UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Ottimizzazione delle Infrastrutture Cloud su AWS: Terraform, Terragrunt e la Gestione Multi-Ambiente

RELATORE
Prof. Arcangelo Castiglione
Università degli Studi di Salerno

RELATORE ESTERNO
Gabriele Previtera
Epsilon SRL

CANDIDATO
Junhuang Chen
Matricola: 0512112650

Anno Accademico 2024-2025

"The only real mistake
is the one from which we learn nothing."

Abstract

L'evoluzione del cloud computing ha trasformato radicalmente il modo in cui vengono gestite le infrastrutture IT, offrendo soluzioni sempre più flessibili e scalabili grazie a piattaforme come Amazon Web Services (AWS). Tuttavia, con l'aumento della complessità delle architetture cloud, diventa essenziale adottare strumenti avanzati per garantire una gestione efficace degli ambienti multipli. Terraform è uno dei principali strumenti per la gestione dell'infrastruttura come codice (IaC), permettendo di definire e controllare le risorse cloud in modo dichiarativo. Nonostante la sua potenza, Terraform presenta alcune limitazioni nella gestione multi-ambiente, in particolare nella condivisione di configurazioni e output tra ambienti diversi e nella gestione degli state backend.

L'attività di ricerca, svolta in collaborazione con Gabriele Previtera, tutor aziendale presso Epsilon SRL, si è concentrata sull'analisi delle problematiche e ha proposto l'adozione di Terragrunt come soluzione per superare i limiti di Terraform. Grazie a Terragrunt, è stato possibile centralizzare la gestione delle configurazioni, ridurre la duplicazione del codice e automatizzare il passaggio di variabili tra moduli. La nuova architettura implementata ha portato a un significativo miglioramento nell'efficienza del deployment e nella riduzione degli errori legati alla configurazione manuale delle dipendenze tra applicazioni.

L'approfondimento ha incluso il confronto pratico tra il metodo tradizionale con Terraform e l'approccio con Terragrunt, evidenziando i vantaggi in termini di scalabilità e automazione. Inoltre, sono stati studiati servizi AWS come S3 per la gestione dello stato remoto e CloudFront per la distribuzione dei contenuti, integrando le best practice per l'infrastruttura come codice.

I risultati ottenuti dimostrano che l'uso di Terragrunt rappresenta una soluzione efficace per ottimizzare la gestione delle infrastrutture cloud moderne, con ulteriori margini di miglioramento grazie all'integrazione di funzionalità avanzate, come l'automazione della configurazione post-deployment tramite hooks e l'integrazione con pipeline CI/CD più avanzate.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Contesto e Motivazione

Negli ultimi anni, il cloud computing ha rivoluzionato la gestione delle infrastrutture IT, offrendo soluzioni scalabili, flessibili ed efficienti. Amazon Web Service (AWS) si distingue come una delle piattaforme cloud più utilizzate a livello globale, fornendo un'ampia gamma di servizi per la gestione e l'orchestrazione delle risorse IT. Tutta via, con l'aumento della complessità delle architetture cloud, diventa cruciale adottare strumenti avanzati per gestire in modo efficace ambienti multipli.

Infrastructure as Code (IaC) rappresenta una soluzione chiave a questa difficoltà, consentendo la definizione e il provisioning delle risorse cloud attraverso codice dichiarativo. In questo contesto, Terraform è uno degli strumenti più diffusi per la gestione delle infrastrutture cloud, ma presenta alcuni limitazioni nella gestione multi-ambiente, in particolare nella condivisione delle configurazioni e nella gestione degli output tra diversi ambienti. Per superare queste problematiche, Terragrunt si propone come estensione di Terraform, introducendo un approccio modulare e centralizzato per semplificare la gestione delle infrastrutture multi-ambiente.

1.2 Obiettivo della Ricerca

Questa tesi si propone di analizzare i limiti di Terraform nella gestione multi-ambiente e di valutare i benefici dell'adozione di Terragrunt come soluzione per migliorare

l'efficienza e la scalabilità delle infrastrutture cloud su AWS. Verranno esplorati i vantaggi di Terragrunt nella centralizzazione delle configurazioni, nella riduzione della ridondanza del codice e nell'automazione del flusso di lavoro.

1.3 Rilevanza e Importanza dello Studio

L'importanza di questa ricerca risiede nella crescente necessità delle aziende di gestire ambienti cloud complessi in modo efficace, riducendo errori e migliorando la manutenibilità dell'infrastruttura. Sebbene esistano diverse soluzioni per la gestione delle configurazioni cloud, l'integrazione tra Terraform e Terragrunt si presenta come una delle più promettenti per la sua flessibilità e semplicità di adozione.

Questa tesi fornirà un'analisi dettagliata delle best practice per l'uso combinato di Terraform e Terragrunt, con un focus sulla gestione degli ambienti di sviluppo. Inoltre, saranno esplorati i servizi AWS utilizzati per supportare tali configurazioni, come S3 per il backend remoto e CloudFront per la distribuzione dei contenuti.

1.4 Domande di Ricerca e Obiettivi Specifici

Le domande di ricerca principali a cui questa tesi cercherà di rispondere sono:

- Quali sono le principali difficoltà nella gestione multi-ambiente con Terraform?
- In che modo Terragrunt può semplificare e migliorare la gestione dell'infrastruttura cloud?
- Quali sono le best pratice per l'adozione di Terragrunt in un contesto aziendale su AWS?
- Quali vantaggi pratici derivano dall'uso di Terragrunt in termini di efficienza operativa e manutenibilità del codice?

Gli obiettivi specifici dello studio includono:

- Analizzare i limiti della gestione multi-ambiente con Terraform.
- Implementare un'infrastruttura cloud su AWS utilizzando Terraform e Terragrunt.
- Confrontare l'efficacia di una gestione basata esclusivamente su Terraform rispetto a una basata su Terragrunt.
- Proporre un modello di best pratice per le aziende che desiderano adottare Terragrunt per la gestione della loro infrastruttura cloud.

1.5 Struttura della Tesi

La tesi è divisa nei seguenti capitoli:

- Introduzione - Presenta il contesto, la motivazione e gli obiettivi dello studio.
- Background - Fornisce una panoramica sui concetti fondamentali del cloud computing, AWS, Terraform e Terragrunt.
- Stato dell'arte - Analizza le attuali soluzioni disponibili e i principali studi sulla gestione multi-ambiente con Terraform.
- Soluzione proposta - Descrive il modello e l'approccio adottato per migliorare la gestione delle infrastrutture multi-ambiente.
- Implementazione della soluzione proposta - Illustra il caso pratico di utilizzo di Terraform e Terragrunt per la gestione di ambienti su AWS.
- Valutazione sperimentale della soluzione proposta - Analizza i benefici e le criticità della soluzione implementata, confrontandolo con l'approccio tradizionale.
- Conclusioni e sviluppi futuri - Riassume i risultati dello studio e propone possibili sviluppi futuri nell'uso di Terragrunt.

Questa struttura guidera il lettore attraverso un percorso logico e approfondito, fornendo sia una solida base teorica che un'analisi pratica delle soluzioni proposte.

Capitolo 2

Background

2.1 Introduzione al Cloud Computing

Il cloud computing è un modello di erogazione di servizi IT (Information Technology) "on demand" tramite Internet. Ha permesso alle aziende di scalare le proprie risorse in modo dinamico e flessibile. Grazie a modelli di servizio come Infrastructure as a Service (IaaS) e Software as a Service (SaaS), invece di possedere e mantenere infrastrutture e software localmente, è possibile gestire ambienti IT senza la necessità di hardware fisico dedicato, gli utenti possono accedere a risorse condivise (come potenza di calcolo, storage e applicazioni) ospitate da provider di servizi cloud. Tra i principali provider di servizi cloud, Amazon Web Service (AWS) si distingue per la sua ampia gamma di servizi e strumenti dedicato all'automazione dell'infrastruttura.

2.1.1 Caratteristiche del Cloud Computing

Il cloud computing presenta diverse caratteristiche fondamentali che lo distinguono dai modelli tradizionali di gestione IT:

- **Scalabilità**: Le risorse possono essere aumentate o ridotte in base alla domanda.
- **Elasticità**: Capacità di adattarsi dinamicamente alle esigenze di utilizzo.
- **Accesso remoto**: Possibilità di accedere ai servizi da qualsiasi dispositivo connesso a Internet.

- **Modello a consumo**: Le risorse vengono pagate in base all'uso effettivo.
- **Automazione**: Processi gestiti tramite strumenti software per ridurre l'intervento manuale.

Queste caratteristiche rendono il cloud computing una scelta sempre più popolare per aziende di tutte le dimensioni.

2.1.2 Chi utilizza il cloud computing?

Il cloud computing è adottato da un'ampia varietà di utenti, che spaziano dalle grandi aziende alle piccole start-up, dalle istituzioni pubbliche ai singoli sviluppatori. Di seguito, alcune delle principali categorie di utilizzatori:

1. **Grandi aziende e imprese multinazionali**.
   Le grandi aziende adottano il cloud computing per migliorare la scalabilità, ridurre i costi infrastrutturali e accelerare il time-to-market. Aziende come Netflix, Airbnb e Spotify utilizzano il cloud per gestire enormi volumi di dati e garantire un'elevata disponibilità dei servizi su scala globale.

2. **Start-up e PMI (Piccole e Medie Imprese)**
   Le startup sfruttano il cloud per ridurre i costi iniziali di infrastruttura, evitando investimenti onerosi in data center fisici. Il modello **pay-per-use** permette alle PMI di crescere in modo flessibile, adattando le risorse alle proprie esigenze.

3. **Enti pubblici e governativi**
   Molti governi e istituzioni pubbliche adottano il cloud per modernizzare le infrastrutture IT, migliorare l'efficienza e garantire sicurezza e conformità normativa. Un esempio è il programma **GovCloud** negli Stati Uniti o l'adozione del **cloud nazionale** in diversi paesi europei.

4. **Sviluppatori e professionisti IT**
   Gli sviluppatori utilizzano il cloud per testare, sviluppare e distribuire applicazioni in ambienti scalabili. Servizi come AWS Lambda, Google Cloud Functions e Azure DevOps semplificano la gestione del ciclo di vita delle applicazioni.

5. **Settori specializzati (Sanità, Finanza, Educazione, e-commerce, gaming, etc.)**

   - **Sanità**: utilizzo del cloud per l'analisi dei dati medici, archiviazione di cartelle cliniche elettroniche e intelligenza artificiale applicata alla diagnostica.
   - **Finanza**: banche e istituti finanziari adottano il cloud per l'analisi dei rischi, il trading ad alta frequenza e la sicurezza dei dati.

- **Educazione**: piattaforme come Google Workspace for Education e Microsoft Azure vengono utilizzate per l'e-learning e la gestione degli studenti.
- **Gaming e intrattenimento**: servizi di cloud gaming come NVIDIA GeForce Now o Xbox Cloud Gaming permettono di giocare senza hardware potente in locale.

2.1.3 Vantaggi del Cloud Computing

L'adozione del cloud computing offre numerosi vantaggi rispetto ai tradizionali modelli IT:

- **Riduzione dei costi**: Non è necessario investire in hardware fisico, e si paga solo per le risorse utilizzate.
- **Manutenzione semplificata**: I provider gestiscono aggiornamenti e sicurezza, riducendo il carico amministrativo.
- **Flessibilità e mobilità**: L'accesso ai dati e alle applicazioni è possibile da qualsiasi luogo.
- **Alta disponibilità**: Grazie alle infrastrutture ridondanti, il cloud garantisce continuità operativa e minimi tempi di inattività.
- **Collaborazione facilitata**: Più utenti possono lavorare simultaneamente sugli stessi progetti grazie alle risorse condivise.

2.2 Amazon Web Services (AWS)

Amazon Web Services (AWS) è una delle piattaforme di cloud computing più diffuse e offre un'ampia gamma di servizi che coprono vari ambiti dell'IT, tra cui calcolo, deployment, storage, reti, l'intelligenza artificiale, sicurezza e la gestione di applicazioni su infrastrutture scalabili e sicure. Grazie alla sua architettura globale e alla sua scalabilità, AWS permette alle azienda di creare infrastrutture flessibili e ad alte prestazioni senza dover investire in hardware fisico. Inoltre, AWS fornisce strumenti avanzati per la gestione dell'automazione, la sicurezza dei dati e l'ottimizzazione dei costi, consentendo alle organizzazioni di adattarsi rapidamente alle esigenze di mercato in continua evoluzione.

2.2.1 Modelli di Servizio nel Cloud Computing su AWS

AWS offre una vasta gamma di servizi cloud, organizzati secondo tre principali modelli di servizio: **Infrastructure as a Service (IaaS), Platform as a Service**

(PaaS) e Software as a Service (SaaS). Questi modelli forniscono diversi livelli di astrazione e gestione delle risorse, consentendo alle aziende di scegliere la soluzione più adatta alle proprie esigenze.

1. **Infrastructure as a Service (IaaS)**
   L'IaaS fornisce infrastruttura IT virtualizzata su richiesta, eliminando la necessità di acquistare e gestire hardware fisico. Con questo modello, gli utenti hanno il controllo sulle risorse come server, rete, storage e sistemi operativi, ma senza doversi occupare della gestione fisica dell'infrastruttura.
   Esempi di servizi AWS IaaS:

   - **Amazon EC2 (Elastic Compute Cloud)**: fornisce capacità di calcolo scalabile in base alle necessità.
   - **Amazon S3 (Simple Storage Service)**: offre storage altamente scalabile e durevole.
   - **Amazon VPC (Virtual Private Cloud)**: consente la creazione di reti isolate nel cloud.
   - **Elastic Load Balancing (ELB)**: distribuisce automaticamente il traffico tra più istanze EC2 per garantire alta disponibilità.
   - **Amazon CloudFront**: un servizio Content Delivery Network (CDN) che distribuisce contenuti (come pagine web, video e API) con bassa latenza e alte prestazioni, sfruttando una rete globale di edge locations.

   Questo modello è ideale per aziende che vogliono un'infrastruttura flessibile e scalabile senza dover gestire data center fisici.

2. **Platform as a Service (PaaS)**
   Il modello PaaS offre un ambiente di sviluppo e deployment completamente gestito, consentendo agli sviluppatori di concentrarsi sulla scrittura del codice senza preoccuparsi dell'infrastruttura sottostante. AWS gestisce l'infrastruttura, il sistema operativo e il runtime, semplificando lo sviluppo e la scalabilità delle applicazioni.
   Esempi di servizi di PaaS:

   - **AWS Lambda**: consente l'esecuzione di codice serverless, senza provisioning di server.
   - **Amazon RDS (Relational Database Service)**: fornisce database gestiti come MySQL, PostgreSQL, e Aurora.
   - **AWS Fargate**: gestisce l'infrastruttura necessaria per eseguire container senza dover gestire server o cluster.

Il PaaS è utile per le aziende che vogliono accelerare il time-to-market, riducendo il lavoro di gestione dell'infrastruttura.

3. **Software as a Service (SaaS)**
   Il modello SaaS fornisce software accessibili via internet senza che l'utente debba installare o gestire applicazioni localmente. AWS ospita molte applicazioni SaaS di terze parti, ma fornisce anche strumenti per le aziende che vogliono sviluppare e distribuire il proprio software in modalità SaaS.
   Esempi di servizi AWS legati al SaaS:

   - **AWS Marketplace**: una piattaforma che consente alle aziende di distribuire e vendere software SaaS su AWS.
   - **Amazon WorkSpaces**: un servizio Desktop-as-a-Service (DaaS) che fornisce desktop virtuali basati su cloud.
   - **Amazon Connect**: un contact center cloud completamente gestito per le aziende.
   - **AWS App Runner**: semplifica il deployment di applicazioni SaaS scalabili senza dover gestire server.

