SiPY
The quote you were looking for.

Diletta Lagomarsini
dilettalagom@gmail.com

Federica Montesano
federica.montesano42@gmail.com

***Abstract*—Il serverless computing ha ottenuto un ruolo di spicco nel paradigma di progettazione di applicazioni, sia perché si riduce il server management, essendo il server stateless, sia perché è un modello pay-as-you-go. In questo articolo discutiamo come questo paradigma sia stato applicato ad una chatbot, descrivendone architettura, implementazione e prestazioni.**

INTRODUZIONE

Parallelamente allo stabilirsi del serverless computing come nuovo paradigma di sviluppo nel cloud, le chatbots sono divenute sempre più rilevanti nell'ambito di information delivery. Implementare un bot in maniera serverless consente di semplificare l'intero processo di sviluppo e di deploy, oltre ad avere notevoli vantaggi in termini di time-to-delivery e costi. L'obiettivo del progetto è quello di sviluppare un'applicazione serverless in grado di interagire con gli utenti attraverso un sistema di messaggistica istantanea.

SiPY è un'app integrata in uno Slack bot e permette di cercare una citazione in funzione delle specifiche richieste dall'utente e restituirla via chat. La comunicazione è favorita dall'utilizzo del client di Slack, una chat (sia web che desktop) intuitiva, che permette all'utente di interagire agevolmente con l'applicazione bot. Il lato server, invece, è ospitato interamente dalla piattaforma Amazon Web Service, configurata utilizzando le risorse custumizzabili messe a disposizione.

***Keywords*—** *serverless, bot, lambda*

ARCHITETTURA

L'applicazione SiPY è composta da un lato client, un lato server e un database cloud distribuito. I linguaggi utilizzati per implementare l'applicazione sono Python e Nodejs: Python è stato il linguaggio principale di sviluppo, mentre Nodejs è stato usato per mantenere l'integrazione con il servizio client utilizzato. Per permettere la comunicazione, infatti, è stata sfruttata la chat del client di Slack, che comunica con il backend solo tramite Nodejs.
In Slack è necessario creare un workspace apposito su cui caricare le applicazioni (in questo caso il bot) e connetterle attraverso canali URL webhook per permettere la comunicazione con altri componenti. Per capire pienamente questo aspetto vedere **Limitazioni**.

Il backend è costituito da una parte totalmente residente sui server Amazon, in particolare sulla regione us-east-1 (Virginia), mentre per la parte di persistenza è stato usato MongoDB integrato su Amazon attraverso il servizio fully managed MongoAtlas. I servizi di AWS utilizzati sono:

- *Amazon Lex*, per creare l'interfaccia di comunicazione con il client, che offre anche un meccanismo di "intelligenza" rispetto al testo inserito dalll'utente [2.1]. Funge inoltre da trigger per una delle funzioni Lambda.
- *AWS Lambda*, è il cuore dell'applicazione serverless: riceve un evento in input da Lex ed effettua la computazione facendo richieste al data storage [2.2].
- *SNS Topic*, utilizzato come trigger per una delle funzioni Lambda [2.3].
- *API Gateway*, per creare un punto di accesso diretto verso il backend per una delle funzioni Lambda [2.4].
- *S3*, utilizzato in fase di deploy per mantenere dati necessari per la creazione del full stack su CloudFormation.

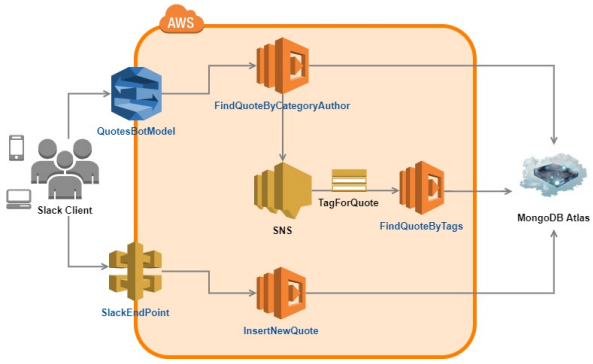

Figure 1. Rappresentazione dell'architettura dell'applicazione e dei servizi AWS utilizzati.

Amazon Lex

Lex fornisce un'interfaccia di interazione tra il client e il backend dell'applicazione: il suo compito è quello di analizzare le frasi di input inviate dall'utente e simulare una conversazione intelligente e di senso compiuto. Il servizio rappresenta un'astrazione del parsing manuale degli input, con controllo automatico degli errori sulle parole chiave.

Configurando in maniera opportuna il Model del bot, Lex permette sia di identificare le richieste dell'utente, sia di rispondere in maniera corretta e coerente alle richieste stesse assumendo comportamenti preconfigurati.

I comportamenti, chiamati "Intent", programmati nel modello sono:

- QuotesIntent, che is occupa delle richieste delle citazioni da parte dell'utente analizzandone i valori: in particolare

ottiene un valore per la categoria e uno per l'autore della citazione.
- HelpIntent, che riconosce le eventuali richieste di aiuto digitate dall'utente per richiedere il formato adatto per interagire con il bot stesso.

La categoria e l'autore sono identificati dagli "Slot" di Lex collegati al "QuotesIntent", che hanno sia un ordinamento di priorità definito dallo sviluppatore sia un insieme di valori preconfigurati, rappresentati i possibili sinonimi per quello Slot. In generale, sono i valori utilizzati per l'addestramento del bot e corrispondono ai valori "Category" e "Author" salvati nel data store.
Ogni Intent è direttamente collegato alla Lambda attraverso il "Fulfillment" che si occuperà di computare la richiesta. I dati, prima parsati e incapsulati in un messaggio json da Lex, vengono inviati alla Lambda, triggerandola. Da quel momento il flusso dell'esecuzione è delegato interamente alla funzione Lambda, dalla sua istanziazione, previa ricezione dei valori di input all'interno dell'evento "event" che l'ha triggerata, all'esecuzione vera e propria della query su MongoDB, compreso il ritorno dei risultati.

