**Relazione progetto Sistemi Cloud**

# AItinerary

**Luca Strano**

Matricola: 1000052662

**Professori:**

Giuseppe Pappalardo
Antonio Salvatore Calanducci

Anno Accademico 2024/2025

Corso di Laurea Magistrale in Informatica

# Indice

# Capitolo 1

# Introduzione

In questo elaborato verrà presentata l'implementazione e l'integrazione di un'applicazione web su Amazon Web Services (AWS), denominata *AItinerary*, un sistema di gestione di itinerari turistici che consente agli utenti di ottenere consigli personalizzati su luoghi da visitare, ristoranti e attività da svolgere in base alla città di interesse e al numero di giorni di permanenza in essa. Di seguito un diagramma che illustra l'architettura generale dell'applicazione:

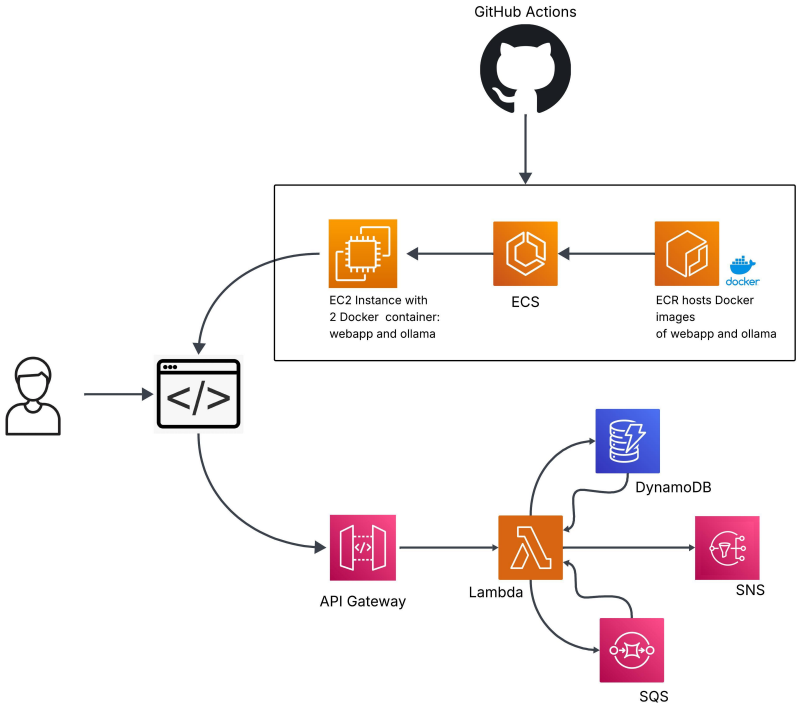


**Figura 1.1:** Schema progetto

## 1.1 Obiettivi

Il presente progetto nasce con l'obiettivo di approfondire l'utilizzo dei servizi offerti da Amazon Web Services, una delle piattaforme più diffuse e consolidate nel settore Cloud Computing. Lo scopo è stato quello di progettare e realizzare una semplice applicazione cloud-native, con l'intento di comprendere in modo pratico l'interazione tra i diversi servizi AWS e le logiche architetturali alla base delle moderne applicazioni distribuite.

Nella realizzazione del progetto sono stati adottati una serie di servizi mirati di AWS, tra cui Lambda, API Gateway, DynamoDB, ECS, SQS ed altri, con l'obiettivo di comprenderne il funzionamento, le modalità di integrazione e i vantaggi offerti rispetto a soluzioni più tradizionali.

Questo lavoro vuole essere quindi un piccolo ma concreto passo verso una più profonda comprensione dell'ecosistema cloud e delle sue potenzialità applicative, offrendo un'esperienza pratica che completi la preparazione teorica acquisita durante il corso.

# Capitolo 2

# Descrizione dell'applicazione

Quella realizzata è un'applicazione web interattiva progettata per generare itinerari di viaggio personalizzati tramite l'ausilio di un modello linguistico di grandi dimensioni (LLM). Il sito presenta un'interfaccia semplice ed essenziale, pensata per facilitare l'interazione da parte dell'utente.

## 2.1 Front-end


**Figura 2.1:** Homepage dell'applicazione

Come si può notare (Figura 2.1), la parte più in alto del front-end consiste in un form in cui l'utente può inserire tre informazioni fondamentali: un'email, il nome della città che intende visitare e il numero di giorni previsti per il soggiorno. Una volta compilati i campi, l'utente può premere il pulsante "Genera Itinerario", che attiva il processo di generazione vera e propria.