   Il SaaS è ideale per le aziende che vogliono offrire software agli utenti finali senza dover gestire la distribuzione e la manutenzione su infrastrutture locali.

2.3 Infrastructure as Code (IaC)

L'Infrastructure as Code (IaC) rappresenta un paradigma fondamentale per la gestione moderna delle infrastrutture cloud. Con IaC, è possibile definire e configurare risorse infrastrutturali tramite codice anziché attraverso processi manuali, garantendo riproducibilità, coerenza e automazione. Questo approccio riduce l'errore umano e facilita il versioning delle configurazioni. Esistono diversi strumenti IaC disponibili, ognuno con le proprie caratteristiche e funzionalità. Tra gli strumenti più diffusi, Terraform è un'opzione open-source che permette di definire e gestire l'infrastruttura su diverse piattaforme cloud (AWS, Azure, GCP) e on-premise.

2.3.1 Vantaggi dell'Infrastructure as Code

L'adozione dell'IaC offre diversi vantaggi, tra cui:

- **Automazione e riproducibilità**: Riduzione di tempi e errori umani grazie all'automazione dei processi e consente di creare ambienti infrastrutturali coerenti e ripetibili, garantendo che le stesse configurazioni vengano utilizzate in tutti gli ambienti.

- **Coerenza e standardizzazione**: Le configurazioni rimangono uniformi in tutti gli ambienti, evitando configurazioni non documentate.
- **Versionamento e tracciabilità**: Grazie all'integrazione con i sistemi di controllo versione (come Git), è possibile tenere traccia delle modifiche e ripristinare versioni precedenti.
- **Scalabilità**: La gestione delle risorse cloud può essere facilmente adattata a esigenze crescenti.

2.4 Terraform e la Gestione dell'Infrastruttura

Terraform, sviluppato da HashiCorp, è uno degli strumenti **IaC** più diffusi e ampiamente adottati per la gestione delle infrastrutture cloud. La sua forza principale risiede nella capacità di **definire e orchestrare risorse cloud** in modo dichiarativo utilizzando **HashiCorp Configuration Language (HCL)**, un linguaggio che permette di scrivere configurazioni leggibili e modulari. La sua sintassi semplice ma potente permette di definire in modo preciso l'infrastruttura desiderata, riducendo la complessità rispetto ad altri metodi di gestione manuale delle risorse.

L'approccio **dichiarativo** di Terraform è uno dei suoi principali vantaggi. Invece di scrivere script imperativi che definiscono **passo per passo** come le risorse devono essere configurate, in Terraform l'utente dichiara semplicemente **cosa** vuole ottenere. Il sistema si occupa automaticamente di determinare **come** realizzare quelle modifiche, con un alto livello di astrazione che riduce al minimo gli errori umani. Questo approccio non solo rende le configurazioni più comprensibili e facili da gestire, ma favorisce anche la **ripetibilità** e la **coerenza** tra ambienti diversi, consentendo una gestione unificata delle risorse .

Terraform supporta una vasta gamma di **provider**, tra cui **AWS, Azure, Google Cloud** e **piattaforme on-premise** come VMware, OpenStack e altri. I **provider** sono essenzialmente plugin che estendono le capacità di Terraform, permettendo a quest'ultimo di interagire con le API di diverse piattaforme e gestire risorse specifiche (come istanze di calcolo, reti, load balancer, database, storage, e molto altro).

Grazie a questa flessibilità, Terraform è uno strumento **multi-cloud** che permette alle organizzazioni di orchestrare e gestire infrastrutture complesse in ambienti ibridi o distribuiti su più piattaforme, con un unico strumento e un'unica configurazione.

Una delle caratteristiche distintive di Terraform è il suo modello di funzionamento basato su un **piano di esecuzione**. Quando si esegue il comando `terraform plan`, Terraform genera un piano di esecuzione che mostra esattamente quali modifiche verranno applicate all'infrastruttura per raggiungere lo stato desiderato definito nel codice HCL. Questo piano include azioni come la creazione, la modifica o la distruzione di risorse. Il piano di esecuzione non solo aiuta a ridurre gli errori

durante il provisioning, ma consente anche di **verificare** le modifiche prima di applicarle effettivamente, aumentando il livello di fiducia nel processo di automazione e riducendo il rischio di conseguenze indesiderate.

2.4.1 Moduli in Terraform

Nel caso di ambienti multipli (ad esempio, **dev, stage e prod**), la gestione delle configurazioni diventa una sfida. In ambienti diversi, le configurazioni possono differire, e senza l'uso di tecniche appropriate, è facile finire con duplicazioni di codice. Una delle soluzioni è l'utilizzo di **moduli**. I moduli sono blocchi di codice HCL riutilizzabili che permettono di definire risorse comuni o gruppi di risorse. I moduli possono essere condivisi tra più progetti o team, aumentando la **manutenibilità** e riducendo la ridondanza nel codice. Ad esempio, un modulo potrebbe definire una configurazione per un server web che può essere utilizzata in diversi ambienti senza modifiche significative al codice, basta passare valori diversi tramite le **variabili**.

2.4.2 L'uso di Variabili

Le **variabili** in Terraform permettono di parametrizzare le configurazioni. Le variabili possono essere definite nel codice stesso, in file separati o passate direttamente dalla riga di comando. Le variabili consentono di adattare facilmente le configurazioni a diversi ambienti senza duplicare il codice. Utilizzando variabili, è possibile creare configurazioni più **flessibili** e **riutilizzabili**, ma è importante gestirle con attenzione per evitare errori, soprattutto in contesti in cui le variabili vengono utilizzate in combinazione con moduli e backend remoti.

2.4.3 Provisioner in Terraform

Un altro concetto avanzato di Terraform è l'uso dei **provisioner**. I provisioner sono strumenti che permettono di eseguire script o comandi sulle risorse dopo che sono state create o modificate. Questo è utile per attività come la configurazione automatica di software, la gestione della sicurezza o altre operazioni personalizzate che devono essere eseguite su una risorsa dopo la sua creazione. Tuttavia, l'uso dei provisioner deve essere **moderato**. I provisioner non sono gestiti nello stesso modo delle risorse e potrebbero generare problemi di **consistenza** se non usati correttamente. Ad esempio, se una risorsa viene ricreata, il provisioner potrebbe essere eseguito nuovamente, creando problemi indesiderati.

2.5 Docker e il Contesto della Containerizzazione

Durante lo studio è stato analizzato anche Docker, strumento open source, uno delle tecnologie di conteinerizzazione più utilizzate. Al suo cuore, Docker permette di "containerizzare" le applicazioni, ovvero di impacchettarle insieme a tutte le loro dipendenze in unità standardizzate chiamate "container". Immagina i container come delle scatole standardizzate che contengono tutto ciò che serve per far funzionare un'applicazione: codice, librerie, impostazioni e dipendenze.

Prima di Docker, la distribuzione del software era spesso un processo complesso e problematico. Gli sviluppatori dovevano affrontare sfide come:

- **"Funziona sulla mia macchina"**: Le applicazioni potevano funzionare perfettamente nell'ambiente di sviluppo, ma fallire in produzione a causa di differenze nelle configurazioni o nelle dipendenze.
- **Complessità delle macchine virtuali (VM)**: Le VM, sebbene fornissero isolamento, erano pesanti e richiedevano molte risorse, rendendo difficile la scalabilità.
- **Difficoltà nella gestione delle dipendenze**: Installare e gestire le dipendenze software poteva essere un compito arduo e soggetto a errori.

Docker è emerso come soluzione a queste sfide, offrendo un approccio più leggero, efficiente e portabile alla distribuzione del software.

2.5.1 Caratteristiche fondamentali

Le principali caratteristiche di Docker sono:

- **Leggerezza e velocità**: I container Docker condividono il kernel del sistema operativo host, eliminando la necessità di emulare un intero sistema operativo. Questo li rende molto più leggeri e veloci delle VM.
- **Portabilità**: I container possono essere eseguiti su qualsiasi sistema che supporta Docker, garantendo che l'applicazione si comporti allo stesso modo in tutti gli ambienti.
- **Isolamento**: I container sono isolati l'uno dall'altro, impedendo alle applicazioni di interferire tra loro e migliorando la sicurezza.
- **Automazione**: Docker permette di automatizzare il processo di creazione, distribuzione e gestione delle applicazioni, semplificando il ciclo di vita del software.

Pensa di dover spedire un'applicazione. Con i metodi tradizionali, dovresti preoccuparti di configurare l'ambiente di destinazione, installare le dipendenze e assicurarti che tutto funzioni correttamente. Con Docker, impacchetti l'applicazione e le sue dipendenze in un container, come se fosse una scatola standardizzata. Questa "scatola" può essere spedita e aperta su qualsiasi sistema che supporta Docker, garantendo che l'applicazione funzioni sempre allo stesso modo.

Docker ha reso la containerizzazione accessibile a un'ampia gamma di sviluppatori e aziende, contribuendo alla diffusione di pratiche di sviluppo moderne come la microservizi e la continuous delivery.

2.5.2 Concetti chiave

Alla base di Docker vi sono alcuni concetti fondamentali.

Il **container** è un'unità software leggera e autonoma che include tutto il necessario per eseguire un'applicazione. Condivide il kernel del sistema operativo host, ma rimane isolato dagli altri container grazie a tecnologie come namespaces e cgroups. Questo lo rende altamente portabile ed eseguibile su qualsiasi sistema che supporti Docker. Le **immagini**, invece, rappresentano modelli di sola lettura che contengono le istruzioni per creare i container. Queste immagini vengono generate a partire da un **Dockerfile**, un file di testo che definisce il sistema operativo di base, le dipendenze software e le configurazioni necessarie. Le immagini possono essere archiviate in registri come **Docker Hub**, un repository pubblico che permette di scaricare immagini già pronte o condividerne di nuove.

Il cuore del sistema è il **Docker Engine**, il componente principale che si occupa di creare e gestire i container, eseguendo i comandi ricevuti dall'utente o da altri software. Tuttavia, nelle architetture moderne, Docker si affida a **containerd**, un runtime container che gestisce direttamente l'esecuzione dei container. Il funzionamento di Docker può essere suddiviso in tre fasi principali. Nella prima fase, lo sviluppatore crea un **Dockerfile**, che descrive le istruzioni per la costruzione dell'immagine dell'applicazione. Il Docker Engine elabora il file, scarica le dipendenze e genera l'immagine corrispondente. Una volta ottenuta l'immagine, questa può essere utilizzata per creare un container, il quale viene eseguito in un ambiente isolato con accesso alle risorse necessarie. Infine, Docker offre strumenti per gestire i container, consentendo di avviarli, arrestarli, riavviarli o rimuoverli, oltre alla possibilità di collegarli tra loro e alla rete host.

L'utilizzo di Docker offre numerosi vantaggi. Le applicazioni containerizzate sono altamente **portabili**, potendo essere eseguite su qualsiasi sistema compatibile con Docker senza necessità di modifiche. L'**isolamento** tra container garantisce che le applicazioni non interferiscano tra loro, migliorando la stabilità dell'ambiente, anche se non raggiunge il livello di separazione di una macchina virtuale. Inoltre, i

container sono estremamente **efficienti**, grazie alla loro leggerezza e rapidità di avvio, consentendo un utilizzo ottimizzato delle risorse. Questo li rende particolarmente adatti a scenari in cui è necessaria una **scalabilità** dinamica, sebbene per la gestione di container su larga scala vengano impiegati strumenti come **Kubernetes** o **Docker Swarm**. Infine, Docker semplifica il processo di **deployment**, riducendo il rischio di errori e minimizzando i tempi di inattività.

Oggi, Docker è uno strumento essenziale per sviluppatori e amministratori di sistema, semplificando la creazione, la distribuzione e la gestione delle applicazioni in modo efficiente e affidabile.

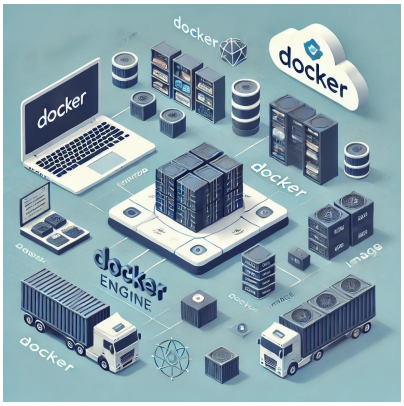


Figura 2.1: Schema illustrativo dei componenti di Docker

Capitolo 3

Stato dell'arte

3.1 Introduzione

La gestione dell'infrastruttura IT ha subito una notevole evoluzione nel corso degli anni. In passato, le infrastrutture venivano configurate e gestite manualmente, con interventi diretti sugli ambienti fisici e virtuali. Questo approccio risultava inefficiente, soggetto a errori e difficilmente scalabile, sopratutto per aziende con ambienti complessi e distribuiti.

Con l'avvento dell'Infrastructure as Code (IaC), la gestione dell'infrastruttura è diventata più strutturata e automatizzata. Strumenti come Terraform hanno permesso di definire l'infrastruttura in modo dichiarativo, rendendo più semplice la creazione, la modifica e la distruzione delle risorse cloud. Tuttavia, Terraform presenta alcune limitazioni quando si tratta di gestire ambienti multipli e infrastrutture complesse, rendendo necessaria l'adozione di strategie avanzate per migliorare la riusabilità e la gestione del codice, aspetti che verranno approfonditi nei prossimi capitoli.

Un aspetto cruciale nell'adozione di Infrastructure as Code è il processo di creazione dell'infrastruttura. In un approccio tradizionale, la configurazione delle risorse veniva effettuata manualmente tramite interfacce grafiche o script personalizzati, comportando tempi di implementazione lunghi, un'elevata probabilità di errori umani e scarsa ripetibilità. Con Terraform e strumenti simili, la creazione dell'infrastruttura avviene in maniera dichiarativa, permettendo agli sviluppatori di descrivere l'architettura desiderata in file di configurazione versionabili o dei veri e propri codici.

Questo metodo garantisce maggiore coerenza tra gli ambienti e facilita il ripristino rapido in caso di guasti o malfunzionamenti.
In questo capitolo, analizzeremo lo stato attuale della gestione delle infrastrutture con un approccio IaC, focalizzandoci sulle soluzioni più diffuse e sulle sfide riscontrate con l'utilizzo di Terraform in diversi contesti.

3.2 Comparativa gestione infrastrutture con approccio IaC

Esistono diversi strumenti che implementano il paradigma IaC, ognuno con approcci e caratteristiche differenti. AWS CloudFormation, AWS CDK, Terraform e Pulumi sono quattro delle soluzioni più adottate per la gestione delle infrastrutture cloud, ma differiscono per metodologia, linguaggio utilizzato, livello di espressività e flessibilità.

3.2.1 AWS CloudFormation

AWS CloudFormation è lo strumento nativo di AWS per definire infrastrutture tramite modelli JSON o YAML. Permette di orchestrare e distribuire risorse AWS, garantendo un'integrazione diretta con l'ecosistema AWS. Tuttavia, la sua natura dichiarativa limita la flessibilità nella gestione dinamica delle infrastrutture.