AWS Lambda

Il servizio Lambda costituisce il cuore dell'applicazione serverless. Le funzioni lambda implementate sono quattro, tre delle quali scritte in Python 3.x e una scritta in NodeJs, per i motivi di integrazione con Slack già accennati e in seguito spiegati in dettaglio [6].

Nella prima versione del progetto era stata implementata una funzione lambda che interagiva con un data store diverso, in particolare DynamoDB, successivamente abbandonato per motivi descritti in **Implementazione**.
Le funzioni sono, dunque:

- *FindQuoteByCategoryAuthor*: alla richiesta di una citazione per categoria e/o autore da parte dell'utente compie una ricerca discriminando le possibili combinazioni degli input. Se la citazione esiste, la ritorna all'utente tramite canale verso Lex, altrimenti passa il controllo della ricerca alla funzione lambda seguente pubblicando la nuova richiesta su un topic di SNS Topic, che triggera la suddetta funzione.
- *FindQuoteByTags*: viene triggerata dalla pubblicazione su topic da parte di FindQuoteByCategoryAuthor. Effettua una ricerca con gli stessi input ricevuti dalla lambda precedente, ma estendendola su altri campi del dataset, in particolare ricerca la categoria richiesta tra i tag della citazione.
- *InsertNewQuote*: è attivata attraverso un comando direttamente da Slack per permettere all'utente di inserire una nuova citazione. Effettua poi una ricerca su MongoDB e se la citazione è già presente non effettua l'inserimento, altrimenti la inserisce. Prima di terminare, ritorna un messaggio di successo o insuccesso all'utente attraverso il canale webhook collegato a Slack.
- *lambdaDynamo*: questa funzione è sostanzialmente equivalente a FindQuoteByCategoryAuthor, le differenze verranno spiegate in **Implementazione**.

Amazon SNS

Amazon SNS è stato utilizzato come trigger per la funzione FindQuoteByTags, che è un subscriber del topic TagForQuote. La funzione FindQuoteByCategoryAuthor, se non trova direttamente una citazione, pubblica sul topic un evento che ha come attributi la categoria e l'autore della citazione ricevuti in input da Lex.

Il vantaggio di un sistema come SNS in un'applicazione serverless è l'asincronia e quindi il non avere vincoli tra funzioni lambda, che per il paradigma serverless non devono poter mantenere uno stato. Infatti le due funzioni riescono a coesistere senza dover interagire tra loro né in maniera sincrona, né dovendo passare per sistemi di caching.
Inoltre, inserendo un meccanismo di "publishing-subscribing" combinato con l'apertura del canale diretto verso il client, è stato possibile evitare sia l'invocazione sincrona tra le lambda che il pagamento del tempo di attesa della prima lambda per l'esecuzione e il ritorno della seconda.

API Gateway

Per permettere l'interazione con il comando di Slack per l'inserimento di una nuova citazione, sono state utilizzate le API Gateway come seconda "porta di ingresso" verso il backend del bot e sono direttamente collegate alla lambda InsertNewQuote. Le API eseguono il marshaling della richiesta dell'utente dal client Slack verso la lambda, che viene triggerata proprio dall'inoltro del messaggio HTTP/POST che le attraversa.

La scelta delle API Gateway è strettamente legata al client di Slack, che impediva la creazione di un comando che fosse compatibile con i valori di input attesi dagli "Intent" di Lex. Infatti per questo tipo di comunicazione, Slack prevede l'utilizzo di un canale URL su cui scambiare pacchetti HTTP, mentre Lex si aspetta di ricevere in ingresso dei json il cui content deve essere formato da stringhe con un formato ben strutturato.

Persistenza

Il data store utilizzato è MongoDB, attraverso il cloud storage MongoDB Atlas per collegare MongoDB ad Amazon. La scelta di MongoDB è stata compiuta dopo aver utilizzato DynamoDB e aver riscontrato limitazioni che verranno spiegate successivamente [6], ma anche per un mapping immediato tra il dataset e la natura json-like di MongoDB.
L'ulteriore vantaggio di MongoDB su DynamoDB per questa applicazione è la possibilità di MongoDB di fare query in maniera svincolata da una chiave primaria, anche in profondità attraverso il documento.

Per capire meglio i motivi che hanno portato a cambiare data store, serve una rapida analisi del dataset ottenuto da [4].
Il file json si presenta come un array di documenti, in cui ogni documento ha la forma seguente:

```
1 {
2     "Quote": "Don't cry because it's over,
          smile because it happened.",
3     "Author": "Dr. Seuss",
4     "Tags": [
```

```
        "cry",
        "crying",
        "experience",
        "happiness",
        "joy",
        "life",
        "optimism",
        "sadness",
        "smiling "
    ],
    "Popularity": 0.15566615566615566,
    "Category": "life"
}
```

I campi “Quote”, “Author” e “Category” sono banalmente la citazione, l’autore e la relativa categoria a cui fa riferimento. “Category” però può anche essere un campo vuoto. La Quote viene restituita all’utente, mentre Category e Author sono utilizzati da FindQuoteByCategoryAuthor e FindQuoteByTags per cercare all’interno di MongoDB. Il campo “Tags” è quello che viene usato per computare FindQuoteBytags, ovvero nel caso in cui la ricerca effettuata dall’utente non corrisponda ad una categoria in “Category”. Infine il campo “Popularity” rappresenta un indice di popolarità della citazione, non utilizzato in questa release [7].

