Università degli studi di Modena e Reggio Emilia
Dipartimento di Scienze Fisiche, Informatiche e Matematiche

Corso di Laurea in Informatica

Progettazione e sviluppo di infrastruttura cloud su AWS

Relatrice:
Prof.ssa Claudia Canali

Candidato:
Dario Bizzarri

Anno Accademico 2024/2025

Elenco delle figure

Indice

Capitolo 1

Introduzione

Lo sviluppo di software moderno richiede spesso la presenza di numerosi componenti che devono interagire tra di loro, e ciascuno di questi componenti comporta necessità ulteriori in fatto di gestione e allocamento di risorse. Una delle soluzioni che sono emerse negli ultimi decenni per arginare la problematica sono i servizi cloud, gestiti dai rispettivi fornitori e studiati per sollevare l'utente dalla maggior parte delle responsabilità che quest'ultimo avrebbe dovuto altrimenti assumersi.

Uno degli esempi più diffusi di servizio cloud è costituito dai servizi di database, pensati per assolvere l'utente da compiti quali gestione di backup e crittografia, in modo da potersi concentrare unicamente sulla realizzazione dei suoi progetti. Tuttavia, la scelta di un servizio di database in cloud non deve essere presa alla leggera, in quanto affidare i dati della propria applicazione a un servizio di terze parti è una scelta di design che può avere dei rischi imprevisti.

È il caso di Turso, servizio di database basato su SQLite, che a inizio 2025 ha riportato un incidente che ha coinvolto alcuni nodi della loro nuova infrastruttura, realizzata in Amazon Web Services (AWS). A causa di questo incidente, i dati di questi nodi sono andati irreparabilmente persi, causando disagi agli utenti colpiti e danneggiando la reputazione del servizio.

La domanda sorge spontanea, dunque: in che modo realizzare un'infrastruttura robusta e a prova di incidente per un servizio di importanza critica come un provider di database? In questa tesi verrà trattato il principale competitor di Turso, ossia il servizio SQLite Cloud, che già da un anno e mezzo prima dell'incidente sopracitato ospita migliaia di nodi senza perdite di dati.

SQLite Cloud è un sistema di gestione di database relazionali (RDBMS), realizzato dalla società omonima a partire dal motore SQLite e distribuito sotto forma di Platform as a Service (PaaS). Si tratta di un servizio cloud progettato per essere altamente scalabile e performabile, conferendo ai clienti la possibilità di creare uno o più nodi in diverse posizioni geografiche, e organizzarli in cluster per garantire la resilienza dei dati immagazzinati e l'alta affidabilità del servizio stesso.

La società si è in principio rivolta a Polarity, ai tempi chiamata Soluzioni Futura, per la pianificazione e la realizzazione di un'architettura cloud che soddisfacesse tutti i requisiti, utilizzando AWS come principale cloud provider.

Il candidato, ai tempi impiegato presso Polarity in qualità di ingegnere del cloud, è stato assegnato al progetto di SQLite Cloud dall'inizio delle operazioni, e si è trattato del primo progetto presso l'azienda che ha seguito nella sua interezza.

Il candidato ha contribuito allo studio dei requisiti per progettare l'architettura, trovando soluzioni per problemi sorti sia all'inizio della pianificazione che durante, e ha realizzato l'intera infrastruttura tramite tecnologie di Infrastructure as Code (IaC) all'avanguardia come CloudFormation e Pulumi.

La scelta di queste tecnologie non è stata casuale: si è trattato del punto d'incontro tra le conoscenze pregresse di entrambi i team e le necessità tecniche dovute ai requisiti del progetto. Dopo aver valutato attentamente le opzioni, si è deciso di sfruttare Pulumi come principale motore di Infrastructure as Code, e integrare alcuni template modulari in CloudFormation all'interno dell'infrastruttura sviluppata tramite Pulumi, così da poter trarre vantaggio dai template pronti all'uso di proprietà di Polarity.

Essendo un servizio a infrastruttura dinamica, ovvero dove nuovi componenti di infrastruttura sarebbero stati creati su richiesta dei clienti, era inoltre necessario sviluppare un'interfaccia che permettesse al servizio di comunicare con l'infrastruttura e espanderla. A tal scopo il candidato ha sviluppato un modulo scritto in Go, il linguaggio nel quale era stato sviluppato il backend del servizio. Questo modulo definisce una serie di funzioni che, tramite il Software Development Kit (SDK) di Pulumi, svolgono operazioni granulari di espansione dell'infrastruttura, come creazione o eliminazione di un nuovo nodo.

Ulteriori necessità del progetto sono state la configurazione personalizzata di un container Docker per Traefik, il load balancer scelto per reindirizzare il traffico ai nodi, lo sviluppo di un plugin Docker personalizzato per gestire il montaggio dei volumi, a seguito dell'End of Life di Amazon Linux 2, e varie operazioni di migrazione, reingegnerizzazione e rivisitazione dei requisiti, durante il periodo di vita dell'infrastruttura.

La presente tesi tratterà dunque l'architettura del servizio, considerando i requisiti di partenza ma notando, ove opportuno, le preoccupazioni emerse in momenti successivi e risolte con miglioramenti in corso d'opera.

Capitolo 2

Requisiti

Il primo passaggio della collaborazione è stato la stesura del primo insieme di requisiti.

Si tratta di una lista di necessità che l'architettura deve soddisfare, realizzata prima di qualsiasi operazione per facilitare lo sviluppo del progetto.

2.1 Lista di requisiti

- Devono essere realizzati template infrastructure-as-code che permettano all'utente di richiedere la creazione di un nuovo cluster nella regione geografica desiderata, a partire da un sottoinsieme di regioni designate.
- Devono essere realizzati template infrastructure-as-code che permettano all'utente di richiedere la creazione di un nuovo nodo all'interno di un cluster di sua proprietà.
- Devono essere realizzati infrastructure-as-code che permettano alla società SQLite Cloud di deployare a comando nuovi load balancer, dedicati a ogni combinazione di utente e regione.
- I template devono essere integrati con la codebase di SQLite Cloud per eseguire applicativamente le operazioni sopracitate, nonchè monitorarne il progresso e comunicarlo all'utente, anche in caso di fallimento.
- Devono essere gestiti tutti gli effetti collaterali (ad esempio inserimento dei nuovi elementi di infrastruttura nel database dedicato) e politiche di retry in caso di errore.
- Deve essere possibile l'eliminazione su richiesta degli elementi di infrastruttura creati dall'utente.
- I template devono essere sviluppati in AWS CloudFormation o AWS Cloud Development Kit (CDK).
- Il deployment dei template deve essere gestito tramite le API standard di AWS o tramite le nuove API di AWS Cloud Control.
- I nodi devono essere eseguiti direttamente su istanze EC2, oppure containerizzati tramite Docker, usando ECS come orchestrator e EC2 o Fargate come capacity provider. In caso venga utilizzato EC2, ciascun utente avrà macchine EC2 dedicate

all'esecuzione dei suoi nodi, e ciascuna macchina EC2 avrà un disco EBS dedicato per lo storage dei nodi.

- Devono essere studiate e implementate politiche di backup basate su snapshot di volumi.
- In caso di fallimenti di istanze EC2 o nodi, è necessario che AWS possa sostituire in maniera trasparente gli elementi falliti. A questo scopo si ipotizza di rappresentare ciascun nodo come un servizio ECS dedicato, con numero di task impostato a 1, e di rappresentare ciascuna istanza EC2 come un AutoScaling Group dedicato, con numero di istanze impostato a 1. In questo modo, le polizze di autoscaling native di ECS e EC2 AutoScaling garantiranno che gli elementi falliti siano sostituiti il prima possibile con nuovi elementi sani.

