Sistemi Distribuiti e Cloud Computing

Project A4: microservice application vs. serverless application

Luca Di Totto
Matricola 0333084
Laurea magistrale in Ingegneria Informatica
Università di Roma Tor Vergata
luca.ditotto@students.uniroma2.eu

Luca Di Marco
Matricola 0333083
Laurea magistrale in Ingegneria Informatica
Università di Roma Tor Vergata
luca.dimarco.01@students.uniroma2.eu

***Abstract*—Questo documento ha l'obiettivo di descrivere la realizzazione di una semplice applicazione implementata seguendo due pattern differenti, microservizi e serverless, e di effettuare delle valutazioni su entrambe le architetture. Verranno presentate le architetture dei due sistemi, le funzionalità offerte e le sfide incontrate durante la realizzazione dell'applicazione.**

I. Introduzione

Il progetto realizzato consiste nell'implementazione di un'applicazione che fornisce una piattaforma web, dove è messo a disposizione dell'utilizzatore uno strumento di riconoscimento facciale. L'utente avrà la possibilità di effettuare il login e successivamente, sarà in grado di selezionare una foto dal proprio dispositivo, con la quale effettuare il riconoscimento facciale delle persone che compaiono nella stessa e, dunque, conosciute dall'algoritmo di riconoscimento. A seguito dell'elaborazione, l'utente visualizzerà a schermo la foto caricata, affiancata dai nomi delle persone presenti, che il sistema è stato in grado di riconoscere. A questo punto verrà offerta all'utente la possibilità di inviare una mail alle persone riconosciute, in modo da informarle della loro presenza all'interno della foto. Un'idea di utilizzo al quale si è pensato è quella di un'azienda che offre come servizio l'utilizzo dell'applicazione in oggetto. Si è ipotizzato infatti che la piattaforma può essere venduta da un'azienda ai propri clienti, la quale avrà il compito di occuparsi della personalizzazione del modello di riconoscimento in base alle richieste dell'acquirente.

Nel progetto l'applicazione viene realizzata seguendo due stili architetturali differenti, a microservizi mediante un cluster Elastic Kubernetes Service e serverless mediante le AWS Lambda.

Nella prima parte di questo documento verranno specificati i dettagli implementativi dell'architettura a microservizi, nella successiva i dettagli dell'architettura serverless. Infine, verranno confrontate le due attraverso considerazione e analisi dei test di carico.

II. Architettura a microservizi

A. Struttura del cluster

L'applicazione è composta da 3 microservizi sviluppati in Python e adotta per la comunicazione il framework sincrono gRPC. Ognuno dei tre microservizi è istanziato all'interno di un container. Per la creazione delle immagini dei container è stato utilizzato Docker e successivamente Kubernetes come tool di orchestrazione dei container. Questi ultimi sono inseriti in pod Kubernetes ed ogni pod inserito in un Deployment per permettere la sua gestione e replicazione. Ogni Deployment viene affiancato da un Service, che nel nostro caso è di tipo Load Balancer per permettere il bilanciamento del carico delle richieste tra le varie repliche all'interno del Deployment. Ad ogni Deployment viene inoltre affiancato un Horizontal Pod Autoscaler (HPA) che, mediante le metriche di utilizzazione fornite dal Metric Server, permette la scalabilità orizzontale dei pod, per far fronte ad un aumento delle richieste. Nel nostro caso le metriche sono basate sull'utilizzazione della CPU dei vari pod ed è impostata ad un limite del 25%. Utilizzando, inoltre, una comunicazione basata su gRPC si è reso necessario l'utilizzo aggiuntivo di un Service Mesh su ogni Deployment, nel nostro caso Linkerd, per permettere di instradare le richieste anche su pod diversi, altrimenti impossibile vista la struttura stessa della comunicazione gRPC. Per lo sviluppo è stato inoltre implementato il design pattern Circuit Breaker all'interno dei metodi che richiamano metodi gRPC in modo da evitare errori in cascata e ripetuti da parte di repliche in errore e non funzionanti.

Nella nostra applicazione troviamo, come già anticipato, tre Deployment: app-deployment responsabile della gestione della web app e della presa delle richieste dell'utente, face-rec-deployment responsabile dell'algoritmo di riconoscimento facciale ed infine mail-deployment responsabile dell'invio delle e-mail.

