del progetto. Nel corso di questo, sono state elaborate più soluzioni, che sono state via via migliorative.

L'origine del problema è stata la natura *serverless* del servizio di code.
Come spiegato nella sezione 4.1.9, la libreria presente non doveva compiere particolari operazioni per modellare questo requisito: era sufficiente recuperare la lista dei *tenant*, scorrerla e periodicamente sincronizzarla per ottenere eventuali nuovi *client-id*. Questo approccio non presenta problemi quando si lavora con un servizio *stateful*, in quanto è sempre possibile sapere a che punto si è arrivati nello scorrere la lista dei *tenant*.
In un ambiente *serverless* questo non è assolutamente possibile e si sono dovute cercare delle vie alternative per risolvere questo problema ed implementare correttamente il requisito.

Una prima soluzione utilizzava la randomizzazione per garantire l'equità tra i clienti. Il meccanismo prevedeva che, ad ogni richiesta di recupero di *task*, venissero eseguiti i seguenti *step*:

1. Recupero di tutti i *tenant* registrati in una determinata coda;
2. Estrazione random di un *tenant* e recupero di *task* da eseguire;
3. Ripetere il punto 2 fino al raggiungimento del numero di *task* richiesto dal consumatore o fino all'esaurimento dei *tenant* da interrogare.

Questo approccio effettivamente era in grado di risolvere il problema della *fairness* e quindi anche di soddisfare correttamente il relativo requisito. Tuttavia presentava delle limitazioni, che non realizzavano la *fairness* in senso stretto.
Con questa soluzione l'equità era garantita solo da un punto di vista statistico. Questo significa che il suo risultato diveniva visibile solo dopo un alto numero di recuperi da parte dei consumatori, garantendone l'efficacia e la correttezza solo dopo un periodo di tempo molto lungo.
Ciò va contro quello che l'azienda definisce come *fairness*, ovvero che l'equità deve essere garantita nello spazio (indipendentemente dal numero di clienti) e nel tempo (sia nel breve che nel lungo periodo).
Un altro aspetto che rende questa soluzione poco corretta, è il numero di clienti che l'azienda possiede al momento. Essendo questo numero sull'ordine delle centinaia, è più difficile che un approccio randomico garantisca equità.

Grazie all'aiuto di alcuni colleghi del *team*, si è trovata una soluzione molto più corretta, che permette di garantire la *fairness* in senso stretto. La soluzione utilizzata è quella descritta nella sezione 4.1.9.

Gestione di *task* con diverse priorità

Un requisito inizialmente presente, poi tolto, è la gestione di *task* con diverse priorità all'interno della stessa coda.

Il requisito prevede che ogni coda sia virtualmente suddivisa in più code di diversa priorità. L'operazione di *dequeue* doveva quindi recuperare dei *task* tenendo conto della priorità, ovvero restituendo prima quelli più prioritari e poi quelli meno prioritari.

Per soddisfare questo requisito è stata implementata una soluzione che prevedeva l'implementazione di esso in parte sul *database* e in parte via codice, durante l'operazione di *dequeue*.

La soluzione prevedeva uno schema leggermente differente per la tabella `messages`, raffigurato dalla tabella 4.30.

Tabella 4.30: Schema della tabella *messages* con priorità

Nome attributo	Descrizione	Tipo
`MessageId`	*UUID (Universally Unique IDentifier)* generato dal servizio	`String`
`QueueAndShardId`	Concatenazione di `queue-id` e `client-id`	`String`
`PriorityAndVisibleAfter`	Concatenazione di priorità del *task* e *datetime* in cui il *task* ritornerà visibile a tutti consumatori della coda	`String`
`EnqueuedDate`	Data e ora in cui il *task* è stato accodato	`String` (*RFC3339*)
`Payload`	Contenuto del *task* (oggetto *JSON*)	`Binary` (*base64*)
`ConsumerId`	*Id* del consumatore che sta eseguendo la *task*	`String`
`Attempts`	Numero di tentativi di esecuzione falliti	`Number`

Si noti in particolare l'attributo `PriorityAndVisibleAfter`: esso è la concatenazione del livello di

priorità del *task* con la fine del suo tempo di invisibilità. La concatenazione, seppur poco intuitiva, si è resa necessaria in quanto si voleva ordinare i *task* per priorità e tempo di invisibilità, ma su *DynamoDB* è possibile avere solo un attributo per la *range key*. Grazie all'ordinamento lessicografico, con questa tabella i *task* risultano essere ordinati prima per priorità e poi per tempo di invisibilità.

Con questa soluzione, lo schema dell'indice `QueueShard-VisibleAfter-Idx` risulta definito come nella tabella 4.31.

Tabella 4.31: Schema del *GSI QueueShard-VisibleAfter-Idx* con priorità

Nome attributo	Descrizione	Tipo
`MessageId`	*UUID* generato dal servizio	`String`
`QueueAndShardId`	Concatenazione di `queue-id` e `client-id`	`String`
`PriorityAndVisibleAfter`	Concatenazione di priorità del *task* e *date-time* in cui il *task* ritornerà visibile a tutti consumatori della coda	`String`

Questo indice permetteva, con una *query* opportunamente definita, di ottenere tutti i *task* visibili con un certo livello di priorità per un determinato *tenant*. Non è risultato possibile trovare un modo per ottenere un certo numero di *task* ordinati per priorità effettuando una singola *query*.
Sull'assunzione delle precedenti affermazioni, il risultato è che, per recuperare un certo numero di *task* in ordine prioritario, dovevano essere eseguite più *query*, in cui ognuna andava alla ricerca di *task* visibili con un certo livello di priorità. Questo porta ad avere un'operazione di *dequeue* più lenta, inefficiente e costosa per via del maggior numero di interrogazioni verso il *database*.