Dopo aver concordato questa lista di requisiti, ha avuto inizio presso Polarity la prima fase di analisi interna e realizzazione di prototipi, durante la quale sono emerse problematiche intrinseche ai requisiti che rendevano difficile (se non impossibile) lo sviluppo di un'architettura stabile, sicura e economicamente sostenibile. Sono state dunque trovate alternative che risolvessero i problemi individuati, e a valle di ciò è stata stilata una revisione della lista di requisiti, in attesa di approvazione da parte di SQLite Cloud.

2.2 Modifiche apportate alla lista di requisiti

- Si è optato per eseguire tutti i nodi su ECS, usando EC2 come capacity provider.
- Tuttavia le istanze EC2, anzichè essere dedicate agli utenti come precedentemente ipotizzato, saranno condivise tra i nodi di tutti gli utenti, divisi per regione. Di conseguenza, se utente A e utente B hanno entrambi un nodo nella stessa regione, è possibile che i nodi dei due utenti vengano eseguiti sulla stessa istanza EC2.
- Allo stesso modo, anzichè deployare e dedicare un nuovo load balancer per ogni combinazione di utente e regione, si è optato per deployare staticamente un singolo load balancer in ciascuna delle regioni designate, dotato di logiche di indirizzazione tali da consentire a ogni utente di raggiungere solo ed esclusivamente i propri nodi in una data regione.
- Anzichè utilizzare le soluzioni di load balancing offerte da AWS, i requisiti hanno reso necessario optare per una soluzione self-hosted, e la scelta è ricaduta su Traefik.
- Questo richiede di sviluppare soluzioni di crittografia in-flight personalizzate, in quanto usando un load balancer personalizzato non possiamo contare sulle soluzioni gestite da AWS.
- Il concetto di cluster di nodi è stato disaccoppiato dal concetto di cluster di ECS, e si è optato invece per un cluster ECS dedicato a ciascuna regione.
- Si è scelto di utilizzare le API standard di AWS per il deployment dei template, i quali saranno realizzati in Pulumi e CloudFormation. Pulumi è un nuovo framework di infrastructure-as-code che si adatta perfettamente alle esigenze di dinamicità dell'architettura, mentre CloudFormation è stato incluso per poter riutilizzare i template già esistenti presso Polarity.

- Essendo impercorribile l'idea di un volume EBS condiviso da tutti i nodi nella stessa macchina EC2, si è optato per creare un volume per ciascun nodo, e assegnare a ogni macchina tutti i volumi dei nodi eseguiti al suo interno.

Le modifiche proposte sono poi state approvate da SQLite Cloud, al che si è potuto procedere al primo design dell'architettura.

Capitolo 3

Amazon Web Services

Amazon Web Services, o AWS, è uno dei tre più grandi cloud provider al mondo, insieme a Microsoft Azure e a Google Cloud. Lanciato nel 2006, in origine era costituito da solamente tre servizi, mentre nel 2025 ne conta più di duecento.

I servizi di AWS spaziano da elementi basilari per la realizzazione di qualsiasi architettura in cloud, come istanze computazionali o volumi di storage, a soluzioni gestite che offrono un'alternativa a sviluppare da zero funzionalità universali, come load balancing o autoscaling, fino a servizi pensati per ridurre al minimo le competenze necessarie per il loro utilizzo, come Elastic Beanstalk e App Runner.

Tutti i servizi di AWS interagiscono perfettamente tra di loro, e sono studiati per garantire alta affidabilità e resilienza, tramite gestione automatica da parte di AWS oppure fornendo tutte le funzionalità necessarie per giungervi autonomamente.

AWS è organizzato in regioni, di cui 17 regioni principali, disponibili a ogni utente sin dal principio, e numerose altre regioni secondarie, attivabili su richiesta. Ogni regione corrisponde a un insieme di data center, distribuiti nel mondo e suddivisi a loro volta in 2 o 3 agglomerati per ciascuna regione, detti zone di disponibilità, o availability zones (AZ). La stragrande maggioranza dei servizi di AWS è provvista di architettura multi-AZ, tale che nel caso in cui un disastro colpisse una delle zone di disponibilità, il servizio non sarebbe compromesso, in quanto le altre zone su cui è distribuito continuerebbero a funzionare indipendentemente. Questa resilienza ha reso AWS un'ottima scelta per infrastrutture costituite da computazioni critiche, per le quali qualsiasi downtime sarebbe particolarmente dannoso.

Nello sviluppo dell'architettura di SQLite Cloud, è stato fatto uso di diversi dei servizi offerti da AWS, elencati e descritti di seguito.

3.1 CloudFormation

CloudFormation è il servizio di infrastructure-as-code proprietario di AWS. Si tratta di un'alternativa peculiare alle altre soluzioni, in quanto può essere usato solo per deployare infrastruttura su AWS, e mentre la maggior parte delle soluzioni IaC vengono eseguite lato client, l'esecuzione di CloudFormation avviene lato server.

L'utente può scrivere un template in JSON o YAML, all'interno del quale definisce esattamente le risorse che vuole vengano create, configurando i rapporti tra le risorse dello stesso template oppure tra le risorse del template e quelle già esistenti nell'account, grazie a un sistema di parametri e output. Quindi, deploya il template su CloudFormation e attende che la creazione delle risorse sia terminata, monitorando la dashboard di CloudFormation in caso di errori.

Nonostante la limitazione ad AWS, si tratta di un servizio molto potente e robusto, ideale per infrastrutture non ibride e dal funzionamento standard.

3.2 CloudWatch

Tutte le funzionalità di logging e gestione delle metriche di AWS sono agglomerate nelle feature di CloudWatch, tramite cui è possibile configurare e consultare fonti di log, analizzare metriche raccolte da tutti i servizi di AWS, e configurare allarmi che svolgano azioni prestabilite in base ai dati ottenuti dalle metriche. I log vengono organizzati in gruppi, e per ogni gruppo è possibile configurare una scadenza, così che i log più vecchi di una certa soglia siano eliminati definitivamente. Quando configurata opportunamente, questa funzionalità permette di diminuire i costi senza compromettere la possibilità di monitorare attentamente il corretto funzionamento dell'infrastruttura.

3.3 CodeSuite

La CodeSuite di AWS è un insieme di servizi pensati per automatizzare la gestione e il deployment del codice di proprietà dell'utente. Tutti i servizi al suo interno sono studiati per interagire tra di loro alla perfezione, e consentono all'utente di architettare pipeline rapide ed efficienti per trasformare codice in applicativi compilati, e eventualmente lanciarli automaticamente sull'infrastruttura configurata.

3.3.1 CodeBuild

CodeBuild riceve da una sorgente il codice sorgente e un file di configurazione, e utilizza le indicazioni contenute nel file di configurazione per compilare un artefatto eseguibile a partire dal codice sorgente ricevuto. Si tratta dello step fondamentale della maggior parte di pipeline CI/CD. CodeBuild è organizzato in progetti: ogni progetto è configurato con un insieme di specifiche, che determinano le risorse a disposizione della build, e devono quindi essere configurate accuratamente affinchè il processo di build non sia ostacolato dalla mancanza di risorse, ma allo stesso tempo non vengano allocate più risorse del necessario.

3.3.2 CodePipeline

CodePipeline è il servizio collante tra tutti i passi di compilazione di un applicativo. Una volta creata una pipeline, è possibile scegliere un'integrazione con un servizio di version control (come GitHub, GitLab o BitBucket), scegliere un progetto di CodeBuild che effettui la compilazione dell'applicativo, e un servizio ECS che esegua l'applicativo compilato.