IMPLEMENTAZIONE

Di seguito verrà descritta l’implementazione nei suoi punti essenziali, dalle funzioni lambda utilizzate alle differenze di implementazione tra FindQuoteByCategoryAuthor e lambdaDynamo, che come accennato precedentemente differiscono per la modalità di effettuare le query, rispettivamente su MongoDB e DynamoDB. Tutte le funzioni implementate sono state scritte in linguaggio Python 3.x, tranne InsertNewQuote che è stata scritta in Nodejs poiché interagisce direttamente con Slack, come era stato accennato precedentemente [2.2]. Successivamente [Fig. 2] e [Fig. 3] si può trovare una rappresentazione delle interazioni tra i componenti del sistema.

FindQuoteByCategoryAuthor

L’input della funzione è un event generato da Lex all’inserimento della richiesta da parte dell’utente. La funzione ha anche due variabili d’ambiente: una riferita al cluster di MongoDB e una riferita al topic di SNS.

Una volta estratti i campi “Author” e “Category” viene chiamate la funzione *getQuote* in cui si discriminano i vari casi di input possibili con un semplice costrutto if-else. In particolare gli input sono valori di “Author” e “Category” del tipo “Jane Austen” e “love”, per cui la ricerca produrrà come risultato una citazione di Jane Austen sul tema “love”, ma è anche possibile che l’utente non abbia una preferenza per l’autore o la categoria, e che quindi scriva “any” per uso o entrambi i campi.
Nel primo caso la ricerca sarà effettuata solo per il campo con valore diverso da “any”, mentre nel secondo caso verrà semplicemente scelta una citazione a caso senza vincoli di autore e categoria.
Ad esempio, se l’utente digita “I want a quote about life by any”, quello che otterrebbe sarebbe una citazione sulla categoria “life” per un autore qualsiasi, e idem nel caso in cui la categoria sia “any”.

```
if author == "any" and category != "any":
    items = list(collection.find({"Category": category}))
elif author != "any" and category == "any":
    items = list(collection.find({"Author": {'$regex':
         author, '$options': 'i'}}))
elif author == "any" and category == "any":
    items = list(collection.find({'$or': [{"Category":
         "quotes"}, {"Category": ""}]}))
else:
    items = list(collection.find({"Category": category,
         "Author": {'$regex': author, '$options': 'i'}}))
```

Le opzioni “*regex*” rendono possibile cercare la citazione anche se non si conosce il nome dell’autore esatto, ma si conosce solo, ad esempio, il cognome. Dopo la query al data store, si effettuano vari controlli sulla riuscita o meno della query, e in caso di successo si prende casualmente una delle citazoni contenute nel vettore di citazioni ottenuto. Infine si risponde all’utente attraverso lo stesso canale di invio di Lex. In caso di insuccesso si passa il controllo alla funzione *calltoTagslambda*, dopo aver creato un messaggio da pubblicare sul topic TagForQuote di SNS, dopodiché la prima lambda termina con un messaggio di accettazione della richiesta.

```
msg = {"Category": category, "Author": author}

response = sns.publish(
              TopicArn=os.environ["SNS_TOPIC_ARN"],
              Message=json.dumps(msg))
```

LambdaDynamo

Analogamente alla funzione precedente [3.1], l’input di questa funzione è un evento generato da Lex. La differenza sostanziale sta nel modo in cui l’input viene considerato per rendere fattibile la query su DyanmoDB. Infatti avendo bisogno di una chiave primaria per la tabella in DynamoDB, si è scelto di utilizzare “Category”. Una chiave primaria non può essere nè NULL nè una stringa vuota, andando in conflitto con la prima versione del file json delle citazioni che invece li considerava valori ammissibili per “Category”. Sono stati quindi cambiati i campi vuoti relativi alla categoria nel dataset con la parola “other”, utilizzando quel valore ogni qual volta l’utente avesse richiesto una citazione di categoria “any”.
Un altro problema si presentava per la situazione analoga del campo “Author”, che poteva essere o non essere inserito dall’utente. Qualora non venisse inserito, la ricerca è effettuata semplicemente per categoria.

```
if (author == ""):
    items = table.query(
        KeyConditionExpression=Key('Category).eq(category))
```

In caso fosse inserito, risultava complesso effettuare una ricerca poiché le uniche query consentite erano per chiave. In questo caso è stata effettuata una query solo per categoria e