Dopo aver illustrato la soluzione e le relative limitazioni in un incontro con altri colleghi del *team*, si è giunti alla conclusione che quanto desiderato non poteva essere risolto in maniera efficiente ed elegante usando *DynamoDB* come sistema di persistenza.

Si è quindi deciso di non implementare questo requisito, in quanto raggiungibile direttamente dall'esterno. Per un servizio Thron è possibile definire più code, con priorità diversa. Sarà esso che dovrà gestire internamente le code prioritarie che si è creato.

Metriche

Non è stato possibile definire ed implementare tutte le metriche desiderate.

Inizialmente erano state prese in considerazione anche altre metriche, più interessanti e che permettevano un controllo più approfondito sulle code. Alcuni esempi:

* **Numero *task* visibili**: indica il numero di *task* che sono in attesa di essere processati all'interno di una coda;
* **Numero *task* invisibili**: indica il numero di *task* che stanno venendo processati dai consumatori.

Non è possibile implementare queste e altre metriche, per due motivi tra di loro correlati.
CloudWatch salva le misurazioni registrate come coppie *key: value*, dove la chiave è rappresentata dal *timestamp* e il valore è quello effettivo della misurazione. Con questo sistema di monitoraggio, se si volesse implementare una metrica che rappresenti il valore attuale di *task* con particolari caratteristiche, si dovrebbe tenere il conto dei *task* che hanno quella particolare caratteristica. In questo modo si potrebbe, periodicamente, registrare una misurazione che rappresenti il conteggio attuale.
Questo però implica che il servizio debba avere una qualche struttura per memorizzare i cambiamenti e periodicamente fare il *push* della misurazione. Si sta quindi parlando della necessità di avere un

servizio *stateful*, che sia in grado di tenere un valore nel tempo.

Date le necessità dettate dalle precedenti motivazioni e l'impossibilità di soddisfarle avendo un servizio *serverless*, non è stato possibile implementare metriche di questo tipo.

Si sarebbero potute realizzare con altri sistemi di monitoraggio, ad esempio *Prometheus*. Esso permette di avere delle metriche che simulano un contatore: sarebbe stato semplice incrementare e decrementare i valori al manifestarsi di alcuni eventi.
Tuttavia non è stato scelto poiché presentava altre limitazioni che lo rendevo meno adatto rispetto a *CloudWatch*.

4.2.5 Limitazioni del servizio

In questa sezione vengono illustrate alcune limitazioni e lacune che il servizio possiede allo stato attuale.

Concurrent dequeues

Ogni coda può possedere più produttori e più consumatori definiti dal servizio Thron che l'ha creata.
Avere più produttori non è un problema, avere più consumatori lo è. In casi sfortunati, è possibile che più consumatori tentino di prelevare lo stesso *task* da eseguire. Quando questo succede potrebbero verificarsi due diversi scenari, in base al *task* da eseguire:

* Eseguire più volte il *task* non comporta un danno alla piattaforma;
* Eseguire più volte il *task* potrebbe portare la piattaforma in uno stato inconsistente.

Limitare questo problema non è stato semplice, poiché, avendo un *datastore* alla base, non è stato possibile implementare correttamente meccanismi che garantissero la semantica *exactly-once-delivery*.
In parte si è rimediato attraverso l'utilizzo di un meccanismo di *optimistic locking*[7]. Nel caso del servizio, vengono effettuati i seguenti passi durante un'operazione di *dequeue*:

1. *Query* sull'indice `QueueShard-VisibleAfter-Idx` per recuperare dei *task* visibili con un determinato *tenant*;
2. Operazione `UpdateItem` sugli *item* precedentemente recuperati con espressione condizionale:
    * Se `VisibleAfter` è uguale a quando è stato recuperato al passo 1, si aggiorna il tempo di invisibilità e si assegna il *task* al consumatore;
    * Se `VisibleAfter` è diverso rispetto a quando è stato recuperato al passo 1, si passa oltre (il *task* è già stato prelevato da qualche altro consumatore).

In questo modo un *task* può venire in un primo momento valutato come valido per il prelievo da più consumatori contemporaneamente. Successivamente solo un consumatore può trovare `VisibleAfter` con lo stesso valore rispetto a quando l'ha letto, sarà lui ad aggiornare l'*item* nel *database* e quindi a prelevare il *task*. A questo punto la condizione non potrà verificarsi per tutti gli altri consumatori, che scarteranno il *task* precedentemente recuperato.

L'approccio utilizzato consente di ridurre drasticamente, ma non eliminare del tutto, la possibilità che lo stesso *task* venga prelevato ed eseguito più volta da consumatori diversi. La nota negativa è che si perde un po' di efficienza nell'esecuzione della *dequeue*, dovendo probabilmente effettuare più *query* per estrarre i *task* da eseguire.

[7] Optimistic locking: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBMapper.OptimisticLocking.html

Undefined behaviour

Alcune funzionalità del servizio necessitano la cooperazione dell'utilizzatore per poter funzionare correttamente.

