

# Contents

<b>1 Architettura Generale</b>	<b>2</b>
<b>2 Frontend</b>	<b>2</b>
2.1 Funzionalità del Frontend . . . . .	3
2.2 Some Screenshot . . . . .	4
<b>3 Backend</b>	<b>4</b>
3.1 Descrizione delle API Backend . . . . .	5
3.1.1 /images . . . . .	5
3.1.2 /albums . . . . .	6
3.1.3 /upload . . . . .	6
3.1.4 /auth . . . . .	6
3.2 DockerFile . . . . .	6
<b>4 Amazon CloudFront</b>	<b>6</b>
4.1 Architettura e configurazione . . . . .	6
<b>5 Load Balancer</b>	<b>7</b>
<b>6 Elastic Container Service (ECS)</b>	<b>7</b>
6.1 Architettura e configurazione . . . . .	7
6.2 Task Definition . . . . .	8
<b>7 Autoscaling del Cluster ECS</b>	<b>9</b>
7.1 Funzionamento della Policy di Autoscaling . . . . .	9
<b>8 Lambda Function e Analisi Immagini</b>	<b>10</b>
<b>9 Amazon S3</b>	<b>11</b>
<b>10 Database</b>	<b>12</b>
<b>11 CI/CD</b>	<b>12</b>
<b>12 Infrastruttura AWS con Terraform</b>	<b>14</b>
<b>13 Conclusioni</b>	<b>14</b>
13.1 Riepilogo dei Servizi AWS utilizzati . . . . .	15

# CloudyGallery: Gestione di Immagini su AWS

Messina Giacomo Giovanni

July 23, 2025

## Introduzione

CloudyGallery è una piattaforma cloud per la gestione intelligente di immagini. Il sistema consente di caricare, organizzare, ricercare e analizzare immagini tramite una dashboard web, integrando funzionalità come il riconoscimento automatico dei tag tramite Amazon Rekognition, la gestione di album personalizzati e la visualizzazione di statistiche interattive.

L'architettura si basa su un frontend React ospitato su S3, un backend FastAPI containerizzato e deployato su ECS, e una funzione Lambda per l'analisi automatica delle immagini. L'infrastruttura è completamente automatizzata tramite Terraform e il ciclo di vita applicativo è gestito da pipeline CI/CD su GitHub Actions.

## 1 Architettura Generale

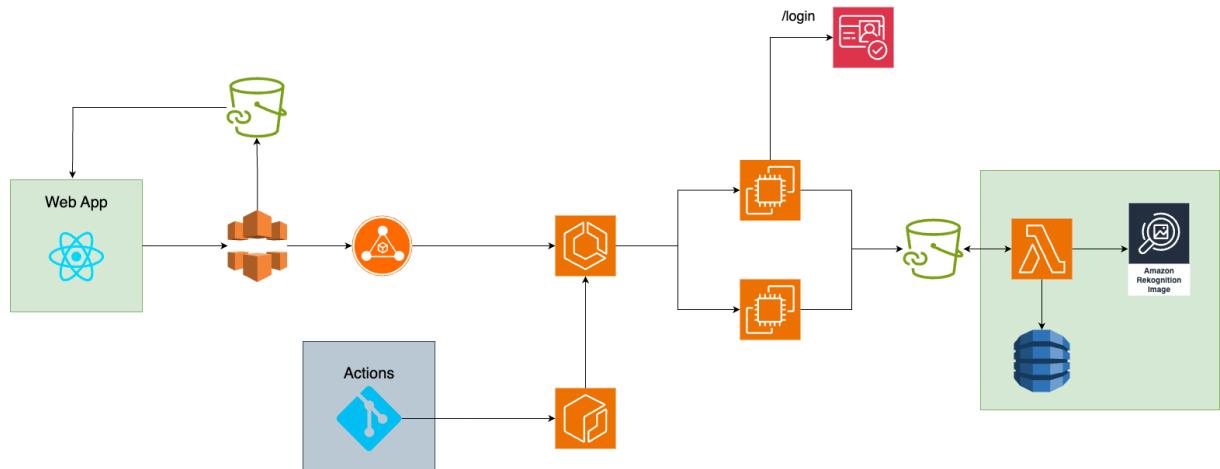


Figure 1: Pipeline e architettura del progetto

## 2 Frontend

Il frontend è sviluppato in React e ospitato come sito statico su un bucket **S3**. Offre una dashboard utente che consente di:

- Caricare nuove immagini tramite drag&drop multiplo.
- Visualizzare le immagini esistenti, raggruppate per tag.
- Effettuare ricerche e filtrare immagini in base ai tag generati da **Amazon Rekognition**.
- Visualizzare statistiche sui tag tramite grafici interattivi (Chart.js).
- Gestire l'autenticazione tramite Amazon Cognito (login, registrazione, conferma account, callback OAuth2).
- Navigare tra dashboard, statistiche e gestione album.

## 2.1 Funzionalità del Frontend

Le principali funzionalità sono:

- **Autenticazione:** Login, registrazione, conferma account e callback OAuth2 tramite Amazon Cognito.
- **Upload immagini:** Caricamento di una o più immagini tramite drag&drop, con anteprima, barra di avanzamento e inserimento di tag personalizzati.
- **Visualizzazione immagini:** Dashboard che mostra tutte le immagini dell'utente, con anteprima, metadati, tag e possibilità di ricerca e filtro.
- **Gestione album:** Creazione, visualizzazione, eliminazione e navigazione tra album personali. Possibilità di spostare immagini tra album.
- **Eliminazione immagini:** Rimozione di immagini con animazione e feedback visivo.
- **Ricerca e filtro:** Ricerca rapida delle immagini per nome, tag o album.
- **Visualizzazione tag:** Mostra i tag generati automaticamente da Amazon Rekognition e quelli inseriti manualmente.
- **Statistiche e grafici:** Pagina dedicata alle statistiche sui tag e sulle immagini, con grafici interattivi (Chart.js).
- **Download:** Download diretto delle immagini e copia del link pubblico.
- **Logout e gestione sessione:** Logout sicuro e gestione automatica della sessione utente.