Nella parte inferiore della pagina invece viene mostrata una top 5 delle città più ricercate all'interno dell'applicativo.

## 2.2 Back-end

Dal punto di vista architetturale, tutta la logica applicativa lato back-end è stata progettata per essere eseguita nel cloud, utilizzando i servizi serverless di AWS. In particolare, alla pressione del pulsante vengono effettuate una serie di richieste verso un API Gateway, il quale funge da punto di ingresso per il sistema. L'API Gateway attiva più funzioni Lambda, ciascuna incaricata di eseguire una specifica operazione, tra cui:

- L'invio dei dati inseriti dall'utente verso il modello che si occuperà di generare un itinerario di viaggio coerente con le informazioni fornite, suddiviso per giornate e con una varietà di suggerimenti su cosa visitare, fare o esplorare nella città scelta.
- L'aggiornamento della classifica delle città più ricercate. Ogni volta che un utente effettua una nuova richiesta, il conteggio relativo alla città inserita viene aggiornato e la classifica viene ricalcolata in tempo reale. Questo permette di visualizzare dinamicamente le cinque destinazioni più popolari tra gli utenti che hanno utilizzato il servizio.

L'applicativo combina quindi funzionalità interattive, generazione dinamica dei contenuti tramite LLM e una semplice componente statistica che arricchisce l'esperienza utente.

# Capitolo 3

# Amazon Web Services

In questo capitolo entreremo nel dettaglio di tutti i servizi offerti da AWS che sono stati implementati all'interno dell'applicazione per fornire un ambiente scalabile e affidabile per l'esecuzione e la gestione di tutti i vari componenti all'interno di essa.

## 3.1 ECS, ECR, EC2

Per il deployment dell'applicazione, è stato scelto Amazon ECS (Elastic Container Service), che sfrutta a sua volta i servizi EC2 ed ECR (Elastic Container Registry). Questa scelta è stata motivata dall'esigenza di avere un controllo più granulare sull'infrastruttura sottostante (es. configurazione delle istanze EC2 e dei load balancer, gestione security group ecc...), oltre che dalla necessità di eseguire due container separati (webapp e ollama) con differenti requisiti di risorse.

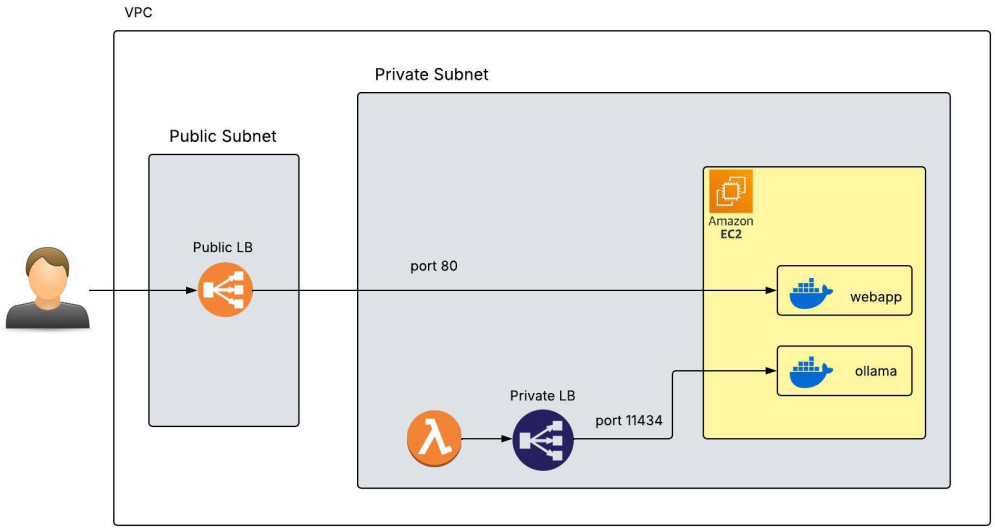


**Figura 3.1:** Architettura di rete

Il processo di provisioning è stato completamente automatizzato tramite script Python con la libreria boto3, e prevede i seguenti passaggi principali:

– Creazione di due load balancer: uno pubblico per rendere accessibile la webapp sulla porta 80, e uno privato dedicato alla comunicazione interna con il servizio ollama che ascolta sulla porta 11434.

```
        lb = elbv2.create_load_balancer(
            Name="ecs-lb",
            Subnets=[SUBNET_PUB_ID, SUBNET_PUB_2_ID],
            SecurityGroups=[LB_PUB_SG],
            Scheme="internet-facing",
            Type="application",
            IpAddressType="ipv4"
        )

        lb_private = elbv2.create_load_balancer(
            Name="ecs-lb-private",
            Subnets=[SUBNET_PRV_ID, SUBNET_PRV_2_ID],
            SecurityGroups=[LB_PRV_SG],
            Scheme="internal",
            Type="application",
            IpAddressType="ipv4",
        )
```

– Configurazione dei Target Group e Listeners che permettono ai load balancer di ascoltare le richieste in ingresso su una determinata porta e inoltrarle ai container corretti dentro l'istanza EC2 in cui sono eseguiti.

– Creazione di un cluster ECS e di un Auto Scaling Group basato su uno specifico launch template EC2, dove vengono specificate l'immagine da utilizzare nell'istanza e il tipo di istanza da lanciare (nel nostro caso una t3.large vista la quantità di risorse necessarie per eseguire il container di ollama).

```
        lt = ec2.create_launch_template(
            LaunchTemplateName="ecs-lt",
            LaunchTemplateData={
                "ImageId": AMI_ID,
                "InstanceType": INSTANCE_TYPE,
                "KeyName": KEY_NAME,
                "IamInstanceProfile": {"Name": INSTANCE_PROFILE_NAME},
                "SecurityGroupIds": [EC2_SG],
                "UserData": base64.b64encode(user_data_script.encode("
                    utf-8")).decode("utf-8")
            }
        )

        autoscaling.create_auto_scaling_group(
            AutoScalingGroupName="ecs-asg",
            LaunchTemplate={"LaunchTemplateName": "ecs-lt"},
            MinSize=1,
            MaxSize=3,
            DesiredCapacity=1,
            VPCZoneIdentifier=SUBNET_PRV_ID,
            TargetGroupARNs=[tg_arn, tg_private_ollama_arn],  #
                Associa entrambi i target group all'ASG
            Tags=[{"Key": "Name", "Value": "ECS Instance", "
                PropagateAtLaunch": True}]
        )

        ecs.create_cluster(clusterName=ECS_CLUSTER_NAME)
```

– Registrazione di 2 Task Definition, uno per il container webapp e uno per ollama, entrambi con configurazioni personalizzate di memoria dedicata.

```
        ecs.register_task_definition(
            family="webapp-task",
            networkMode="bridge",
            requiresCompatibilities=["EC2"],
            cpu="256",
            memory="512",
            executionRoleArn=EXECUTION_ROLE_ARN,
            containerDefinitions=[
                {
                    "name": "webapp",
                    "image": WEBAPP_IMAGE,
                    "memory": 512,
                    "essential": True,
                    "portMappings": [
                        {"containerPort": 5000, "hostPort": 80}
                    ]
                }
```

```
            ]
        )

        ecs.register_task_definition(
            family="ollama-task",
            networkMode="bridge",
            requiresCompatibilities=["EC2"],
            cpu="1024",
            memory="7168",
            executionRoleArn=EXECUTION_ROLE_ARN,
            containerDefinitions=[
                {
                    "name": "ollama",
                    "image": OLLAMA_IMAGE,
                    "memory": 7168,
                    "essential": True,
                    "portMappings": [
                        {"containerPort": 11434, "hostPort": 11434}
                    ],
                    "mountPoints": [
                        {
                            "sourceVolume": "ollama-data",
                            "containerPath": "/root/.ollama",
                            "readOnly": False
                        }
                    ]
                }
            ],
            volumes=[
                {
                    "name": "ollama-data",
                    "host": {
                        "sourcePath": "/var/app/data/ollama"
                    }
                }
            ]
        )
```

– Deploy dei 2 ECS Service distinti, ciascuno collegato al rispettivo Task e target group del load balancer.

```
        ecs.create_service(
            cluster=ECS_CLUSTER_NAME,
            serviceName="webapp-service",
            taskDefinition="webapp-task",
            desiredCount=1,
            launchType="EC2",
            loadBalancers=[
                {
                    "targetGroupArn": tg_arn,
                    "containerName": "webapp",
                    "containerPort": 5000
                }
            ],
            deploymentConfiguration={"maximumPercent": 100, "
                minimumHealthyPercent": 0}
        )

        ecs.create_service(
```

```
            cluster=ECS_CLUSTER_NAME,
            serviceName="ollama-service",
            taskDefinition="ollama-task",
            desiredCount=1,
            launchType="EC2",
            loadBalancers=[
                {
                    "targetGroupArn": tg_private_ollama_arn,
                    "containerName": "ollama",
                    "containerPort": 11434
                }
            ],
            deployment Configuration={"maximumPercent": 100, "
                minimumHealthyPercent": 0}
        )
```