Un esempio è la gestione della *fairness*: nel caso in cui un consumatore non riportasse, in una richiesta di *dequeue*, l'ultimo *tenant* che gli è stato precedentemente comunicato, il servizio ricomincerebbe ogni volta a scorrere la lista dei *tenant* dall'inizio.
Questo di fatto fa decadere la garanzia della *fairness* del servizio.

Un altro aspetto da considerare è la possibilità che l'*id* di un consumatore non sia univoco. Se in una stessa coda vi sono più consumatori con lo stesso *id*, il servizio non è ovviamente in grado di distinguerli e potrebbe permettere loro di effettuare operazioni che normalmente non dovrebbero essere loro consentite.

Tuttavia alcune funzionalità non possono essere implementate o erogate diversamente per via della natura *serverless* del servizio.
Seppur questi aspetti possano rappresentare un rischio per il servizio e le sue funzionalità, bisogna tenere in considerazione che esso verrà utilizzato solo internamente dagli altri servizi della piattaforma Thron. Perciò ci si può basare sull'assunzione che questi possano essere modificati e correttamente integrati per rispettare le limitazioni del servizio di code e quindi utilizzare le sue funzionalità nel modo corretto.

Bad pattern

Molte delle funzionalità erogate dal servizio riprendono le *API* che venivano esposte dalla libreria esistente. È stata fatta questa scelta per mantenere coerenza con essa e non privare i servizi utilizzatori di funzionalità che magari precedentemente utilizzavano largamente.

Una funzionalità in particolare è stata molto dibattuta, durante gli incontri di allineamento con gli altri membri del *team*.
L'operazione di *update* prevede la possibilità di aggiornare un *task* all'interno della coda, per poterlo eseguire in più *step* successivi.
Questo approccio rappresenta un cattivo *pattern* per la gestione degli *item* all'interno di una coda. Nella teoria si dovrebbero considerare gli *item* di una coda come immutabili e quindi non dovrebbe mai essere possibile modificarli.
Il mancato completamento del *task* potrebbe lasciare la piattaforma in uno stato inconsistente e, aggiungendoci il problema delle *concurrent dequeues*, la situazione potrebbe diventare ancora più complessa.

Nel corso degli incontri il problema è stato sottolineato più volte, senza però prendere una decisione concreta su come eliminarlo, per cui è rimasto sul prodotto finale.
Una soluzione, utilizzabile anche allo stato attuale, potrebbe essere quella di eseguire uno *step*, eliminare il *task* dalla coda e accodare un nuovo *task* aggiornato.

4.2.6 Estensioni del servizio

In questa sezione si presentano alcuni *open point*, non trattati all'interno dello *stage* poiché esterni al progetto, ma comunque utili al fine dell'utilizzo del servizio.

Autenticazione

Non è stato implementato nessun tipo di autenticazione per la fruizione delle funzionalità del servizio.

Al momento è possibile effettuare richieste ai vari *endpoint* da qualsiasi *host*.
Un possibile servizio *AWS* da utilizzare potrebbe essere *Amazon Cognito*[8]. Esso permette di

[8] Amazon Cognito: https://aws.amazon.com/cognito/

integrare una fase di autenticazione quando una richiesta arriva all'*API Gateway*. Essendo un servizio di *AWS*, può essere facilmente integrato nell'infrastruttura del servizio di code.

Sonde

Avendo definito delle metriche che permettono di monitorare lo stato delle code, un punto interessante potrebbe essere la definizione di allarmi a partire da esse.

Gli allarmi consentono di automatizzare le fasi di controllo. In particolare consentono di essere avvisati preventivamente rispetto al verificarsi di un problema, in modo da poter intervenire anticipatamente e risolvere il problema.

Un esempio di allarme definibile a partire dalle metriche implementate, potrebbe essere una sonda che vada a porre un limite inferiore al numero di *task* prelevati dai consumatori. Se in un certo istante si va sotto la soglia, l'allarme scatta. Questo potrebbe essere utile, in quanto potrebbe indicare che i consumatori sono congestionati e che quindi la coda si sta riempiendo ma non riesce ad essere svuotata.

Per definire sonde interessanti e utili sarebbe necessario effettuare uno studio approfondito, anche sulla base di quanto è già stato implementato sulla libreria esistente.

Capitolo 5

Verifica e validazione

In questo capitolo vengono descritti i processi di verifica e validazione del prodotto, descrivendo i tool utilizzati e le metodologie applicate per valutare il corretto funzionamento e la qualità del prodotto.

5.1 Verifica

In questa sezione vengono illustrati gli strumenti e i metodi utilizzati per verificare la correttezza di tutto il materiale prodotto durante il progetto.

5.1.1 Documentazione

Un obiettivo dello *stage* era quello di produrre documentazione sul servizio realizzato, illustrandone in particolare i passi di sviluppo e le scelte architetturali prese.

Per realizzarlo è stata prodotta documentazione di vario tipo: dalla descrizione del codice alla descrizione testuale del servizio.
La documentazione non è da intendersi come un insieme di documenti scritti, ma come una serie di nozioni che sono state raccolte in *file* diversi a seconda del *topic*.
In seguito viene elencata la serie di documenti prodotti durante lo sviluppo del progetto.

README

La descrizione del prodotto, in tutte le sue sfaccettature, è stata riportata in un *file markdown*.
Esso, denominato ***README.md***, è presente nella cartella radice del *repository* e funge da sua presenazione.
Il suo scopo è quello di guidare uno sviluppatore nella navigazione all'interno del *repository*, ma anche capire il contenuto e lo scopo del progetto che sta al suo interno.