## 2.2 Some Screenshot

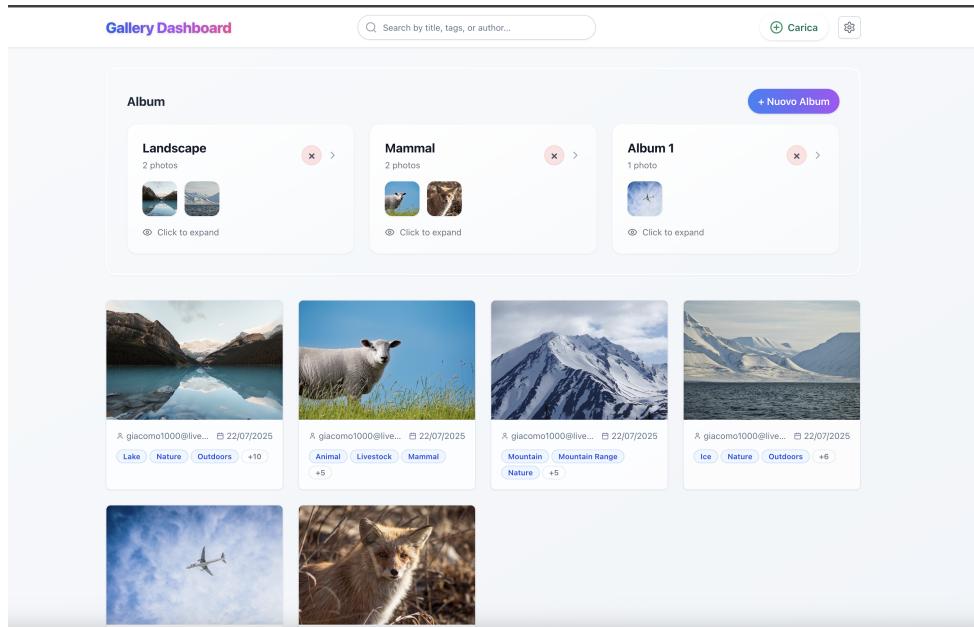


Figure 2: Dashboard

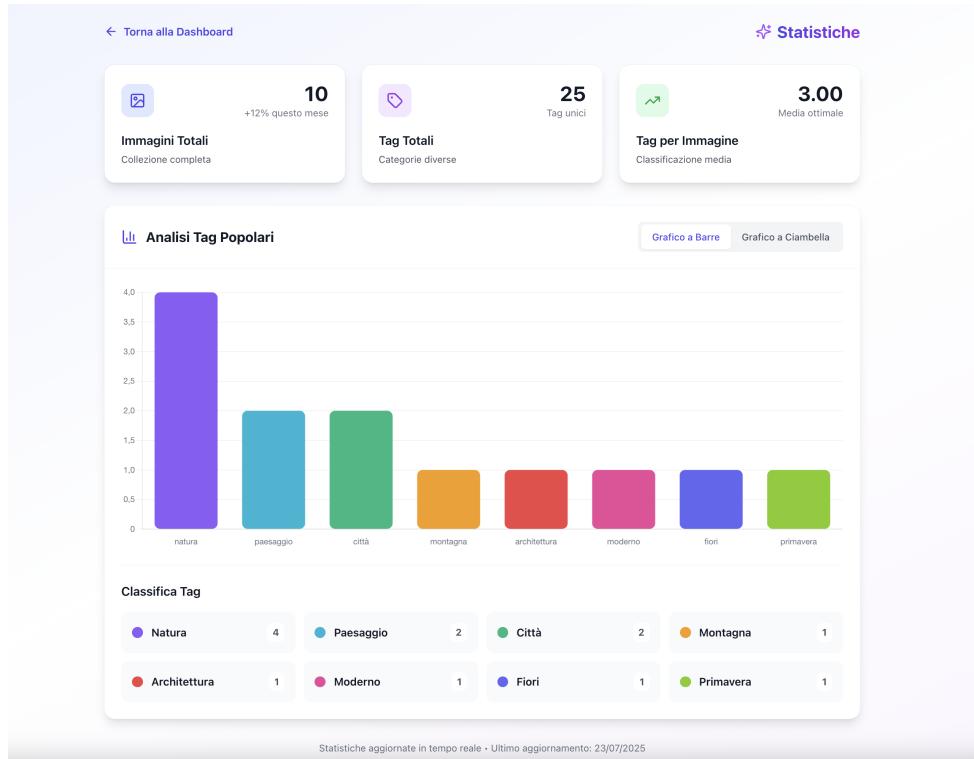


Figure 3: Stats

## 3 Backend

Il backend è sviluppato in **Python** con **FastAPI**, containerizzato tramite **Docker** e deployato su un cluster **ECS EC2** con supporto per il Service Auto Scaling.

Le principali funzionalità offerte dalle API sono:

- Gestione degli utenti tramite autenticazione federata con **Amazon Cognito**.
- Upload delle immagini su **S3** e gestione dei metadati.
- Ricerca delle immagini per tag, ottenuti tramite analisi automatica.
- Gestione degli album e delle statistiche utente.

Il backend espone le API organizzate principalmente sotto due prefissi:

- **/auth**: endpoint per l'autenticazione e la gestione degli utenti.
- **/api**: endpoint per funzionalità applicative come upload, gestione immagini e album.