**ECR** ospita le immagini Docker dei due container dell'applicazione:

– **Immagine webapp**: contenente il front-end dell'applicazione e le chiamate alle varie API. Tramite Flask il server viene avviato e si mette in ascolto sulla porta 5000, nel Task Definition invece viene definita la mappatura delle porte tra host (istanza EC2 che esegue il container) e container come segue:

```
    {
      "portMappings": [
        {
          "hostPort": 80,
          "containerPort": 5000
        }
      ],
    }
```

Nel nostro caso il container espone la porta interna 5000 mentre l'istanza EC2 espone la 80. Il load balancer pubblico indirizza le richieste in ingresso sulla porta 80 verso l'istanza EC2, la quale le inoltra alla porta 5000 del container.

– **Immagine ollama**: contenente il servizio ollama in ascolto sulla porta 11434, che riceve le richieste (ovvero i dati forniti dall'utente nel form), interroga localmente il modello gemma:3-1b, e restituisce l'itinerario generato.

## 3.2 DynamoDB

Per la gestione dei dati dinamici e persistenti legati alle richieste degli utenti e alla generazione della classifica delle città più cercate, è stato utilizzato il servizio DynamoDB, un database NoSQL completamente gestito.

Nel progetto sono state definite due tabelle principali:

- **CityStats**: contiene, per ogni città (utilizzata come partition key), un contatore che rappresenta il numero di volte che quella città è stata cercata dagli utenti. I dati contenuti in questa tabella vengono aggiornati da una funzione Lambda ogni volta che l'utente invia una nuova richiesta. Un'altra funzione Lambda legge i dati da questa tabella per estrarre le cinque città più cercate, che verranno poi mostrate nella pagina web principale.
- **IntineraryRequests**: contiene le richieste di itinerario effettuate dagli utenti. La partition key è il campo `request_id`, un identificativo univoco generato per ogni richiesta, poi altri campi sono i dati forniti dall'utente riguardo città e numero di giorni di permanenza, un campo `status` che identifica lo stato della richiesta (Pending, Completed o Failed) e infine il campo risposta che conterrà l'itinerario generato (quando lo status è Completed) oppure un codice di errore (in caso di Failed).

Il campo status risulta fondamentale per gestire in modo asincrono il ciclo di vita di una richiesta. Quando una richiesta viene inviata, viene inserita nella tabella con stato Pending. La gestione dell'elaborazione asincrona avviene tramite una coda SQS a cui è attaccata una funzione Lambda incaricata di processare le richieste man mano che giungono nella coda.

## 3.3 API Gateway

Per facilitare la comunicazione tra l'interfaccia utente dell'applicazione e la logica di back-end, è stato configurata un'API REST (TravelAppAPI) utilizzando il servizio API Gateway. Questo servizio consente di esporre diverse risorse e metodi HTTP, ognuno collegato a una specifica funzione Lambda, nel nostro caso specifico, che esegue l'elaborazione richiesta.

Attraverso API Gateway sono state create diverse route:

- **/classifica**: espone un metodo GET per prelevare dalla tabella CityStats le prime 5 città più cercate, e un metodo POST per aggiornare tale tabella dopo ogni richiesta fatta dall'utente.

- **/richiesta_itinerario**: espone un metodo POST che conserva la richiesta effettuata dall'utente all'interno della tabella ItineraryRequests marcandola come "Pending", i dati della richiesta inoltre vengono anche inseriti all'interno della coda SQS in attesa di essere consumati.

- **/risultato_itinerario**: espone un metodo GET per recuperare dalla tabella ItineraryRequests le risposte fornite da ollama solo nel caso in cui lo stato delle richieste è "COMPLETED" o "FAILED". Viene eseguito un poll periodico di questa funzione perchè ollama impiega qualche minuto per elaborare la richiesta e restituire una risposta.

Inoltre, è stata abilitata per ogni route la gestione del Cross-Origin Resource Sharing (CORS), fondamentale per permettere alle applicazioni web di effettuare chiamate API da domini differenti senza blocchi di sicurezza. Il servizio API Gateway si occupa anche di gestire i permessi di invocazione, garantendo che solo le richieste autorizzate possano raggiungere le funzioni Lambda.

