

Music Database Query App for Distributed Systems on Cloud

Lorenzo Maria Alberto Paoria

22 luglio 2025

Sommario

Il progetto "Music Database Query App for Distributed Systems on Cloud" rappresenta un'applicazione distribuita per la gestione di query su database musicali implementata su infrastruttura cloud AWS.

1 L'idea

L'idea del progetto nasce nell'ambito del corso di Sistemi Cloud e Laboratorio. Avendo già sviluppato un'applicazione per il progetto del corso di Sistemi Distribuiti e Laboratorio, ho deciso di riutilizzarla come base. La sfida principale consiste nel progettare un'architettura cloud in grado di gestire carichi di lavoro variabili, garantire la persistenza dei dati e mantenere prestazioni ottimali anche in presenza di picchi di traffico.

La soluzione permette a più client di connettersi simultaneamente al server, eseguire query sul database musicale e ricevere risultati in tempo reale, il tutto gestito attraverso una CLI.



Figura 1: Logo

2 Tecnologie utilizzate

2.1 Amazon Web Services (AWS)

AWS fornisce l'infrastruttura cloud per l'hosting dell'applicazione. I servizi utilizzati includono:

- **EC2**: Istanza virtuale per l'hosting del server applicativo
- **RDS**: Database PostgreSQL per l'uso applicativo
- **VPC**: Rete virtuale privata per la sicurezza
- **Security Groups**: Firewall
- **SNS**: Servizio di notifiche per il monitoraggio
- **SQS**: Code di messaggi per il logging
- **NLB**: Network Load Balancer per la distribuzione del carico

2.2 Java

Linguaggio di programmazione principale per lo sviluppo dell'applicazione. Java è stato scelto per:

- Portabilità multi-piattaforma
- Robustezza nella gestione delle eccezioni
- Ricco ecosistema di librerie per database e networking
- Supporto nativo per la programmazione multi-threaded

2.3 PostgreSQL

Sistema di gestione database relazionale scelto per:

- Affidabilità e conformità ACID
- Supporto per transazioni complesse
- Ottimizzazione per carichi di lavoro misti (lettura/scrittura)
- Integrazione nativa con Amazon RDS

2.4 Docker

Containerizzazione dell'applicazione per:

- Deployment consistente tra ambienti diversi
- Isolamento delle dipendenze
- Scalabilità orizzontale
- Facilità di gestione e aggiornamento

2.5 Python

Utilizzato per gli script di automazione dell'infrastruttura:

- Deploy automatizzato delle risorse AWS
- Aggiornamento dei secrets GitHub (indirizzo ip server e chiave)
- Aggiornamento degli indirizzi ip, dns e rds endpoint
- Monitoraggio della coda di messaggi

2.6 GitHub

Utilizzato come sostegno in quanto:

- Sistema di versionamento del codice
- Automazione dei processi di CI/CD tramite Actions

3 Architettura del Sistema

3.1 Architettura Generale

Il sistema adotta un'architettura client-server distribuita su cloud con i seguenti componenti principali:

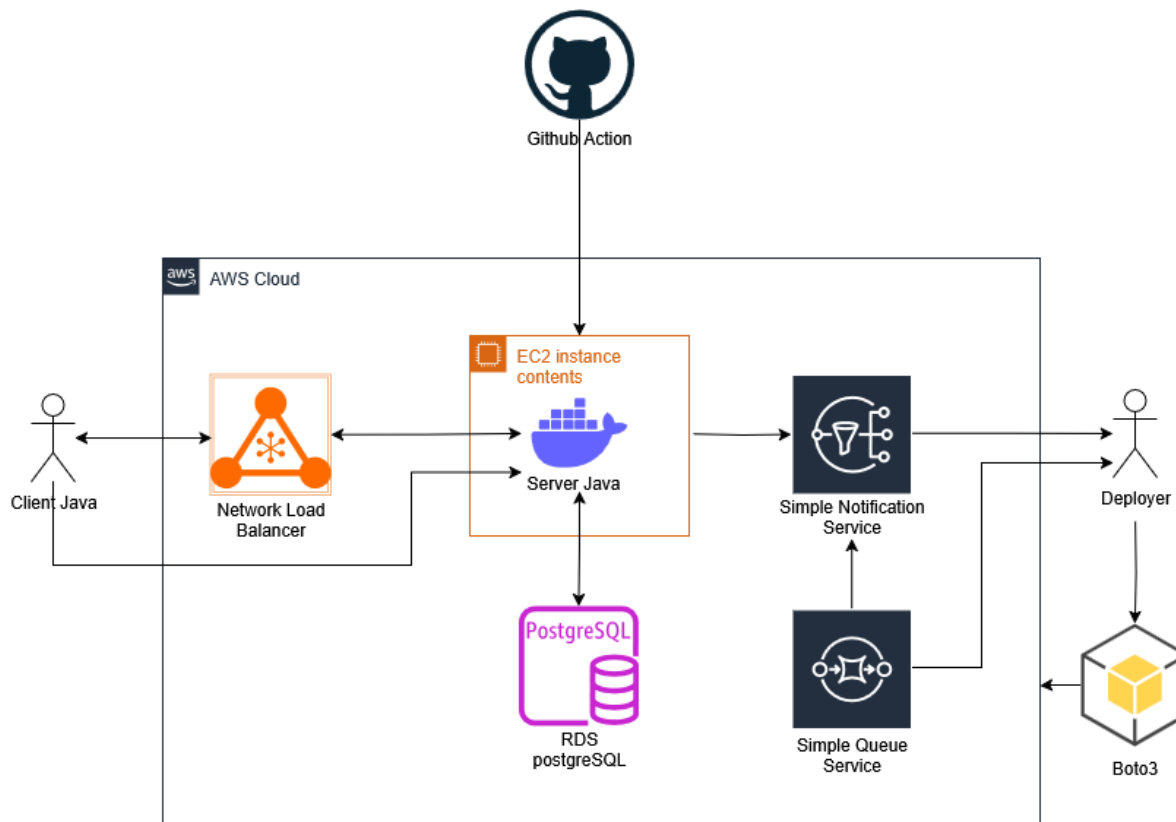


Figura 2: Diagramma dell'architettura generale del sistema

- **Client Java**: Interfaccia CLI Java per l'interazione utente
- **Deployer**: Amministratore del l'intero sistema
- **Server Java**: Server Java multi-threaded ospitato su container Docker runnato su EC2
- **RDS postgresQL**: PostgreSQL database gestito tramite Amazon RDS
- **Network Load Balancer**: Load Balancer per la distribuzione del traffico che usa TCP
- **Simple Notification Service**: SNS per notificare lo stato del server
- **Simple Queue Service**: SQS per il logging
- **GitHub Actions**: per il CI/CD

3.2 Struttura del Database

Il Database è progettato per la memorizzazione e gestione dei dati relativi ad una piattaforma di streaming musicale.