Per il salvataggio delle foto caricate dall'utente e successivamente modificate e riconosciute dall'algoritmo di face-recognition ci siamo appoggiati su di un bucket S3. Per l'invio delle mail invece abbiamo utilizzato un semplice client SMTP che inoltra le varie e-mail su indirizzi salvati all'interno di un database ospitato dal servizio DynamoDB. Si rimanda al capitolo *IV sezione E* per l'immagine descrittiva dell'architettura appena discussa.

B. Impostazioni ed utilizzo del cluster EKS

La struttura a microservizi è stata interamente ospitata mediante l'utilizzo del servizio Elastic Kubernetes Service di Amazon AWS. All'interno dello stesso è stato aggiunto un gruppo di nodi destinati ai vari servizi e Deployment. Ogni nodo ospita una macchina EC2 di tipo t4g.small o superiore per motivi di capacità di calcolo da parte del Deployment di riconoscimento facciale. Nel nostro caso le macchine EC2 t4g.small sono di tipologia ARM, ma è possibile optare anche per macchine in architettura x86.

C. Funzionalità

1) Login

Il servizio di login mette a disposizione dell'utente 5 campi di input in cui lo stesso può inserire le proprie credenziali AWS (AWS access key, AWS access key ID e AWS session token), attraverso le quali il sistema andrà a creare una cartella nel bucket S3, corrispondente all'ID utente, nella quale verranno salvate le foto elaborate, ed e-mail e password di un indirizzo di posta elettronica che verrà successivamente utilizzate per l'invio della mail. A seguito del click sul bottone di submit da parte dell'utente, tutti i dati inseriti vengono salvati nei cookie del browser in modo da essere rintracciabili ed utilizzabili durante tutto il ciclo di vita dell'applicazione. Successivamente l'utente viene indirizzato verso la pagina di caricamento della foto.

2) Upload e Face recognition

Il servizio di upload permette all'utilizzatore di selezionare una foto dal proprio computer e caricarla, in modo da far partire il riconoscimento facciale. L'applicazione, quindi, accede a S3 con le credenziali AWS salvate precedentemente e salva l'immagine nella cartella denominata con lo stesso ID AWS di chi l'ha caricata (se non esiste, viene creata). A questo punto viene invocata la funzione che effettua il riconoscimento facciale, che recupera la foto presente nella cartella del bucket S3, la elabora, cerca volti conosciuti ed infine, modifica la foto aggiungendo un quadrato rosso su tutti i volti noti, con i nomi associati. Finite tutte le modifiche, la foto aggiornata viene salvata nuovamente sul bucket sovrascrivendo la precedente. Viene successivamente ricaricata la pagina web dove viene mostrata a schermo la nuova immagine modificata e vengono visualizzati di fianco questa l'elenco dei nomi delle persone riconosciute nella foto offrendo la possibilità notificare le stesse con una mail.

3) Send Email

Il servizio di send e-mail viene invocato al click del bottone associato ai nomi all'interno della foto precedentemente ottenuta: prende in ingresso il nome della persona riconosciuta nella foto e mediante una query verso il database di indirizzi su DynamoDB recupera la e-mail dell'interessato. Successivamente mediante un semplice client SMTP inoltra la e-mail al destinatario associato inserendo come corpo un URL diretto verso la foto modificata.

D. Design pattern

All'interno di ogni metodo che esegue chiamate gRPC si è introdotto il pattern Circuit Breaker, in modo da evitare che un problema o malfunzionamento su un determinato servizio si ripercuota in cascata sugli altri e porti ad un sovraccarico inutile all'interno dei Deployment e un aumento degli errori tali da rendere il comportamento dell'applicazione non predicibile. L'invocazione del metodo richiamato da gRPC non è dunque immediato, ma mediato da un proxy. Quest'ultimo ha il compito di aprire il circuito nel momento in cui il servizio invocato solleva un malfunzionamento. Dopo un numero di tentativi, impostato a 2 nel nostro progetto, se il malfunzionamento persiste, il circuito viene aperto e viene interrotta la comunicazione tra client e servizio per un tempo di timeout (5 secondi). Al termine di questo timeout l'interruttore viene chiuso nuovamente permettendo il passaggio di un numero di richieste ridotto. Se quest'ultime vanno a buon fine, il servizio riprende il suo normale funzionamento, altrimenti viene aperto nuovamente l'interruttore e viene fatto ripartire il timeout precedentemente introdotto.