Questa architettura ha permesso di mantenere separata l'interfaccia di accesso dalla logica di back-end, facilitando la scalabilità e la manutenzione dell'applicazione, oltre a garantire una comunicazione efficiente e sicura tra i vari componenti.

## 3.4 SQS

Per garantire affidabilità nell'elaborazione si è deciso di gestire le richieste degli utenti in modo asincrono tramite l'utilizzo del servizio SQS (Simple Queue Service). Questo il workflow implementato nell'applicazione:

- Quando l'utente clicca sul pulsante "Genera Itinerario", i dati inseriti vengono inviati tramite una funzione lambda (collegata all'API Gateway) alla coda SQS.

- Ogni nuovo messaggio inserito in coda triggera una seconda funzione lambda, la quale costruisce il prompt da inviare a ollama, attende la risposta (anche in caso di errore) e la salva nella tabella ItineraryRequests di DynamoDB, oltre che inviarla in un topic SNS per notificare l'utente via e-mail.

Una soluzione sincrona, con una lambda collegata direttamente all'API gateway per inviare le richieste a ollama, è stata scartata a causa dei vincoli temporali imposti da API Gateway, infatti quest'ultimo ha un timeout massimo di 30 secondi circa, mentre ollama con le risorse a disposizione impiega

tra 2 e 3 minuti per generare una risposta, superando ampiamente tale limite, quindi questo avrebbe comportato un fallimento sistematico delle richieste compromettendo l'esperienza d'uso dell'utente.

## 3.5 SNS

Nel progetto, Amazon SNS (Simple Notification Service) viene utilizzato per notificare l'utente via email non appena la generazione dell'itinerario è stata completata. Quando un utente compila il form sul front-end e invia una richiesta, una funzione Lambda si occupa della gestione dei dati inseriti: oltre a inviare un messaggio alla coda SQS per l'elaborazione asincrona della richiesta, la Lambda effettua una sottoscrizione ad un topic SNS, utilizzando l'indirizzo e-mail fornito dall'utente. Per ricevere le notifiche, l'utente dovrà confermare la sottoscrizione tramite l'email automatica inviata da SNS.

Una volta che la richiesta viene processata da ollama e viene generata una risposta, un'altra funzione Lambda, oltre ad aggiornare la tabella ItineraryRequest, pubblicherà un messaggio nel topic SNS, contenente la risposta con l'itinerario, che verrà quindi recapitato a tutti i destinatari iscritti al topic, notificando così direttamente via e-mail l'utente che aveva effettuato la richiesta.

## 3.6 Lambda

Nel contesto di un'architettura serverless, AWS Lambda rappresenta uno dei servizi fondamentali offerti da Amazon Web Services per eseguire codice in risposta a eventi, senza la necessità di gestire o configurare server. Le funzioni Lambda vengono eseguite solo quando vengono invocate, riducendo i costi e migliorando la scalabilità dell'applicazione.

Nel progetto sviluppato, tali funzioni sono state progettate per reagire a trigger specifici, come chiamate HTTP tramite API Gateway o messaggi pubblicati in code SQS. Questa scelta ha permesso di implementare un flusso di elaborazione distribuito, scalabile e facilmente manutenibile, in linea con i principi delle architetture cloud-native.

Ogni funzione Lambda è stata isolata per svolgere un compito ben definito, facilitando il riutilizzo, la tracciabilità e il monitoraggio tramite strumenti come CloudWatch.

Nel seguito di questo paragrafo, si descrivono nel dettaglio le 5 funzioni Lambda implementate, evidenziando per ciascuna il ruolo svolto, le risorse AWS collegate e il meccanismo di attivazione.

### 3.6.1 LAMBDA 1: update_database

La funzione update_database ha il compito di aggiornare un contatore associato a una città all'interno della tabella DynamoDB CityStats, o eventualmente inserire quella città all'interno della tabella se è la prima volta che viene cercata. Per ogni città, il contatore rappresenta il numero totale di volte che quella città è stata cercata nell'applicazione.

```
    table.update_item(
        Key={'city': citta},
        UpdateExpression='SET #cnt = if_not_exists(#cnt, :zero) + :uno',
        ExpressionAttributeNames={
            '#cnt': 'count'  # alias per l'attributo count
        },
        ExpressionAttributeValues={
            ':uno': 1,
            ':zero': 0
        }
    )
```

La funzione è invocata tramite API Gateway, in risposta a una richiesta HTTP POST (verso la route /classifica) generata quando l'utente preme il pulsante "Genera Itinerario" nel front-end dell'applicazione.