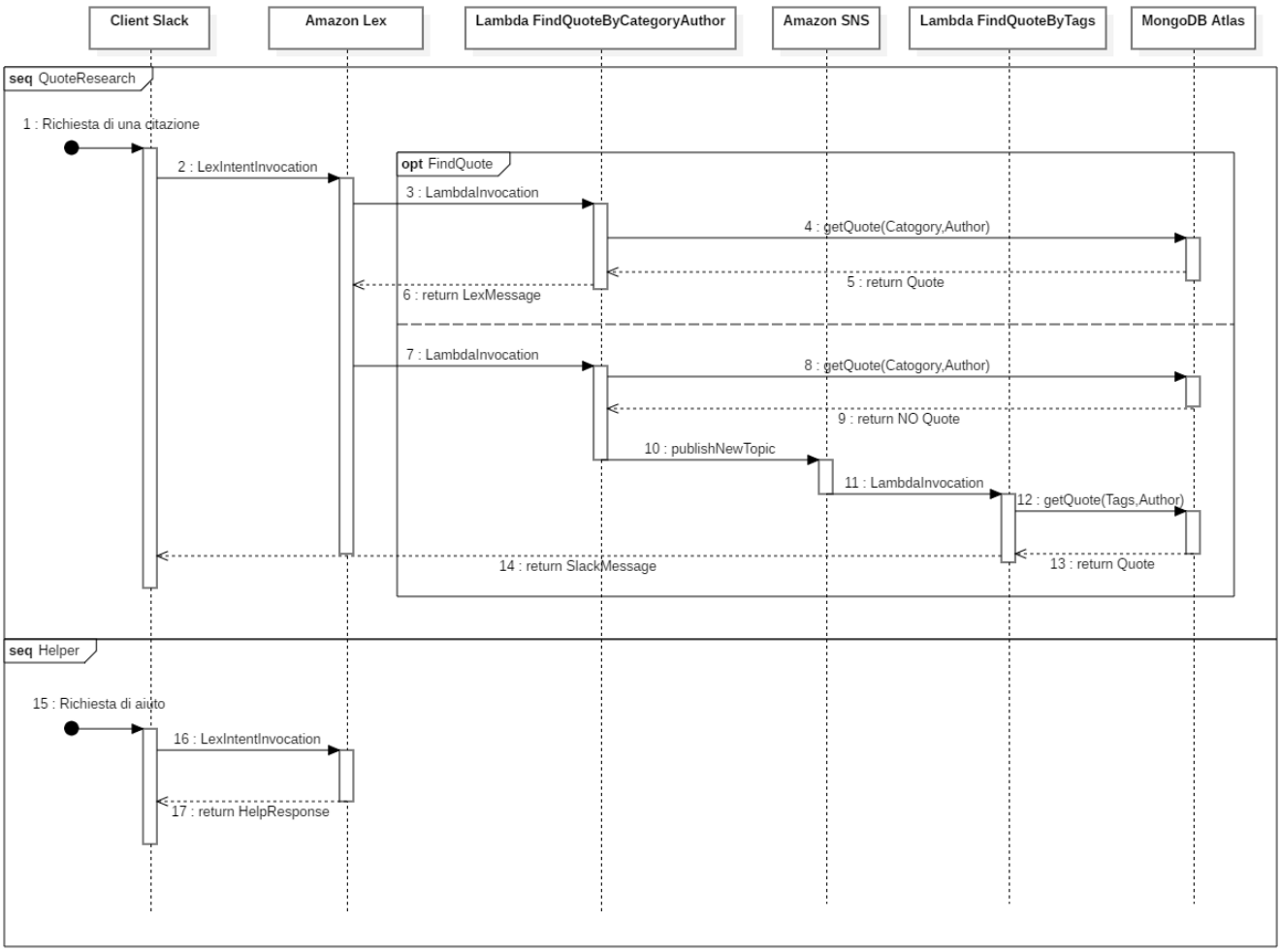

Figure 2. Sequence Diagram dei flussi di esecuzione che utilizzano *Amazon Lex* come punto di accesso alle risorse

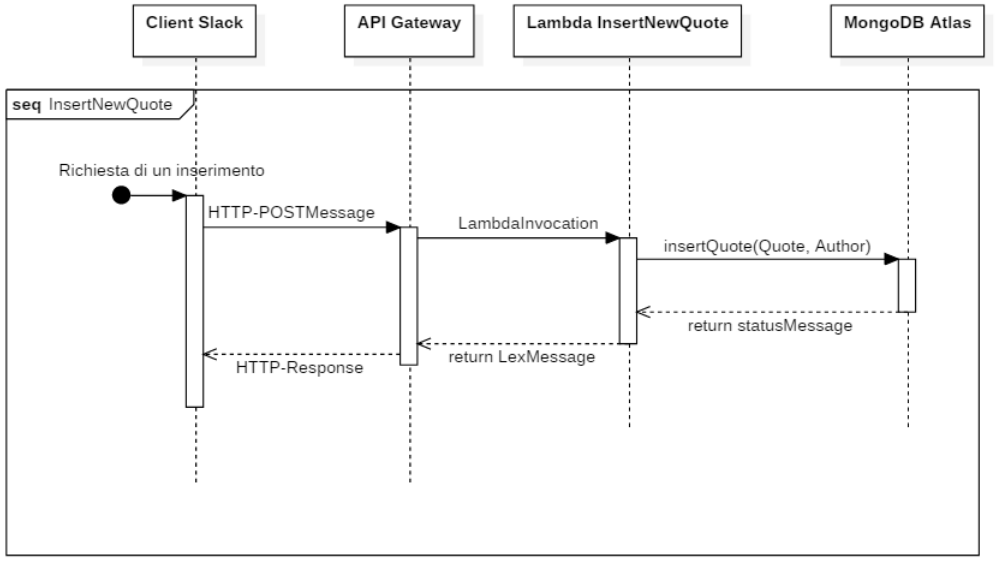

Figure 3. Sequence Diagram dei flussi di esecuzione che utilizzano *API Gateway* come punto di accesso alle risorse

successivamente sono state scremate tutte le citazioni dell'autore richiesto, aumentando dunque la complessità del lavoro della funzione.

```
else:
    itemsCat = table.query(
        KeyConditionExpression=Key('Category').eq(category))
    response = []
    for i in range(len(itemsCat['Items'])):
        if (author in itemsCat['Items'][i]['Author']):
            response.append(itemsCat['Items'][i])
```

FindQuoteByTags

Questa funzione è triggerata dalla pubblicazione sul topic TagForQuote, da cui può leggere il messaggio. Anch'essa ha come variabile d'ambiente l'uri del cluster di MongoDB, con l'aggiunta del webhook di Slack per poter inviare la risposta direttamente alla chat.