E. Strumenti software utilizzati

Gli strumenti utilizzati per la creazione dell'architettura a microservizi sono i seguenti:

- Docker: piattaforma software per la virtualizzazione a livello di sistema operativo utilizzata per la creazione e distribuzione di applicazioni. Nel caso in esame è stato utilizzato per la creazione di immagini per i container che ospitano le varie funzionalità dei servizi precedentemente introdotti, da utilizzare successivamente all'interno dei pod Kubernetes;
- Kubernetes: piattaforma per la gestione e l'orchestrazione dei container in maniera scalabile e distribuita. Nel caso in esame ogni container è ospitato all'interno di un Pod, dove quest'ultimo viene gestito all'interno di un Deployment (insieme di Pod). Ogni Deployment è affiancato ad un Service che permette di avere un accesso dall'esterno al Deployment;
- Minikube: tool per l'esecuzione di un cluster Kubernetes in locale;
- gRPC: framework open-source di Remote-Procedure-Call per la comunicazione tra i vari servizi distribuiti tramite Kubernetes;
- Flask: micro-framework Python utilizzato per l'implementazione di un semplice server Web;
- Linkerd: servizio mesh per Kubernetes, che si occupa di migliorare le performance e la stabilità dei servizi in esecuzione, attraverso l'installazione di micro-proxy accanto a ciascuna istanza del servizio, in modo da manipolare il traffico in entrata e in uscita da questo senza introdurre una latenza eccessiva. Attraverso i Service Mesh di Linkerd, viene gestita la scalabilità orizzontale dei pod, evitando che l'applicazione smetta di rispondere negli istanti in cui sta effettuando scaling.
- DynamoDB: servizio di database NoSQL interamente gestito che combina prestazioni elevate e prevedibili con una scalabilità ottimale. Esso consente di scaricare gli oneri di gestione e dimensionamento di un database distribuito in modo da non doversi più preoccupare di provisioning dell'hardware, installazione e configurazione, replica, applicazione di patch al software e dimensionamento del cluster. Nell'applicazione realizzata a microservizi, viene utilizzato per il salvataggio delle credenziali utente ed indirizzo e-mail;
- Amazon S3: Amazon Simple Storage Service (Amazon S3) è un servizio di archiviazione di oggetti che offre scalabilità, disponibilità dei dati, sicurezza e prestazioni all'avanguardia nel settore. Nell'applicazione è utilizzato per il salvataggio delle immagini caricate dall'utente;
- Amazon EKS: servizio gestito da Amazon AWS per la creazione e distribuzione di cluster Kubernetes;
- Amazon VPC: servizio Amazon per la creazione di reti virtuali private e pubbliche in cui avviare il cluster

EKS e permettere la comunicazione all'interno e all'esterno dello stesso.

- Locust: framework Python per la creazione di load test.
- Amazon S3: Amazon Simple Storage Service (Amazon S3) è un servizio di archiviazione di oggetti che offre scalabilità, disponibilità dei dati, sicurezza e prestazioni all'avanguardia nel settore. Utilizzato, nel nostro caso, per il salvataggio delle immagini caricate dall'utente e modificate dal meccanismo di face-recognition;
- Client SMTP: libreria Python per la creazione di un semplice client SMTP per l'inoltro di e-mail.

F. *Test e risultati*

I seguenti risultati riguardano il cluster Kubernetes precedentemente descritto, inserito all'interno di un contesto EKS distribuito dove vengono previsti un minimo di 8 nodi fino ad un massimo di 25.
Le macchine t4g.small presentano le seguenti caratteristiche:

- 2 vCPU;
- 2 GiB di memoria.

Le configurazioni dei pod nei vari deployment sono le seguenti:

- App-deployment:
  - Request: CPU = 250m, Memory = 500 Mi;
  - Limit: CPU = 500m, Memory = 1Gi
- Face-rec-deployment:
  - Request: CPU = 500m, Memory = 1 Gi;
  - Limit: CPU = 1, Memory = 1.5Gi
- Mail-deployment:
  - Request: CPU = 250m, Memory = 500 Mi;
  - Limit: CPU = 500m, Memory = 1Gi

Testando il nostro cluster mediante l'utilizzo di Locust ed effettuando test di carico abbiamo potuto notare che nel caso di upload di foto e successivo riconoscimento facciale i tempi di risposta si assestano ad un massimo di circa 21 s per un massimo di 15 utenti connessi simultaneamente e 130 richieste. Nel caso di invio e-mail invece, il tempo massimo diminuisce fino a circa 14 s per un massimo di 15 utenti e 160 richieste. Nella simulazione dei test abbiamo inoltre constato due fallimenti nel processo di upload poiché in contemporanea dell'operazione di scaling all'interno del cluster stesso. Quest'ultima operazione ha causato, anche se per un brevissimo tempo, un disservizio nei Deployment interessati. Si rimanda al capitolo *IV sezione E* per le tabelle riassuntive dei dati sui tempi di risposta ottenuti. Da considerare, inoltre, che la macchina ospitante l'istanza di Locust è stata la macchina di sviluppo locale: il cluster veniva acceduto mediante l'external-IP del servizio di Web App.