### 3.6.2 LAMBDA 2: get_top_list

La funzione get_top_list ha il compito di recuperare le 5 città più ricercate all'interno dell'applicazione, consultando i valori memorizzati nella tabella DynamoDB CityStats. L'ordinamento è basato sul campo count, che rappresenta il numero di richieste registrate per ogni città. Viene quindi eseguito uno scan sull'intera tabella CityStats, applicato un ordinamento decrescente delle città in base al campo count e selezionati solo i primi 5 risultati.

```
    # Scan tutta la tabella
    response = table.scan()
    items = response.get('Items', [])

    # Ordina per conteggio decrescente
    items.sort(key=lambda x: x.get('count', 0), reverse=True)

    # Prendi top 5
    top_5 = items[:5]
```

Questa funzione è invocata da API Gateway in seguito a una richiesta HTTP GET indirizzata alla route /classifica. Viene richiamata nel momento in cui l'utente accede alla pagina principale dell'applicazione, così che possa vedere da subito le città più popolari, e ogni volta che viene effettuata una nuova richiesta di generazione dell'itinerario premendo il pulsante "Genera Itinerario".

### 3.6.3 LAMBDA 3: request_itinerary

La funzione request_itinerary rappresenta il punto di ingresso principale per la generazione asincrona degli itinerari. Dopo che l'utente ha compilato il form dell'applicazione indicando e-mail, città e numero di giorni di permanenza, questa Lambda:

- Registra la richiesta in una tabella DynamoDB (ItineraryRequests) assegnando un request_id univoco e marcando lo stato come "PENDING".

```
        table.put_item(Item={
            'request_id': request_id,
            'status': 'pending',
            'timestamp': int(time.time()),
            'citta': citta,
            'giorni': giorni
        })
```

- Invia un messaggio con le stesse informazioni a una coda SQS, il quale va a triggerare un'ulteriore funzione lambda che consuma il messaggio appena inserito.

```
        sqs.send_message(
            QueueUrl=SQS_URL,
            MessageBody=json.dumps({
                'request_id': request_id,
                'citta': citta,
                'giorni': giorni
            })
        )
```

- Effettua una sottoscrizione al topic SNS utilizzando l'indirizzo email fornito dall'utente.

```
        sns.subscribe(
            TopicArn=arn_topic,
            Protocol='email',
```

```
            Endpoint=email
        )
```

Viene invocata tramite API Gateway in seguito a una richiesta HTTP POST verso la route /richiesta-itinerario.

### 3.6.4 LAMBDA 4: process_itinerary

process_itinerary è una funzione Lambda che viene triggerata automaticamente ogni volta che un nuovo messaggio viene inserito nella coda SQS. Il suo obiettivo è elaborare la richiesta di itinerario presente nel messaggio tramite i seguenti passaggi:

– Genera un prompt personalizzato in base alla città e al numero di giorni specificati dall'utente ed effettua una chiamata POST al servizio ollama per generare l'itinerario, passando il prompt appena creato. In base all'esito della richiesta viene definito un attributo status che può essere "COMPLETED" o "FAILED".

```
        try:
            # Costruzione del prompt per Ollama
            prompt = f"Crea un itinerario di {giorni} giorni a {citta
                }, in italiano."

            # Chiamata al servizio Ollama per generare l'itinerario
            response = requests.post(OLLAMA_URL, json={
                "model": "gemma3:1b",
                "prompt": prompt,
                "stream": False
            }, timeout=600) # Timeout di 600 secondi, ollama impiega
                qualche minuto per rispondere

            # Controllo della risposta
            response.raise_for_status()
            risposta = response.json().get("response", "Nessuna
                risposta ricevuta.")
            status = 'completed'
            print(f"Stato della risposta: {status}")
        except Exception as ollama_error:
            print(" Errore Ollama:", str(ollama_error))
            risposta = f"Errore Ollama: {str(ollama_error)}"
            status = 'failed'
```