Vantaggi di AWS CloudFormation

AWS CloudFormation è stato il primo strumento a supportare la gestione dell'infrastruttura come codice su AWS e, per lungo tempo, è stato il primo a fornire il supporto alle nuove funzionalità di AWS. Grazie alla sua stretta integrazione con il cloud di Amazon, ogni volta che vengono introdotte nuove risorse o funzionalità, CloudFormation è generalmente il primo strumento a implementarle, consentendo agli utenti di sfruttare tempestivamente le novità e le innovazioni offerte dalla piattaforma AWS.

Svantaggi di AWS CloudFormation

Uno degli aspetti più critici di CloudFormation è il suo approccio dichiarativo, che pur essendo vantaggioso in termini di prevedibilità e automazione, può risultare rigido quando si tratta di infrastrutture dinamiche e modulari. Rispetto a strumenti come Terraform, CloudFormation spesso richiede una maggiore quantità di codice per ottenere lo stesso risultato, aumentando la complessità della gestione.

Inoltre, i modelli CloudFormation tendono a diventare molto articolati man mano che le infrastrutture crescono, rendendo difficile la loro manutenzione e comprensione. La gestione degli stack, sebbene sia un meccanismo utile per organizzare il provisioning

e il rollback delle risorse, introduce alcune limitazioni in termini di personalizzazione: modificare o eliminare singole risorse all'interno di uno stack può risultare complicato e talvolta richiedere la ricreazione dell'intero stack.

Infine, la rigidità della sintassi rende complesso implementare logiche avanzate o dinamiche, come cicli o condizioni più sofisticate, che in altri strumenti possono essere gestite più facilmente.

AWS CloudFormation è uno strumento potente e affidabile per la gestione dell'infrastruttura AWS. Tuttavia, è importante considerare le sue limitazioni e valutare se è la soluzione più adatta alle proprie esigenze. In alcuni casi, potrebbe essere preferibile utilizzare strumenti IaC alternativi, come Terraform, che offrono una maggiore flessibilità e modularità.

3.2.2 AWS CDK (Cloud Development Kit)

AWS CDK è una soluzione che permette di definire infrastrutture utilizzando linguaggi di programmazione comuni come TypeScript, Python e Java. A differenza di CloudFormation, che utilizza una sintassi dichiarativa, CDK adotta un approccio imperativo, consentendo agli sviluppatori di definire le risorse in modo programmatico.

Vantaggi di AWS CDK

Uno dei maggiori punti di forza di CDK è la sua capacità di sfruttare i "construct" e le funzionalità avanzate dei linguaggi di programmazione, come cicli, condizioni e funzioni riutilizzabili. Questo consente di creare infrastrutture più modulari e manutenibili rispetto ai template statici di CloudFormation.

Un altro aspetto rilevante è la sua integrazione nativa con CloudFormation: una volta definita l'infrastruttura con CDK, il framework genera automaticamente il codice CloudFormation corrispondente e lo distribuisce tramite gli stack AWS. Questo significa che gli utenti possono beneficiare della potenza di CDK senza dover rinunciare alle funzionalità di CloudFormation, come il rollback automatico e la gestione delle dipendenze.

Inoltre, CDK supporta concetti come **L2 Constructs**, che sono astrazioni di alto livello delle risorse AWS, e **L3 Constructs**, che permettono di definire intere architetture con pochi comandi. Questo consente di semplificare la definizione dell'infrastruttura e ridurre significativamente la quantità di codice necessario.

Svantaggi di AWS CDK

Tuttavia, CDK presenta alcune sfide, come la necessità di una conoscenza approfondita del linguaggio di programmazione scelto e una maggiore complessità rispetto agli strumenti puramente dichiarativi. Inoltre, l'integrazione con sistemi multi-cloud non è altrettanto agevole come con Terraform.

3.2.3 Pulumi

Pulumi è una piattaforma di infrastruttura come codice che si distingue per la sua capacità di utilizzare linguaggi di programmazione generici come JavaScript, TypeScript, Python, Go e .NET per definire e gestire infrastrutture cloud. A differenza di altri strumenti IaC, che si basano su linguaggi specifici per la configurazione (come HCL di Terraform o JSON/YAML di CloudFormation), Pulumi consente agli sviluppatori di scrivere codice più dinamico e complesso, sfruttando la potenza e la flessibilità dei linguaggi di programmazione moderni.

Vantaggi

- **Flessibilità e espressività**: Pulumi permette di utilizzare linguaggi di programmazione completi, offrendo una maggiore flessibilità rispetto agli altri strumenti IaC. Gli sviluppatori possono sfruttare librerie, moduli e strumenti di debugging nativi dei linguaggi di programmazione.
- **Compatibilità multi-cloud**: Pulumi supporta diversi provider cloud (AWS, Azure, Google Cloud, Kubernetes, e altri), rendendolo adatto per gestire infrastrutture multi-cloud in modo uniforme.
- **Esperienza di sviluppo familiare**: Utilizzando linguaggi di programmazione comuni, gli sviluppatori possono applicare le proprie competenze esistenti senza dover imparare nuovi linguaggi di configurazione o DSL (Domain Specific Languages).

Svantaggi

- **Curva di apprendimento**: Sebbene l'uso di linguaggi di programmazione comuni possa sembrare vantaggioso, può anche comportare una curva di apprendimento per gli sviluppatori che non sono abituati a gestire infrastrutture cloud attraverso codice, soprattutto quando si tratta di gestione di risorse di basso livello.
- **Comunità più piccola**: Rispetto a Terraform e CloudFormation, Pulumi ha una comunità e un ecosistema di moduli più piccoli, il che potrebbe limitare la disponibilità di risorse o soluzioni pronte all'uso.
- **Potenziale complessità nelle operazioni avanzate**: L'utilizzo di linguaggi di programmazione per definire l'infrastruttura può portare a una maggiore complessità in scenari molto complessi, specialmente per chi è abituato a strumenti più dichiarativi.

3.2.4 Terraform

Terraform rappresenta un'alternativa potente e flessibile per la gestione dell'infrastruttura come codice. A differenza di CloudFormation e CDK, Terraform è indipendente dal provider cloud e consente di gestire infrastrutture su AWS, Azure, Google Cloud e molte altre piattaforme. Utilizza un linguaggio dichiarativo chiamato HCL che permette di descrivere lo stato desiderato dell'infrastruttura senza specificare esplicitamente i passaggi per raggiungerlo.

3.2.5 Stato in Terraform

La gestione dello **stato** in Terraform è un concetto fondamentale per il corretto funzionamento dello strumento ed è essenziale per mantenere coerenza tra la configurazione dichiarativa e l'infrastruttura effettiva. Senza un meccanismo di stato, Terraform non sarebbe in grado di capire quali risorse ha già creato e quali devono essere modificate, rimosse o aggiunte.

Cos'è lo stato in Terraform?

Lo **stato in Terraform** è un file (tipicamente chiamato `terraform.tfstate`) che contiene un **registro dettagliato** di tutte le risorse che Terraform sta gestendo. Questo file tiene traccia di:

1. **L'elenco delle risorse** create da Terraform, insieme ai loro identificatori univoci (come ID delle istanze AWS EC2, nomi di bucket S3, ecc.).
2. **Gli attributi delle risorse**: per esempio, l'indirizzo IP assegnato a una macchina virtuale, le configurazioni di rete, i parametri del database.
3. **Le dipendenze tra le risorse**: se una risorsa dipende da un'altra, questa relazione viene registrata nel file di stato.
4. **L'ultimo stato noto dell'infrastruttura** rispetto alla configurazione Terraform, in modo che Terraform possa calcolare le differenze quando si eseguono nuove modifiche.

Perché lo stato è necessario?

Terraform non interroga continuamente i provider cloud per verificare lo stato attuale delle risorse, perché questo sarebbe inefficiente e molto lento. Invece, Terraform utilizza il file di stato come **riferimento locale** per determinare quali modifiche applicare.

Quando si esegue `terraform plan`, Terraform confronta lo stato salvato con la configurazione attuale definita nei file `.tf` per determinare quali modifiche sono necessarie. Questo confronto aiuta a:

- **Evitare di ricreare risorse già esistenti**, riducendo il rischio di interruzioni accidentali.
- **Eseguire modifiche mirate**, aggiornando solo ciò che è necessario.
- **Identificare il drift**, ovvero discrepanze tra lo stato registrato e l'infrastruttura reale, che possono verificarsi a causa di modifiche manuali effettuate al di fuori di Terraform.

Dove viene memorizzato lo stato?

Per impostazione predefinita, lo stato di Terraform viene salvato localmente in un file `terraform.tfstate`. Tuttavia, questo approccio ha **limiti significativi**, specialmente quando si lavora in team o su infrastrutture complesse. Alcuni problemi dello stato locale sono:

- **Difficoltà di collaborazione**: se più persone lavorano sullo stesso progetto, ciascuno avrebbe il proprio file di stato locale, causando problemi di sincronizzazione.
- **Perdita di stato**: se il file viene cancellato accidentalmente o il computer viene danneggiato, si perde la traccia dell'infrastruttura gestita.
- **Rischio di modifiche concorrenti**: più utenti potrebbero tentare di modificare l'infrastruttura simultaneamente, creando conflitti.

Per risolvere questi problemi, Terraform supporta **backend remoti**, che permettono di **memorizzare lo stato in un servizio centralizzato**. Alcuni backend comuni sono:

- **AWS S3 con DynamoDB** (per il blocco dello stato e la prevenzione di modifiche simultanee)
- **Google Cloud Storage (GCS)**
- **Azure Blob Storage**
- **Terraform Cloud e Terraform Enterprise**

Quando lo stato è memorizzato in un backend remoto, Terraform recupera lo stato prima di ogni operazione (`terraform init` e `terraform plan`, assicurando che tutti i membri del team abbiano accesso alla versione più aggiornata.

Blocco dello stato per prevenire conflitti

Quando più persone lavorano contemporaneamente su Terraform, è importante evitare che due utenti eseguano modifiche allo stesso tempo, perché ciò potrebbe portare a **stati incoerenti**. Per questo motivo, Terraform offre la funzionalità di **state locking** (blocco dello stato). Ad esempio, se si utilizza **AWS S3 con DynamoDB**, DynamoDB può essere configurato per creare un **lock** sul file di stato ogni volta che un utente sta eseguendo `terraform apply`. Questo impedisce ad altri utenti di eseguire modifiche contemporaneamente, evitando errori e conflitti.

Drift e gestione delle modifiche manuali

Un altro problema comune nella gestione dell'infrastruttura è il **drift**: ovvero, situazioni in cui lo stato registrato da Terraform non corrisponde più alla realtà. Il drift si verifica tipicamente quando qualcuno modifica direttamente le risorse nel cloud senza passare per Terraform.

Per identificare queste discrepanze, è possibile eseguire `terraform plan`, che mostrerà eventuali differenze tra lo stato registrato e l'infrastruttura effettiva. Se necessario, si può eseguire `terraform apply` per riportare l'infrastruttura allo stato desiderato.

Best Pratice per la gestione dello stato

1. **Utilizzare backend remoti** per garantire che lo stato sia sempre accessibile e aggiornato.
2. **Abilitare il blocco dello stato** per prevenire modifiche concorrenti.
3. **Eseguire regolarmente** `terraform plan` per rilevare eventuali drift e mantenere l'infrastruttura coerente.
4. **Non modificare manualmente il file di stato**: si raccomanda di lasciare che Terraform gestisca il file di stato. Modifiche manuali sono sconsigliate, salvo casi particolari.
5. **Proteggere il file di stato**: poiché contiene informazioni sensibili (come ID di risorse e variabili), dovrebbe essere crittografato o gestito con permessi adeguati.

In conclusione, la gestione dello stato è un aspetto critico di Terraform. Senza un sistema di stato affidabile, Terraform non sarebbe in grado di tenere traccia delle risorse e applicare modifiche in modo sicuro ed efficiente. Per garantire la massima affidabilità e collaborazione tra team, è essenziale adottare backend remoti, abilitare il blocco dello stato e seguire le migliori pratiche per la sua gestione.

3.2.6 Vantaggi di Terraform

L'utilizzo di Terraform porta numerosi vantaggi che lo rendono uno strumento fondamentale per la gestione delle infrastrutture moderne:

- **Indipendenza dal provider**: Terraform è uno strumento **multi-cloud** che supporta una vasta gamma di provider. Questa indipendenza dai singoli provider consente alle organizzazioni di gestire infrastrutture su più piattaforme senza dipendere da un unico fornitore. Inoltre, l'adozione di una piattaforma di orchestrazione unificata permette di evitare il lock-in con un singolo provider, garantendo maggiore **flessibili** e **portabilità** delle risorse.

- **Dichiaratività e prevedibilità**: L'approccio dichiarativo permette di descrivere in modo chiaro e conciso l'infrastruttura desiderata, senza la necessità di gestire ogni singolo passaggio operativo. Questo approccio aumenta la **prevedibilità** del provisioning e riduce significativamente il rischio di errori. Grazie al piano di esecuzione (`terraform plan`, gli utenti possono visualizzare le modifiche prima che vengano applicate, consentendo di verificare che le operazioni siano conformi alle aspettative.

- **Semplicità di apprendimento e linguaggio intuitivo**: Terraform utilizza il linguaggio di configurazione **HCL** (HashiCorp Configuration Language), che è stato progettato per essere **leggibile e comprensibile** anche da chi non ha esperienza con linguaggi di programmazione complessi. La sua sintassi chiara e concisa facilita l'apprendimento e l'adozione dello strumento. Inoltre, la documentazione estesa e la vasta community di supporto permettono di risolvere rapidamente dubbi e problemi, rendendo Terraform accessibile anche ai principianti.

- **Modularità**: Terraform permette di scrivere configurazioni altamente **modulari**. I moduli consentono di raggruppare risorse correlate e di riutilizzare codice in più contesti, migliorando la manutenibilità e riducendo la duplicazione del codice. I moduli possono essere sviluppati in modo indipendente, consentendo di costruire e gestire infrastrutture in modo più scalabile ed efficiente.

- **Automazione e scalabilità**: Terraform si integra facilmente con altre **pratiche DevOps** come **CI/CD** e **Git** per automatizzare il provisioning e la gestione delle risorse. Utilizzando strumenti di integrazione continua, Terraform consente di aggiornare e scalare le infrastrutture in modo rapido e sicuro. Inoltre, l'approccio dichiarativo e la gestione centralizzata delle risorse tramite moduli e variabili rende la gestione dell'infrastruttura non solo automatica ma anche **scalabile**, adattabile a progetti di qualsiasi dimensione.

3.2.7 Svantaggi di Terraform

Nonostante i numerosi vantaggi, Terraform presenta alcuni svantaggi che potrebbero influire sul suo utilizzo in scenari complessi:

- **Difficile da usare con logiche complesse**: Terraform è uno strumento dichiarativo, il che significa che descrive lo stato desiderato dell'infrastruttura, ma non è progettato per gestire logiche operative molto complesse. Quando è necessario implementare logiche intricate o flussi di lavoro avanzati, come condizioni dinamiche, cicli complessi o dipendenze variabili tra risorse, Terraform può risultare meno flessibile rispetto ad altri strumenti di gestione dell'infrastruttura. In tali casi, è possibile che gli utenti debbano ricorrere a soluzioni esterne o script per superare queste limitazioni.
- **Poca flessibilità nelle operazioni dinamiche**: Sebbene Terraform sia potente nella gestione di infrastrutture dichiarative, può risultare meno flessibile rispetto ad altri strumenti per operazioni altamente dinamiche. La mancanza di un linguaggio di programmazione completo limita la possibilità di eseguire operazioni complesse che richiedono un controllo più fine sulle risorse. Inoltre, la gestione delle risorse in tempo reale può essere complicata, rendendo difficile l'adattamento immediato alle modifiche in tempo reale.