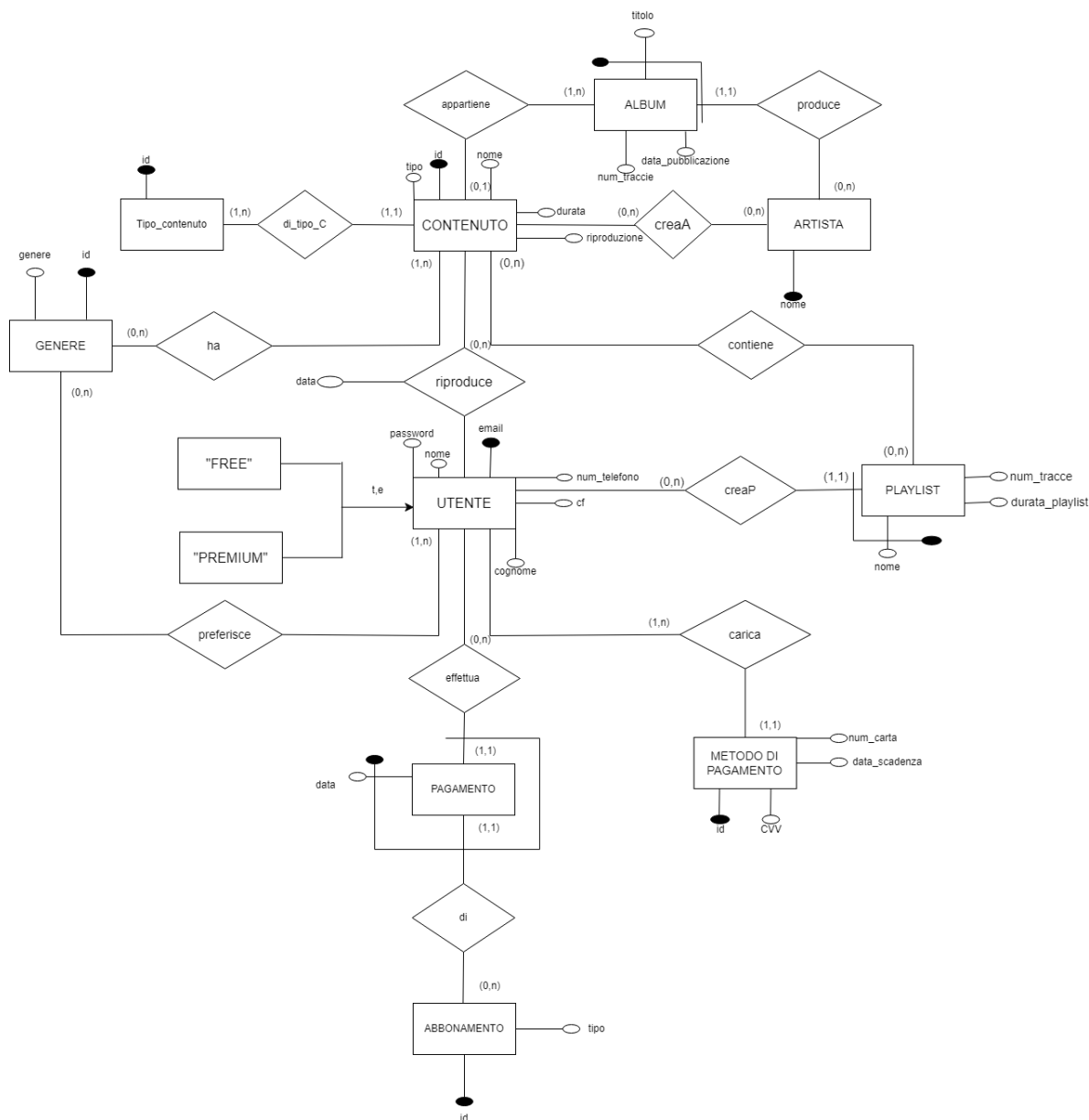


Figura 3: Schema ER del database musicale

4 Panoramica

La cartella ‘scripts/infrastructure’ contiene gli script Python per l’automazione del deploy e gestione dell’infrastruttura AWS. Ogni script ha un ruolo specifico.

4.1 Panoramica dei File Infrastructure

- **deploy_music_app.py**: Script principale per il deploy completo dell’infrastruttura
- **setup_nlb.py**: Configurazione del Network Load Balancer che è opzionale per il funzionamento base dell’applicativo
- **update_github_secrets.py**: Aggiornare i secrets GitHub per il corretto funzionamento della Action
- **update_java_config_on_ec2.py**: Aggiornamento configurazione Java in locale e push della repo
- **monitor_sqs.py**: Monitoraggio in tempo reale della coda di messaggi
- **user_data_script.sh**: Script di inizializzazione per l’istanza EC2
- **deploy_config.json**: File con le specifiche di tutti i componenti creati generato automaticamente alla fine del deploy dell’infrastruttura
- **my-ec2-key.pem**: Chiave privata per accesso SSH alle istanze EC2

5 Script

Il sistema utilizza script Python per l'automazione del deploy su AWS:

5.1 deploy_music_app.py - Script Principale

Listing 1: Struttura principale dello script di deploy

```
1 def main():
2
3     # Pulizia delle risorse AWS
4     if "--clean" in os.sys.argv:
5         delete_resources(ec2, rds, KEY_PAIR_NAME,
6                           DB_INSTANCE_IDENTIFIER,
7                           'MusicAppRDSSecurityGroup',
8                           'MusicAppEC2SecurityGroup',
9                           skip_rds)
10
11     return
12
13     # Deploy EC2 e RDS
14     ec2_client = boto3.client('ec2', region_name=REGION)
15     rds_client = boto3.client('rds', region_name=REGION)
16
17     # Step 1: Gestione credenziali AWS
18     aws_credentials = read_aws_credentials()
19
20     # Step 2: Creazione Key Pair EC2
21     key_pair_name = get_key_pair(ec2_client, KEY_PAIR_NAME)
22
23     # Step 3: Configurazione VPC e Security Groups
24     vpc_id, rds_sg_id, ec2_sg_id = create_vpc_and_security_groups(
25         ec2_client, rds_client)
26
27     # Step 4-5: Deploy RDS e inizializzazione database
28     rds_endpoint = deploy_rds_instance(rds_client, rds_sg_id)
29     initialize_database(rds_endpoint, schema_sql, data_sql)
30
31     # Step 6-7: Setup notifiche SNS e code SQS
32     topic_arn = setup_sns_notification(REGION, topic_name, email)
33     queue_url = setup_sqs_logging_queue(REGION, queue_name, topic_arn)
34
35     # Step 8: Deploy istanze EC2
36     deploy_ec2_instances(ec2_client, ec2_sg_id, user_data_script)
37
38     # Step 9: Salvataggio informazioni nel deploy_config.json
39     config = {
40         "server_public_ip": server_public_ip,
41         "server_private_ip": server_private_ip,
42         "rds_endpoint": rds_endpoint,
43         "db_username": DB_MASTER_USERNAME,
44         "db_password": DB_MASTER_PASSWORD,
45         "db_name": DB_NAME,
46         "key_pair_name": key_pair_name_actual
47     }
```