Di seguito un estratto dal file principale `main.py` che mostra la struttura di routing:

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from routes.upload import router as upload_router
from routes.images import router as images_router
from routes.auth import router as auth_router
from routes.album import router as album_router

app = FastAPI(title="AWS Backend API", version="1.0.0")

app.include_router(auth_router, prefix="/auth", tags=["authentication"])
app.include_router(upload_router, prefix="/api", tags=["upload"])
app.include_router(images_router, prefix="/api", tags=["images"])
app.include_router(album_router, prefix="/api", tags=["albums"])

@app.get("/api/health")
async def health_check():
    return {"status": "healthy"}
```

### 3.1 Descrizione delle API Backend

Di seguito una panoramica delle principali API REST implementate nel backend FastAPI per la gestione di immagini, album e autenticazione.

#### 3.1.1 /images

- **GET /images/{user\_id}**: Restituisce tutte le immagini dell'utente, con metadati, tag e raggruppamento per album.
- **DELETE /images/{user\_id}/{filename}**: Elimina una specifica immagine dell'utente dal bucket S3.
- **POST /images/move**: Sposta un'immagine da un album a un altro per l'utente specificato.
- **GET /tags/{user\_id}**: Restituisce tutti i tag associati alle immagini dell'utente, con conteggio di frequenza.

### 3.1.2 /albums

- **GET /albums/{user\_id}**: Elenca tutti gli album (cartelle) dell'utente presenti su S3.
- **POST /albums**: Crea un nuovo album (cartella) per l'utente su S3.
- **DELETE /albums/{album\_name}/{user\_id}**: Elimina un album e tutte le immagini contenute per l'utente.

### 3.1.3 /upload

- **POST /upload**: Permette l'upload di una nuova immagine su S3, con metadati opzionali (nome, tag, userId). Restituisce l'URL pubblico dell'immagine.
- **DELETE /delete/{filename}**: Elimina una immagine dal bucket S3 dato il nome file.

### 3.1.4 /auth

- **POST /exchange-code**: Scambia un authorization code Cognito per i token di accesso (OAuth2 flow).

## 3.2 DockerFile

Infine, il seguente Dockerfile definisce il processo di build e avvio dell'applicazione:

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY src/
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

## 4 Amazon CloudFront

Amazon CloudFront è un servizio di Content Delivery Network (CDN) che distribuisce contenuti a bassa latenza e alta velocità in tutto il mondo. Nel progetto, CloudFront è utilizzato per distribuire sia il frontend che il backend attraverso un'unica distribuzione, sfruttando origini multiple.

### 4.1 Architettura e configurazione

La distribuzione CloudFront gestisce due origini principali:

- **Origine Frontend**: Il frontend statico è ospitato in un bucket Amazon S3 configurato per il web hosting. CloudFront distribuisce i file statici (HTML, CSS, JavaScript, immagini) direttamente dal bucket, garantendo un rapido caricamento e caching a livello globale.

- **Origine Backend:** Le richieste API verso il backend FastAPI sono instradate tramite CloudFront verso un endpoint distinto, utilizzando un path prefix dedicato `/api`. Questo permette di esporre il backend attraverso lo stesso dominio CloudFront usato per il frontend, mantenendo una gestione centralizzata e semplificando la configurazione CORS e di sicurezza.

## 5 Load Balancer

Nell’architettura AWS utilizzata, il **Load Balancer** svolge un ruolo cruciale nel gestire e distribuire il traffico in ingresso verso il backend.

Il Load Balancer è posizionato *a valle* di Amazon CloudFront, ovvero:

- Le richieste esterne arrivano innanzitutto su CloudFront, che funge da Content Delivery Network (CDN) e punto di ingresso globale.
- CloudFront indirizza il traffico verso due origini principali: il frontend (contenuti statici) e il backend (API).
- Le richieste al backend passano poi attraverso il Load Balancer, che distribuisce il traffico tra le istanze ECS in esecuzione, garantendo alta disponibilità e scalabilità.

Questa configurazione consente di:

- Migliorare le prestazioni grazie alla cache globale di CloudFront.
- Bilanciare il carico di lavoro tra le risorse backend.
- Isolare e gestire in modo efficiente la scalabilità delle applicazioni backend.

## 6 Elastic Container Service (ECS)

Amazon Elastic Container Service (ECS) è un servizio di orchestrazione di container altamente scalabile e ad alte prestazioni che supporta container Docker. Nell’infrastruttura del progetto, ECS viene utilizzato per eseguire e gestire il backend containerizzato in ambiente EC2.

### 6.1 Architettura e configurazione

Il backend è distribuito tramite ECS con le seguenti caratteristiche principali:

- **Cluster ECS basato su EC2:** il servizio ECS utilizza istanze EC2 che formano un cluster nel quale vengono eseguiti i container. Questo permette un controllo più diretto sulle risorse di calcolo.
- **Task Definition:** La definizione della task specifica:
  - L’immagine Docker del backend, ospitata su Amazon Elastic Container Registry (ECR).
  - Le risorse allocate al container, come CPU e memoria.

- Il mapping delle porte necessarie per esporre l'applicazione (porta 8000).
  - La configurazione del logging, che utilizza CloudWatch Logs per il monitoraggio centralizzato.
  - L'inclusione di variabili ambiente caricate da un file presente in un bucket S3 dedicato.
- **Networking:** Il cluster è contenuto all'interno di una VPC dedicata con subnet pubbliche configurate per consentire l'accesso esterno ai container, tramite il mapping delle porte configurato nella task definition.

## 6.2 Task Definition

La **Task Definition** rappresenta la configurazione di esecuzione del container ECS. Di seguito è riportata la definizione utilizzata:

```

"taskDefinitionArn": "arn:aws:ecs:us-east-1:839351546441:task-
    definition/definizione-processo-aws-backend:49",
"containerDefinitions": [
{
  "name": "aws-backend",
  "image": "839351546441.dkr.ecr.us-east-1.amazonaws.com/awscloud/
    project:c7a5d1f",
  "cpu": 100,
  "portMappings": [
    {
      "name": "aws-backend-800-tcp",
      "containerPort": 8000,
      "hostPort": 8000,
      "protocol": "tcp",
      "appProtocol": "http"
    }
  ],
  "essential": true,
  "environment": [],
  "environmentFiles": [
    {
      "value": "arn:aws:s3:::backend-s3-config/.env",
      "type": "s3"
    }
  ],
  "mountPoints": [],
  "volumesFrom": [],
  "logConfiguration": {
    "logDriver": "awslogs",
    "options": {
      "awslogs-group": "/ecs/definizione-processo-aws-backend",
      "mode": "non-blocking",
      "awslogs-create-group": "true",
      "max-buffer-size": "25m",
      "awslogs-region": "us-east-1",
      "awslogs-stream-prefix": "ecs"
    },
    "secretOptions": []
  },
  "systemControls": []
}
]

```

```

        },
    ],
    "family": "definizione-processo-aws-backend",
    "taskRoleArn": "arn:aws:iam::839351546441:role/ecsTaskExecutionRole",
    "executionRoleArn": "arn:aws:iam::839351546441:role/
        ecsTaskExecutionRole",
    "networkMode": "bridge",
    "revision": 49,
    "volumes": [],
    "status": "ACTIVE",
    "cpu": "200",
    "memory": "384"
}

```

## Punti chiave

- **Risorse allocate:** 200 unità CPU e 384 MB di memoria.
- **Port Mapping:** la porta 8000 del container è mappata alla porta 8000 dell'host EC2 (TCP/HTTP).
- **Variabili ambiente:** caricate dinamicamente da un file .env memorizzato su S3.
- **Logging:** configurazione con driver awslogs, log inviati al gruppo CloudWatch /ecs/definizione-processo-aws-backend.
- **IAM Roles:** utilizzo del ruolo ecsTaskExecutionRole per l'accesso a ECR e S3.
- **Network Mode:** impostato su bridge, consentendo al container di comunicare con la rete host.

## 7 Autoscaling del Cluster ECS

Il cluster ECS è configurato per ridimensionarsi automaticamente attraverso un **Capacity Provider** con *Managed Scaling*. Questa funzionalità utilizza una policy di tipo **Target Tracking Scaling** che mantiene costantemente la metrica CapacityProviderReservation al valore target del 100%.

### 7.1 Funzionamento della Policy di Autoscaling

La policy di autoscaling monitora il livello di utilizzo delle risorse del cluster (vCPU e memoria) e agisce come segue:

- **Scale Out:** se l'utilizzo complessivo delle risorse ECS supera il target del 100%, vengono avviate nuove istanze EC2 all'interno dell'Auto Scaling Group (ASG) associato.
- **Scale In:** se l'utilizzo scende al di sotto della soglia, le istanze EC2 non necessarie vengono terminate, riducendo i costi.

Questa configurazione garantisce che il cluster mantenga un numero di istanze ottimale in funzione del carico di lavoro, evitando sprechi di risorse e migliorando l'efficienza.

## 8 Lambda Function e Analisi Immagini

La funzione Lambda viene attivata automaticamente dal caricamento di una nuova immagine su S3. Il suo compito è:

- Prelevare l'immagine dal bucket S3.
- Analizzarla tramite **Amazon Rekognition** per estrarre i tag (label) più rilevanti.
- Salvare i risultati (tag, metadati, timestamp) su una tabella **DynamoDB**.

La funzione è scritta in Python e utilizza i client boto3 per l'integrazione con i servizi AWS. Gestisce errori di formato, oggetti mancanti e problemi di analisi, loggando ogni fase per facilitare il debug.

Codice Python della Lambda:

```
1 import boto3
2 import json
3 from datetime import datetime
4 import urllib.parse
5 from decimal import Decimal
6
7 dynamodb = boto3.resource('dynamodb', region_name='us-east-1')
8 table = dynamodb.Table('ImageLabels')
9
10 def lambda_handler(event, context):
11     print("Evento ricevuto:", json.dumps(event, indent=2))
12     try:
13         bucket = event['Records'][0]['s3']['bucket']['name']
14         key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'],
15                                         encoding='utf-8')
16         print(f"Bucket: {bucket}")
17         print(f"Key originale: {key}")
18         supported_formats = ['.jpg', '.jpeg', '.png', '.gif', '.bmp', '.webp']
19         if not any(key.lower().endswith(fmt) for fmt in supported_formats):
20             print(f"Formato file non supportato: {key}")
21             return {
22                 'statusCode': 400,
23                 'body': json.dumps({'error': 'Formato file non supportato'})
24             }
25         rekognition = boto3.client('rekognition', region_name='us-east-1')
26         s3_client = boto3.client('s3', region_name='us-east-1')
27         try:
28             s3_client.head_object(Bucket=bucket, Key=key)
29             print(f"Oggetto S3 trovato: s3://{bucket}/{key}")
30         except Exception as e:
31             print(f"Oggetto S3 non trovato: {e}")
32             return {
33                 'statusCode': 404,
34                 'body': json.dumps({'error': f'Oggetto non trovato: {str(e)}'})
35             }
36         print(f"Inizio analisi immagine: s3://{bucket}/{key}")
37         response = rekognition.detect_labels(
38             Image={
39                 'S3Object': {
40                     'Bucket': bucket,
41                     'Name': key
42                 }
43             },
44             MaxLabels=15,
45             MinConfidence=70
46         )
47         labels = []
48         label_names = []
49         for label in response['Labels']:
50             confidence_value = Decimal(str(round(label['Confidence'], 2)))
51             labels.append({
52                 'Name': label['Name'],
53                 'Confidence': confidence_value
54             })
55             label_names.append(label['Name'])
56         print(f"Labels trovati ({len(labels)}):", label_names)
57         item = {
58             'ImageKey': key,
59             'Bucket': bucket,
60             'Labels': labels,
```