3.2.8 Comparativa

Caratteristica	**CloudFormation**	**CDK**	**Terraform**	**Pulumi**
Provider cloud	AWS	AWS, tramite CloudFormation	Multi-cloud	Multi-cloud
Linguaggio	JSON/YAML	TypeScript, Python, Java, C#	HCL	JavaScript, TypeScript, Python, Go, .NET
Approccio	Dichiarativo	Imperativo	Dichiarativo	Imperativo
Complessità	Alta	Media	Media	Media
Flessibilità	Media	Alta	Alta	Alta

Tabella 3.1: Confronto tra CloudFormation, CDK, Terraform e Pulumi

3.3 Gestione infrastrutture con Terraform

Nella gestione dell'infrastruttura con Terraform, una pratica comune è l'organizzazione del codice in un repository singolo con una chiara separazione tra la configurazione degli ambienti e i moduli riutilizzabili. Questo approccio consente di mantenere

un'infrastruttura modulare e scalabile, semplificando la gestione e il deployment delle risorse.

L'infrastruttura viene definita attraverso file `.tf` contenenti la configurazione dichiarativa delle risorse. Utilizzando una combinazione di variabili, moduli e file di stato, Terraform permette di applicare modifiche controllate e riproducibili agli ambienti cloud.

3.3.1 Creazione e gestione dell'infrastruttura con Terraform

L'approccio di Terraform si basa su un insieme di file di configurazione che descrivono le risorse da creare. Un tipico flusso di lavoro con Terraform include i seguenti passaggi:

1. **Scrittura della configurazione**: Il codice dell'infrastruttura viene definito utilizzando HCL.
2. **Inizializzazione (`terraform init`)**: Terraform scarica i provider necessari per gestire le risorse specificate.
3. **Pianificazione (`terraform plan`)**: Terraform analizza lo stato attuale dell'infrastruttura e genera un piano delle modifiche da apportare.
4. **Applicazione (`terraform apply`)**: Le risorse vengono create, modificate o distrutte in base alla configurazione e al piano generato.
5. **Distruzione (`terraform destroy`)**: Se necessario, Terraform può eliminare tutte le risorse definite nel codice.

3.3.2 Esempio di struttura del codice Terraform

Una delle pratiche comuni nell'uso di Terraform è la separazione tra il codice riutilizzabile (moduli) e la configurazione specifica per un ambiente. Un esempio di struttura è la seguente:

```
AWS Shared/
├── dev/
│   ├── main.tf
│   ├── variables.tf
│   ├── outputs.tf
│   ├── shared.auto.tfvars
├── application/
│   ├── main.tf
│   ├── vpc.tf
│   ├── ...
│   ├── variables.tf
│   ├── outputs.tf
```

Figura 3.1: Struttura del progetto Terraform.

- **Cartella** `dev/`: Contiene la configurazione specifica per l'ambiente di sviluppo, con variabili come `ami_id`, `Name` ed `Environment`.
- **Cartella** `application/`: Contiene il modulo delle risorse come EC2, che definisce la configurazione di una macchina virtuale, una VPC, subnet e altre risorse associate.

Esempio di codice:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "5.78.0"
    }
  }
  required_version = ">= 1.2.0"
}

resource "aws_vpc" "prova_vpc" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "public_subnet" {
  vpc_id     = aws_vpc.prova_vpc.id
  count      = length(var.public_subnet_cidrs)
  cidr_block = element(var.public_subnet_cidrs, count.index)
  tags = {
    Name = "Public Subnet ${count.index + 1}"
  }
}

resource "aws_instance" "prova_instance" {
  ami                    = var.ami_id
  instance_type          = var.instance_type
  subnet_id              = aws_subnet.public_subnet[1].id
  associate_public_ip_address = true
  security_groups        = [aws_security_group.ec2_sg.id]
  tags = merge(
    var.shared_tags,
    var.inst_tags
```

```
33   )
34 }
```

In questo modulo, la VPC e le subnet vengono definite prima di creare un'istanza EC2, che utilizza parametri variabili per una maggiore flessibilità.

3.3.3 Concetto di repository singolo e rilascio negli ambienti con GitLab Flow

Una strategia comune nella gestione delle infrastrutture con Terraform è l'uso di un **repository singolo** che contiene la configurazione per più ambienti (sviluppo, test, produzione). Questo approccio semplifica il controllo delle versioni e garantisce coerenza tra gli ambienti.

Con **GitLab e Terraform**, il rilascio delle modifiche segue il modello **GitLab Flow**, in cui:

- Le modifiche vengono implementate e testate in branch specifici per lo sviluppo.
- Una volta verificate, vengono fuse al branch `main` o in un branch dedicato (`staging`, `production`) per essere applicate negli ambienti di produzione.
- Le pipeline CI/CD di GitLab automatizzano il deploy delle modifiche negli ambienti corrispondenti.
- Terraform utilizza il **backend remoto di GitLab** per gestire lo stato dell'infrastruttura, evitando conflitti tra le esecuzioni.

Questo approccio garantisce una gestione strutturata dell'infrastruttura, riduce il rischio di errori manuali e permette un'implementazione controllata e scalabile dell'infrastruttura cloud.

3.4 Gestione infrastrutture con Terraform in contesti multi ambiente

Quando si gestiscono infrastrutture con Terraform in contesti multi ambiente, la logica rimane simile alla gestione di un singolo ambiente, ma viene ampliata per supportare ambienti distinti come sviluppo (`dev`), staging (`staging`) e produzione (`prod`).

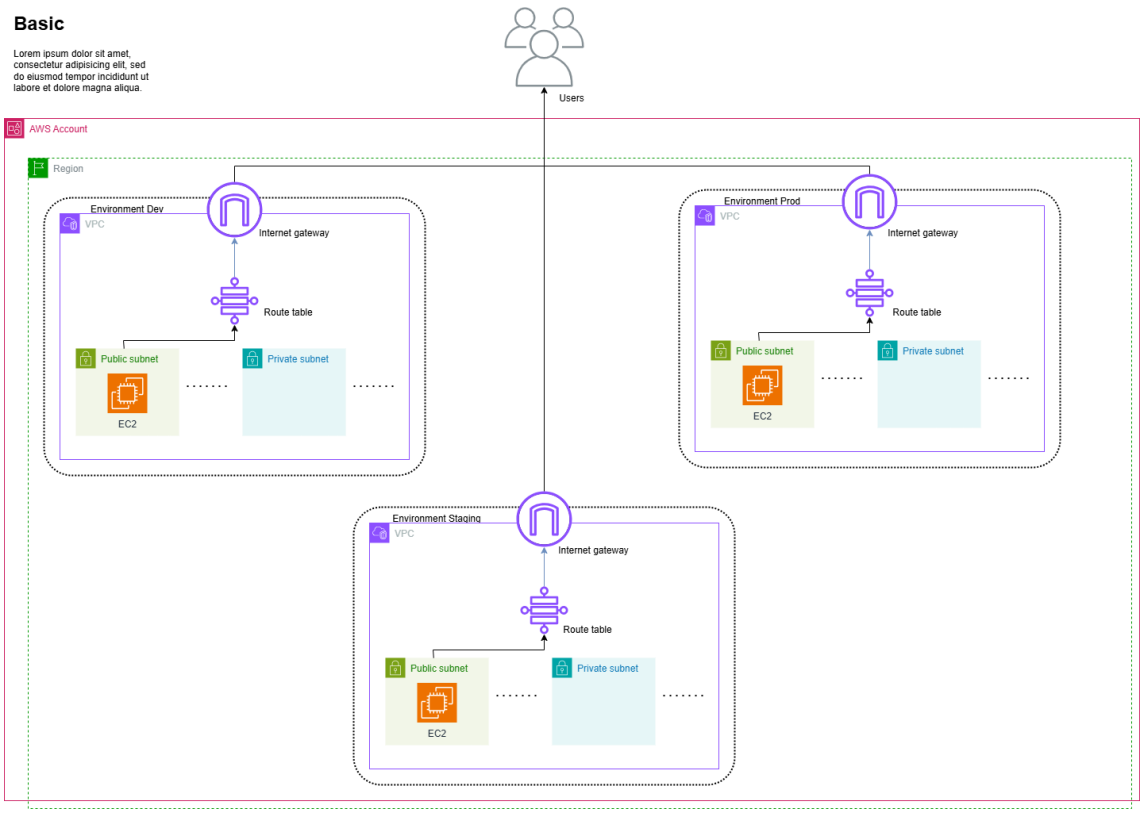


Figura 3.2: Struttura del progetto Terraform in contesti multi ambiente.

Questa architettura rappresenta un'infrastruttura cloud su AWS, progettata per supportare tre ambienti distinti: Development (Dev), Staging e Production (Prod). Ciascun ambiente è ospitato all'interno della propria Virtual Private Cloud (VPC), garantendo un isolamento completo e un livello superiore di sicurezza.

All'interno di ogni VPC, la rete è strutturata con subnet pubbliche e private. Le subnet pubbliche ospitano istanze EC2 accessibili dall'esterno, mentre quelle private sono riservate a risorse interne, come database e servizi che non devono essere esposti direttamente a internet. La connessione alla rete globale è garantita dall'Internet Gateway, mentre le route table regolano il traffico, assicurando che le subnet pubbliche possano comunicare con l'esterno secondo regole predefinite.

L'architettura è stata progettata con un approccio modulare e scalabile, così da adattarsi facilmente a diverse esigenze operative e supportare in modo efficace i processi di test. L'uso di subnet private contribuisce a rafforzare la sicurezza, limitando la superficie di attacco e proteggendo le risorse più sensibili.

Questa separazione consente di mantenere configurazioni specifiche per ogni ambiente, garantendo non solo maggiore sicurezza, stabilità e controllo delle modifiche, ma anche un'infrastruttura più flessibile e adatta alla crescita dell'applicazione.

La struttura del repository in questo caso si espande per includere più ambienti:

```
AWS Shared/
├── environments/
│   ├── dev/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
│   │   ├── shared.auto.tfvars
│   ├── staging/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
│   │   ├── shared.auto.tfvars
│   ├── prod/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
│   │   ├── shared.auto.tfvars
├── application/
│   ├── main.tf
│   ├── vpc.tf
│   ├── ...
```

Figura 3.3: Organizzazione in cartelle del progetto Terraform in multi ambiente.

- **Cartella** `live/dev`: Contiene la configurazione per l'ambiente di sviluppo.
- **Cartella** `live/staging`: Ambiente di staging, che replica sviluppo con alcune modifiche.
- **Cartella** `live/prod`: Ambiente di produzione con configurazioni ottimizzate.

Ogni ambiente dispone di un proprio file `*.auto.tfvars` che definisce parametri specifici dell'ambiente, come `instance_type`, `ami_id` di un istanza oppure il tag di ambiente chiamto `Environment` Questi file permettono di personalizzare l'infrastruttura per ciascun contesto senza dover modificare il codice dei moduli condivisi. I moduli riutilizzabili, definiti nella cartella `modules/`, vengono richiamati nei diversi ambienti passando loro i parametri appropriati. Questo approccio riduce la duplicazione del codice e garantisce coerenza tra gli ambienti, semplificando la gestione, il testing e l'evoluzione dell'infrastruttura nel tempo.

3.5 Gestione infrastrutture con Terraform in ambienti complessi

In ambienti complessi, la gestione delle infrastrutture richiede un'organizzazione chiara delle risorse e delle loro dipendenze. Spesso, esistono risorse che devono essere distribuite in ambienti separati, ma che sono interdipendenti tra loro. Questo comporta un processo di deployment che prevede:

1. **Identificare le dipendente tra infrastrutture**

2. **Provisioning delle infrastrutture in modo indipendente tra loro**
3. **Collegamento delle infrastrutture tra di loro**

Sebbene questo approccio per quanto composto solo da tre passi può nascondere delle insidie dovute alla scarsa tracciabilità delle dipendenze tra applicazioni che spesso gestite da team diversi da quello infrastrutturale, può generare errori nel collegamento tra le infrastrutture, in quanto queste attività sono fatte manualmente, il che spesso si traduce in ritardi e disservizi.

3.5.1 Esempio pratico

Per comprendere al meglio i problemi che potrebbero presentarsi nel deployare infrastrutture complesse in diversi ambienti, utilizziamo la seguente architettura come esempio:

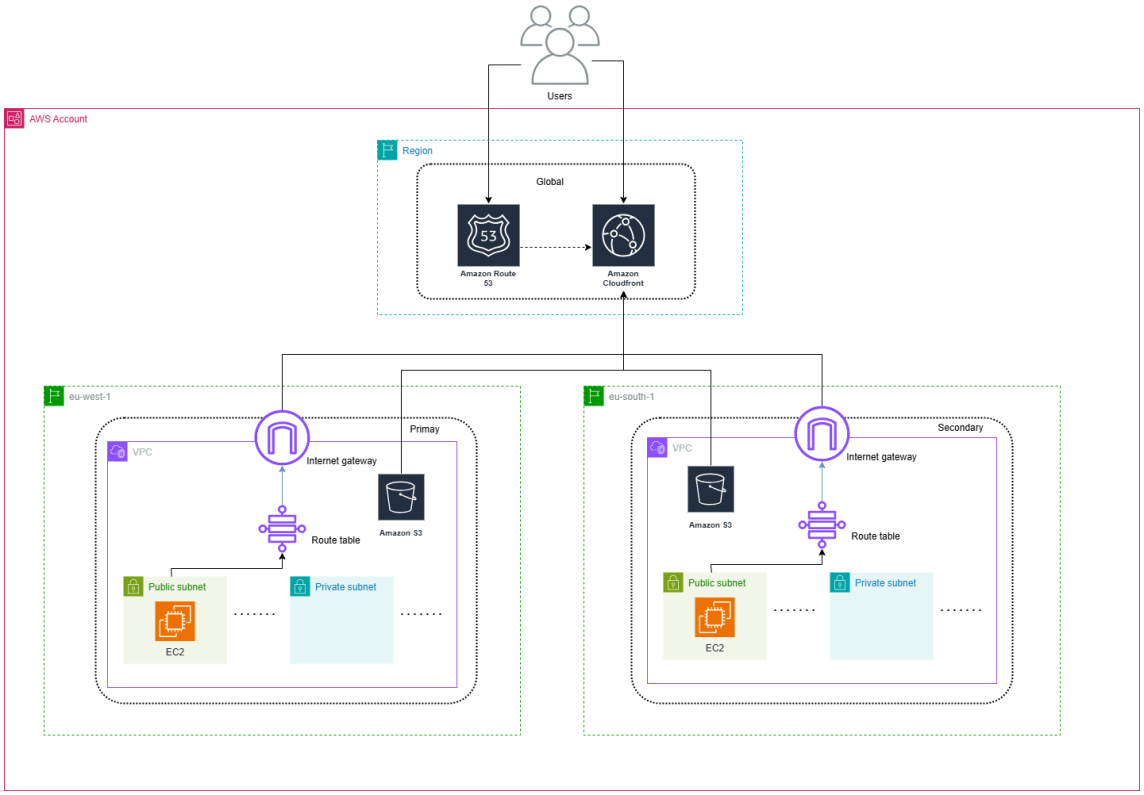


Figura 3.4: Struttura del progetto Terraform ambienti complessi.

In questa nuova architettura, si introduce una nuova struttura di **Disaster Recovery** (DR) per aumentare la resilienza. Mentre prima ogni ambiente (Dev, Staging e Prod) era isolato nella propria VPC, ora l'infrastruttura è distribuita su più regioni AWS con risorse globali per garantire continuità operativa.

Un elemento chiave è la struttura **Global**, che ospita servizi come **Amazon Route 53** e **CloudFront** per gestire il traffico tra regioni. Route 53 si occupa del DNS, mentre CloudFront, funge da CDN, migliora la distribuzione dei contenuti in modo bilanciato e riduce la latenza. Se l'ambiente primario subisce un guasto, Route 53 può reindirizzare il traffico alla regione secondaria, associando continuità operativa

L'architettura prevede due ambienti in regioni distinte: una **primaria** in **eu-west-1** e una **secondaria** in **eu-south-1**, ciascuna con la propria VPC, subnet pubbliche e private, e istanze EC2 per le applicazioni.

Tuttavia, rispetto all'architettura precedente, si nota l'aggiunta delle **dipendenze** tra le strutture con CloudFront, il che comporta che la struttura presenta una dipendenza diretta tra le diverse componenti: il layer globale non può essere deployato prima che le infrastrutture delle singole regioni siano già operative. Infatti, Route 53 e CloudFront devono essere configurati facendo riferimento alle risorse esistenti, come le istanze EC2, i load balancer o gli S3 bucket presenti nelle regioni. Questo significa che il processo di provisioning deve essere eseguito in fasi:

1. **Prima si deployano le infrastrutture indipendenti**, ovvero gli ambienti regionali di AWS Customer 1 e AWS Customer 1 Recovery.

2. **Solo successivamente si può procedere con il deployment dell'infrastruttura globale**, che può quindi configurare Route 53 e CloudFront utilizzando i riferimenti alle risorse esistenti.