I principali punti elencati nel *file* sono:

* Descrizione delle *feature* principali del servizio;
* Spiegazione di com'è stata implementata la *fairness*;
* Descrizione delle funzionalità per gestire code e *task*;
* Struttura del servizio, con descrizione dei singoli servizi *AWS* utilizzati;
* Funzionalità aggiuntive;
* Limitazioni del servizio.
* Struttura del *repository*;
* Istruzioni per compilare, testare ed eseguire il *deploy*.

Specifica *OpenAPI*

Avere una descrizione dettagliata e corretta delle *API REST* esposte è fondamentale per poter permettere ai servizi Thron di potersi integrare con il servizio di code.
A tal fine è stato prodotto un *file yaml*, contenente il dettaglio degli *endpoint* esposti secondo la specifica *OpenAPI v3*.

Codice

Tutto il codice prodotto è stato documentato attraverso dei commenti all'interno dei *file* sorgente. Si è scelto di utilizzare il meccanismo dei commenti, poiché *Go* fornisce la possibilità di creare della documentazione (come se fosse una libreria) a partire dai commenti definiti nel codice.
La loro struttura deve rispettare delle regole e delle convenzioni.

Sono state descritte tutte le strutture definite, illustrandone lo scopo e descrivendo singoli campi al loro interno.

A livello di funzioni si è descritto:

* Scopo della funzione;
* Parametri e loro scopo;
* Eventuali passi delicati o poco chiari all'interno del corpo.

Come detto precedentemente, è possibile, tramite un comando fornito dal linguaggio, creare tutta la documentazione del progetto a partire dai commenti. Questo rende le operazioni di consultazione e comprensione del codice molto più semplici e veloci da portare a termine.

Pagine *confluence*

Molti appunti presi e curati durante il percorso di *stage* sono stati riportati e gestiti sul *wiki* aziendale, mantenuto su *Confluence*[1].

In queste pagine web si è riportato:

* *Sketch* dei requisiti e delle metriche del servizio;
* Decisioni fondamentali prese per l'implementazione dei requisiti funzionali fondamentali;
* *Output* dei vari incontri con altri membri del *team*;
* Limitazioni e *open point* del servizio.

5.1.2 *Testing* del servizio

In questa sezione vengono descritte le attività di test e gli strumenti utilizzati al fine di controllare la correttezza e il buon funzionamento del servizio realizzato.

Test sul codice

Per verificare la correttezza del codice sono stati definiti dei test di unità e di sistema.

I primi hanno riguardato per lo più le singole funzioni, andando a testare il corretto funzionamento della logica implementata nel caso medio, nei casi limite e nei casi oltre il limite.
Per lo sviluppo di questi test, si è deciso di utilizzare il *pattern AAA*.
Come *tool* di sviluppo per i test, è stato utilizzato un *framework* interno di *Go*, che permette di definire una serie di situazioni da testare con relativo *output* atteso, per poi eseguirle in serie.

[1] Confluence: https://www.atlassian.com/software/confluence

Si è utilizzata anche una libreria esterna, *testify*[2], che fornisce le comuni asserzioni, presenti in altri linguaggi (come nel *framework JUnit* di *Java*).

I test di sistema sono stati realizzati per verificare il corretto funzionamento, ad alto livello, delle funzionalità esposte dal servizio.
Essi sono stati scritti in *Go*, seguendo il *pattern AAA*.
Ogni test effettua i seguenti passi:

* Prepara una richiesta *HTTP* verso un determinato *endpoint*, con parametri e *body* variabili in base al test che si sta effettuando *(Arrange)*;
* Invia la richiesta al servizio *(Act)*;
* Fa delle asserzioni sulla risposta, verificandone lo *status code* e il contenuto del *body (Assert)*.

In una prima fase di *testing*, quando ancora il servizio eseguiva solamente in locale, si è utilizzato *Postman*[3]. Esso permette di effettuare, in modo semplice e veloce, richieste *HTTP custom*.
Si possono salvare anche dei *set* di richieste predefinite. Questa funzionalità è stata utilizzata per esportare un insieme di richieste predefinite, utili per un utente che vuole capire il funzionamento del servizio in modo rapido e comodo.

È possibile, tramite un comando del linguaggio, lanciare tutti i test definiti e verificare l'esito positivo per ciascuno di essi.

Simulazione casi d'uso

Per testare il corretto funzionamento del servizio in contesti d'uso più attinenti alla realtà, si è deciso di simulare dei casi d'uso comuni che il servizio potrebbe dover affrontare.

Per poter simulare delle situazioni reali è stato necessario definire delle strutture che andassero ad emulare il comportamento di un *producer* e un *consumer* di un generico servizio Thron.
A tal fine sono state definite due principali strutture:

* **Produttore**: permette l'accodamento di nuovi *task* all'interno di una coda;
* **Consumatore**: espone funzionalità per recuperare *task* da una coda e gestirli attraverso operazioni di *update*, *safe remove* e *increment attempts*.