3.3.3 Integrazione con GitHub

L'utente può collegare il proprio account GitHub alla suite, aggiungendo al proprio account un'automazione che rileverà modifiche apportate a un insieme configurabile di repository e attiverà l'esecuzione di azioni conseguenti da parte di altri servizi della suite.

3.4 Elastic Compute Cloud (EC2)

EC2 è il servizio computazionale fondamentale offerto da AWS, tramite il quale è possibile creare istanze dotate di varie specifiche. Ogni istanza è una macchina virtuale eseguita sugli hardware distribuiti di AWS e gestita tramite un hypervisor, dedicata all'utente, che ne può controllare i tempi di esecuzione e la configurazione.

L'utente può inoltre impostare il livello di accessibilità a ciascuna delle sue istanze tramite la funzionalità dei security groups, un firewall di quarto livello che può essere usato per determinare su quali porte l'istanza può ricevere traffico e da quali porte può inviare traffico. Una corretta configurazione di security groups e routing a livello di rete virtuale permette all'utente di ottenere un'infrastruttura sicura e accessibile solamente ove necessario.

3.4.1 EC2 Auto Scaling

EC2 Auto Scaling è il servizio proprietario di AWS pensato per scalare automaticamente qualsiasi infrastruttura basata su EC2. L'utente può configurare un autoscaling group e decidere quante istanze deve contenere, quante può contenerne al massimo e quante può contenerne al minimo, quindi può configurare delle regole di scaling automatico per alterare il numero desiderato di istanze in base a metriche quali utilizzo medio di CPU o memoria, richieste in entrata e salute delle istanze. Il meccanismo di scaling può essere inoltre integrato con vari altri servizi di AWS, quali EventBridge e CloudWatch.

3.4.2 Elastic Block Storage (EBS)

EBS è la principale soluzione di storage per le istanze EC2, nonchè la soluzione default per i dischi root. Si tratta di un sistema di storage a blocchi, dischi virtuali che possono essere collegati via rete alle istanze alle quali sono assegnati. È possibile collegare più volumi a una stessa istanza, ma un volume non può essere collegato a più istanze contemporaneamente. Come le istanze EC2, anche i volumi EBS sono disponibili in una vasta gamma di dimensioni e specifiche.

3.4.3 Amazon Linux

Le istanze EC2 possono eseguire numerosi sistemi operativi, da Linux a Windows, passando per MacOS. Tuttavia, Linux costituisce la stragrande maggioranza non solo delle istanze EC2 al mondo, ma di tutti i server al mondo, siano essi in cloud o on-premises. Per questo motivo, Amazon ha sviluppato Amazon Linux, una distribuzione di Linux ottimizzata per EC2 e basata su CentOS e RedHat Enterprise. Il progetto SQLite Cloud è stato avviato negli ultimi mesi di vita di Amazon Linux 2, e ha proseguito la sua esistenza anche dopo il lancio di Amazon Linux 2023, una nuova versione più ottimizzata e compatibile con strumenti moderni.

3.5 Elastic Container Repository (ECR)

ECR è un servizio di immagazzinamento di immagini Docker compilate, simile a DockerHub ma integrato con tutti i sistemi e i servizi nativi di AWS. L'utente può creare una repository e caricarvi immagini manualmente, oppure tramite una pipeline che compili per lui l'immagine. Dopodichè, i servizi di ECS possono essere configurati per utilizzare sempre la versione più recente di una determinata immagine immagazzinata su ECR, e sostituiranno automaticamente le proprie task quando rilevano che nella repository selezionata è disponibile una nuova versione.

3.6 Elastic Container Service (ECS)

ECS è una delle due soluzioni di container orchestration offerte da AWS, insieme a Elastic Kubernetes Service (EKS, che come si può dedurre dal nome è basato su Kubernetes). Tramite ECS è possibile definire dei cluster, e al loro interno dei servizi, corrispondenti a una determinata immagine Docker, della quale vengono deployate un numero configurabile di task per servizio, corrispondenti ai container. I servizi ECS possono essere eseguiti su EC2, passando al cluster un insieme di istanze o autoscaling group da utilizzare come capacity provider, oppure su Fargate, un'alternativa serverless offerta da AWS più costosa, ma che solleva l'utente dalla responsabilità di amministrare istanze EC2 per l'esecuzione dei suoi container.

3.7 Identity and Access Management (IAM)

IAM è uno dei servizi fondamentali di AWS, in quanto amministra la gestione dei permessi e delle autorizzazioni per tutte le identità in tutti i servizi offerti.

Tramite IAM è possibile configurare utenti, gruppi e ruoli, e assegnare a ciascuna di queste entità un insieme di autorizzazioni, sotto forma di polizze. Queste polizze possono essere anche configurate lato risorsa per numerosi servizi (tra cui KMS e S3), per permettere un controllo ancora più granulare degli accessi.

Ciascuna polizza è definita in JSON, e organizza le autorizzazioni in blocchi detti statements. Ciascuno statement può autorizzare o negare una determinata azione, dove l'azione è rappresentata da una lista di permessi, stringhe corrispondenti a permessi riconosciuti dalla API di AWS. È poi necessario specificare l'insieme di risorse sulle quali questo statement ha effetto, con la possibilità di utilizzare l'asterisco (*) per indicare grandi insiemi di risorse. Infine, IAM mette a disposizione una serie di parametri aggiuntivi per raffinare ulteriormente il controllo dei permessi, come un parametro condition che autorizza l'azione solamente se sono soddisfatte una serie di condizioni booleane.

Le polizze IAM possono essere configurate solo lato identità, solo lato risorsa o su entrambi i lati, sortendo effetti diversi in base ai casi: qualsiasi negazione esplicita avrà la meglio su qualsiasi autorizzazione esplicita, e qualsiasi operazione sarà negata a meno che non sia presente lato identità o lato risorsa almeno una autorizzazione esplicita. Le due eccezioni alla regola sono gli accessi tra account separati e gli accessi a chiavi KMS: in questo caso, è necessaria un'autorizzazione esplicita sia lato identità che lato risorsa.

IAM permette inoltre di amministrare più account AWS contemporaneamente, rendendoli parte della stessa organizzazione IAM; quindi, è possibile configurare utenti con credenziali SSO (single sign-on) che sfruttino l'organizzazione IAM per ricevere i permessi dovuti sugli account dovuti. Una pratica molto diffusa è la creazione di un account gestionale, un account staging per testare le nuove funzionalità e un account di produzione, l'unico account esposto al pubblico.

3.8 Key Management System (KMS)

KMS è la soluzione di crittografia at-rest fornita da AWS per tutti i servizi offerti. Tramite KMS, è possibile creare chiavi gestite da AWS o personalizzate, e configurare la propria infrastruttura affinchè utilizzi determinate chiavi per cifrare o decifrare determinate risorse (come volumi EBS o parametri SSM). Quindi, è possibile configurare polizze IAM e assegnarle alle chiavi e alle identità per stabilire quali entità sono autorizzate a usare quali chiavi per decifrare quali risorse.

3.9 Route 53

Route 53 è il servizio di DNS proprietario di AWS. Tramite Route 53 è possibile sia acquistare e registrare un dominio, sia gestire domini registrati presso entità di terze parti (quali Namecheap o Register.it).

Route 53 permette di gestire molteplici hosted zone per ogni dominio, impostando record di tutte le classiche tipologie (A, AAAA, CNAME, MX), oppure record alias, una tipologia speciale di AWS che consente di puntare a un insieme di risorse di AWS, lasciando che sia Route 53 a gestire responsabilità come fallback e controlli di salute.