```
AWS Customer 1/
├── environments/
│   ├── dev/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
│   │   ├── shared.auto.tfvars
│   ├── prod/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
│   │   ├── shared.auto.tfvars
│   ├── ...
├── application/
│   ├── main.tf
│   ├── ec2.tf
│   ├── ...
```

(a) Struttura replicata per la primaria e secondaria.

```
Global/
├── environments/
│   ├── dev/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
│   │   ├── shared.auto.tfvars
│   ├── prod/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── outputs.tf
│   │   ├── shared.auto.tfvars
│   ├── ...
├── application/
│   ├── main.tf
│   ├── cloudfront.tf
│   ├── ...
```

(b) Struttura che dipende dalle precedenti.

Figura 3.5: Struttura progetto Terraform in ambienti complessi

Ulteriormente, oltre alle dipendenze con il layer globale, si possono avere anche dipendenze tra le infrastrutture primarie e secondarie, complicando ulteriormente la fase di provisioning.

3.5.2 Limitazioni della gestione manuale dei riferimenti tra ambienti

Uno dei principali problemi riscontrati nella gestione dell'infrastruttura con Terraform riguarda il modo in cui i moduli prendono i valori in input, in particolare nel caso di CloudFront nel modulo `global`. Questo dipende dagli ambienti **dev e prod**, poiché necessita del DNS pubblico delle istanze EC2 di ciascun ambiente. Tuttavia, il processo per ottenere e utilizzare questi riferimenti risulta complesso e poco scalabile:

1. **Deploy degli ambienti separatamente**:
   - È necessario eseguire il deploy degli ambienti (dev, prod) tramite il comando `terraform apply` per creare le istanze EC2.
   - Le istanze vengono generate con indirizzi DNS pubblici in AWS.
2. **Recupero manuale dei DNS pubblici**:
   - Bisogna accedere alla console AWS, andare nella sezione EC2 e copiare manualmente i valori del Public DNS per ogni istanza creata.
3. **Assegnazione dei DNS al modulo global**:
   - I DNS devono essere inseriti nei file di variabili (*.auto.tfvars) per essere utilizzati nel modulo global.
4. **Applicazione della configurazione nel modulo global**:
   - Dopo l'aggiornamento dei file di variabili, si deve eseguire terraform apply nel modulo global, configurando CloudFront con gli origin corretti e collegando le istanze EC2 tramite Route 53.

Questo processo, già macchinoso per pochi ambienti, diventa insostenibile quando si devono gestire **60 infrastrutture** su più account AWS, con un impatto significativo sia in termini operativi che economici. La gestione manuale dei riferimenti tra ambienti introduce ritardi, errori e costi elevati, soprattutto a causa delle continue modifiche e della necessità di interventi manuali.

Per ottimizzare questa operazione, è necessario introdurre un nuovo strumento che semplifichi il recupero dei riferimenti alle risorse e ne garantisca l'integrazione senza intervento manuale.

Per ottimizzare questa operazione, è necessario introdurre un nuovo strumento che semplifichi il recupero dei riferimenti alle risorse e ne garantisca l'integrazione senza intervento manuale. Per questo motivo, è stato necessario adottare **Terragrunt**, che consente una gestione più efficiente e automatizzata delle dipendenze tra ambienti, riducendo il rischio di errori e ottimizzando il provisioning dell'infrastruttura.

Capitolo 4

Soluzione proposta

4.1 Introduzione

Nella gestione dell'infrastruttura con Terraform, abbiamo individuato alcune limiti legate alla ripetitività delle operazioni, alla gestione manuale dei riferimenti tra infrastrutture e alla necessità di eseguire più comandi separati per garantire un corretto deployment. Questi fattori aumentano il rischio di errori e rendono il processo meno efficiente.

Inoltre, quando si gestiscono infrastrutture su AWS con Terraform, spesso si deve affrontare la sfida della gestione multi-ambiente. Un classico esempio di organizzazione di un progetto Terraform prevede diverse cartelle per ogni infrastruttura, come visto in capitoli precedenti dove ogni infrastruttura viene replicato con delle variazioni di configurazione.

Tuttavia, con la crescita dell'infrastruttura, la duplicazione del codice tra ambienti diversi diventa difficile da mantenere e soggetta a errori. Anche l'uso di moduli Terraform aiuta a ridurre la ripetizione, ma la configurazione dei moduli stessi, l'impostazione delle variabili e la gestione backend dello stato remoto possono diventare un onere significativo.

Per risolvere queste difficoltà, si propone l'uso di **Terragrunt** che automatizza e ottimizza la gestione dell'infrastruttura. Inoltre permette di ridurre la duplicazione del codice, gestire in modo centralizzato la configurazione degli ambienti e semplificare il flusso di deploy.

4.2 Terragrunt

Terragrunt è un wrapper per Terraform/OpenTofu progettato per migliorare la gestione di infrastrutture multi-ambiente, eliminando la necessità di ripetere codice e semplificando la configurazione. A differenza di Terraform, che richiede la gestione manuale di ambienti separati con codice duplicato, Terragrunt permette di centralizzare la configurazione, automatizzare il backend e applicare impostazioni comuni a tutti gli ambienti, riducendo il rischio di errore e facilitando la manutenzione. Il suo obiettivo principale è mantenere il codice **DRY (Don't Repeat Yourself)**, un principio di sviluppo software che mira a ridurre la duplicazione del codice per renderlo più facile da mantenere e meno soggetto a errori. Consideriamo, ad esempio, una tipica struttura di progetto Terraform con ambienti multipli, in cui ogni ambiente (prod, qa, stage) ha la stessa infrastruttura composta da un'applicazione, un database MySQL e una VPC. Senza strumenti aggiuntivi, la gestione separata di questi ambienti porta a una ripetizione significativa del codice, aumentando la complessità e il rischio di inconsistenze. Terragrunt aiuta a risolvere questo problema centralizzando la configurazione, automatizzando la gestione dello stato remoto e applicando impostazioni comuni per massimizzare il riutilizzo del codice e ridurre il sovraccarico di manutenzione.

4.2.1 Caratteristiche di Terragrunt

Terragrunt utilizza un file di configurazione chiamato `terragrunt.hcl` che permette di:

- **Centralizzare la configurazione**: Definire una struttura di cartelle più pulita e modulare.
- **Riutilizzare il codice**: Eliminare la necessità di duplicare i file Terraform per ogni ambiente.
- **Gestire automaticamente lo stato remoto**: Configurare un backend S3 e un meccanismo di locking centralizzato senza doverlo ripetere in ogni modulo.
- **Applicare politiche comuni**: Definire impostazioni di sicurezza, provider e altri parametri a livello globale.

Struttura di un progetto con Terragrunt

Con Terragrunt, la struttura del progetto diventa più modulare e organizzata:

```
project-root/
├── terragrunt.hcl
├── live/
│   ├── dev/
│   │   └── terragrunt.hcl
│   ├── dev recovery/
│   │   └── terragrunt.hcl
│   ├── prod/
│   │   └── terragrunt.hcl
│   ├── ...
│   │
│   └── global/
│       └── terragrunt.hcl
└── modules/
    ├── AWS Customer 1/
    │   ├── main.tf
    │   ├── ec2.tf
    │   └── ...
    └── global/
        ├── main.tf
        ├── cloudfront.tf
        └── ...
```

Figura 4.1: Esempio di struttura del progetto integrando Terragrunt.

Invece di replicare tutti i file `.tf`, ogni modulo può essere richiamato da file `terragrunt.hcl`, semplificando la manutenzione e garantendo la coerenza tra ambienti.

4.3 Funzionalità di Terragrunt

Terragrunt offre diverse funzionalità avanzate che semplificano la gestione di infrastrutture multi-ambiente in Terraform. Queste funzionalità permettono di ridurre la duplicazione del codice, migliorare l'automazione e garantire una maggiore coerenza tra gli ambienti. Di seguito, vediamo come Terragrunt affronta alcuni problemi comuni nella gestione dell'infrastruttura.

4.3.1 Keep your backend configuration DRY

Senza Terragrunt, ogni ambiente in Terraform deve definire manualmente la configurazione del backend per il salvataggio dello stato, portando a duplicazione del codice. Ad esempio, nel file `main.tf` di ogni ambiente:

```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state"
    key            = "dev/terraform.tfstate"
    region         = "us-east-1"
    encrypt        = true
    lock_table     = "terraform-lock"
  }
}
```

In questo approccio, ogni ambiente (dev, prod, ecc.) ha una configurazione backend separata, aumentando il rischio di errori e incoerenze.

Con Terragrunt, è possibile centralizzare questa configurazione in un unico file `terragrunt.hcl` nella root del progetto:

```
remote_state {
  backend = "s3"
  config = {
    bucket         = "my-terraform-state"
    key            = "${path_relative_to_include()}/terraform.tfstate"
    region         = "us-east-1"
    encrypt        = true
    lock_table     = "terraform-lock"
  }
}
```

Grazie a questa configurazione centralizzata, tutti i moduli e gli ambienti ereditano automaticamente la stessa configurazione del backend, riducendo il rischio di errori e semplificando la gestione dello stato remoto. Invece di ripetere il backend S3 in ogni file `main.tf`, si può definire una configurazione centrale in un file `terragrunt.hcl` nella root del progetto. Tutti i moduli figlio possono ereditare questa configurazione includendolo senza doverla ripetere.

4.3.2 Keep your Terraform CLI arguments DRY

Terragrunt sostituisce direttamente i comandi di Terraform come apply, plan e destroy, aggiungendo anche funzionalità avanzate per semplificare l'uso di Terraform. Ad esempio, invece di eseguire `terraform apply`, Terragrunt permette di lanciare il comando:

```
terragrunt apply
```

Inoltre, Terragrunt introduce i flag come parametri aggiuntivi che consentono di aggiungere una personalizzare per l'esecuzione dei comandi Terragrunt

```
terragrunt apply --terragrunt-include-dir /path/to/dirs/to/include
```

Anche più volte:

```
terragrunt apply --terragrunt-include-dir /path/to/dirs/to/include
        --terragrunt-include-dir /another/path/to/dirs/to/include
```

4.3.3 Eseguire comandi Terraform su più moduli contemporaneamente

La gestione di più ambienti (dev, prod, qa, uat ecc.) diventa più semplice grazie alla possibilità di definire variabili e configurazioni riutilizzabili.

Con Terragrunt, è possibile eseguire un comando Terraform su tutti i moduli contemporaneamente invece di eseguirlo singolarmente per ciascun modulo. Ad esempio:

```
terragrunt run-all apply
```

Questo comando cerca ricorsivamente le unità terragrunt nell'albero della directory corrente e eseguirà il comando specificato, in questo caso terraform apply, in ordine di dipendenza su tutti i moduli presenti nella struttura del progetto, semplificando notevolmente la gestione dell'infrastruttura.

4.3.4 Dependency Management

Terragrunt offre strumenti per la gestione delle dipendenze tra diversi moduli, garantendo che i componenti dell'infrastruttura vengano applicati nel corretto ordine. Ad esempio, se un'applicazione dipende dalla creazione di un database, Terragrunt si assicurerà che quest'ultimo venga creato prima dell'applicazione stessa.

Se hai un database che deve essere creato prima dell'applicazione, puoi definire questa relazione in `terragrunt.hcl` utilizzando `dependency`:

```
dependency "database" {
  config_path = "../database"