Il messaggio letto da topic è un dizionario, dunque necessita di un parsing in formato json per interagire con il data store. Si passa quindi il controllo alla funzione *getQuoteByTags* che esegue la query in maniera analoga a quanto visto in FindQuoteByCategoryAuthor. Dopo aver interrogato il data store, sia in caso di successo che di insuccesso, crea un canale verso Slack e invia il messaggio di risposta in formato json.

```
def slack_callout(quote):
    try:
        response = {"text": quote,
                    "channel": "#%s" %
                         (os.environ['SLACK_CHANNEL'])}
        slack_response =
             requests.post(os.environ['SLACK_CHANNEL'],
             json=response, headers={'Content-Type':
             'application/json'})
        return slack_response.status_code
    except:
        return "error"
```

InsertNewQuote

Dopo aver digitato il comando "*\insert*" nella chat di Slack, l'utente può effettuare l'inserimento di una nuova citazione con il relativo autore, rispettando il formato suggerito. La richiesta viene inviata attraverso l'URL delle API Gateway e infine triggera la lambda InsertNewQuote. È scritta nel linguaggio Nodejs, come già evidenziato, per problemi di compatibilità con il client.
Dopo aver parsato i valori di input e aver istanziato una connessione verso il data store, la lambda effettua due tipi di query:

- controlla se la citazione che sta per essere inserita è già presente;
- effettua un nuovo inserimento solamente se la citazione non esiste.

```
const collection = client.db("test").collection("quotes");
collection.findOne({Quote:newQuoteJSON.Quote},
     function(err, result) {
        if(!result){
            collection.insertOne(newQuoteJSON,
                 function(err, res) {
                   context.succeed("Your quote has been
                        inserted. Try me c:");});
        }else{
            context.succeed("Your quote is already here.
                 Find it ;)");
        }
```

Nella funzione *createJSON* ogni nuova citazione è ricostruita esattamente secondo il formato json descritto in **Persistenza**.

TEST

Sono stati effettuati Unit test sulle singole funzioni direttamente dalla piattaforma AWS Lambda e tramite script bash. Integration e System test sono stati effettuati anch'essi tramite script bash e tramite beta testing dall'applicazione Slack.

```
bot_name='QuotesBotModel'
bot_alias='SiPY'
user_id='client'

category='life'
author='any'

for i in $(seq 1 300); do
   post_text()
   {
       cmd="aws lex-runtime post-text --region us-east-1
            --bot-name=$bot_name --bot-alias=$bot_alias
            --user-id=$user_id"
       $cmd --input-text "Give me a quote about $category
            by $author"
   }
   r=$(post_text 'Give me a quote about $category by
        $author')
   i=$(( i + 1 ))
done
```

Per il monitoraggio di latenza e throughput è stato utilizzato Cloudwatch e il modulo boto3 di python per abilitare python ad interagire con AWS, il modulo time è stato usato per misurare i tempi da codice. Di seguito alcuni grafici che mostrano i tempi in secondi per ogni funzione e i confronti tra le funzioni stesse. I grafici sono stati computati con Matlab.

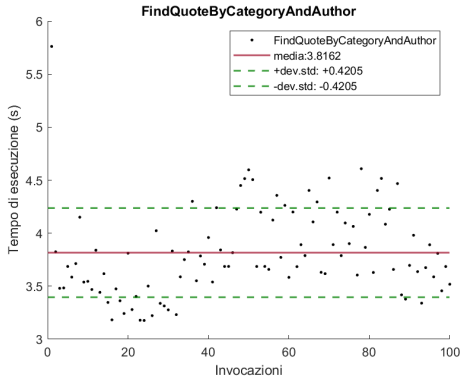

Figure 4. Tempi di risposta di FindQuoteByCategoryAndAuthor per 100 invocazioni.

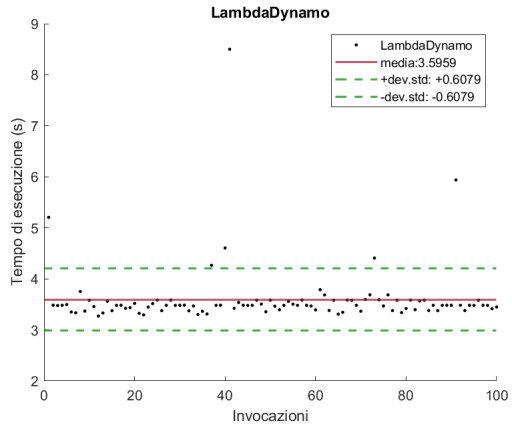

Figure 5. Tempi di risposta di LambdaDynamo per 100 invocazioni.

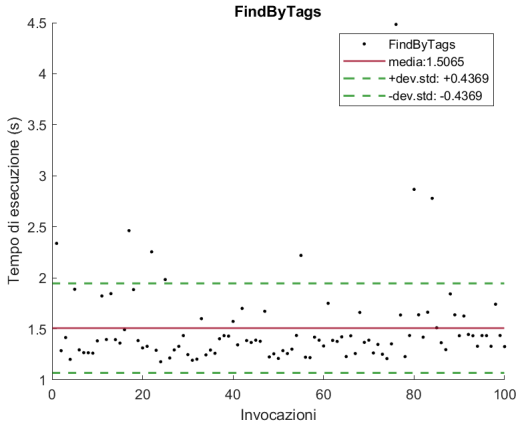

Figure 6. Tempi di risposta di FindQuoteByTags per 100 invocazioni.

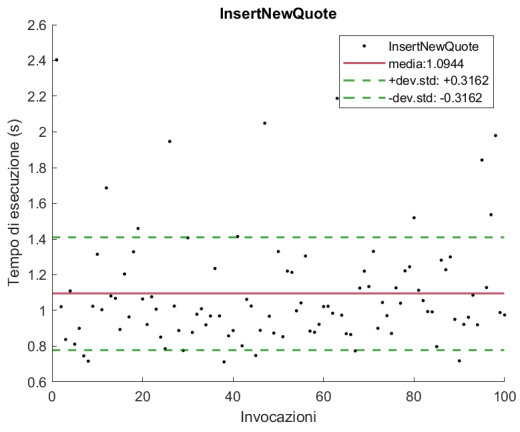

Figure 7. Tempi di risposta di InsertNewQuote per 100 invocazioni.