Route 53 consente inoltre di impostare per ogni record una polizza di routing, così da poter controllare facilmente regole speciali di indirizzazione del traffico. Ad esempio, è possibile creare più record con lo stesso nome e impostati con la polizza di routing 'latenza', ma che puntino a due diverse risorse situate in regioni diverse di AWS. In questo modo, l'utente sarà indirizzato alla risorsa con la minore latenza per il suo collegamento.

3.10 Systems Manager (SSM)

Uno dei più versatili servizi di AWS, Systems Manager è composto da una serie di funzionalità utili per manipolare i sistemi in funzione sulle varie soluzioni di computazione offerte da AWS, in particolare EC2.

3.10.1 SSM Session Manager

Il gestore di sessioni di SSM consente di accedere alle istanze EC2 direttamente dalla dashboard di AWS, senza bisogno di configurare porte o chiavi SSH. Si tratta quindi di un potente strumento per la gestione rapida delle risorse da parte degli amministratori.

3.10.2 SSM Run Command

Run Command permette di definire dei documenti, ovvero degli script bash che vengono salvati nell'account. È poi possibile eseguire programmaticamente questi script, selezionando un'istanza EC2 di destinazione e passando al documento eventuali variabili d'ambiente per alterarne dinamicamente l'esecuzione. Si tratta di uno strumento molto potente per gestire sistemi distribuiti e coordinare operazioni che richiedono procedure speciali eseguite direttamente all'interno delle macchine.

3.10.3 SSM Parameter Store

Con Parameter Store è possibile immagazzinare stringhe di piccole o grandi dimensioni sotto forma di parametri, con la possibilità di crittografare parametri di importanza critica, e recuperare qualsiasi parametro programmaticamente, ammesso che l'identità che esegue il recupero sia provvista dei permessi necessari per leggere i parametri e effettuare eventuali decifrazioni.

Parameter Store include inoltre una serie di parametri default gestiti direttamente da AWS, come ad esempio stringhe identificatrici di immagini precompilate da AWS per il lancio di istanze EC2 ottimizzate.

3.11 Simple Storage Service (S3)

Uno dei primi servizi lanciati da AWS, S3 consente di immagazzinare e servire file in maniera efficiente e distribuita. Tutti i file sono organizzati in bucket, di proprietà di un determinato account e contraddistinti da un indirizzo univoco. I bucket sono poi a loro volta organizzati in prefissi, concettualmente simili a un sistema di directory, e l'accesso ai file può essere manipolato tramite polizze di accesso assegnate a ciascun bucket, definite nel formato IAM. In questo modo, è possibile limitare l'accesso a un insieme ristretto di identità, che siano interne all'account AWS proprietario del bucket o esterne.

3.12 Virtual Private Cloud (VPC)

VPC è la base di tutte le infrastrutture di AWS (ad eccezione delle infrastrutture serverless), e consente di creare delle reti virtuali private, isolate dal resto della rete globale e separate da tutte le reti virtuali degli altri utenti di AWS. È quindi possibile organizzare la rete in sottoreti, gestire il traffico tramite tabelle di routing, e collegare le sottoreti all'internet pubblico aggiungendo un internet gateway o un NAT gateway alla rete virtuale, e connettendolo alle sottoreti tramite tabelle di routing.

Tutte le reti sono realizzate tramite l'uso di indirizzi IP privati e pubblici, e le sottoreti sono realizzate a partire da suddivisioni dell'insieme di IP privati assegnato alla rete virtuale, tramite Classless Inter-Domain Routing (CIDR).

Infine, è possibile collegare tra di loro più reti virtuali, che siano nella stessa regione o in regioni diverse, grazie alle funzionalità di peering offerte da AWS VPC, per poi controllare il traffico tramite inserimento di ulteriori regole nelle tabelle di routing.

Capitolo 4

Altre tecnologie

4.1 Docker

Una delle fondamentali innovazioni della nuova generazione di infrastrutture web è stato il passaggio da applicativi eseguiti direttamente sulle macchine alla containerizzazione e distribuzione degli applicativi.

Un container è una tipologia particolare di macchina virtuale, ottimizzata per poter eseguire molteplici container identici in una stessa macchina. I vantaggi dell'approccio sono molteplici, tra cui ottimizzazione delle risorse, riduzione del collo di bottiglia causato dall'esecuzione di un singolo applicativo con tutte le risorse del sistema assegnate, rimozione a livello applicativo del singolo punto di rottura e maggiore modularietà della distribuzione dell'applicativo e totale indipendenza dall'ambiente di esecuzione, in quanto qualsiasi container di qualsiasi sistema operativo può essere eseguito su qualsiasi sistema operativo host, a patto che sia stato compilato per l'architettura corretta.

La diffusione dei container ha portato un'evoluzione dei paradigmi di sviluppo e ingegneria del software, in quanto era necessario garantire che l'applicativo potesse funzionare correttamente anche quando erano presenti più istanze dello stesso processo. I vantaggi di questo cambiamento sono stati tuttavia unanimemente accolti come un'importante innovazione nello sviluppo software, e ad oggi i container sono una delle basi dello sviluppo web.

Docker è l'implementazione che ha fatto da pioniere per il concetto di container, ed è ancora oggi la soluzione principale, con migliaia di servizi che ne fanno ampio uso.

Tra questi servizi ovviamente troviamo SQLite Cloud, sviluppato sin dal principio in un'ottica di containerizzazione tramite Docker, tecnologia con la quale anche Polarity è estremamente competente.

Di seguito un esempio di utilizzo di Docker: definendo un cosiddetto Dockerfile è possibile stabilire una serie di operazioni da svolgere per compilare l'immagine Docker a partire dalla codebase. L'esempio mostrato è il Dockerfile del container dedicato al load balancer Traefik, altra tecnologia approfondita in seguito.

```
FROM traefik:v3.5

WORKDIR /

RUN apk update && apk add --no-cache curl inetutils-telnet lsof vim openrc

COPY ./traefik.yml /traefik.yml

EXPOSE 8080
EXPOSE 8090
EXPOSE 8860
EXPOSE 9860

CMD traefik --configfile=/traefik.yml
```

Figura 4.1: Dockerfile dell'immagine Traefik.

4.2 Linux

Tutto il lato computazionale dell'infrastruttura è basato su Linux, usando Alpine come distribuzione per i container Docker e Amazon Linux come distribuzione per le istanze EC2.

Linux è un ecosistema open-source altamente modulare, evolutosi intorno all'omonimo kernel e diventato in breve tempo un pilastro del web, utilizzato in circa l'80% dei server globalmente. L'enorme quantità di programmi, moduli e funzionalità realizzate dalla comunità open-source per Linux lo rende la scelta migliore per qualsiasi esigenza computazionale, ragion per cui è stato individuato fin da subito come l'opzione migliore sia da Polarity che da SQLite Cloud.

Per quanto riguarda le distribuzioni, la scelta di Alpine per i container è a sua volta una pratica comune, in quanto si tratta di una distribuzione particolarmente leggera e di conseguenza ottima per l'esecuzione in ambienti a basse risorse. Amazon Linux, invece, è la scelta ovvia per istanze EC2, in quanto ottimizzato per l'esecuzione in cloud e integrato alla perfezione con la maggior parte dei servizi AWS.