  mock_outputs = {
    db_endpoint = "mock-endpoint"
```

```
  }
}

inputs = {
  database_endpoint = dependency.database.outputs.db_endpoint
}
```

In questo modo, il modulo che rappresenta l'applicazione aspetterà che il database sia creato prima di eseguire la sua configurazione. Chiaramente la risorsa che si intende utilizzare deve essere dichiarato come output nel modulo.

4.4 Vantaggi di Terragrunt

- **Miglior riutilizzo del codice**: Una sola definizione del modulo può essere riutilizzata in più ambienti.
- **Gestione centralizzata dello stato remoto**: Evita errori e incongruenze tra ambienti.
- **Minore rischio di errore umano**: Le configurazioni sono più facili da mantenere e meno soggette a modifiche manuali errate.
- **Maggiore coerenza tra gli ambienti**: Definire impostazioni di sicurezza, provider e altri parametri a livello globale.

4.5 Svantaggi di Terragrunt

Terragrunt si rivela quindi una soluzione potente per chi desidera mantenere un'infrastruttura Terraform più pulita, scalabile e facilmente gestibile, riducendo il carico di manutenzione e migliorando la coerenza tra ambienti. Tuttavia, è importante considerare anche alcuni svantaggi. Ad esempio, l'uso di Terragrunt introduce una curva di apprendimento aggiuntiva, poiché gli utenti devono comprendere sia Terraform che le configurazioni specifiche di Terragrunt. Inoltre, l'adozione di un ulteriore livello di astrazione può rendere il debug più complesso in caso di problemi con l'infrastruttura. L'integrazione di Terragrunt con moduli Terraform esistenti potrebbe non essere sempre fluida, e in alcuni casi potrebbe richiedere modifiche aggiuntive. Inoltre, l'introduzione di un ulteriore strato di configurazione aumenta la complessità, specialmente per infrastrutture meno complesse, dove l'uso di Terragrunt potrebbe risultare eccessivo. Infine, la gestione di molteplici ambienti o progetti paralleli con Terragrunt può diventare difficile se non si seguono best practices, aumentando la possibilità di incoerenze nelle configurazioni.

Capitolo 5

Implementazione della soluzione proposta

5.1 Introduzione

Dopo aver illustrato i problemi della gestione multi-ambiente con Terraform e i vantaggi introdotti da Terragrunt, in questo capitolo descriveremo nel dettaglio l'implementazione della soluzione proposta per integrarla ad un progetto esistente. Analizzeremo la struttura dei moduli, le relazioni tra i componenti e il processo che ha portato alla soluzione finale.

5.2 Processo di Deployment delle Applicazioni con Terraform

Nella configurazione tradizionale con Terraform, ogni ambiente aveva una copia separata dei file Terraform, portando a una gestione ripetitiva e potenzialmente soggetta a errori specialmente nei casi in cui si dovesse utilizzare Terraform per deployare infrastrutture costituite da diversi componenti software in forte relazione tra loro in diversi ambienti.

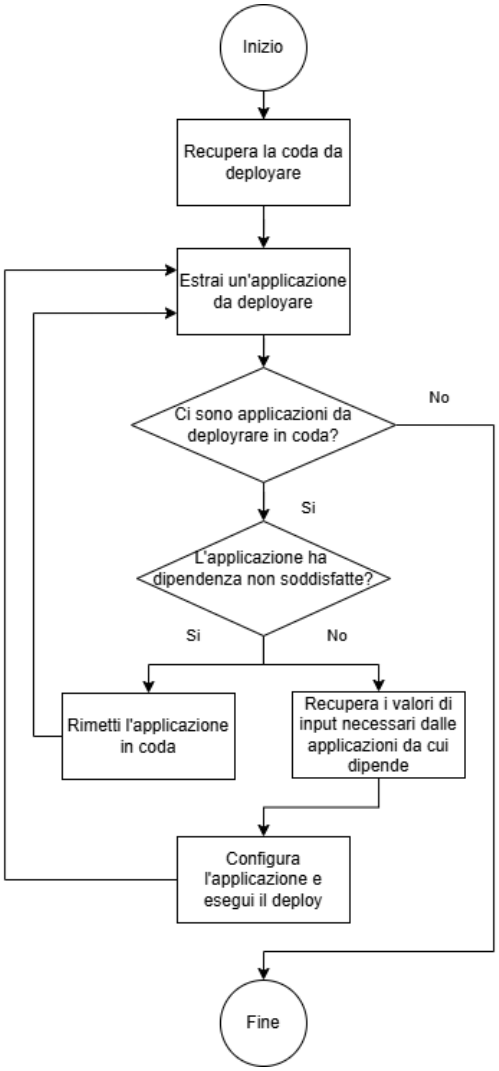


Figura 5.1: Flow chart su deploy delle applicazioni con dipendenze in Terraform

Il deployment di un insieme di applicazioni con Terraform segue un flusso ben definito per garantire che ogni componente venga installato nel momento giusto, rispettando eventuali dipendenze e configurazioni.

Il processo inizia con il recupero della coda delle applicazioni da deployare, ovvero l'insieme delle applicazioni che devono essere installate in un determinato ambiente. Una volta ottenuta la lista delle applicazioni, viene fatto un'estrazione dalla coda. Prima di procedere, è necessario verificare se ci sono applicazioni in attesa nella coda. Se la coda è vuota, il processo si conclude, poiché tutte le applicazioni sono già state installate. Se invece ci sono ancora elementi nella lista, si esamina l'applicazione selezionata per determinare se ha delle dipendenze che non sono ancora state soddisfatte.

Nel caso in cui l'applicazione dipenda da un'altra che non è ancora stata deployata, viene rimessa in coda e si passa alla successiva. Questo assicura che il deployment segua un ordine logico e che nessuna applicazione venga installata prima che i suoi

prerequisiti siano stati soddisfatti. Se invece l'applicazione è pronta per essere deployata, vengono recuperati i valori di input necessari, come configurazioni o parametri derivanti dalle applicazioni da cui dipende. Una volta ottenuti questi valori, Terraform procede alla configurazione e all'esecuzione del deployment.

Questo processo viene ripetuto fino a quando tutte le applicazioni nella coda non sono state installate con successo. In questo modo si evita il rischio di deployare applicazioni nell'ordine sbagliato o senza le informazioni necessarie per il loro corretto funzionamento.

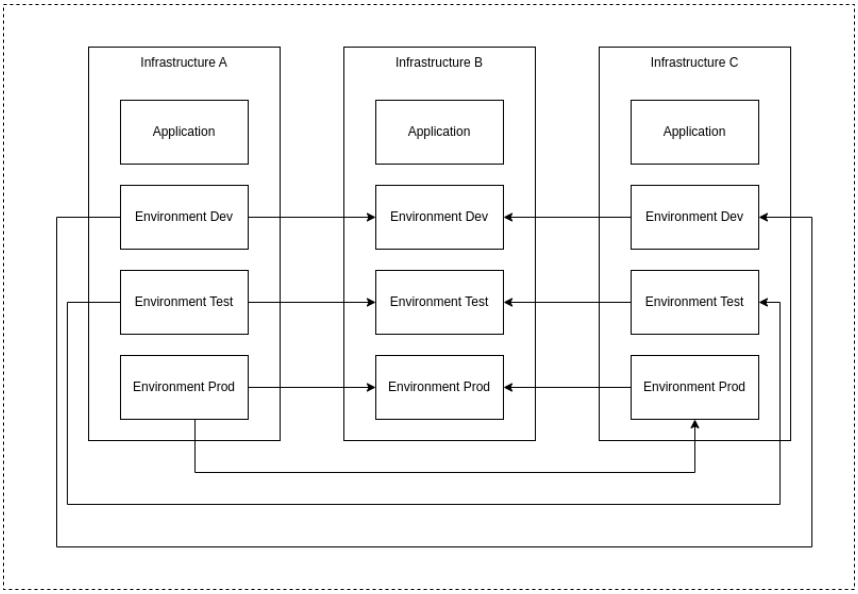


Figura 5.2: Rappresentazione logica di un ambiente deployato con Terraform

Deployment delle Applicazioni e Connessione Manuale

Dopo aver eseguito il deployment delle applicazioni, l'utente può connetterle manualmente tra di loro. Questo avviene attraverso la configurazione dei parametri di input e output di ogni applicazione, in modo che possano comunicare correttamente. Ad esempio, un'applicazione che necessita di un database potrebbe aver bisogno di ricevere come input l'endpoint del database già deployato.

Per gestire questa connessione in Terraform è necessario che l'utente configuri manualmente queste connessioni, modificando i file di configurazione o aggiornando le variabili d'ambiente. Questo può avvenire tramite strumenti di orchestrazione, configurazioni manuali nei file **.tfvars**, oppure tramite sistemi di gestione dei segreti come AWS Secrets Manager o HashiCorp Vault.

5.2.1 Infrastruttura come Entità Autonoma

Ogni infrastruttura creata con Terraform è un'entità a sé stante all'interno dell'ambiente in cui viene deployata. Questo significa che può esistere indipendentemente

dalle altre infrastrutture e che la sua configurazione è definita in modo autonomo. Tuttavia, affinché le diverse infrastrutture possano comunicare tra loro, è necessario stabilire un meccanismo di connessione basato su variabili di input e output. Le variabili di output di una determinata infrastruttura possono essere utilizzate come input per un'altra, creando un collegamento tra le due entità.

5.3 Soluzione alla Dipendenza

Mentre con una soluzione basata su Terragrunt, ogni ambiente è composto da applicazioni già legate tra loro e il deployment avviene con un solo comando, riducendo la complessità della configurazione. Questo approccio permette di gestire le infrastrutture in modo più modulare e organizzato, evitando la necessità di configurare manualmente ogni singola applicazione e stabilire le connessioni tra di esse dopo il deployment.

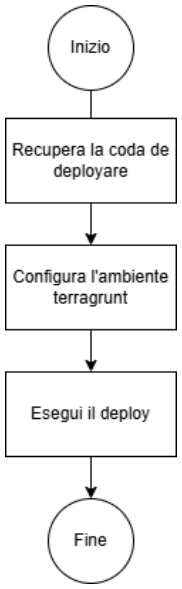


Figura 5.3: Flow chart su deploy delle applicazioni con dipendenze in Terragrunt

Il diagramma fornito illustra un processo di deployment più semplificato rispetto a quello tradizionale con Terraform. Si parte sempre dal recupero della coda delle applicazioni da deployare, ovvero l'insieme delle infrastrutture e dei servizi che devono essere installati. Tuttavia, a differenza dell'approccio precedente, non è necessario verificare e risolvere manualmente le dipendenze tra le applicazioni, perché queste sono già organizzate in un'unica struttura logica.

Dopo aver recuperato la coda, il passaggio successivo consiste nella configurazione dell'ambiente Terragrunt. Qui entra in gioco il principale vantaggio di Terragrunt: invece di dover definire ripetutamente le configurazioni per ogni singola applicazione,

è possibile gestire la configurazione dell'intero ambiente in modo centralizzato. Terragrunt si occupa di riutilizzare moduli Terraform esistenti, applicando una struttura coerente per tutti gli ambienti, che siano di sviluppo, testing o produzione.

Una volta configurato l'ambiente, il deployment viene eseguito con un unico comando, permettendo di installare tutte le applicazioni in modo orchestrato e senza dover intervenire manualmente per gestire input e output tra le varie infrastrutture. Questo semplifica notevolmente il processo, riducendo il rischio di errori e velocizzando le operazioni.

Alla fine del processo, l'intero ambiente è operativo senza necessità di ulteriori interventi. Grazie a questo approccio, Terragrunt rende il deployment più scalabile e meno soggetto a problemi legati alla gestione delle dipendenze tra le applicazioni, facilitando l'amministrazione delle infrastrutture cloud in ambienti complessi.

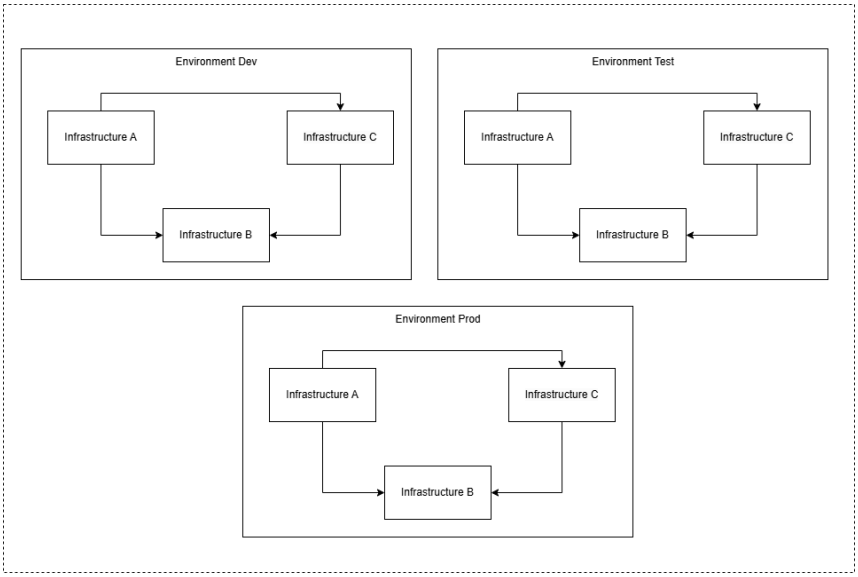


Figura 5.4: Rappresentazione logica di un ambiente deployato con Terragrunt

5.4 Implementazione di Terragrunt: Fasi e Strategie

Per evolvere le modalità di deploy delle infrastrutture utilizzando Terragrunt è stato necessario:

1. Revisionare le infrastrutture per renderle dei moduli riutilizzabili.
2. Risolvere dipendenze tra i moduli.
3. Predisporre delle regole che favoriscano l'integrazione dei moduli.

5.5 Realizzazione moduli applicativi

L'adozione di Terragrunt ha richiesto di riorganizzare l'infrastruttura in moduli Terraform riutilizzabili, separando le risorse per favorire la modularità e la scalabilità. Ogni infrastruttura viene gestito come un modulo indipendente dall'ambiente in cui verrà deployato e salvato in repository git o una registry Terraform al fine di poterne gestire il versionamento. Quest'ultimo è fondamentale ai fini della tracciabilità delle dipendenze tra moduli.

La nuova organizzazione delle infrastrutture in moduli richiede la presenza di 3 elementi principali:

- **Definizione delle risorse Terraform**: contenente la configurazione delle risorse AWS.
- **Variabili di configurazione** (variable.tf): per rendere il modulo parametrizzabile
- **Output** (output.tf): per esportare gli parametri delle risorse da utilizzare in altri moduli o anche semplicemente come output.

Le infrastrutture definite devono essere organizzate in moduli separati i quali vengono poi istanziati nei diversi ambienti. In questa configurazione, le definizioni delle risorse vengono collocate all'interno di una directory dedicata ai moduli, mentre le configurazioni specifiche degli ambienti vengono gestite separatamente in un'altra directory. Questo approccio permette di mantenere una chiara separazione tra la logica infrastrutturale e le configurazioni ambientali, facilitando la gestione multi-ambiente e riducendo la duplicazione del codice.

5.6 Risoluzione dipendenze tra i moduli

Uno dei principali ostacoli nella gestione delle infrastrutture con Terraform è rappresentato dalle dipendenze tra i moduli. In un'architettura complessa, i vari componenti dell'infrastruttura devono interagire tra loro, e questa interconnessione può generare vincoli che devono essere risolti affinché il deployment avvenga senza errori.

In particolare, si può incorrere in dipendenze cicliche, ossia situazioni in cui due moduli dipendono reciprocamente l'uno dall'altro. Un esempio tipico si verifica quando il modulo A necessita di un valore generato dal modulo B, mentre B, a sua volta, ha bisogno di un valore derivante da A. Questo loop impedisce a Terraform di stabilire un ordine corretto di esecuzione, bloccando il deployment.

Per risolvere le dipendenze cicliche tra i moduli in Terragrunt, si possono adottare due strategie:

1. **Sostituire gli output con valori costruiti**: Se la dipendenza tra i due moduli riguarda un valore che può essere determinato a priori in base a una convenzione nota, è possibile sostituire l'output del modulo dipendente con una stringa costruita manualmente. Ad esempio, se il modulo B richiede la variabile `s3_module_a_id` dal modulo A, ma quest'ultimo a sua volta dipende da `s3_module_b_id` del modulo B, si può intervenire costruendo manualmente il valore del bucket S3 in base a una formula predefinita. Tuttavia, questa soluzione non è sempre applicabile, poiché non tutte le risorse possono essere derivate in modo deterministico.

2. **Isolare le risorse critiche in un modulo separato**: Un'altra strategia più strutturata consiste nel rimuovere le risorse che generano la dipendenza ciclica e spostarle in un modulo indipendente, che non appartiene né a A né a B. In questo modo, i moduli A e B potranno dipendere da questo modulo centrale senza introdurre cicli tra loro. Ad esempio, se il problema riguarda la creazione di un bucket S3 condiviso, invece di gestirlo nei moduli A e B, si può creare un modulo separato S3_common, che fornisce l'output necessario ad entrambi.

5.7 Integrazione tra moduli

L'integrazione tra moduli è un aspetto fondamentale nella costruzione di un'infrastruttura modulare, poiché consente alle diverse componenti di comunicare tra loro in modo strutturato. In questo contesto, quando un'applicazione A utilizza funzionalità fornite da un'altra applicazione B, è necessario che il modulo A abbia accesso ai riferimenti di B o addirittura possa utilizzare risorse create nell'infrastruttura di B.

L'integrazione avviene tramite il passaggio di variabili di input e output tra i moduli. Se il modulo A è progettato correttamente, avrà delle variabili di input i cui valori dipendono dagli output di B. Analogamente, B definirà **esplicitamente gli output** necessari affinché possano essere utilizzati dagli altri moduli.

Un esempio pratico può riguardare un'applicazione web deployata su AWS che necessita di un database gestito da un altro modulo. Supponiamo di avere due moduli:

- **Modulo A**: Applicazione web
- **Modulo B**: Database RDS

Per far sì che l'applicazione web possa connettersi al database, è necessario che A riceva come input l'endpoint del database creato da B. In Terraform, questo viene realizzato con il seguente approccio:
Nel modulo B (Database RDS) definiamo l'output dell'endpoint:

```
output "rds_endpoint" {
  value = aws_db_instance.my_database.endpoint
}
```

Nel modulo A (Applicazione Web), definiamo una variabile di input:

```
variable "database_endpoint" {}

resource "aws_ecs_task_definition" "app" {
  container_definitions = jsonencode([
    {
      name  = "web-app"
      image = "my-app-image"
      environment = [
        {
          name  = "DATABASE_URL"
          value = var.database_endpoint
        }
      ]
    }
  ])
}
```

E successivamente, nella configurazione di Terragrunt, colleghiamo i due moduli:

```
dependency "database" {
  config_path = "../database"
}

inputs = {
  database_endpoint = dependency.database.outputs.rds_endpoint
}
```

Questo approccio garantisce che l'applicazione web possa ricevere dinamicamente l'endpoint del database senza che l'utente debba configurarlo manualmente.

Con l'approccio classico di Terraform, invece, questa integrazione veniva gestita manualmente recuperando gli output di B e inserendoli come variabili di input in A. Questo comportava un maggiore sforzo nella gestione delle configurazioni, aumentando il rischio di errori e rendendo il deployment meno scalabile.

Grazie all'uso di Terragrunt, il legame tra i moduli viene costruito in modo automatico, migliorando la coerenza dell'infrastruttura e facilitando la gestione multi-ambiente.

Capitolo 6

Valutazione sperimentale della soluzione proposta

In questo capitolo viene presentata un'analisi sperimentale della soluzione proposta, valutandone il comportamento attraverso casi d'uso concreti. L'obiettivo è verificare l'efficacia dell'approccio adottato rispetto ai problemi individuati nella gestione delle infrastrutture cloud.

A tal fine, sono state implementate e testate diverse architetture su ambienti distinti utilizzando due approcci: il metodo classico con Terraform e l'approccio basato su Terragrunt.

6.1 Deploy della struttura con Terraform

L'approccio tradizionale con Terraform ha richiesto la definizione di moduli per ogni componente dell'infrastruttura, garantendo riutilizzabilità e modularità. Ogni ambiente è stato configurato separatamente, replicando il codice Terraform con parametri distinti per ogni contesto operativo.

Il processo di deploy è stato gestito con i seguenti passi:

1. **Deploy dello strato middleware**: in ogni account AWS è stato prima necessario effettuare il deployment dei componenti middleware, che includono elementi come il Certificate Manager, Load Balancer e il routing con Route 53.

Questo processo è stato eseguito separatamente per ogni ambiente, garantendo che le risorse fossero disponibili prima della fase successiva.

2. **Recupero degli output del middleware**: una volta completato il deployment del middleware, è stato necessario estrarre manualmente gli output generati, come gli endpoint, gli ID delle risorse e le configurazioni di sicurezza.

3. **Deploy dello strato applicativo**: successivamente, si è proceduto al deployment dello strato applicativo (Application o AI Developer Resource). Dato che queste risorse dipendono direttamente dal middleware, i valori recuperati manualmente sono stati inseriti come input nelle configurazioni Terraform dello strato applicativo.

4. **Ripetizione per ogni account**: questo processo è stato ripetuto per ogni account AWS coinvolto, con la necessità di gestire separatamente gli stati Terraform e di garantire la coerenza tra le configurazioni dei vari ambienti.

Questo metodo ha evidenziato alcune difficoltà nella gestione multi-ambiente, come la necessità di mantenere aggiornati più file di configurazione e la gestione complessa dello stato.

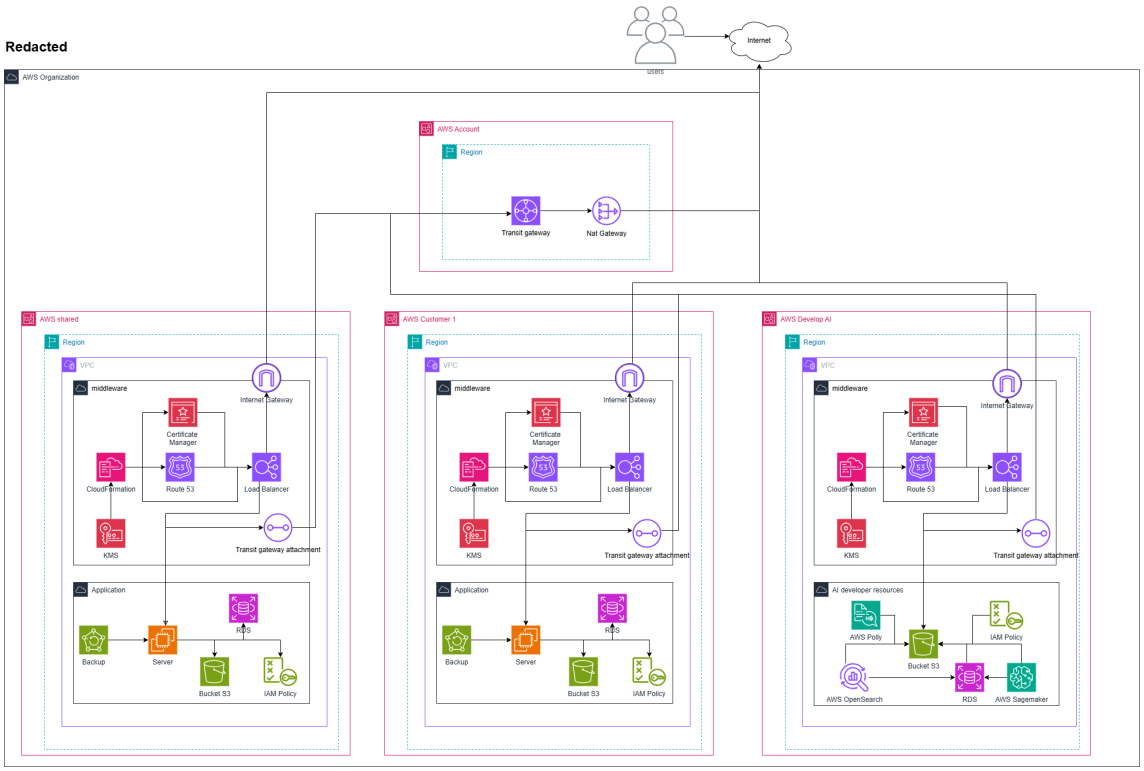


Figura 6.1: Progetto Redacted

L'architettura rappresentata nel diagramma mostra un'infrastruttura AWS suddivisa in più account e ambienti.

AWS Shared

L'ambiente **AWS Shared** contiene i servizi condivisi tra tutti gli altri account, fornendo una base comune per la gestione dell'infrastruttura. Questa ambiente include un **Internet Gateway** per la connessione esterna, una **VPC** personalizzato per l'ambiente, **Route 53** per la gestione del DNS, un **Load Balancer** per il bilanciamento del traffico e **Certificate Manager** per la gestione dei certificati SSL. L'automazione viene gestita attraverso **CloudFormation**, mentre la protezione dei dati sensibili è garantita da **KMS (AWS Key Management Service)**. Le risorse di backend includono server, backup e bucket S3, con accesso regolato da **policy IAM**. Inoltra, la comunicazione tra gli ambienti avviene tramite il **Transit Gateway**, che assicura connettività sicura e scalabile.

AWS Customer 1

L'ambiente **AWS Customer 1** è simile all'ambiente condiviso, ma è dedicato a un cliente specifico. Questo ambiente dispone di un proprio VPC, un Load Balancer, Route 53 per la gestione del DNS e CloudFormation per l'automazione dell'infrastruttura. La gestione delle chiavi è affidata a **KMS**, mentre le risorse applicative comprendono server, backup, bucket S3 e policy IAM. Anche in questo caso, la connettività con l'ambiente esterna è gestita tramite il Transit Gateway.

AWS Develop AI

L'ambiente **AWS Develop AI** è progettato per supportare applicazioni di intelligenza artificiale e machine learning. Oltre alla struttura del middleware, comune degli altri ambienti, include servizi specializzati come **AWS Polly, OpenSearch e SageMaker** insieme ad un **RDS (Relational Database Service)**. Questo ambiente utilizza un'infrastruttura simile a quella degli altri account, con middleware per la gestione del traffico, gestione DNS e bilanciamento del carico. Le risorse AI sono collegate tramite Transit Gateway, garantendo una comunicazione fluida tra gli ambienti e facilitando l'integrazione con gli altri servizi dell'infrastruttura.

Questa suddivisione permette di isolare i servizi condivisi da quelli specifici per ogni cliente o ambiente di sviluppo, migliorando la sicurezza, facilitando la scalabilità e ottimizzando la gestione delle risorse cloud.

Struttura cartelle Terraform

L'organizzazione del progetto Terraform in file segue una struttura, illustrata in figura, che viene replicata per soddisfare le diverse esigenze dei clienti.

```
Middleware
└── environment
    ├── AWS Shared
    │   ├── main.tf
    │   ├── middleware.auto.tfvars
    │   ├── variables.tf
    │   └── outputs.tf
    ├── AWS Customer 1
    │   ├── main.tf
    │   ├── middleware.auto.tfvars
    │   ├── variables.tf
    │   └── outputs.tf
    ├── AWS Develop AI
    │   ├── main.tf
    │   ├── middleware.auto.tfvars
    │   ├── variables.tf
    │   └── outputs.tf
    └── application
        ├── main.tf
        ├── security-groups.tf
        ├── load-balancer.tf
        └── ...
```

(a) Struttura di Middleware.

```
Application
├── environment
│   ├── AWS Shared
│   │   ├── main.tf
│   │   ├── application.auto.tfvars
│   │   ├── variables.tf
│   │   └── outputs.tf
│   ├── AWS Customer 1
│   │   ├── main.tf
│   │   ├── application.auto.tfvars
│   │   ├── variables.tf
│   │   └── outputs.tf
│   └── AWS Develop AI
│       ├── main.tf
│       ├── application.auto.tfvars
│       ├── variables.tf
│       └── outputs.tf
└── application
    ├── main.tf
    └── ...
```

(b) Struttura di Application.

Figura 6.2: Struttura del progetto Terraform in cartelle

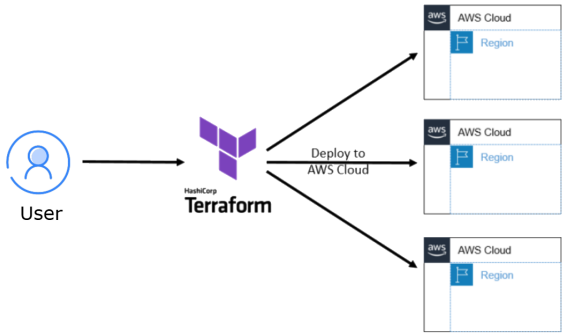


Figura 6.3: Approccio tradizionale dell'utente con Terraform

6.2 Deploy della struttura con Terragrunt

L'adozione di Terragrunt ha permesso di semplificare e automatizzare molte delle complessità riscontrate nell'approccio tradizionale con Terraform. Grazie alla sua capacità di gestire la logica di infrastruttura a livelli superiori e di automatizzare il passaggio di variabili tra moduli, il processo di deploy è stato notevolmente ottimizzato.

I principali passi seguiti con Terragrunt sono stati:

1. **Organizzazione dei moduli in una struttura gerarchica**: Terragrunt ha permesso di strutturare i moduli Terraform in un'unica gerarchia logica, evitando la duplicazione di codice tra ambienti e account diversi.

2. **Deploy automatizzato dello strato middleware**: anziché eseguire il deploy manualmente per ogni ambiente, Terragrunt ha consentito di eseguire l'infrastruttura middleware in modo centralizzato, recuperando automaticamente gli output necessari.

3. **Passaggio automatico di variabili tra gli strati**: gli output del middleware sono stati resi disponibili direttamente per lo strato applicativo, eliminando la necessità di inserirli manualmente.

4. **Deploy dello strato applicativo con dipendenze risolte**: una volta completato il deploy del middleware, le applicazioni (Application o AI Developer Resource) sono state distribuite senza la necessità di recuperare manualmente gli output.

5. **Gestione automatica degli stati Terraform**: grazie all'uso di backend remoti condivisi, Terragrunt ha migliorato la gestione separata degli stati Terraform per ogni ambiente, semplificando l'orchestrazione e riducendo il rischio di incoerenze.

Grazie a questo approccio, è stato possibile ridurre significativamente il tempo necessario per il deployment delle infrastrutture su più ambienti e garantire una maggiore coerenza tra le configurazioni.

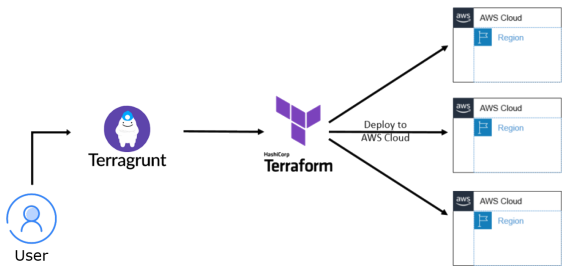


Figura 6.4: Nuovo approccio dell'utente con Terragrunt

6.3 Gestione dello stato con Terragrunt

Una delle principali semplificazioni introdotte da Terragrunt riguarda la gestione dello stato Terraform. In questo progetto, è stato adottato un approccio centralizzato per la configurazione dello stato, sfruttando un file `terragrunt.hcl` posizionato nella root della gerarchia di configurazione.

Il file `terragrunt.hcl` a livello di root contiene le configurazioni comuni a tutte le parti dell'infrastruttura, inclusa la definizione del backend per lo stato Terraform. Ad esempio, se lo stato viene archiviato in un bucket S3, la configurazione potrebbe essere simile alla seguente:

```
remote_state {
  backend = "s3"
  generate = {
    path = "backend.tf"
    if_exists = "overwrite_terragrunt"
  }

  config = {
    bucket = "my-terraform-state-bucket"
    key = "path/to/statefile/${path_relative_to_include}/tf.tfstate"
    region = "eu-west-1"
    encrypt = true
    dynamodb_table = "terraform-lock"
  }
}
```

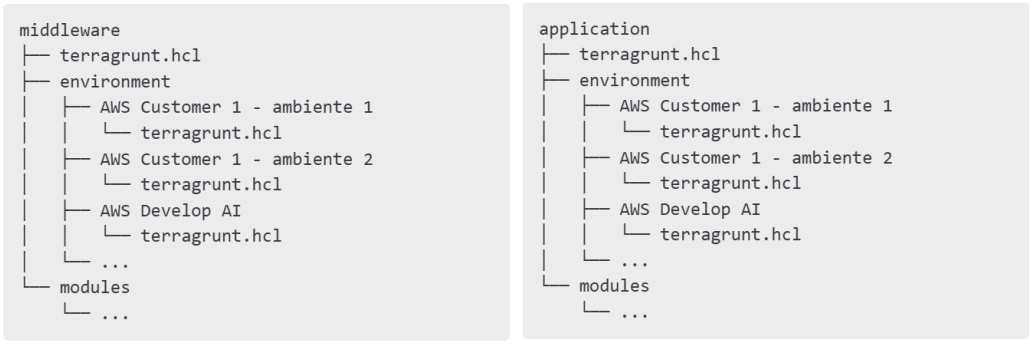
Essendo definito a livello di root, ogni modulo e ambiente che include questa configurazione erediterà automaticamente la gestione dello stato, evitando di dover specificare manualmente il backend in ogni singolo modulo. Questo approccio garantisce che tutti gli ambienti utilizzino una configurazione coerente per la gestione dello stato e previene problemi di inconsistenza e perdita di dati.

Grazie a questa configurazione centralizzata, Terragrunt permette di:

- **Evitare duplicazioni**: tutti i moduli condividono le stesse impostazioni di backend senza necessità di ripeterle.
- **Migliorare la sicurezza**: lo stato viene archiviato in un backend remoto e protetto, con meccanismi di blocco per prevenire modifiche concorrenti.
- **Facilitare la gestione multi-ambiente**: ogni ambiente può utilizzare lo stesso stato centralizzato o avere configurazioni specifiche se necessario.

6.4 Struttura Dopo l'Adozione di Terragrunt

L'introduzione di Terragrunt permette di centralizzare la gestione della configurazione e di rendere più modulare l'infrastruttura:



(a) Struttura di Middleware. (b) Struttura di Application.

Figura 6.5: Struttura del progetto in cartelle integrando Terragrunt

Questi due domini utilizzano Terraform per definire l'infrastruttura e Terragrunt per gestire le configurazioni e il riutilizzo del codice in ambienti multipli.

Nella cartella **modules** sono definiti i moduli Terraform, i quali fanno riferimento a sorgenti esterne di versioning (come repository Git) per garantire un migliore controllo delle versioni e una maggiore riutilizzabilità.

I file `terragrunt.hcl` presenti nei vari ambienti ereditano le configurazioni di base definite nel file `terragrunt.hcl` **root** solo se sono esplicitamente inclusi, evitando ripetizioni e semplificando la gestione multi-ambiente.

6.5 Diagrammi di Struttura

Per comprendere meglio le differenze tra le due architetture, possiamo rappresentarle con diagrammi.

6.5.1 Struttura Modulare Prima della Soluzione

In questa rappresentazione, ogni ambiente è gestito separatamente e richiede la definizione esplicita dei moduli e delle risorse.

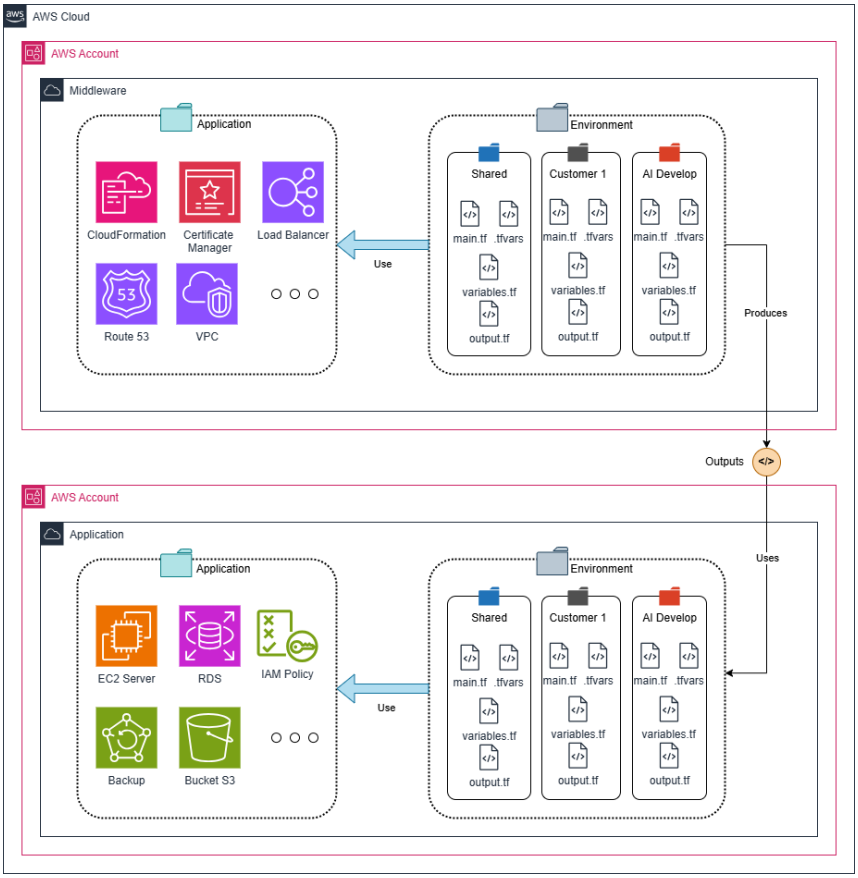


Figura 6.6: L'architettura del progetto con solo Terraform

Questa architettura multi-account su AWS consente una chiara separazione tra gestione infrastrutturale e applicazioni, migliorando la sicurezza e l'organizzazione delle risorse. Il primo account agisce come un livello di gestione centralizzato, mentre il secondo si concentra sull'elaborazione e l'archiviazione dei dati. Grazie a servizi come Load Balancer, Route 53 e KMS, è possibile garantire un'infrastruttura scalabile, sicura e ben organizzata, con policy di accesso rigorose e una forte integrazione tra i due ambienti.

6.5.2 Struttura Modulare Dopo l'Adozione di Terragrunt

Dopo l'ottimizzazione, gli ambienti condividono le stesse configurazioni e dipendono da un'unica sorgente.

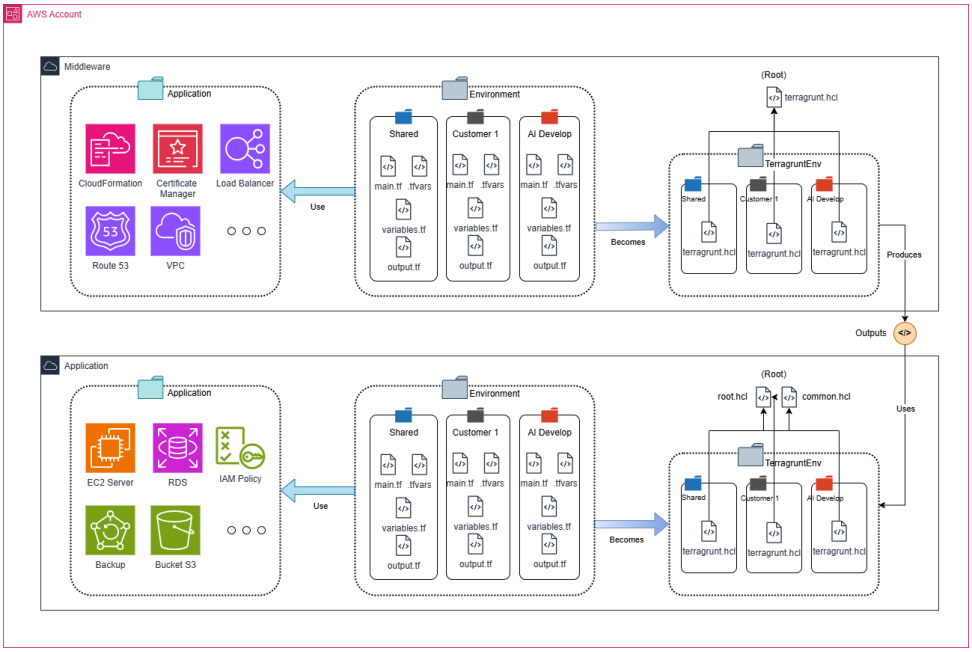


Figura 6.7: L'architettura del progetto con l'introduzione di Terragrunt

L'architettura illustrata nel diagramma rappresenta un'evoluzione della gestione dell'infrastruttura AWS attraverso l'adozione di **Terragrunt** come strumento di gestione per Terraform. Ogni ambiente utilizza un file `terragrunt.hcl`, che richiama i moduli definiti nella cartella `modules`. Questo approccio riduce drasticamente la duplicazione del codice e facilita la gestione delle configurazioni.

Il file `terragrunt.hcl` o `root.hcl` e `common.hcl` posto nella posizione più esterna contiene le configurazioni comuni per tutti gli ambienti, tra cui la definizione del provider e la configurazione del backend. Grazie a questo file, è possibile centralizzare le impostazioni condivise, garantendo coerenza e semplificando la gestione dell'infrastruttura.

6.6 Risoluzione dipendenze

Terragrunt semplifica la gestione delle dipendenze utilizzando i blocchi di dipendenza (`dependency`). In questo caso, possiamo definire la relazione tra i due moduli direttamente nei file di configurazione di Terragrunt.

Modulo A (Middleware)

Nel modulo A definiamo l'output dell'id della vpc:

```
output "vpc_id" {
  value = try(module.vpc[0].vpc_id, null)
}
```

Modulo B (Application)

Nel modulo B, invece di inserire manualmente l'id della VPC, possiamo definire una variabile di input:

```
variable "vpc_id" {}
```

Configurazione di Terragrunt

Nella configurazione di Terragrunt, specifichiamo la dipendenza del modulo B dal modulo A usando il blocco `dependency` e nell'input definiamo lo stesso nome definito per variabile:

```
dependency "customer_1" {
  config_path = "/path/to/module A/customer_1-middleware"
}

inputs = {
  vpc_id = dependency.customer_1.outputs.vpc_id
}
```

In questo modo sarà possibile accedere non solo al VPC id del modulo A, ma anche ad altri output definiti nello stesso modulo.

Se il modulo dell'infrastruttura **non contiene già l'output richiesto** e **non si vuole modificare il modulo direttamente**, è possibile **generare nuovi output al momento del deployment** utilizzando il file terragrunt.hcl nel modulo A.

Ad esempio, se il modulo Terraform di A non fornisce la VPC id come output, possiamo dichiararlo direttamente.

```
generate "extra_outputs" {
  path      = "generated_outputs.tf"
  if_exists = "overwrite_terragrunt"
  contents  = <<EOF
  output "vpc_id" {
    value = try(module.vpc[0].vpc_id, null)
  }
  EOF
}
```

In questo modo, senza modificare il modulo originale, **Terragrunt genera automaticamente un nuovo file di output** che può essere utilizzato dagli altri moduli. Questa soluzione è utile quando si lavora con moduli gestiti esternamente o versionati in un repository centrale. Quando viene lanciato modulo B, questo attenderà che venga prima deployato modulo A per poi recuperare i valori di output necessari per l'esecuzione.

6.7 Relazioni tra i Componenti

Un altro aspetto chiave dell'implementazione è la gestione delle relazioni tra i componenti infrastrutturali.

- **Applicazione → Moduli**: L'applicazione non definisce più direttamente le risorse, ma richiama i moduli predefiniti.
- **Moduli → Ambienti**: Gli ambienti istanziano i moduli specificandone i parametri attraverso `terragrunt.hcl`.
- **Gestione delle dipendenze**: Con la `dependency` in Terragrunt, è possibile stabilire l'ordine di esecuzione tra i vari componenti tramite l'utilizzo di un Grafo Aciclico Diretto (DAG).

Capitolo 7

Conclusioni e sviluppi futuri

Nel presente lavoro di tesi è stata analizzata l'infrastruttura cloud multi-account su AWS e l'automazione del deployment tramite l'uso di Terraform e Terragrunt. Sono stati implementati, testati e confrontati entrambi gli approcci, con particolare attenzione ai vantaggi offerti da Terragrunt nella gestione multi-ambiente e nella riduzione della complessità operativa.

Grazie all'adozione di Terragrunt, è stato possibile ottenere una gestione più strutturata delle configurazioni, riducendo significativamente la duplicazione del codice e centralizzando la gestione dello stato di Terraform. Questo ha comportato una riduzione dell'80% nei tempi di deployment per ambienti contenenti fino a 60 applicazioni, eliminando completamente gli errori derivanti dal collegamento manuale tra le applicazioni.

Vantaggi ottenuti

L'adozione di Terragrunt ha portato numerosi benefici concreti:

- **Riduzione dell'80% dei tempi di deployment**: il tempo necessario per configurare e distribuire l'infrastruttura si è drasticamente ridotto, aumentando l'efficienza operativa.
- **Eliminazione del 100% degli errori di collegamento tra applicazioni**: la gestione centralizzata ha evitato configurazioni errate o incoerenze tra ambienti.

- **Migliore organizzazione del codice**: grazie alla riduzione della duplicazione e alla strutturazione delle configurazioni, il codice è più leggibile e mantenibile.
- **Scalabilità migliorata**: la soluzione proposta consente di gestire più ambienti e applicazioni in modo più efficace.

7.1 Miglioramenti e sviluppi futuri

Nonostante i risultati ottenuti, ci sono diversi aspetti che potrebbero essere migliorati ed estesi per rendere la soluzione ancora più efficiente:

1. **Automazione della configurazione post-deployment**: Attualmente, dopo il deployment dell'infrastruttura, alcune configurazioni devono essere effettuate manualmente. Un possibile sviluppo futuro prevede l'integrazione di **hooks** in Terragrunt per automatizzare la configurazione post-deployment, ad esempio attraverso script che configurano i servizi appena creati.
2. **Integrazione con pipeline CI/CD avanzate**: Sebbene Terragrunt abbia semplificato la gestione dell'infrastruttura, un ulteriore passo avanti sarebbe l'integrazione con pipeline CI/CD più avanzate, per permettere una gestione end-to-end dell'infrastruttura e delle applicazioni in modo completamente automatizzato.
3. **Migliore gestione della sicurezza**: L'uso di AWS IAM e KMS ha migliorato la protezione delle credenziali e dei dati, ma è possibile rafforzare ulteriormente la sicurezza implementando meccanismi di **scansione statica del codice Terraform** e **controlli di conformità automatizzati**.
4. **Monitoraggio e logging avanzato**: L'infrastruttura potrebbe beneficiare di un'integrazione più stretta con strumenti di monitoraggio come AWS CloudWatch, Prometheus o Grafana, per ottenere una maggiore visibilità sulle performance e lo stato dei servizi deployati.

Questa tesi ha dimostrato che l'adozione di Terragrunt per la gestione delle infrastrutture cloud porta significativi vantaggi in termini di automazione, riduzione degli errori e ottimizzazione dei tempi di deployment. Tuttavia, vi sono ancora ampie possibilità di miglioramento e innovazione, che potranno essere esplorate nelle future evoluzioni di questo progetto.

Bibliografia

[1] HashiCorp, *Terraform Documentation*,
https://developer.hashicorp.com/terraform/docs.

[2] Gruntwork, *Terragrunt Documentation*,
https://terragrunt.gruntwork.io/docs/.

[3] Gruntwork, *Terragrunt - Hooks*
https://terragrunt.gruntwork.io/docs/features/hooks/

[4] AWS, *AWS Well-Architected Framework*,
https://aws.amazon.com/architecture/well-architected/.

[5] AWS, *AWS Best Practices for Infrastructure as Code*,
https://docs.aws.amazon.com/wellarchitected/latest/framework/
best-practices-infrastructure-as-code.html.

[6] HashiCorp, *Terraform Best Practices*,
https://developer.hashicorp.com/terraform/tutorials.

[7] AWS Community, *Terragrunt for Multi-Region/Multi-Account Deployments*,
https://community.aws/content/2gAzYFJaAXQMH44XzQbV7wMCY8R/
terragrunt-for-multi-region-multi-account-deployments.

[8] DEV Community, *Multi-account AWS deployments with Terragrunt*,
https://dev.to/aws-builders/
multi-account-aws-deployments-with-terragrunt-4kod.

[9] Krishna Wattamwar *Terragrunt vs Terraform: DevOps tools*
https://www.linkedin.com/pulse/
terragrunt-vs-terraform-devops-tools-krishna-wattamwar-agzbf/

[10] GitLab, *Introduction to GitLab Flow*,
https://docs.gitlab.co.jp/ee/topics/gitlab_flow.html