Caratteristiche principali:

- **Idempotenza:** L'operazione può essere eseguita più volte senza causare problemi, in quanto, se le risorse da creare sono già presenti, vengono semplicemente riutilizzate.
- **Rollback:** Supporta una pulizia completa o parziale delle risorse create. Utilizzando gli argomenti `--clean` `--nords`, è possibile eliminare tutte le risorse generate, mantenendo intatto il database (la cui creazione richiede tempi lunghi). Oppure è possibile effettuare una pulizia completa tramite l'argomento `--clean`
- **Logging:** Fornisce un output dettagliato per ogni operazione, organizzato in sezioni come `[SECTION]`, `[INFO]`, `[SUCCESS]`, `[WARNING]` e `[ERROR]`.
- **Gestione degli errori:** Implementa una gestione robusta degli errori legati ai servizi AWS.

5.2 setup_nlb.py - Network Load Balancer

Script dedicato alla configurazione del NLB:

Listing 2: Configurazione Network Load Balancer

```
1 def main():
2
3     # Pulizia risorse create
4     if "--clean" in os.sys.argv:
5         cleanup_nlb_resources()
6     return
7
8     # lettura della configurazione di deploy esistente
9     deploy_config = read_deploy_config()
10
11     # inizializzazione dei client AWS
12     ec2_client = boto3.client('ec2', region_name=REGION)
13     elbv2_client = boto3.client('elbv2', region_name=REGION)
14
15     # STEP 1: recupero delle informazioni di rete
16     vpc_id, subnet_ids = get_default_vpc_and_subnets(ec2_client)
17
18     # STEP 2: identificazione dell'istanza server
19     server_instance_id = get_server_instance_id(ec2_client)
20
21     # STEP 3: creazione del target group
22     target_group_arn = create_target_group(elbv2_client, vpc_id)
23
24     # STEP 4: creazione del Network Load Balancer
25     nlb_arn, nlb_dns = create_nlb(elbv2_client, subnet_ids)
26
27     # STEP 5: configurazione del listener
28     listener_arn = create_listener(elbv2_client, nlb_arn, target_group_arn)
29
30     # STEP 6: registrazione dell'istanza nel target group
31     register_target(elbv2_client, target_group_arn, server_instance_id)
32
33     # STEP 7: aggiornamento della configurazione di deploy
34     update_deploy_config(nlb_dns, NLB_PORT)
```

Caratteristiche principali:

- **Idempotenza:** L'operazione può essere eseguita più volte senza causare problemi.
- **Rollback:** Supporta una pulizia completa delle risorse create attraverso l'argomento `--clean`
- **Logging:** Fornisce un output dettagliato per ogni operazione anche esso suddiviso in sezioni.
- **Gestione degli errori:** Implementa una gestione robusta degli errori legati ai servizi AWS.
- **Indirizzamento:** Distribuzione automatica del traffico.
- **Health Check:** Monitoraggio ogni 30 secondi.
- **Scalabilità:** Supporto per multiple istanze.
- **Performance:** Latenza ultra-bassa (Layer 4) in quanto lavora tramite TCP.

5.3 update_github_secrets.py - Integrazione CI/CD

Script per aggiornare i GitHub Secrets utili per il funzionamento delle Actions:

Listing 3: Aggiornamento GitHub Secrets

```
1 def main():
2
3     script_dir = os.path.dirname(os.path.abspath(__file__))
4
5     # STEP 1: caricamento del token GitHub dall'ambiente
6     github_token = load_environment()
7
8     # STEP 2: recupero delle informazioni del repository Git
9     owner, repo = get_repo_info()
10
11     # STEP 3: verifica dell'esistenza dei file necessari
12     config_file = os.path.join(script_dir, "deploy_config.json")
13     pem_file = os.path.join(script_dir, "my-ec2-key.pem")
14
15     # STEP 4: lettura delle configurazioni e della chiave privata
16     with open(config_file, 'r') as f:
17         config = json.load(f)
18
19     with open(pem_file, 'r') as f:
20         pem_content = f.read()
21
22     # STEP 5: recupero della chiave pubblica del repository
23     public_key_info = get_public_key(owner, repo, github_token)
24
25     # STEP 6: aggiornamento del secret EC2_HOST
26     print(f"[STEP] Aggiornamento del secret EC2_HOST con valore: {config['server_public_ip']}")
27     if not update_secret(owner, repo, github_token, "EC2_HOST",
28                          config['server_public_ip'],
29                          public_key_info['key_id'],
30                          public_key_info['key']):
31         success = False
32
33     # STEP 7: aggiornamento del secret EC2_SSH_KEY
34     print("[STEP] Aggiornamento del secret EC2_SSH_KEY in corso...")
35     if not update_secret(owner, repo, github_token, "EC2_SSH_KEY",
36                          pem_content,
37                          public_key_info['key_id'],
38                          public_key_info['key']):
39         success = False
40
41     # STEP 8: verifica del risultato finale
42     if success:
43         print("[SUCCESS] Tutti i secrets sono stati aggiornati correttamente!")
44         print("[INFO] Ora puoi procedere con il push per triggerare il deploy!")
45     else:
46         print("[ERROR] Alcuni secrets non sono stati aggiornati correttamente.")
```

Caratteristiche principali:

- **Crittografia:** Secrets crittografati con chiave pubblica del repo
- **API GitHub:** Utilizzo sicuro delle API GitHub
- **Validazione:** Controlli di integrità sui dati

5.4 update_java_config_on_ec2.py - Aggiornamento Remoto

Script per aggiornare la configurazione Java in locale e fare una push alla repository GitHub:

Listing 4: Aggiornamento configurazione via SSH

```
1 def main():
2
3     # lettura della configurazione di deploy
4     with open("deploy_config.json", "r") as f:
5         config = json.load(f)
6
7     # STEP 1: determinazione della configurazione client (NLB/EC2)
8     nlb_enabled = config.get("nlb_enabled", False)
9     if nlb_enabled and "nlb_dns" in config and "nlb_port" in config:
10         # configurazione NLB
11         CLIENT_TARGET_HOST = config["nlb_dns"]
12         CLIENT_TARGET_PORT = str(config["nlb_port"])
13         print(f"[INFO] Configurazione client per Network Load Balancer: {
14             CLIENT_TARGET_HOST}:{CLIENT_TARGET_PORT}")
15     else:
16         # configurazione EC2
17         CLIENT_TARGET_HOST = SERVER_EC2_PUBLIC_IP
18         CLIENT_TARGET_PORT = SERVER_APPLICATION_PORT
19         print(f"[INFO] Configurazione client per connessione diretta EC2: {
20             CLIENT_TARGET_HOST}:{CLIENT_TARGET_PORT}")
21
22     print("[STEP] Avvio del processo di aggiornamento della configurazione...")
23
24     # STEP 2: costruzione dei percorsi ai file di configurazione
25     script_dir = os.path.dirname(os.path.abspath(__file__))
26     project_root = os.path.abspath(os.path.join(script_dir, "..", ".."))
27
28     server_config_path = os.path.join(project_root, "mvnProject-Server", "src", "main",
29         "java", "com", "example", "config", "DatabaseConfig.java")
30     server_db_properties_path = os.path.join(project_root, "mvnProject-Server", "src",
31         "main", "java", "com", "example", "config", "database.properties")
32     client_config_path = os.path.join(project_root, "mvnProject-Client", "src", "main",
33         "java", "com", "example", "DatabaseClient.java")
34
35     # STEP 3: aggiornamento dei file di configurazione Java
36     update_local_java_config(
37         server_ip=SERVER_EC2_PRIVATE_IP,
38         server_port=SERVER_APPLICATION_PORT,
39         rds_endpoint=RDS_ENDPOINT,
40         db_username=DB_USERNAME,
41         db_password=DB_PASSWORD,
42         client_server_ip=CLIENT_TARGET_HOST,
43         client_server_port=CLIENT_TARGET_PORT,
44         server_config_path=server_config_path,
45         server_db_properties_path=server_db_properties_path,
46         client_config_path=client_config_path
47     )
48
49     # STEP 4: commit e push delle modifiche per triggerare GitHub Actions
50     git_commit_and_push()
```

Caratteristiche principali:

- **Auto-detection:** Rileva automaticamente presenza NLB tramite deploy_config.json
- **Configuration Management:** Aggiorna configurazioni Java in locale
- **Git Integration:** Commit e push automatici che avvieranno la GitHub Action

5.5 monitor_sqs.py - Monitoraggio Real-time

Sistema di monitoraggio per log distribuiti:

Listing 5: Monitoraggio SQS in tempo reale

```
1 def monitor_queue():
2     # inizializzazione del client SQS
3     sqs_client = boto3.client('sqs', region_name=REGION)
4
5     # recupero dell'URL della coda
6     queue_url = sqs_client.get_queue_url(QueueName=QUEUE_NAME)['QueueUrl']
7
8     # visualizzazione delle informazioni di avvio
9     print(f"[INFO] Monitoraggio coda SNS avviato")
10    print(f"[INFO] Coda: {QUEUE_NAME}")
11    print(f"[INFO] In ascolto di nuovi messaggi...")
12    print("-" * 60)
13
14    message_count = 0
15
16    # loop principale di monitoraggio
17    while True:
18        try:
19            # lettura e visualizzazione dei nuovi messaggi
20            new_message_count = read_and_display_messages(sqs_client, queue_url)
21            message_count += new_message_count
22
23            # indicatore di attesa se non ci sono nuovi messaggi
24            if new_message_count == 0:
25                current_time = datetime.datetime.now().strftime('%H:%M:%S')
26                print(f"[WAIT] {current_time} - In ascolto... (Tot: {message_count})",
27                    end='\r')
28
29            except KeyboardInterrupt:
30                # gestione dell'interruzione da tastiera
31                print(f"\n\n[SUCCESS] Monitoraggio terminato. Messaggi processati: {
32                    message_count}")
33                break
34            except Exception as e:
35                # gestione degli errori con retry automatico
36                print(f"\n[ERROR] Errore durante il monitoraggio: {e}")
37                time.sleep(5)
38
39 def main():
40     monitor_queue()
```

Caratteristiche principali:

- **Real-time:** Visualizzazione immediata dei nuovi messaggi
- **Timezone:** Conversione automatica fuso orario

5.6 user_data_script.sh - Inizializzazione EC2

Script bash per l'inizializzazione automatica delle istanze EC2:

Listing 6: Script di inizializzazione EC2

```
1 #!/bin/bash
2 sudo dnf update -y
3 # installa Git, Docker e AWS CLI
4 sudo dnf install -y git docker awscli
5 sudo systemctl enable docker
6 sudo systemctl start docker
7 sudo usermod -aG docker ec2-user
8 newgrp docker
9
10 # configura AWS CLI per ec2-user
11 sudo -u ec2-user mkdir -p /home/ec2-user/.aws
12 sudo -u ec2-user cat <<EOF > /home/ec2-user/.aws/credentials
13 [default]
14 aws_access_key_id=AWS_ACCESS_KEY_ID_PLACEHOLDER
15 aws_secret_access_key=AWS_SECRET_ACCESS_KEY_PLACEHOLDER
16 aws_session_token=AWS_SESSION_TOKEN_PLACEHOLDER
17 EOF
18
19
20 # clono il repository GitHub e imposto i permessi
21 git clone https://github.com/lorenzopaoria/repo.git $APP_DIR
22 sudo chown -R ec2-user:ec2-user $APP_DIR
23
24 # verifico che il Dockerfile esista nel repository
25 if [ -f "$APP_DIR/Dockerfile.dockerfile" ]; then
26     # rinomino il Dockerfile esistente per Docker
27     mv "$APP_DIR/Dockerfile.dockerfile" "$APP_DIR/Dockerfile"
28     echo "Using existing Dockerfile from repository"
29 else
30     echo "Warning: Dockerfile.dockerfile not found in repository"
31     exit 1
32 fi
33
34 # build della Docker image
35 echo "Building the Docker image..."
36 cd $APP_DIR
37 docker build -t music-server-app .
38
39 # run del container Docker
40 echo "Running the Docker container..."
41 docker run -d -p 8080:8080 --name musicapp-server music-server-app
42
43 # notifica SNS di completamento setup del server (eseguito come ec2-user)
44 sudo -u ec2-user aws sns publish --region us-east-1 --topic-arn $(message)
```

Caratteristiche principali:

- **Zero Configuration:** Setup completamente automatico
- **Docker Integration:** Containerizzazione dell'app server con porta 8080 esposta
- **AWS Integration:** Configurazione automatica AWS CLI
- **Notification:** Notifica SNS al completamento