La scelta di Linux comporta un importante ruolo del linguaggio bash nello sviluppo dei sistemi e delle automazioni in funzione nell'infrastruttura. Ogni distribuzione Linux è difatti provvista di una shell, con sh e bash comunemente riconosciute come le più diffuse. Di seguito è riportato un esempio di codice bash utilizzato per automatizzare operazioni sulle macchine che compongono l'architettura del progetto.

```
populate_volume_info() {
    for volume_id in $volume_ids; do
        # done
        echo "(4/7) DEBUG: done"
    done
}

check_iam_credentials() {
local max_attempts=30
local attempt=0
local success=false

echo "(4/7) DEBUG: Checking for IAM credentials..."

# Loop until IAM credentials are available or max attempts are reached
while [ $attempt -le $max_attempts ]; do
# Fetch IAM role credentials from the metadata service
local credentials=$(curl -s http://169.254.169.254/latest/meta-data/iam/info --header "X-aws-ec2-metadata-token: $token")

# Check if the 'Code' attribute in the response is 'Success'
if echo "$credentials" | grep -q '"Code" : "Success"'; then
success=true
echo "(4/7) DEBUG: IAM credentials are available."
break
fi

echo "(4/7) DEBUG: Waiting for IAM credentials to be available..."
sleep 10
((attempt++))
done

if [ "$success" = true ]; then
echo "(4/7) DEBUG: IAM credentials are ready. Proceeding with the script..."
else
echo "(4/7) DEBUG: Failed to obtain IAM credentials after $max_attempts attempts."
exit 1  # Exit with an error status if credentials are not available
fi
}

# check IAM credentials
echo "(4/7) DEBUG: check IAM credentials"
check_iam_credentials
echo "(4/7) DEBUG: done"

# volume attachment
echo "(4/7) DEBUG: volume attachment"

populate_volume_info

# done
echo "(4/7) DEBUG: done"
echo "(4/7) DEBUG:"
```

Figura 4.2: Estratto di codice bash in uso nell'infrastruttura.

4.3 Pulumi

Una limitazione importante di CloudFormation come servizio infrastructure-as-code è la scarsa adeguatezza per realizzare infrastrutture dinamiche: difatti, CloudFormation è pensato per architetture con un insieme di parti prestabilito a priori, a eccezione di piccoli elementi scalabili quali autoscaling group e servizi ECS. Per un'infrastruttura altamente dinamica come quella richiesta da SQLite Cloud, era necessario un sistema più flessibile, e dopo attente ricerche si è deciso di fare uso di Pulumi.

La grande innovazione di Pulumi rispetto ad altri sistemi IaC, come CloudFormation, Terraform o CDK, è che i template di infrastruttura possono essere scritti tramite linguaggi classici di programmazione, come JavaScript, TypeScript, C# o Go. Questo ha consentito a Polarity di sviluppare template che si integrassero facilmente con la codebase di SQLite Cloud, scritta prevalentemente in Go, nonchè di approfittare delle conoscenze pregresse del team dedicato in TypeScript per velocizzare lo sviluppo dell'infrastruttura statica.

Un'importante funzionalità di Pulumi per le necessità del progetto è la possibilità di sviluppare codice di infrastruttura in due modi:

- sotto forma di template, scrivendo tutte le risorse da creare in uno o più file del linguaggio designato e eseguendo l'utilità da riga di comando di Pulumi per effettuare il deployment dell'infrastruttura così dichiarata. L'utilità legge il template e

lo interpreta per generare una lista di risorse previste, dopodichè la confronta con lo stato attuale dell'infrastruttura, salvato in una posizione predefinita (per esempio, un bucket S3). Ogni differenza individuata viene presentata all'utente nella preview delle modifiche all'infrastruttura, in attesa di conferma. Questo approccio è stato scelto per sviluppare l'infrastruttura statica.

- sotto forma di codice, tramite un vero e proprio SDK (software development kit). Questo approccio permette di integrare il codice di creazione delle risorse in una codebase esistente, e svolgere le operazioni di modifica dell'infrastruttura senza bisogno di eseguire manualmente l'utilità da riga di comando. Si tratta dunque dell'approccio più adatto per sviluppare funzioni Go da chiamare tramite il backend di SQLite Cloud, al quale è delegato il compito di espandere dinamicamente l'infrastruttura in completa autonomia.

Altra funzionalità di Pulumi della quale è stato fatto ampio uso è la suddivisione delle risorse in stack e la condivisione di informazioni tra gli stack mediante import e export. Uno stack è un insieme di risorse generate e maneggiate insieme, solitamente utilizzato per raggruppare elementi dell'infrastruttura logicamente a stretto contatto tra di loro. Nel codice che definisce uno stack è inoltre selezionare determinate proprietà delle risorse che compongono suddetto stack, e renderle accessibili agli stack esterni, tramite la procedura di export. Altri stack possono quindi accedere a queste proprietà, ottenendone i valori tramite la procedura di import, e utilizzando questi valori per creare le loro risorse.

Un banale esempio dell'utilità di questo sistema è il seguente: supponendo di scrivere uno stack di risorse dedicato alla creazione di un'istanza EC2, e volendo creare poi un altro stack all'interno del quale definire un volume EBS da assegnare a questa istanza, è necessario che il secondo stack conosca l'ID dell'istanza del primo stack. Per ottenere questo risultato è sufficiente esportare dal primo stack l'ID dell'istanza, e importarlo all'interno del secondo stack, per poi referenziarlo nelle proprietà di configurazione del volume EBS.

Nelle prossime due figure, un esempio dell'applicazione di questo principio nel codice Pulumi utilizzato all'interno dell'architettura: le variabili necessarie per la creazione di risorse in template esterni sono state esportate dal template di origine e importate nel template di destinazione.

```
export const ecrRepositoryRepositoryUrl = ecrRepository.repositoryUrl
export const ecrRepositoryName = ecrRepository.name
export const traefikTaskRoleArn = globalTraefikTaskRole.arn
export const traefikInstanceRoleName = traefikInstanceRole.name
export const envBucketName = globalEnvironmentBucket.bucket
export const envBucketArn = globalEnvironmentBucket.arn
export const nodeTaskRole = globalNodeTaskRole.arn
export const taskExecutionRole = globalTaskExecutionRole.arn
export const nodeEcrRepository = nodeEcrRepo.repositoryUrl
export const hostedZoneId = prodHostedZoneId;
export const hostedZoneName = prodHostedZoneName;
```

Figura 4.3: Esportazione di variabili da un template Pulumi.

```
const globalStackRef = new StackReference(`organization/sqlitecloud/sqlite-global`);

const taskExecutionRole =  globalStackRef.getOutput("taskExecutionRole");
const hostedZoneId = globalStackRef.getOutput("hostedZoneId");
const ecrRepositoryRepositoryUrl = globalStackRef.getOutput("ecrRepositoryRepositoryUrl");
const ecrRepositoryName = globalStackRef.getOutput("ecrRepositoryName");
const traefikTaskRoleArn = globalStackRef.getOutput("traefikTaskRoleArn");
const traefikInstanceRoleName = globalStackRef.getOutput("traefikInstanceRoleName");
```

Figura 4.4: Importazione delle stesse variabili in un secondo template Pulumi.

4.3.1 TypeScript

TypeScript è un superset di JavaScript, un linguaggio di programmazione e scripting pensato per lo sviluppo web. Mentre JavaScript è tipizzato dinamicamente, il principale pregio di TypeScript è l'introduzione di tipi statici, che facilitano lo sviluppo e riducono drasticamente la comparsa di errori a tempo di esecuzione.

Tra i linguaggi di programmazione supportati da Pulumi, Polarity ha scelto di sviluppare l'infrastruttura statica in TypeScript per bilanciare la difficoltà di apprendere un nuovo paradigma di sviluppo di infrastructure-as-code con la faimiliarità di un linguaggio utilizzato estensivamente all'interno dell'azienda, sia dai reparti di sviluppo, sia in passato dai membri del reparto infrastruttura, incluso il candidato.