III. ARCHITETTURA SERVERLESS

A. *Struttura dall'applicazione*

L'applicazione serverless è composta da sei funzioni Lambda:

- Una per la funzionalità di login;
- Una per l'inserimento della foto tramite upload su S3;
- Una per l'invocazione di Rekognition per eseguire il riconoscimento facciale;
- Una per la funzionalità di invio e-mail;
- Due funzioni aggiuntive non richiamabili dall'utente per, rispettivamente, la creazione di una collection Rekognition e per l'aggiunta di immagini campione per eseguire il riconoscimento facciale.

Ogni funzione Lambda, ad esclusione delle ultime due, è associata ad un API Gateway fornito da Amazon per permettere l'invocazione della funzione da remoto tramite URL. Il client Web viene ospitato da Amazon Amplify, connesso al repository GitHub con all'interno tutti i file necessari per il corretto funzionamento del lato Web. Si rimanda al capitolo *IV sezione E* per l'immagine descrittiva dell'architettura appena discussa.

B. *Impostazioni ed utilizzo delle funzioni Lambda*

Le funzioni Lambda comunicano tra loro mediante richieste XML inviate dal client Web ospitato all'interno di Amplify. In questo caso non ci sono particolari da tenere a mente riguardo scalabilità delle funzioni Lambda poiché è una funzionalità offerta già all'interno di Amazon e gestita dallo stesso.

C. *Funzionalità*

Le funzionalità offerte sono le medesime dell'applicazione con architettura a microservizi. Le uniche differenze riguardano la funzione di riconoscimento facciale che utilizza il servizio Rekognition offerto da Amazon e non più la libreria Python e la funzionalità di login, che non inserisce i dati immessi dall'utente all'interno di un cookie come nell'architettura precedente, ma, bensì all'interno di una tabella dedicata di DynamoDB.

D. *Design Pattern*

Precedentemente, nel caso di applicazione a microservizi, il riconoscimento facciale veniva implementato mediante una libreria Python Open Source. Nel caso dell'architettura serverless la funzionalità è stata implementata utilizzando il servizio Amazon Rekognition: questo poiché all'interno delle funzioni Lambda non è stato possibile importare le librerie utilizzate precedentemente per un problema di incompatibilità. Inoltre, le due funzioni Lambda aggiuntive, introdotte precedentemente, sono utilizzabili esclusivamente dal gestore del servizio per la creazione di una collection Rekognition e per l'addestramento di essa. Non sono in nessun modo richiamabili dall'utente esterno (client).

E. *Strumenti software utilizzati*

- Amazon Lambda: servizio di calcolo basato su eventi serverless, gestito da Amazon AWS, che permette di eseguire codici per qualsiasi tipo di applicazione o servizio back-end senza la necessità di gestire server. Esegue il codice su un'infrastruttura di elaborazione ad alta disponibilità e gestisce tutta l'amministrazione delle risorse di elaborazione, compresa la manutenzione del server e del sistema operativo, il provisioning e la scalabilità automatica della capacità;
- Amazon API Gateway: Amazon API Gateway è un servizio AWS per la creazione, la pubblicazione, la gestione, il monitoraggio e la protezione di API REST, HTTP e WebSocket a qualsiasi livello. Attraverso la REST API creata, viene richiamata una funzione Lambda specifica tramite richiesta HTTP. L'utilizzo di Gateway API fornisce agli utenti un endpoint HTTP sicuro per richiamare la funzione

Lambda e può aiutare a gestire grandi volumi di chiamate alla funzione limitando il traffico e convalidando e autorizzando automaticamente le chiamate API;