5.7 Dockerfile.dockerfile - File configurazione del container

Listing 7: dockerfile per container

```
1 # uso di un'immagine ufficiale di Maven con Java 17 come immagine di base
2 FROM maven:3.9-eclipse-temurin-17
3
4 # imposta la directory di lavoro all'interno del contenitore
5 WORKDIR /app
6
7 # copia l'intero progetto all'interno del contenitore
8 COPY . .
9
10 # Questo compilerà il codice e scaricherà le dipendenze
11 RUN mvn -f mvnProject-Server/pom.xml clean install
12
13 # Imposta la directory di lavoro sulla cartella del progetto server per il comando
14 # successivo
15 WORKDIR /app/mvnProject-Server
16
17 # espone la porta 8080 per consentire il traffico all'applicazione
18 EXPOSE 8080
19
20 # Questo è il comando che verrà eseguito all'avvio del contenitore
21 CMD ["mvn", "-Pserver", "exec:java"]
```

Caratteristiche principali:

- **Immagine Base Ufficiale:** Usa l'immagine Docker ufficiale `maven:3.9-eclipse-temurin-17` con Java 17 preinstallato
- **Build Automatizzato:** Esegue automaticamente `mvn clean install` per compilare il progetto e risolvere le dipendenze
- **Directory Setup:** Imposta directory di lavoro chiare per build e runtime
- **Port Mapping:** Espone la porta 8080 per accesso esterno al server
- **Avvio del Server:** Esegue il server Java via Maven usando il profilo `server` all'avvio del container

5.8 Deploy Docker Container on EC2 - GitHub Actions Workflow

Workflow YAML per il deploy automatico dell'applicazione su un'istanza EC2 usando Docker e GitHub Actions.

Listing 8: Workflow GitHub Actions per il deploy EC2 con Docker

```
1 name: Deploy Docker Container on EC2
2
3 on:
4   push:
5     branches: [ main, master ]
6
7 jobs:
8   deploy:
9     runs-on: ubuntu-latest
10    steps:
11      - name: Checkout code
12        uses: actions/checkout@v4
13
14      - name: Validate EC2 connection
15        uses: appleboy/ssh-action@v1.0.3
16        with:
17          host: ${ secrets.EC2_HOST }
18          username: ec2-user
19          key: ${ secrets.EC2_SSH_KEY }
20          timeout: 60s
21          script: |
22            echo "Connessione EC2 stabilita con successo"
23            whoami
24            docker --version || echo "Docker non installato"
25
26      - name: Deploy to EC2
27        uses: appleboy/ssh-action@v1.0.3
28        with:
29          host: ${ secrets.EC2_HOST }
30          username: ec2-user
31          key: ${ secrets.EC2_SSH_KEY }
32          timeout: 600s
33          command_timeout: 30m
34          script: |
35            set -e
36
37            cd /home/ec2-user/repo
38
39            git fetch origin
40            git reset --hard origin/main
41
42            docker ps -q --filter "publish=8080" | xargs -r docker stop || true
43            docker ps -aq --filter "publish=8080" | xargs -r docker rm || true
44
45            sleep 5
46
47            docker system prune -f
48
49            echo "Ricostruendo il container Docker..."
50            docker build -f Dockerfile.dockerfile -t musicapp-server
51
52            docker run -d --name musicapp-server -p 8080:8080 --restart unless-stopped
53              musicapp-server
54
55            sudo -u ec2-user aws sns publish --region us-east-1 --topic-arn $(message)
56
57            echo "Deploy completato!"
```

Caratteristiche principali:

- **CI/CD Automatica:** Deploy automatico su EC2 ad ogni push su main
- **Connessione Sicura:** Connessione SSH al server EC2 tramite GitHub Secrets
- **Repo Sync:** Sincronizzazione forzata con `git fetch` e `git reset --hard`

- **Gestione Porta 8080:** Terminazione forzata di processi attivi e verifica disponibilità
- **Pulizia Docker:** Cleanup completo di container e immagini inutilizzate
- **Build & Run:** Costruzione della Docker image e avvio container con policy di riavvio
- **Health Check:** Verifica che il container sia attivo e che l'app risponda sulla porta 8080
- **Notifica SNS:** Invio notifica al topic SNS AWS al termine del deploy

6 Workflow Operativo del Sistema

6.1 Processo di Deploy Completo

Il sistema implementa un workflow di deploy completamente automatizzato che segue questa sequenza:

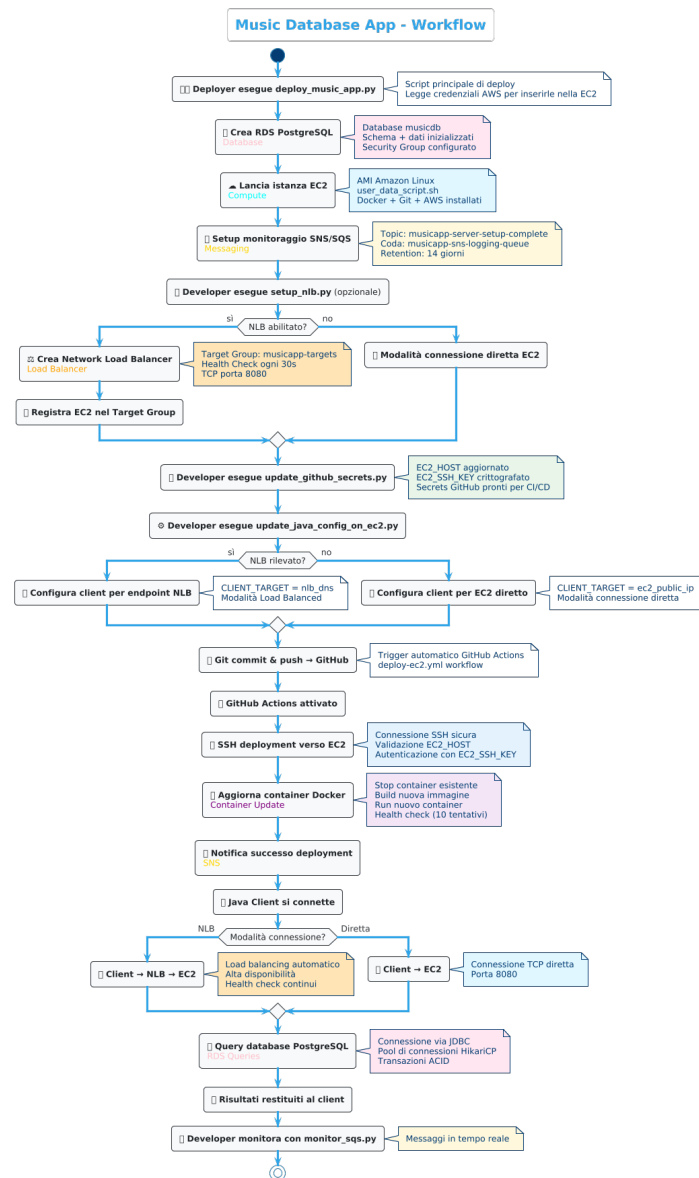


Figura 4: Diagramma del workflow

1. Inizializzazione Infrastruttura

```
1 python deploy_music_app.py
```

2. Configurazione Load Balancer (Opzionale)

```
1 python setup_nlb.py
```

3. Sincronizzazione GitHub Secrets

```
1 python update_github_secrets.py
```

4. Aggiornamento Configurazione

```
1 python update_java_config_on_ec2.py
```

5. Monitoraggio Sistema

```
1 python monitor_sqs.py
```

6.2 Sistema di Pulizia Risorse

Il sistema include opzioni flessibili per la pulizia delle risorse:

Pulizia completa:

```
1 python deploy_music_app.py --clean
```

- Elimina tutte le risorse AWS (EC2, RDS, NLB, SNS, SQS)
- Tempo stimato: 15-20 minuti (causa eliminazione RDS)
- Per documentazione dopo aver eliminato SQS si deve aspettare 1 minuto per poterlo ricreare

Pulizia parziale (mantiene database):

```
1 python deploy_music_app.py --clean --nords
```

- Elimina EC2, NLB, SNS, SQS ma mantiene RDS
- Tempo stimato: 2-3 minuti

Pulizia solo NLB:

```
1 python setup_nlb.py --clean
```

- Elimina solo Network Load Balancer
- Mantiene tutto il resto attivo

6.3 Prerequisiti e Configurazione

Il sistema richiede la seguente configurazione:

Listing 9: Configurazione credenziali AWS

```
1 # File ~/.aws/credentials
2 [default]
3 aws_access_key_id = YOUR_ACCESS_KEY_ID
4 aws_secret_access_key = YOUR_SECRET_ACCESS_KEY
5 aws_session_token = YOUR_SESSION_TOKEN # opzionale per credenziali temporanee come
    Learner Lab
```

Listing 10: Configurazione GitHub per CI/CD

```
1 # File .env nella root del progetto
2 GITHUB_TOKEN=your_github_token
3 GITHUB_TOKEN_API=your_api_token
4 REPO=username/repository-name
5 SECRET_NAME=deployment_secrets
6 PEM_PATH=path/to/private/key
```

6.4 Tempi di Deployment

- **Deploy completo da zero:** 15-20 minuti
- **Setup Network Load Balancer:** 3-5 minuti
- **Aggiornamento configurazione:** 1-2 minuti
- **Pulizia completa risorse:** 15-20 minuti (incluso RDS)
- **Pulizia parziale (no RDS):** 2-3 minuti