– Aggiorna la tabella DynamoDB ItineraryRequests, in base al request_id della richiesta, salvando la risposta ricevuta e modificando lo stato da "PENDING" a "COMPLETED" se tutto va bene o "FAILED" in caso di errore.

```
        table.update_item(
            Key={'request_id': request_id},
            UpdateExpression='SET #s = :s, risposta = :r',
            ExpressionAttributeNames={'#s': 'status'},
            ExpressionAttributeValues={
                ':s': status,
                ':r': risposta
            },
            ConditionExpression='attribute_exists(request_id)'
        )
```

– Pubblica un messaggio nel topic SNS, contenente l'itinerario generato da ollama, per notificare l'utente via e-mail.

```
        subject = "Il tuo itinerario e' pronto!" if status == "
            completed" else "Errore nella generazione"
        sns.publish(
            TopicArn=arn_topic,
            Subject=subject,
            Message=risposta
        )
```

### 3.6.5 LAMBDA 5: result_itinerary

La funzione result_itinerary permette di recuperare tutti i dati associati a una richiesta di itinerario, identificata tramite uno specifico request_id, precedentemente inserita nella tabella DynamoDB ItineraryRequests.

```
    response = table.get_item(Key={'request_id': request_id})
    item = response.get('Item')
```

È pensata per essere invocata lato front-end, dopo che l'utente ha effettuato una richiesta, con un polling periodico, per verificare se l'elaborazione è completata ed ottenere eventualmente la risposta generata da ollama.

La funzione viene invocata tramite API Gateway in seguito a una richiesta HTTP GET, verso la route /risultato-itinerario, con il parametro requestId passato nella query string. Dal lato client, viene eseguito un polling con setInterval() ogni 5 secondi, analizzando il campo status della risposta per determinare quando fermare il polling e mostrare il risultato.

```
    const intervalId = setInterval(async () => {
        try {
            const resultResponse = await fetch("${apiUrl}/risultato-
                itinerario?requestId=${requestId}");
            if (!resultResponse.ok) throw new Error("Errore nel recupero
                risultato");
```

```
5
6             const resultData = await resultResponse.json();
7
8             if (resultData.status === "completed") {
9                 clearInterval(intervalId);
10                rispostaLlm.textContent = resultData.risposta || "Nessuna
                      risposta.";
11                risultato.style.display = "block";
12                loading.style.display = "none";
13            } else if (resultData.status === "failed") {
14                clearInterval(intervalId);
15                rispostaLlm.textContent = "Errore nella generazione dell'
                      itinerario.";
16                risultato.style.display = "block";
17                loading.style.display = "none";
18            }
19        } catch (err) {
20            clearInterval(intervalId);
21            rispostaLlm.textContent = "Errore durante il polling: " + err.
                  message;
22            risultato.style.display = "block";
23            loading.style.display = "none";
24         }
25    }, 5000);
```

## 3.7 Altri servizi AWS utilizzati

Oltre ai servizi principali descritti nei capitoli precedenti, durante la progettazione e lo sviluppo dell'applicazione sono stati utilizzati altri servizi AWS fondamentali per la gestione dell'infrastruttura, del codice e del monitoraggio:

- **S3**: è stato utilizzato per la conservazione dei pacchetti ZIP contenenti il codice delle funzioni Lambda.
- **CloudWatch**: particolarmente utile per il monitoraggio dei log generati dalle funzioni lambda, in particolare per verificare il corretto inserimento dei dati nelle tabelle DynamoDB, tracciare eventuali errori nelle chiamate all'istanza ollama, diagnosticare eventuali fallimenti nei messaggi ricevuti nella coda SQS.
- **VPC (Virtual Private Cloud)**: è stato utilizzato per la creazione e la configurazione dell'infrastruttura di rete su cui si basa l'applicazione, in particolare per la definizione di una VPC dedicata, delle subnet pubbliche e private, dei vari security group personalizzati e di un NAT Gateway fondamentale per permettere alle istanze EC2 di agganciarsi al cluster ECS creato (le istanze EC2 vengono infatti create all'interno di una subnet privata).

# Capitolo 4

# CI/CD: GitHub Actions

L'obiettivo principale della pipeline CI/CD implementata tramite GitHub Actions è quello di automatizzare il processo di build, push e deploy delle immagini dei container webapp e ollama su Amazon ECS, ma solo nel caso in cui queste siano state effettivamente modificate. Questo approccio selettivo consente di ridurre i tempi di deploy e ottimizzare l'uso delle risorse.