- Amazon DynamoDB: servizio di database NoSQL interamente gestito che combina prestazioni elevate e prevedibili con una scalabilità ottimale. Esso consente di scaricare gli oneri di gestione e dimensionamento di un database distribuito in modo da non doversi più preoccupare di provisioning dell'hardware, installazione e configurazione, replica, applicazione di patch al software e dimensionamento del cluster. Nell'applicazione serverless è stato utilizzato per il salvataggio delle credenziali utente ed indirizzi e-mail;
- Amazon Rekognition: software di riconoscimento ad autoapprendimento di Amazon AWS che può essere utilizzato per identificare oggetti o persone. Tra gli svariati casi d'uso che questo software può avere è presente il riconoscimento facciale, che permette di cercare nelle immagini i volti che corrispondono a quelli presenti in un container, noti come raccolte di volti (o collection). Nell'applicazione è stato utilizzato per effettuare il riconoscimento dei volti all'interno delle immagini caricate dall'utente.
- Amazon S3: Amazon Simple Storage Service (Amazon S3) è un servizio di archiviazione di oggetti che offre scalabilità, disponibilità dei dati, sicurezza e prestazioni all'avanguardia nel settore. Nell'applicazione è utilizzato per il salvataggio delle immagini caricate dall'utente;
- Client SMTP: libreria Python per la creazione di un semplice client SMTP per l'inoltro di e-mail.

F. *Test e risultati*

Nella nostra implementazione, le funzioni Lambda presentano una memoria di 128 MB ed utilizzano una concorrenza account non riservata con un limite pari a 1000. Testando la nostra applicazione serverless mediante l'utilizzo di Locust ed effettuando test di carico abbiamo potuto notare che nel caso di upload di foto e successivo riconoscimento facciale i tempi di risposta si assestano ad un massimo di circa 4 s per un numero di utenti pari a 15 connessi simultaneamente e 300 richieste. Nel caso di invio e-mail invece, il tempo massimo diminuisce fino a circa 1.3 s per un numero di utenti pari a 15 connessi simultaneamente e 300 richieste. Aumentando gradualmente gli utenti connessi simultaneamente fino ad un massimo di 30, i valori precedentemente illustrati non variano. Nella simulazione dei test abbiamo inoltre constato un solo fallimento dovuto alla sola scadenza delle credenziali temporanee di AWS e non alla implementazione delle funzioni o al processo di scalabilità delle stesse. Si rimanda al capitolo *IV sezione E* per le tabelle riassuntive dei dati sui tempi di risposta ottenuti. Da considerare, inoltre, che la macchina ospitante l'istanza di Locust è stata la macchina di sviluppo locale: le funzioni Lambda venivano accedute mediante l'URL messi a disposizione dagli API Gateway.

IV. Conclusioni

A. *Tempi di risposta*

Dai risultati ottenuti è possibile vedere come, dal punto di vista del tempo di risposta, l'architettura serverless si è rivelata essere la più rapida, con una differenza di circa 17 secondi per la funzionalità di upload e riconoscimento facciale e di circa 12 secondi per la funzionalità di inoltro e-mail. Da tener conto è la differenza di implementazione riguardo l'algoritmo di riconoscimento facciale: nel caso dell'applicazione a microservizi è stato scelto un algoritmo disponibile nella libreria Python; per quanto riguarda l'applicazione serverless è stato scelto il servizio proprietario di Amazon Rekognition. Quest'ultimo potrebbe essere uno dei motivi del sostanziale divario tra i due tempi di risposta. Un secondo motivo molto importante è che, nel caso del serveless le richieste vengono inoltrate direttamente alla funzione Lambda, mediante richieste XML al suo API Gateway. Nel caso, invece, di richieste nell'applicazione a microservizi, una singola richiesta passa per più endpoint, come il Service Load Balancer del Deployment e per il servizio di Service Mesh di Linkerd, che introduce un quantitativo di delay non evitabile.

B. *Implementazione*

La facilità di implementazione e deploy trovata nello sviluppo dell'applicazione in architettura serverless non si riflette nell'implementazione e deploy dell'architettura a microservizi, la quale lascia molto carico di lavoro al programmatore per quanto riguarda la gestione del bilanciamento del carico, della scalabilità orizzontale e della gestione di possibili servizi non funzionanti. Caratteristiche che, nell'ambiente delle funzioni Lambda, troviamo implementate e gestite da Amazon AWS stesso, lasciando al programmatore solo l'onere di gestire il codice dell'applicazione stessa. Di contro, però, nell'architettura a microservizi abbiamo un più alto grado di controllo nella gestione delle singole macchine, della gestione della loro comunicazione e del loro deploy.