```

60     'LabelNames': label_names,
61     'Timestamp': datetime.utcnow().isoformat(),
62     'TotalLabels': Decimal(len(labels)),
63     'ProcessedBy': 'ImageAnalysisFunction'
64   }
65   table.put_item(Item=item)
66   print(f"Dati salvati in DynamoDB per: {key}")
67   return {
68     'statusCode': 200,
69     'body': json.dumps({
70       'message': 'Analisi completata con successo',
71       'imageKey': key,
72       'bucket': bucket,
73       'labelsFound': len(labels),
74       'labels': label_names
75     })
76   }
77 except Exception as e:
78   error_msg = f"Errore durante l'analisi: {str(e)}"
79   print(f"{error_msg}")
80   import traceback
81   print("Stack trace completo:")
82   traceback.print_exc()
83   return {
84     'statusCode': 500,
85     'body': json.dumps({
86       'error': error_msg,
87       'imageKey': key if 'key' in locals() else 'unknown',
88       'bucket': bucket if 'bucket' in locals() else 'unknown',
89     })
90   }

```

## 9 Amazon S3

L'infrastruttura AWS utilizza Amazon S3 come soluzione di storage fondamentale, con diversi bucket dedicati a scopi specifici per garantire modularità, sicurezza e scalabilità:

- **Bucket di configurazione:**
  - Nome: `backend-s3-config`
  - Funzione: ospita i file di configurazione `.env` utilizzati sia dal backend che dal frontend.
  - Accesso: configurato per consentire l'accesso in lettura selettivo, ad esempio tramite ruoli IAM associati ai task ECS, in modo da mantenere riservatezza e sicurezza delle variabili di ambiente.
  - Lifecycle: la risorsa Terraform specifica `force_destroy = true`, che permette la distruzione del bucket anche se contiene oggetti, utile in fase di sviluppo e test per evitare blocchi nella gestione del bucket.
- **Bucket per il frontend (Static Website Hosting):**
  - Utilizzato per ospitare tutti i file statici dell'interfaccia utente, come HTML, CSS, JavaScript e asset multimediali.
  - Configurato come sito web statico, abilitando la possibilità di servirlo direttamente via HTTP senza dover passare da un server web tradizionale.
  - Integrato con CloudFront per fornire contenuti distribuiti globalmente, riducendo latenza e migliorando le performance.
  - Accessi e permessi: impostati in modo da consentire l'accesso pubblico ai contenuti statici, mentre eventuali risorse protette possono essere gestite tramite politiche di sicurezza più restrittive.

- **Bucket per immagini utenti:**

- Destinato a memorizzare le immagini caricate dagli utenti dell'applicazione.
- Le immagini possono essere processate da servizi AWS come Rekognition per l'analisi automatica, e quindi archiviate in modo efficiente.
- Permessi: configurati per permettere l'upload da parte delle applicazioni client o backend, con controlli di accesso precisi per evitare esposizioni accidentali.
- Possibile integrazione con funzionalità di versioning, replica o lifecycle policy per backup e gestione dei dati a lungo termine.

## 10 Database

La tabella DynamoDB utilizzata nel progetto si chiama `ImageLabels` ed è strutturata come segue:

- **Chiave primaria (hash key):** `ImageKey` di tipo `String (S)`, che identifica in modo univoco ogni immagine.
- **Attributi memorizzati per ogni item:**
  - `Bucket`: nome del bucket S3 originale in cui è salvata l'immagine.
  - `Labels`: un array di oggetti, ciascuno contenente:
    - \* `Name`: il nome della label rilevata dall'analisi dell'immagine.
    - \* `Confidence`: il valore di confidenza associato alla label, rappresentato come numero decimale.
  - `LabelNames`: un array contenente solo i nomi delle labels rilevate, utile per ricerche rapide.
  - `Timestamp`: timestamp in formato ISO 8601 che indica il momento in cui l'immagine è stata processata.
  - `TotalLabels`: numero totale di labels rilevate per l'immagine, espresso come valore decimale.
  - `ProcessedBy`: stringa che identifica la funzione Lambda che ha effettuato l'analisi.

## 11 CI/CD

La pipeline di Continuous Integration e Continuous Deployment (CI/CD) è realizzata tramite **GitHub Actions** e garantisce un processo automatizzato e affidabile per la pubblicazione del progetto. Il flusso funziona nel seguente modo:

- **Trigger:** la pipeline si avvia automaticamente al push sul branch `main` nella cartella `backend/**`.
- **Build e push:** viene creata un'immagine Docker del backend, taggata con l'hash del commit (`IMAGE_TAG`) e inviata al registry **Amazon ECR**.

- **Deploy su ECS:** la definizione della task ECS viene aggiornata con la nuova immagine e il servizio ECS viene riallineato, garantendo un aggiornamento senza interruzioni.
- **Frontend:** un workflow dedicato si occupa del deploy della parte frontend su un bucket **Amazon S3**.

### Workflow per frontend:

```

name: Build and Deploy Frontend to S3

on:
  push:
    branches:
      - main
    paths:
      - 'frontend/**'

jobs:
  build-frontend:
    environment: aws
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: frontend

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Node.js
        uses: actions/setup-node@v4
        with:
          node-version: 18

      - name: Install AWS CLI
        run: pip install awscli

      - name: Scarica .env.production da S3
        run: aws s3 cp s3://${{ vars.AWS_S3_BUCKET_CONFIG }}/.env.production .env.production
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          AWS_DEFAULT_REGION: us-east-1

      - name: Install dependencies
        run: npm install

      - name: Mostra .env.production
        run: cat .env.production

      - name: Build frontend
        run:
          NODE_ENV=production CI=' npm run build

      - name: Upload build artifact
        uses: actions/upload-artifact@v4
        with:
          name: frontend-build
          path: frontend/build

deploy-frontend:
  needs: build-frontend
  environment: aws
  runs-on: ubuntu-latest
  env:
    AWS_S3_BUCKET: ${{ vars.AWS_S3_BUCKET }}

  steps:

```

```

- name: Download build artifact
  uses: actions/download-artifact@v4
  with:
    name: frontend-build
    path: build-output

- name: Deploy to S3
  uses: jakejarvis/s3-sync-action@v0.5.1
  with:
    args: --delete
  env:
    AWS_S3_BUCKET: ${{ env.AWS_S3_BUCKET }}
    AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
    AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    AWS_REGION: us-east-1
    SOURCE_DIR: ./build-output

```

## 12 Infrastruttura AWS con Terraform

L'infrastruttura AWS viene definita tramite **Terraform**, convertendo il template CloudFormation in codice dichiarativo. Il file `main.tf` definisce le risorse principali:

- **VPC e Subnet** per la rete interna.
- **S3 Bucket** per configurazioni e file di input/output.
- **IAM Roles e Policies** per autorizzare Lambda ed ECS a interagire con i servizi AWS.
- **DynamoDB** per la memorizzazione dei metadati.
- **Lambda Function** per analisi immagini.
- **ECS Task Definition** per il backend containerizzato.

Per eseguire il deployment:

1. Installare Terraform (`>=1.3.0`).
2. Inizializzare il progetto: `terraform init`.
3. Validare la configurazione: `terraform validate`.
4. Approvare e creare le risorse: `terraform apply`.

## 13 Conclusioni

In questo progetto sono stati analizzati e configurati vari componenti dell'infrastruttura cloud per realizzare un sistema scalabile, sicuro e performante. L'uso di servizi gestiti ha permesso di ridurre la complessità operativa e garantire una migliore affidabilità.

### 13.1 Riepilogo dei Servizi AWS utilizzati

Di seguito un elenco dei principali servizi utilizzati:

- **Amazon ECS** (Elastic Container Service): gestione dei container e dei task.
- **Amazon EC2**: istanze per eseguire i container tramite il cluster ECS.
- **Amazon ECR** (Elastic Container Registry): repository per le immagini Docker.
- **Amazon S3**: storage per file di configurazione e immagini.
- **Amazon CloudWatch**: monitoraggio e logging.
- **Amazon DynamoDB**: database NoSQL per lo storage dei dati delle analisi.
- **Amazon Rekognition**: analisi automatica delle immagini.
- **Amazon Cognito**: autenticazione e gestione degli utenti.
- **AWS Lambda**: funzioni serverless per l'elaborazione degli eventi.
- **Amazon VPC**: rete privata virtuale per l'infrastruttura.
- **Auto Scaling Group**: gestione dinamica della capacità delle istanze.
- **Cognito**: autenticazione e gestione utenti.