Una volta definite le strutture e le loro funzioni, che integrano le funzionalità del servizio di code, si è passati alla definizione e *testing* dei seguenti casi d'uso:

* ***Fairness***: il test propone una situazione di sbilanciamento del carico all'interno di una coda, con molti *task* associati ad un *tenant* mentre pochi *task* per tutti gli altri *tenant*. Lo scopo è quello di verificare che sia garantita la *fairness* indipendentemente dal livello di carico;
* ***Long task handling***: il test simula il caso in cui un *task* debba essere elaborato in più *step*. Ogni volta che il *task* viene prelevato dalla coda, viene aggiornato fittiziamente e si controlla che esso torni visibile sulla coda dopo il tempo prefissato (e che quindi non venga eliminato);
* **Code prioritarie**: dato che il requisito funzionale per la gestione di *task* con priorità diversa è stato accantonato, si è deciso di definire un test per verificare che la soluzione proposta nella sezione 4.2.4 funzioni correttamente. Il test va a definire più code, una per ogni livello di priorità, e controlla che sia possibile implementare una logica prioritaria a partire da queste code;
* ***Dead letter queue***: il test mira a verificare che, con un certo livello di carico, se un *task* non viene eseguito con successo un certo numero di volte, esso venga rimosso dalla coda principale e spostato nella *dead letter queue*. L'asserzione verifica che dopo un certo numero di volte che si è prelevato il *task* sfortunato, esso non venga mai più prelevato;

[2]Testify: https://github.com/stretchr/testify
[3]Postman: https://www.postman.com/

- **Produttori e consumatori**: il test simula una situazione in cui più produttori e più consumatori interagiscono in modo concorrente con una stessa coda. In particolare, il test va a verificare che un *task* non venga mai prelevato più volte da consumatori diversi, verificando che la soluzione descritta nella sezione 4.2.5 sia effettivamente corretta.

Per i test sopracitati non è sempre stato possibile trovare delle asserzioni valide per determinare se il test fosse stato superato con successo o meno. Infatti in alcuni casi la verifica dell'esito positivo del test poteva essere fatta solo manualmente.

In generale tutti i test dei casi d'uso effettuati hanno avuto esito positivo, rafforzando ulteriormente la base solida derivante dai test di unità e di sistema.

5.2 Validazione

In questa sezione viene descritto il processo di validazione del materiale prodotto durante il progetto.

5.2.1 Documentazione

La validazione della documentazione prodotta durante lo stage è stata effettuata dal *tutor* aziendale. Assumendo il ruolo di committente, egli stabiliva se la documentazione fosse conforme e corretta rispetto alle attese. Dopo aver richiesto eventuali correzioni, essa veniva pubblicata negli ambienti consoni.

5.2.2 Codice

La validazione del codice è stata effettuata maggiormente dal *tutor* aziendale, il quale periodicamente andava ad analizzare il contenuto del *repository*. Ogni fase di verifica produceva delle critiche costruttive e dei consigli da applicare nella fase successiva. Questo tipo di validazione mirava a verificare che il codice avesse certe priorità, come leggibilità, testabilità, manutenibilità e correttezza.

Un'ulteriore fase di validazione è stata fatta in parte da un altro membro del *team*, più esperto sul mondo *Go*. Grazie a questa è stato possibile apprendere ed applicare alcune *best practice* del linguaggio.

5.2.3 Presentazione finale

L'ultimo giorno di *stage* si è tenuta una presentazione, dove ho avuto modo di illustrare il progetto sviluppato durante i due mesi di lavoro.

L'esito della presentazione è stato, a mio parere, positivo. Non sono state evidenziate nuove criticità e i partecipanti concordavano, in linea generale, con la logica e le scelte effettuate in fase di progettazione.

La presentazione, in un certo senso, ha contribuito ad una validazione ad alto livello del progetto.

Capitolo 6

Conclusioni

In questo ultimo capitolo viene analizzato retrospettivamente il progetto di stage, focalizzandosi sul raggiungimento degli obiettivi, sulle conoscenze acquisite e/o rinforzate, e sulla valutazione personale di questo percorso.

6.1 Prodotto realizzato

Come descritto nei precedenti capitoli, il servizio realizzato consente ai servizi Thron di creare e gestire *task* all'interno delle code. Ogni servizio ha la possibilità di creare più code da poter utilizzare per accodare e recuperare *task*. Per ogni *task* vengono messe a disposizione una serie di funzionalità ausiliarie per consentire ai consumatori di poter gestire al meglio un *task* in esecuzione. Il servizio espone anche una serie di metriche per misurare lo stato delle code.

Tra le caratteristiche fondamentali del servizio vi è la garanzia della *fairness*, che permette una gestione equa dell'erograzione di funzionalità della piattaforma Thron ai clienti utilizzatori.
Il servizio inoltre implementa alcuni concetti dei classici sistemi di code, come *dead letter queue* e *exactly-once-delivery*.

6.2 Raggiungimento degli obiettivi

Come descritto nella sezione 4.2.2, il prodotto soddisfa quasi tutti i requisiti individuati nell'attività di analisi dei requisiti.
In particolare, sono stati soddisfatti tutti i requisiti funzionali realizzabili (esclusi quelli indicati nella sezione 4.2.4), tutti i requisiti qualitativi e tutti i requisiti di vincolo.
A questi sono state aggiunte delle funzionalità aggiuntive, per rendere il prodotto più completo e migliore. L'elenco di tali *feature* è descritto nella sezione 4.2.6.