Confronto fra Data Store

Come già spiegato in precedenza, per diminuire il tempo di risposta di tutto il sistema sono stati fatti dei test per confrontare gli accessi ai due diversi data store per la query computata da FindQuoteByCategoryAuthor.

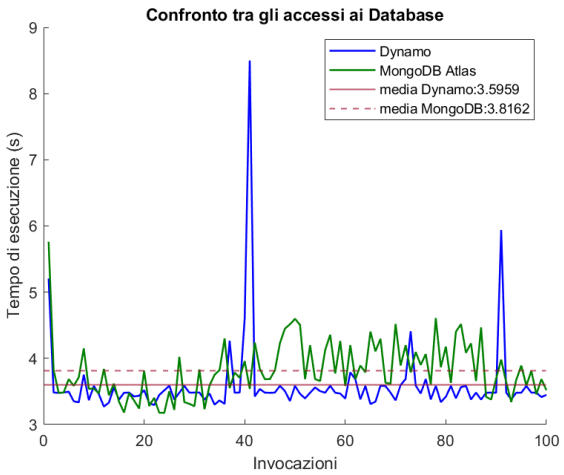

Figure 8. Tempi di risposta dei data store per 100 invocazioni.

Per il test sono stati scelti i valori (love, Jane Austen): è evidente la velocità di risposta di MongoDB rispetto a Dynamo, tanto da dimezzare il tempo di risposta medio [Fig. 9]. Questo avviene per la fluidità della struttura di MongoDB, che consente di effettuare query internamente al documento, al contrario di DynamoDB che permette solo una query per chiave. Inoltre per effettuare una richiesta su DynamoDB che non coinvolga la chiave primaria, è necessario definire indici secondari, che impattano però sulle prestazioni. Dunque come spiegato in [3.2] si è preferito cercare direttamente l'autore inserito dall'utente dall'array dei risultati della query per categoria, ma anche questa soluzione peggiora i tempi di DynamoDB rispetto a MongoDB.

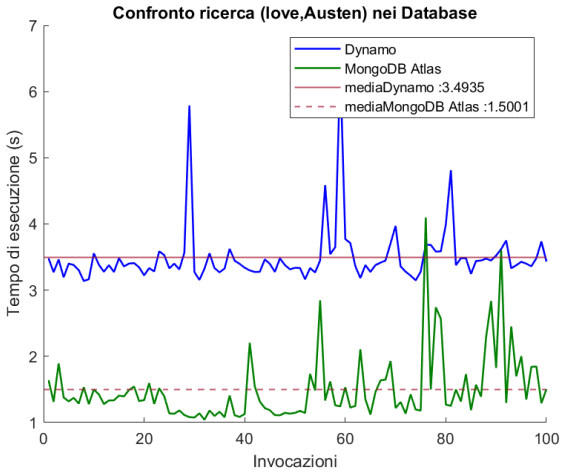

Figure 9. Tempi di risposta dei data store per (love, Jane Austen).

Il secondo caso di test è quello in cui l'utente non sceglie il campo "Author", specificando solo la categoria. In questo caso i valori scelti sono (life,any) e non c'è particolare differenza tra i due [Fig. 10]. Questo risultato è spiegabile perché manca il campo "Author", e quindi non è necessaria la scrematura

successiva che invece viene fatta nel caso (love, Jane Austen).

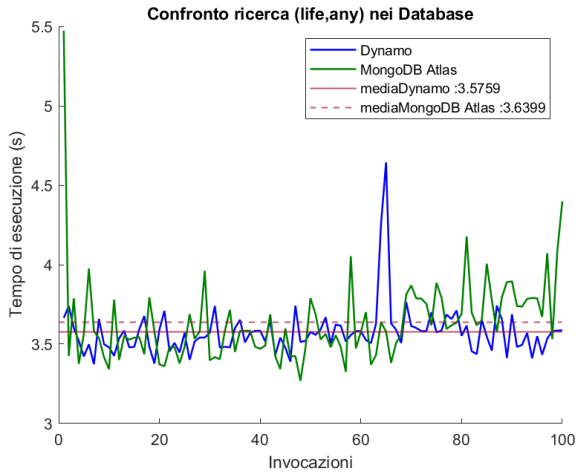

Figure 10. Tempi di risposta dei data store per (life,any).

L'ultimo caso considerato è quello (any,any), in cui l'utente chiede una citazione qualsiasi [Fig. 11]. Dovendo restituire tutte le citazioni con "Category" uguali sia a *any* che a *other*, MongoDB "perde" questo confronto a favore di DynamoDB che invece cerca sempre per chiave con valore *other*. Il motivo per cui è stato necessario indicare con "other" la parola "any" è stato discusso precedentemente [3.2]. Per motivi di incoerenza tra dataset e rappresentazione in DynamoDB sia in termini lessicali sia in termini di struttura delle tabelle stesse, si è deciso di continuare lo sviluppo usando MongoDB.

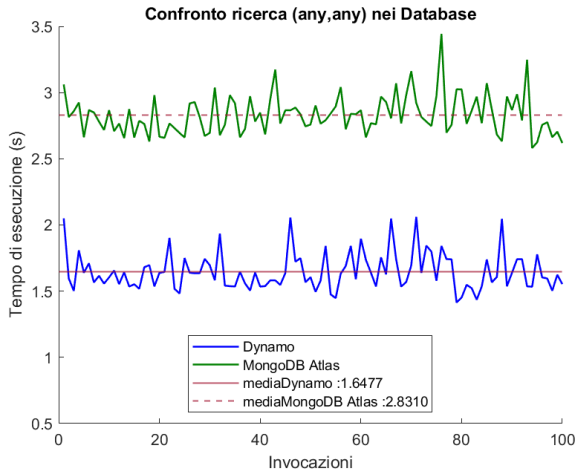

Figure 11. Tempi di risposta dei data store per (any,any).