C. *Costi*

I costi di gestione e di deploy dell'architettura a microservizi sono decisamente più alti rispetto ai costi dell'architettura serverless a parità di carico: nel caso di cluster EKS, infatti, sono da tener presente costi di gestione delle macchine EC2 che ospitano i vari nodi, costi delle VPC utilizzate e dei Gateway per la connessione verso l'esterno. Dal punto di vista del serverless, invece il costo è proporzionale al quantitativo di richieste che arrivano all'applicativo: si paga ciò che si usa. I costi, invece, di servizi come S3 e DynamoDB sono presenti in entrambe le architetture.

D. *Vendor Lock-In*

In entrambe le architetture troviamo un forte utilizzo di servizi come S3 o DynamoDB. Quest'ultimi possono portare ad un deciso vendor lock-in nel momento in cui si decida di portare su un'altra piattoforma (come Google) le due architetture. Dal punto di vista dell'architettura a microservizi il legame con Amazon AWS è decisamente ridotto avendo un algoritmo di riconoscimento facciale non proprietario. Dal punto di vista serverless invece, utilizzando il servizio Amazon Rekognition presenta un più alto grado di possibile vendor lock-in, portando quindi la sua migrazione ad essere più difficile o più dispendiosa in termini di tempo.

E. Figure e tabelle

Si allegano di seguito le tabelle riassuntive dei tempi di risposta precedentemente discussi e le immagini esplicative delle due diverse architetture precedentemente introdotte ed illustrate.

TABLE I. TEMPI DI RISPOSTA MICROSERVIZI (15 UTENTI)

Method	***Name***	***# Richieste***	***50%ile (ms)***
GET	/	287	250
POST	/sendemail	160	14000
POST	/upload	133	21000

Fig. 1. Tempi di risposta per l'architettura a microservizi

TABLE II. TEMPI DI RISPOSTA SERVERLESS (15 UTENTI)

Method	***Name***	***# Richieste***	***50%ile (ms)***
POST	/login	351	713
POST	/sendemail	334	1402
POST	/upload	275	1419
POST	/facerec	298	2625

Fig. 2. Tempi di risposta per l'architettura serverless

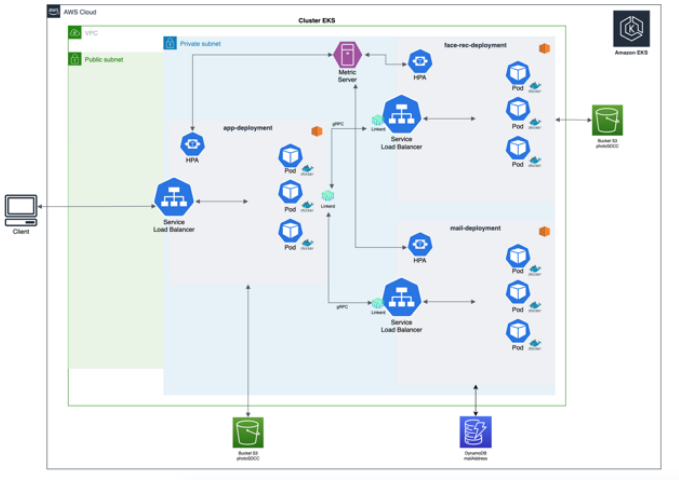

Fig. 3. Architettura a microservizi (cluster EKS)

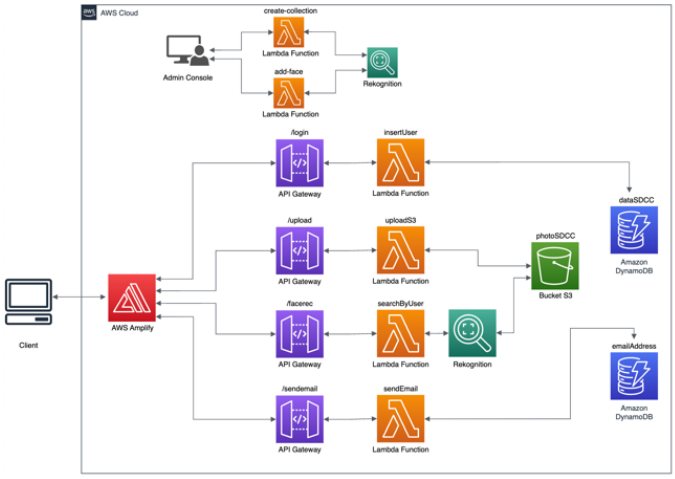

Fig. 4. Architettura serverless (funzioni Lambda)