Di seguito, un estratto dal codice TypeScript sviluppato per il progetto.

```
// peering connections

for (let i = 0; i < utils.length-1; i++) {
  if (!utils[i].create) {
    continue;
  }
  for (let j = i+1; j < utils.length; j++) {
    if (!utils[j].create) {
      continue;
    }
    const peeringConnection = new aws.ec2.VpcPeeringConnection(`peering-cluster-${utils[i].region}-${utils[j].region}`, {
      vpcId: vpcs[i].id,
      peerVpcId: vpcs[j].id,
      peerRegion: utils[j].region,
      tags: {
        Name: `peering-cluster-${utils[i].region}-${utils[j].region}-sqlite`
      }
    }, {
      provider: utils[i].provider
    });

    new aws.ec2.VpcPeeringConnectionAccepter(`peering-cluster-accepter-${utils[i].region}-${utils[j].region}`, {
      vpcPeeringConnectionId: peeringConnection.id,
      autoAccept: true,
      tags: {
        Name: `peering-cluster-${utils[i].region}-${utils[j].region}-sqlite`
      }
    }, { provider: utils[j].provider });

    new aws.ec2.Route(`route-from-${utils[i].region}-to-${utils[j].region}`, {
      routeTableId: routeTables[i].id,
      destinationCidrBlock: `${utils[j].vpcIp[0]}.0.0/14`,
      vpcPeeringConnectionId: peeringConnection.id,
    }, {
      provider: utils[i].provider
    });

    new aws.ec2.Route(`route-from-${utils[j].region}-to-${utils[i].region}`, {
      routeTableId: routeTables[j].id,
      destinationCidrBlock: `${utils[i].vpcIp[0]}.0.0/14`,
      vpcPeeringConnectionId: peeringConnection.id,
    }, {
      provider: utils[j].provider
    });
  }
}
```

Figura 4.5: Esempio di applicazione di TypeScript nello sviluppo del codice di infrastruttura.

4.3.2 Go

Go (o Golang) è un linguaggio di programmazione compilato e tipizzato staticamente, pensato per essere efficiente e veloce. È caratterizzato da una gestione automatica della memoria (detta garbage collection), una sintassi molto semplice e un'alta trasferibilità del codice su molteplici sistemi operativi e architetture.

L'infrastruttura dinamica di SQLite Cloud è stata sviluppata in Go per venire incontro alle esigenze della società omonima, in quanto il backend è stato scritto esclusivamente in Go. Per questo motivo, il candidato ha intrapreso un percorso di studio finalizzato ad acquisire una competenza sufficiente per poter scrivere infrastruttura in Pulumi e Go nella maniera più naturale possibile.

Di seguito, un estratto dal codice Go sviluppato per il progetto.

```go
instances = pulumi.All(instances, enabledRegions).ApplyT(
    func(v []interface{}) []interface{} {
        instances := v[0].([]interface{})
        enabledRegions := v[1].([]interface{})
        var result []interface{}
        for _, elem := range instances {
            elemMap := elem.(map[string]interface{})
            region := elemMap["region"].(string)
            if contains(enabledRegions, region) {
                result = append(result, elem)
            }
        }
        return result
    },
).(pulumi.ArrayOutput)

// RESOURCES

// records

instances.ApplyT(func(v []interface{}) error {
    for _, elem := range v {
        elemMap := elem.(map[string]interface{})
        region := elemMap["region"].(string)
        ip := elemMap["ip"].(string)

        _, err = route53.NewRecord(ctx, stackPrefix+"cluster-"+id+"-record-"+region, &route53.RecordArgs{
            Name:   hostedZoneName.ApplyT(func(v string) string { return id + "." + v }).(pulumi.StringOutput),
            ZoneId: hostedZoneId,
            Type:   pulumi.String("A"),
            Ttl:    pulumi.Int(300),
            LatencyRoutingPolicies: route53.RecordLatencyRoutingPolicyArray{
                &route53.RecordLatencyRoutingPolicyArgs{
                    Region: pulumi.String(region),
                },
            },
            SetIdentifier: pulumi.String(region),
            Records: pulumi.StringArray{
                pulumi.String(ip),
            },
        })
        if err != nil {
            return err
        }
    }
    return nil
})
```

Figura 4.6: Esempio di applicazione di Go nello sviluppo del codice di infrastruttura.

4.4 Traefik

A causa delle esigenze di routing del design di SQLite Cloud, si identificarono presto come inutilizzabili o troppo costose le soluzioni di load balancing offerte da AWS (quali Elastic Load Balancer e Network Load Balancer). Per questo motivo, la scelta è ricaduta su Traefik, un load balancer applicativo sviluppato in Go.

Tra le funzionalità principali di Traefik che lo rendono un ottimo candidato per le necessità dell'architettura, troviamo un'incredibile integrazione con ECS, rivaleggiata solamente dai bilanciatori di carico di AWS. Tuttavia, Traefik consente di reindirizzare non solo traffico

HTTP, ma anche TCP, squalificando automaticamente l'Elastic Load Balancer di AWS, e a un costo di esecuzione molto inferiore rispetto al Network Load Balancer.

Inoltre, pur essendo un load balancer self-hosted, Traefik possiede funzionalità impressionanti di aggiornamento dinamico delle impostazioni di routing, con la possibilità di monitorare un cluster ECS in attesa di modifiche (ad esempio, creazione di un nuovo servizio o rimozione di un servizio esistente) e aggiornare le proprie polizze di indirizzamento del traffico in pochissimi secondi.

Infine, la funzionalità cruciale che rende Traefik perfetto per i requisiti è la possibilità di configurare delle etichette sui container di destinazione, e manipolare il routing tramite queste etichette. Questa capacità rende possibile identificare i nodi di un cliente con lo stesso ID, e fare sì che Traefik reindirizzi il traffico proveniente da uno specifico record DNS solamente ai nodi di quel cliente.

È riportato di seguito il file di configurazione per Traefik utilizzato nell'architettura. Questo file viene incluso nell'immagine Docker che viene compilata e eseguita su tutte le regioni.

```yaml
tls:
  stores:
    default:
      defaultCertificate:
        certFile: "/etc/ssl/traefik/cert.pem"
        keyFile: "/etc/ssl/traefik/key.pem"
  certificates:
    - certFile: "/etc/ssl/traefik/cert.pem"
      keyFile: "/etc/ssl/traefik/key.pem"
      stores:
        - default
providers:
  ecs:
    autoDiscoverClusters: true
  file:
    filename: "/traefik.yml"
    watch: true
api:
  dashboard: false
  insecure: false
entryPoints:
  tls:
    address: ":8860"
  internal:
    address: ":9860"
  http:
    address: ":8090"
    transport:
      respondingTimeouts:
        readTimeout: "20m"
  websocket:
    address: ":4000"
log: # ERROR for prod, DEBUG for staging
  level: ERROR
  noColor: true
```

Figura 4.7: Configurazione Traefik in uso nell'architettura.