Per concludere questo paragrafo, di seguito troviamo i grafici relativi al throughput delle tre funzioni lambda. I grafici [Fig. 12], [Fig. 13] e [Fig. 14] sono stati effettuati lanciando 100 istanze delle funzioni e prendendo i risultati di CloudWatch, in particolare l'evento Invocations, che rappresenta il numero di volte in cui le funzioni vengono invocate in un periodo customizzato di 5 minuti. I picchi di throughput bassi potrebbero essere dovuti al throttling applicato da MongoDBAtlas nella versione utilizzata.

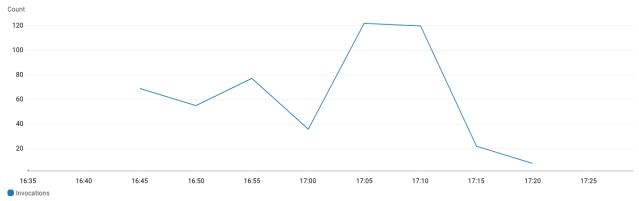

Figure 12. Throughput di FindQuoteByCategoryAuthor

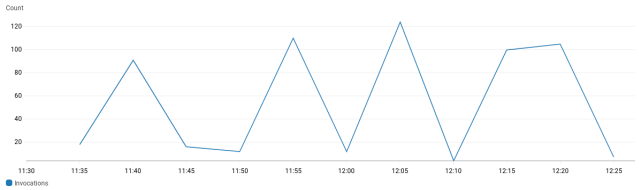

Figure 13. Throughput di FindQuoteByTags

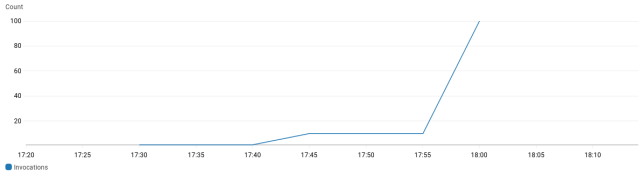

Figure 14. Throughput di InsertNewQuote

DEPLOY

Il deploy dell'applicazione è suddivisono in tre parti, poichè non è stato possibile creare una distribuzione dell'applicazione completa e direttamente scaricabile come viene spiegato in **Limitazioni**. In particolare:

- la creazione della logica di elaborazione sempre nei server di Amazon;
- la creazione del modello del bot come risorsa Amazon;
- la creazione e configurazione finale dell'applicazione lato client.

Tutti i componenti, che costituiscono l'architettura, possono essere ricreati in ogni account Amazon, seguendo le istruzioni descritte nel file README.md. Per renderlo possibile è stato creato un file .yaml contenente l'elenco delle risorse con le relative opzioni e i riferimenti di linking. Il template coincide con un Amazon stack, che comprende:

- il ruolo IAM di esecuzione con le relative policies di esecuzione;
- il TopicSNS;
- la sottoscrizione della lambda;
- le tre lambda già descritte compresi gli eventi e le variabili di ambiente utilizzate;
- le API Gateway compresa la descrizione del mapping.

Tutti i codici delle lambda sono caricati e mantenuti all'interno di un'istanza S3, che viene acceduta in fase di deploy del template. Grazie al servizio CloudFormation è anche possibile visualizzare tutta la struttura così come è vista dalla piattaforma di sviluppo.[Fig. 16]

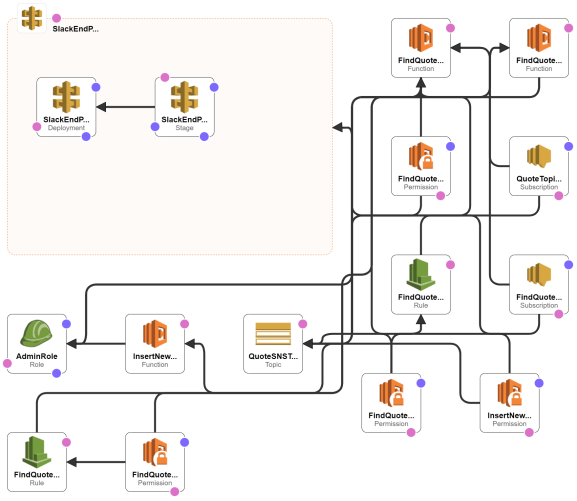

Figure 15. Rappresentazione grafica del template.

Per ricreare il modello del bot invece, è possibile sia effettuare l'import dei singoli componenti (Intent, Slot, Utterances,...) descritti sotto forma di file json attraverso Amazon CLI, oppure importare direttamente il modello completo zippato all'interno della console del servizio Amazon Lex. Le operazioni più importanti restano il building e il publishing del modello, che permettono di generare i channels per la comunicazione *Postback URL* e *OAuth URL* da utilizzare in fase di configurazione del workspace di Slack.

Per quanto riguarda il client, dopo aver creato un account e un workspace, basta aggiungere una nuova applicazione bot Slack, configurata attraverso i channels appena creati per iniziare a chattare con SiPY.