La pipeline è configurata per attivarsi ad ogni push sul branch main ed esegue innanzitutto un controllo per capire se sono stati modificati file all'interno delle cartelle `webapp` o `ollama`, il tutto facendo un confronto tra il commit attuale (HEAD) con quello precedente (HEADˆ).

```
1     WEBAPP_DIFF=$(git diff --name-only HEADˆ -- webapp/ | wc -l)
2
3     if [ "$WEBAPP_DIFF" -gt 0 ]; then
4         echo "WEBAPP_CHANGED=true" >> $GITHUB_ENV
5     fi
```

Solo se WEBAPP_CHANGED o OLLAMA_CHANGED sono impostate a true, viene eseguita la build e il push delle nuove immagini docker nei rispettivi repository ECR. In questo modo si evita di buildare e deployare le immagini dei container che non hanno subito modifiche.

```
1     if: env.WEBAPP_CHANGED == 'true'
2     run: |
3         echo "Building and pushing webapp image"
4         docker build -t $ECR_WEBAPP:latest ./webapp
5         docker push $ECR_WEBAPP:latest
```

Per fare in modo che, in seguito ad una modifica nel repository GitHub, vengano aggiornati solo i container modificati, la pipeline sfrutta alcuni co-

mandi che permettono di interagire dinamicamente con l'infrastruttura ECS. La pipeline infatti scarica le definizioni attuali dei 2 task ECS (che conterranno le configurazioni dei container, incluse le immagini, la quantità di memoria riservata per ciascuno ecc.). La task definition viene poi modificata per aggiornare il riferimento all'immagine Docker, puntando dunque alla nuova immagine pushata precedentemente nel repository ECR.

```
1    aws ecs describe-task-definition --task-definition $TASK_FAMILY_WEBAPP |
         jq '.taskDefinition' > taskdef-webapp.json
2
3    jq --arg IMAGE "$ECR_WEBAPP:latest" '.containerDefinitions |= map(if .
         name=="webapp" then .image=$IMAGE else . end)' taskdef-webapp.json >
         taskdef-updated.json
```

Quindi dopo aver aggiornato la task definition, la pipeline registra una nuova revisione del task su ECS. Successivamente viene aggiornato il servizio ECS affichè utilizzi la nuova versione del task appena registrato e viene attesa la fine del processo di aggiornamento da parte di ECS.

```
1    aws ecs register-task-definition --cli-input-json file://new-taskdef-
         webapp.json
2    aws ecs update-service --cluster $ECS_CLUSTER --service
         $ECS_SERVICE_WEBAPP --task-definition $NEW_TASK_DEF_ARN
3    aws ecs wait services-stable --cluster $ECS_CLUSTER --services
         $ECS_SERVICE_WEBAPP
```

L'approccio descritto rispetta l'obiettivo posto inizialmente e quindi consente di implementare una strategia di Continuous Deployment selettiva, in cui l'aggiornamento dei servizi ECS avviene in modo mirato esclusivamente per i container effettivamente modificati. Questo tipo di automazione risulta essere particolarmente efficiente e scalabile per ambienti containerizzati con più microservizi o componenti indipendenti.

# Capitolo 5

# Conclusioni

Nel complesso, il progetto ha raggiunto pienamente gli obiettivi prefissati, permettendo di consolidare e approfondire la conoscenza della maggior parte dei servizi offerti da AWS e trattati durante il corso. L'intero processo di sviluppo e deployment ha rappresentato un'occasione concreta per mettere in pratica concetti teorici e affrontare problematiche reali legate al provisioning, alla scalabilità e alla gestione dei microservizi in ambiente cloud.

Nonostante i risultati positivi, è importante sottolineare che alcune funzionalità aggiuntive e ottimizzazioni non sono state implementate a causa delle limitazioni imposte dall'ambiente AWS Academy. Ad esempio, l'utilizzo di istanze EC2 più performanti per l'esecuzione dei container avrebbe potuto migliorare sensibilmente i tempi di risposta e la stabilità del sistema rispetto alla t3.large disponibile. Inoltre, per la gestione automatizzata dell'invio delle email, l'integrazione con il servizio Amazon SES (Simple Email Service) avrebbe potuto rappresentare una soluzione più efficiente e professionale rispetto ad SNS, ma anch'essa non è risultata accessibile nel contesto accademico.

Queste limitazioni non hanno tuttavia compromesso l'efficacia dell'esperienza: al contrario, hanno contribuito a sviluppare una maggiore consapevolezza nella scelta delle soluzioni cloud in base alle risorse disponibili, rafforzando le competenze progettuali e operative in un contesto realistico.