Capitolo 5

Design architetturale

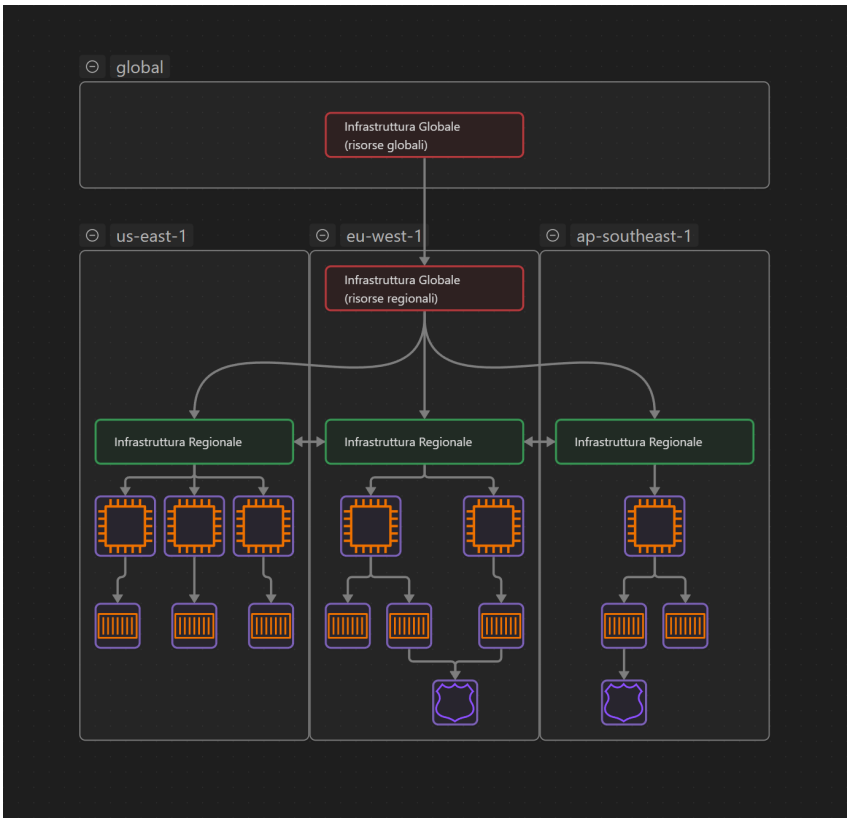

Figura 5.1: Suddivisione logica dell'infrastruttura nei tre livelli di seguito approfonditi.

L'infrastruttura è stata divisa in tre livelli, ciascuno caratterizzato da un differente grado di replicazione:

5.1 Infrastruttura Globale

L'infrastruttura globale include tutti gli elementi dei quali è necessaria una singola copia all'interno dello stesso ambiente.

Tra questi elementi vi sono alcuni che sono nativamente globali, come i ruoli IAM, e non sono legati ad alcuna regione di AWS; altri, invece, devono essere deployati in una determinata regione, nonostante poi le risorse da essi generati siano accessibili ovunque, come i bucket S3 e le repository ECR. Per soddisfare questa necessità è stato determinato arbitrariamente che la regione Irlanda sarebbe divenuta la regione globale, e avrebbe quindi ospitato sia gli elementi regionali dell'infrastruttura globale, sia gli elementi della copia irlandese dell'infrastruttura regionale.

5.1.1 Virtual Private Cloud (VPC)

Gli elementi regionali dell'infrastruttura globale sono organizzati all'interno di una rete virtuale dedicata, suddivisa in quattro sottoreti, di cui due pubbliche e due private, secondo le migliori pratiche raccomandate da AWS. Entrambe le sottoreti pubbliche sono connesse all'internet pubblico mediante un internet gateway, e ciascuna di esse ospita un NAT gateway che consente alle subnet private il traffico con l'esterno tramite protocollo NAT. A ciascun NAT gateway è assegnato un indirizzo IP elastico, e tutte le connessioni sono gestite tramite tabelle di routing, popolate con regole di routing che stabiliscono l'organizzazione del traffico. Di seguito è riportato uno schema riassuntivo della tabella di routing.

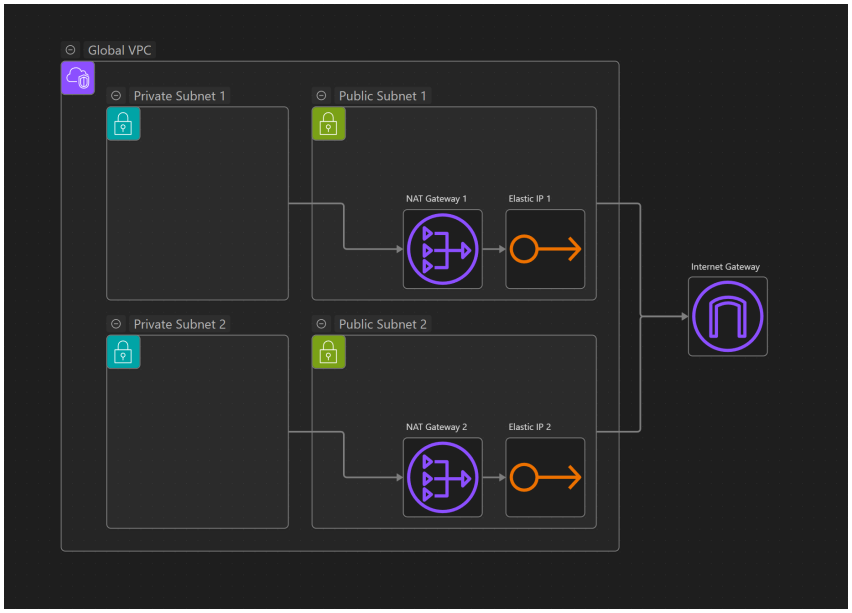

Figura 5.2: Routing dell'infrastruttura globale.

5.1.2 Bucket di S3

L'applicativo dei nodi di SQLite Cloud necessita di una serie di variabili d'ambiente per il corretto funzionamento. Si tratta tuttavia di variabili sensibili e dipendenti dall'ambiente, ragion per cui non è possibile inserirle direttamente nel codice o nell'immagine Docker. La soluzione consiste nel realizzare un file di variabili d'ambiente (comunemente denominato ėnv), e renderlo accessibile a tutti i nodi in maniera sicura. Su AWS, questo obiettivo viene spesso raggiunto tramite la creazione di un bucket S3 accessibile solo ai nodi (e eventualmente alle pipeline che li compilano, nel caso in cui siano presenti varia-

bili necessarie a tempo di compilazione), e il caricamento di un file contenente tutte le variabili all'interno di questo bucket.

È il caso di SQLite Cloud, in quanto si è deciso di creare un bucket appositamente per l'immagazzinamento dei file d'ambiente, e di consentirvi accesso solamente a nodi e pipeline.

5.1.3 Pipeline CI/CD

CI/CD è un acronimo per Continuous Integration/Continuous Delivery, una pratica standard suddivisa in due parti:

- Per Continuous Integration si intende l'integrazione continua di tutte le modifiche al codice apportate dagli sviluppatori, in un unico flusso di cambiamenti. Si tratta del principale ruolo dei sistemi di version control come Git, implementato pienamente da piattaforme quali GitHub o GitLab.
- Per Continuous Delivery si intende poi il rilascio continuo di versioni approvate del codice sintetizzato tramite Continous Integration, grazie a una completa automazione degli strumenti di compilazione e deployment dell'applicativo.

Il risultato di tale pratica è la possibilità di lavorare contemporaneamente all'applicativo, per poi rilasciare nel minor tempo possibile la versione successiva, una volta che è stata testata attentamente.

Fino a qualche anno fa, AWS forniva sia soluzioni di Continuous Integration che di Continuous Delivery, ma dopo la chiusura del servizio CodeCommit nel 2024 AWS si specializza esclusivamente in Continuous Delivery, lasciando il compito di supportare Continuous Integration a servizi quali GitHub, GitLab e BitBucket, permettendo l'interazione tra le due parti tramite integrazioni apposite.

Nel caso di SQLite Cloud, è stato usato GitHub come servizio di Continuous Integration; quindi, tramite la creazione di una connessione da GitHub ad AWS, è stato possibile integrare il servizio a una pipeline di Continuous Delivery architettata tramite CodePipeline e CodeBuild.