Dal punto di vista degli obiettivi dello *stage*, si possono fare le seguenti considerazioni:

* **Analisi e inquadramento del problema**: grazie allo studio di fattibilità svolto e allo studio della libreria esistente, è stato possibile capire su quali tematiche porre maggiormente il *focus*. Questo ha permesso di trovare soluzioni alternative a problemi non facilmente risolvibili;

* **Realizzazione del prodotto**: tutte le attività previste per lo sviluppo del prodotto, dall'analisi dei requisiti alla validazione finale, sono state portate a termine con successo senza particolari intoppi. Alla luce di questo è possibile affermare che il prodotto finale ha le caratteristiche attese e rispecchia le aspettative;

* **Redazione della documentazione**: come descritto nella sezione 5.1.1, è stata prodotta la documentazione richiesta sulle scelte architetturali, sulle funzionalità esposte e sulle decisioni fondamentali prese.

* ***Report* finale dei risultati**: nella presentazione finale, registrata dall'azienda, sono stati esposti i risultati e le considerazioni finali sul prodotto. Grazie ad alcune domande da parte dei colleghi che hanno patecipato, si è fatto anche il punto della situazione in merito allo stato di avanzamento del servizio nell'ottica di un suo uso in un ambiente di produzione.

L'autovalutazione appena descritta è stata confermata, nel corso delle settimane, dagli incontri avuti con il *tutor* aziendale e con gli altri membri del *team*.

6.3 Conoscenze acquisite

Per la realizzazione di questo progetto sono state fondamentali molto le nozioni apprese durante il corso di studi.

Oltre ad aver avuto l'opportunità di sfruttare ed approfondire le conoscenze sopracitate, lo *stage* ha permesso di incrementare notevolmente il mio bagaglio di conoscenze.
Vengono ora elencate le principali conoscenze acquisite:

* *Database non relazionali*: nozioni fondamentali per la realizzazione delle tabelle su *DynamoDB* e il loro successivo corretto utilizzo. L'apprendimento di questo modello di *database* ha permesso di sfruttare al meglio le funzionalità del sistema di persistenza;
* *Servizi AWS*: conoscenza di alcuni servizi del mondo *AWS*, essenziali per la realizzazione di un'infrastruttura in linea con lo stile degli altri servizi della piattaforma Thron.
* *Architettura a microservizi*: concetti e *best practice* basilari di un'architettura a microservizi. L'apprendimento di queste nozioni è stato fondamentale per realizzare un servizio che possa essere integrato con l'architettura della piattaforma Thron;
* *Tecnologie per lo sviluppo*: l'utilizzo delle tecnologie descritte nella sezione 4.1.10, ha permesso di ampliare notevolmente il mio bagaglio tecnologico.

Posso inoltre affermare di aver ricevuto e fatto tesoro di molte *best practice* su concetti di programmazione e sull'approccio per la risoluzione di un problema.

In linea più generale, questo progetto ha permesso di migliorare le mie *soft skill*, attraverso l'interazione con altri membri del *team*, ma anche grazie alla definizione di *deadline* da rispettare.

Lo *stage* presso THRON si è svolto, a mio parere, in modo ottimo.
I due mesi trascorsi in azienda sono risultati molto leggeri e piacevoli, questo soprattutto grazie ad un ambiente di lavoro giovanile e molto motivante.
In generale mi sono sempre sentito, nel mio piccolo, parte integrante del *team*. Ho partecipato con molto piacere agli eventi e cerimonie organizzate dal *team*.
Credo fortemente che il *team* in cui sono stato inserito sia stato uno dei punti chiave per la buona riuscita dello *stage*.

Grazie ad un ottimo rapporto con il *tutor* aziendale, non ho mai dovuto affrontare difficoltà bloccanti in autonomia. Emanuele era sempre pronto e disponibile ad ascoltare dubbi e perplessità per fornirmi una risposta che mi permettesse di continuare correttamente il lavoro.
Un altro aspetto che ho molto apprezzato è stato l'interesse dei membri del *team* verso il progetto che stavo realizzando. Ho avuto più volte, attraverso vari incontri, la possibilità di confrontarmi con colleghi molto più esperti di me per ricevere consigli e critiche su quanto prodotto.
È capitato anche che qualche collega proponesse qualcosa di sua spontanea volontà. Questo ha permesso di migliorare, giorno dopo giorno, la qualità del prodotto.

Per quanto concerne la parte di realizzazione del progetto, tutto è proseguito secondo quanto programmato. Non vi sono mai state difficoltà tali da causare un cambiamento sulla pianificazione

iniziale.
Personalmente ho trovato più difficile la prima parte di inquadramento del problema. Non è stato semplice l'approccio con la libreria esistente, scritta in un linguaggio a me sconosciuto e basata sulla programmazione funzionale.
Tolto questo primo periodo, tutti gli altri sono stati svolti con un buon grado di autonomia.
Ho avuto modo di affrontare tematiche diverse e di risolvere problemi di diversa natura. Questo mi ha permesso di spaziare su *topic* diversi e di rimanere costantemente motivato e propenso ad apprendere nuovi concetti.

Concludendo, questo progetto di *stage* è stato per me molto soddisfacente per come si è svolto, per il prodotto finale realizzato, per le conoscenze che ho potuto apprendere e per le persone che ho potuto conoscere.