Configurazione e esecuzione

Il bot Slack interagisce con il bot Lex attraverso la configurazione dei channels di comunicazione:

1. creare tutte le risorse server utilizzando il template yaml;
2. importare, buildare e pubblicare il modello del bot Lex;
3. creare il workspace in Slack, aggiungendo una applicazione "bot" e ottenere i codici **App Credentials**, come *Client ID, Client Secret, Verification Token*;
4. compilare il format "Slack" sotto la voce "Channel", dopo aver pubblicato con successo il modello del bot Lex, utilizzando gli **App Credentials**;
5. attivare gli **Event Subscriptions** e **Interactive Components** del bot Slack, utilizzando i codici **Callback URLs** che saranno visibili dopo l'esecuzione corretta del punto precedente;
6. attivare **Slash Commands** del bot Slack, creando un comando "*\insert*" utilizzando la *Request URL* generata dopo il deploy delle API Gateway;
7. attivare gli **Incoming Webhooks** del bot Slack e sostituire il valore del *Webhook URL* in corrispondenza della variabile di ambiente *SLACK-CHANNEL* della lambda FindQuoteByTags;

Per iniziare a chattare con SiPY basta accedere al proprio workspace, aggiungere l'applicazione bot appena creata cliccando sul tasto "+" e infine inviare la prima richiesta.

LIMITAZIONI

La prima limitazione incontrata durante la fase di sviluppo riguarda l'utilizzo di DynamoDB come data store [3.2], poiché per poter effettuare query più complesse e aggregate è necessario definire indici secondari, oppure utilizzare la funzione *scan* invece di *query*, che agisce per chiave. Entrambe le soluzioni avrebbero impattato troppo sulle prestazioni, per cui si è deciso di utilizzare MongoDB.
Nonostante la comodità della gestione automatica del cluster offerta da MongoDB Atlas, anche in questo caso l'utilizzo della versione gratuita rappresenta un collo di bottiglia prestazionale, poichè non permette di effettuare in media un numero superiore di 0.32 query/s.

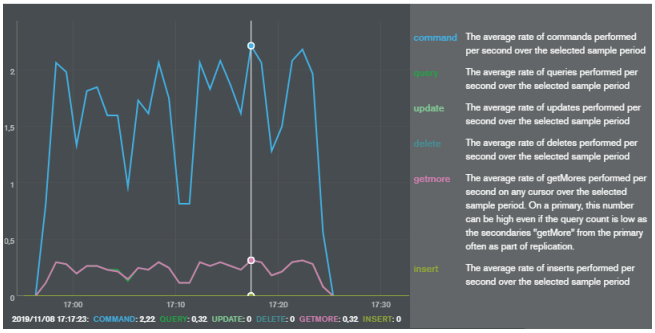

Figure 16. Monitoring di MongoDB Atlas per 400 richieste.

Un'altra limitazione fondamentale riguarda l'utilizzo del client Slack nella sua versione gratuita, poiché oltre ad avere funzionalità limitate, come nel caso del monitoring delle prestazioni, impedisce la gestione completa delle applicazioni create all'interno del workspace, vincolandole esclusivamente al suo proprietario.
Il bot infatti risulta pubblicato e visibile all'interno del workspace di sviluppo, anche per gli eventuali collaboratori, ma per poter essere utilizzato dall'esterno bisognerebbe eseguire un upgrade del workspace e pubblicare ufficialmente l'applicazione nel Market place di Slack. Per questo motivo, il webhook creato per far comunicare FindQuoteByTags con Slack è unico, dunque la risposta per quella query non è visibile da chiunque.

Infine il debugging delle applicazioni serverless è generalmente complesso e, anche se con AWS è possibile monitorare l'applicazione con la piattaforma CloudWatch, questa deve essere ben configurata anche con eventi ad hoc. L'IDE messo a disposizione da AWS Lambda non consente, dunque, un debugging live, ma è necessario sempre creare allarmi verso

CloudWatch. Per quanto riguarda il deploy i template yaml delle singole risorse, dopo esser stati scaricati perdono la formattazione di alcuni componenti, risultando errati in fase di deploy completo dello stack.

SVILUPPI FUTURI

Un miglioramento riguarda un nuovo criterio di scelta delle citazioni. Anziché scegliere la citazione randomicamente tra quelle restituite dopo aver effettuato le query dal data store, si può sfruttare il campo "Popularity" contenuto nel dataset per ogni citazione. Come spiegato in [2.5] il campo è un numero compreso tra 0 e 1 che rappresenta un indice di popolarità per la citazione. Quello che si può fare è istanziare una nuova funzione lambda in grado di scegliere le citazioni con un valore superiore (o inferiore, a seconda della semantica) ad una certa soglia di popolarità.

Per abbattere i tempi di cold-start, che sono un problema generale di Lambda, esistono approcci di warm up che consistono nell'avere una ulteriore funzione lambda che fa ping alle altre funzioni lambda che devono essere tenute warm: questo fa in modo che l'architettura di AWS Lambda non faccia lo shutdown e il successivo start up dei container usati per eseguire le lambda.

Per migliorare i tempi di latenza in generale si può utilizzare Lambda@Edge insieme a CloudFront per avvicinare i servizi all'utente, sviluppando di fatto un'applicazione serverless ed edge computing.

REFERENCES

[1] www.aws.amazon.com.

[2] docs.atlas.mongodb.com

[3] www.freecodecamp.org/news/how-to-design-a-serverless-async-api-6cfd68939459/

[4] www.github.com/akmittal/quotes-dataset

[5] www.baeldung.com/java-aws-lambda

[6] www.docs.aws.amazon.com/en_us/lex/latest/dg/gs-cli-create-order-flowers.html

[7] www.medium.com/@cu_tech/create-a-slack-slash-command-with-aws-lambda-83fb172f9a74

[8] www.mongodb.com/blog/post/serverless-development-with-nodejs-aws-lambda-mongodb-atlas

[9] https://www.mongodb.com/compare/mongodb-dynamodb?lang=it-it