Glossario

Amazon Web Services piattaforma di proprietà del gruppo *Amazon* che offre servizi di *cloud computing*, elaborazione e distribuzione di contenuti. I loro servizi vengono utilizzati per creare applicazioni flessibili, scalabili e affidabili. 4, 8, 10, 29, 31, 33–35, 37, 47–50, 54, 55, 57, 62

API (*Application Programming Interface*) è un modo per due o più macchine di comunicare. Tipicamente espongono un servizio (delle funzionalità) ad altre componenti *software*, al fine di rendere più semplice il lavoro di queste. 2–4, 12, 28, 31, 32, 35, 38, 47, 54, 58, 66

Assert Act Arrange è un *pattern* per scrivere test di unità che abbiano una struttura uniforme. Questo li rende anche facilmente leggibili e comprensibili. 58, 59

At-Least-Once-Delivery è una modalità di prelievo di *task* da una coda. Questo meccanismo garantisce che, per ogni *task*, possano essere fatti più tentativi di prelievo per quel *task* e quindi esso possa essere processato più volte, ma mai perso. 8, 11

Base64 è un gruppo di schemi di codifica da binario a testo che rappresentano dati binari in sequenze di 24 bit. 35, 37, 51

Best effort è una modalità di erogazione di *task* ad un consumatore di una coda. Significa che l'ordinamento con cui i *task* arrivano in coda, potrebbe non essere lo stesso con cui essa vengono prelevati. In una logica *First-In-First-Out* questo significa che l'ordine in senso stretto potrebbe non essere sempre rispettato. 8

Cloud-native è un approccio alla scrittura di *software* che siano scalabili e possano eseguire in ambienti *cloud*. 2, 3

Container è un'unità *software* che racchiude il codice e tutte le sue dipendenze, affinché l'applicazione possa essere eseguita nella stessa maniera in ambienti computazionali diversi. Il concetto è uno dei principi di base su cui si basa *Docker*. 46

CRUD (*Create, Read, Update and Delete*) rappresentazione le quattro operazioni basilari di uno sistema di persistenza. 33

DAM (*Data Asset Management*) è una categoria per prodotti software definita da Forrester. Il DAM è uno strumento che permette alle società che investono e gestiscono contenuti digitali, di ottenere efficienza nei processi e accorciare il time-to-market. 1, 2, 12, 14

Dead Letter Queue particolare tipo di coda che raccoglie tutti i *task* che, per vari motivi, non riescono ad essere processati dai consumatori. I motivi più frequenti sono per via di errori lato *client* o *server*. Gli elementi di questa coda dovrebbero poi essere presi in considerazione per eventuali analisi o nuovi tentativi. 8, 9, 12, 27, 36, 44, 48, 49, 59, 61

Deploy rappresenta uno stato del ciclo di vita del *software*, che consiste nel rilascio di un sistema software o di un'applicazione, con la relativa installazione e la sua messa in esercizio. 31

Endpoint è un luogo digitale esposto tramite un'*API*, dal quale l'*API* riceve le richieste dei *client* e invia le risposte. Ogni *endpoint* è un *URL* che fornisce la posizione di una risorsa sul server dove le *API* sono esposte. 29, 31–33, 38, 47, 49, 50, 54, 58, 59

Exactly-Once-Delivery è una modalità di prelievo di *task* da una coda. Questo meccanismo garantisce, per ogni *task*, che esso venga prelevato esattamente una volta, evitando che possa essere processato più volte. 53, 61

Go linguaggio di programmazione *general-purpose*, sviluppato da Google, compilato e a tipizzazione statica. Assomiglia a C, ma ha il *garbage collector*, non supporta le classi e implementa la concorrenza in stile CSP (*Communicating Sequential Processes*). 28, 34, 47, 58–60

HTTP (*HyperText Transfer Protocol*) è un protocollo di rete, che permette lo scambio di informazioni e risorse sul web tra due macchine collegate in una rete. 31, 59

Java linguaggio di programmazione *general-purpose* ad livello, basato su classi e orientato alla programmazione ad oggetti. Designato per avere il minor numero di dipendenze di implementazione possibile. 28, 59

JSON (*JavaScript Object Notation*) è un formato di serializzazione basato su testo per lo scambio di dati, principalmente tra un *server* e una applicazione *web*. 8, 35, 37, 47, 51

Marshalling è un processo che trasforma un oggetto di un linguaggio di programmazione in un formato adatto all'archiviazione o alla trasmissione. l processo viene comunemente chiamato serializzazione. 34, 35, 47

NoSQL è un tipo di *database* realizzati per modelli di dati specifici con schemi flessibili. Sono molto utilizzati per la facilità di sviluppo, le funzionalità che offrono e la scalabilità delle prestazioni. 34

On demand modalità di accesso a delle risorse *cloud* solo quando necessario, pagando in base all'utilizzo e non in base a un canone fisso. Tipica modalità utilizzata quando si deve far fronte a variazioni di carico. 8, 34

OpenAPI è una specifica per la definizione di servizi *RESTful*, leggibile dalle macchine per descrivere, produrre, consumare e visualizzare servizi web RESTful. 32, 33, 58

PIM (*Product Information Management*) è il processo di gestione delle informazioni richieste per la commercializzazione e la vendita dei prodotti. Funga da supporto alle strategie di *marketing* multicanale. 1

REST (*REpresentational State Transfer*) è un'architettura *software* che impone condizioni sul funzionamento di una *API*. Si può utilizzare per supportare una comunicazione su vasta scala ad elevate prestazioni e affidabile. Si può facilmente implementare e modificare, apportando visibilità e portabilità multi piattaforma a qualsiasi sistema. 12, 28, 31, 32, 38, 47, 58

RFC3339 formato per rappresentare un istante nel tempo. Il formato è il seguente: `2019-10-12T07:20:50.52Z`. 35–37, 51

Round Robin è un algoritmo di pianificazione dei lavori considerato molto equo, in quanto consente a ciascuna componente di un sistema di eseguire i propri lavori per un determinato periodo di tempo. 46

RPC (*Remote Procedure Call*) è un modello programmazione distribuita che permette di eseguire una procedura di una macchina in uno spazio di indirizzi differente. La chiamata risulta essere una normale chiamata a procedura, senza che il programmatore sappia i dettagli di quello che viene durante l'interazione remota. 28

SaaS (*Software as a Service*) è un modello di distribuzione del software applicativo dove un produttore di software sviluppa, opera e gestisce un'applicazione che mette a disposizione dei propri clienti via Internet previo abbonamento. 1

Scala è un linguaggio di programmazione *general-purpose*, staticamente tipizzato che dia supporta il paradigmi ad oggetti che quello funzionale. È stato designato per essere conciso e molte delle sue decisioni di progettazione mirano ad affrontare le critiche a Java. 2

SCRUM è un *framework agile*, incrementale e iterativo, per lo sviluppo di prodotti, applicazioni e servizi. Rappresenta un vero e proprio *framework*, ovvero una modalità strutturata e pianificata di gestione e organizzazione del lavoro, su cui è possibile costruire soluzioni complesse. 1

SDK (*Software Development*) è un insieme di strumenti che consente lo sviluppo di *software* o *firmware* per una specifica piattaforma o uno specifico servizio. Può essere composto da *API*, compilatori, *debugger*, *framework*, documentazione e molto altro. È un modo per facilitare l'adozione della specifica piattaforma o dello specifico servizio da parte degli sviluppatori. 31, 34, 35

Serverless è un modello di esecuzione dove un *cloud provider* alloca, su richiesta, delle macchine per eseguire del codice. Questo tipo di architetture sono dette *stateless*, ovvero senza la capacità di mantenere uno stato interno nel corso delle varie esecuzioni. 2, 3, 8, 28, 29, 46, 47, 53, 54

Tenant è un gruppo di utenti che condivide l'accesso con particolari privilegi ad un'istanza *software*. Nel caso di Thron, un *tenant* rappresenta un'azienda cliente che usufrisce della piattaforma Thron, la quale condivide l'accesso tra i membri al suo interno . 2, 3, 11, 13, 35–40, 45, 46, 50–54, 59

UML (*Unified Modeling Language*) è un linguaggio di modellazione e specifica basato sul paradigma object-oriented. Dvolge un'importantissima funzione di 'lingua franca' nella comunità della progettazione e programmazione a oggetti. Gran parte della letteratura di settore usa tale linguaggio per descrivere soluzioni analitiche e progettuali in modo sintetico e comprensibile a un vasto pubblico. 13–15, 17–23

Unmarshalling è un processo che converte una rappresentazione a basso livello di un'informazione in una struttura (oggetto) ad alto livello. Il processo viene comunemente chiamato deserializzazione. 33, 47

UUID (*Universally Unique IDentifier*) è un identificativo usato nelle infrastrutture *software*, composto da 16 *bytes*. 35–37, 51, 52

Bibliografia

Siti web consultati

Amazon API Gateway. URL: https://aws.amazon.com/api-gateway/.

Amazon CloudFormation. URL: https://aws.amazon.com/cloudformation/.

Amazon CloudWatch. URL: https://aws.amazon.com/cloudwatch/.

Amazon Cognito. URL: https://aws.amazon.com/cognito/.

Amazon DynamoDB. URL: https://aws.amazon.com/dynamodb/.

Amazon EventBridge. URL: https://aws.amazon.com/eventbridge/.

Amazon Lambda. URL: https://aws.amazon.com/lambda/.

Amazon Serverless Application Model. URL: https://aws.amazon.com/serverless/sam/.

Amazon Simple Notification Service. URL: https://aws.amazon.com/sns/.

Amazon Simple Queue Service. URL: https://aws.amazon.com/sqs/.

AWS SDK v2 per Go. URL: https://aws.github.io/aws-sdk-go-v2/docs/.

Comandi di base per Go. URL: https://pkg.go.dev/cmd/go.

Configurarare REST API usando OpenAPI e Amazon API Gateway. URL: https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-import-api.html.

Confluence. URL: https://www.atlassian.com/software/confluence.

Confronto tra servizi out-of-the-box per la gestione di code. URL: https://www.beabetterdev.com/2021/09/10/aws-sqs-vs-sns-vs-eventbridge/.

Documentazione Go. URL: https://go.dev/doc/.

Lambda Proxy Integration. URL: https://docs.aws.amazon.com/apigateway/latest/developerguide/set-up-lambda-proxy-integrations.html.

Optimistic locking per Amazon DynamoDB. URL: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBMapper.OptimisticLocking.html.

Postman. URL: https://www.postman.com/.

Prometheus. URL: https://prometheus.io/.

Specifica OpenAPI v3. URL: https://swagger.io/specification/.

Sviluppo di una funzione Lambda con Go. URL: https://dev.to/preethamsathyamurthy/serverless-golang-rest-api-with-aws-lambda-4cn6#cli-step-2.

Testare Lambda in locale con Amazon API Gateway. URL: https://andmoredev.medium.com/how-to-run-api-gateway-aws-lambda-and-dynamodb-locally-91b75d9a54fe.