Il codice committato a un branch designato della repository GitHub viene quindi prelevato da CodePipeline e passato a un progetto CodeBuild, il quale legge un file apposito presente nella repository (solitamente chiamato buildspec.yml) e svolge le operazioni elencate al suo interno, tra cui solitamente si trova la compilazione di un'immagine Docker. L'immagine viene poi caricata su una repository ECR, pronta per essere recuperata e utilizzata da tutti i servizi.

Buona parte delle pipeline prevede poi solitamente un terzo step, tramite cui il servizio ECS che utilizza l'applicativo compilato viene notificato da CodePipeline della presenza di una nuova versione, e sostituisce immediatamente tutti i task con nuovi task aggiornati. Tuttavia, nel caso di SQLite Cloud, la presenza di più di un servizio (migliaia, infatti) che utilizzano lo stesso applicativo rende questa pratica infattibile. Come alternativa, il team di SQLite Cloud ha sviluppato uno script che viene lanciato manualmente al seguito di ciascun aggiornamento, incaricato di notificare tutti i servizi in tutte le regioni e sostituire tutti i task contemporaneamente.

Di tutti gli elementi coinvolti nella pipeline, solo uno è regionale, e deve essere inserito in una data VPC: il progetto di CodeBuild. Il motivo è presto detto: la compilazione di un applicativo richiede risorse variabili in base alla complessità del procedimento, ragion per cui AWS lascia all'utente la scelta delle specifiche (memoria e CPU) da conferire al progetto, con tariffe crescenti proporzionalmente alle specifiche richieste. Dietro le quinte, AWS sta infatti dedicando macchine di computazione al processo di build, e queste macchine devono trovarsi in una specifica regione, e eventualmente in una VPC.

Sebbene inserire il progetto CodeBuild in una VPC sia facoltativo, nel caso di SQLite Cloud ciò è reso obbligatorio dal fatto che il processo di compilazione dell'immagine Docker parte da un'immagine esistente disponibile su DockerHub. Tuttavia, i progetti CodeBuild non inseriti in una VPC (e quindi privi di un indirizzo IP proprio) effettuano tutte le richieste all'internet pubblico tramite gli indirizzi IP condivisi di AWS, i quali vengono utilizzati giornalmente da migliaia di processi. Di conseguenza, è estremamente difficile, se non virtualmente impossibile, riuscire a completare la richiesta a DockerHub senza incorrere nei limiti di traffico imposti a ciascun indirizzo IP.

La soluzione è dunque collocare il progetto di CodeBuild nelle reti private della VPC, così che le richieste a DockerHub siano effettuate tramite l'indirizzo IP di uno dei NAT gateway connessi a suddette reti private.

Ultimo elemento importante da considerare sul funzionamento della pipeline è che ogni step genera un insieme di file, denominato 'artefatto'. Tutti gli artefatti sono salvati temporaneamente in un bucket S3 dedicato, per permettere allo step successivo di recuperare l'artefatto su cui deve lavorare. L'infrastruttura globale è quindi composta da due bucket S3 separati, uno per i file di variabili d'ambiente e l'altro per gli artefatti di compilazione.

Tutte le risorse sopracitate compongono una pipeline dedicata esclusivamente alla compilazione automatica dell'applicativo dei nodi. Tuttavia, un'altra immagine cruciale al funzionamento dell'architettura è quella del load balancer Traefik. A causa della molto più bassa frequenza di aggiornamenti apportati a questa immagine rispetto a quella dei nodi, però, si è deciso di non realizzare un'intera pipeline per costruirla, e limitarsi a creare una repository di ECR sulla quale caricare manualmente l'immagine, dopo averla compilata personalmente.

È riportato uno schema della pipeline per la compilazione del nodo.

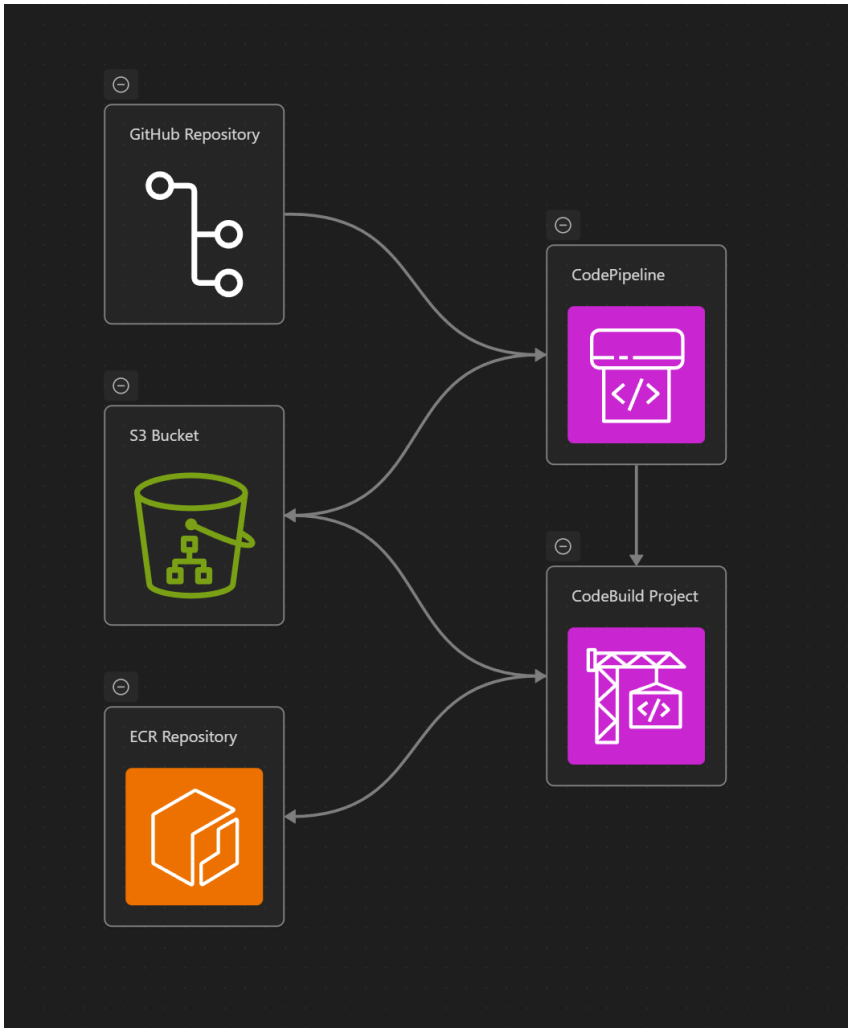

Figura 5.3: Design della pipeline CI/CD.

5.1.4 Ruoli di servizio

Affinchè la pipeline funzioni, è necessario che i servizi gestiti da AWS che la compongono siano autorizzati a svolgere tutte le operazioni su tutte le risorse necessarie. Per questo motivo, è necessario creare dei cosiddetti ruoli di servizio, ciascuno dei quali è dedicato a un determinato servizio AWS e contiene specificate tutte le autorizzazioni necessarie per far funzionare globalmente l'intero servizio.

Essendo Polarity già provvista di un template CloudFormation con tutti i ruoli di servizio impostati alla perfezione, si è deciso di riutilizzare questo template, creando lo stack di CloudFormation come una risorsa dello stack di Pulumi. Di conseguenza, Pulumi controlla il deployment degli stack di CloudFormation, e ne amministra il funzionamento come se fosse un utente.

Grazie al sistema di parametri del template di CloudFormation, sono stati creati solamente i ruoli necessari al funzionamento di questa specifica architettura, escludendo quelli non utilizzati. Difatti, solamente CodePipeline e CodeBuild necessitavano di ruoli di servizio, mentre il resto dell'architettura poteva funzionare tramite ruoli classici.

5.1.5 Schema riassuntivo dell'infrastruttura globale.

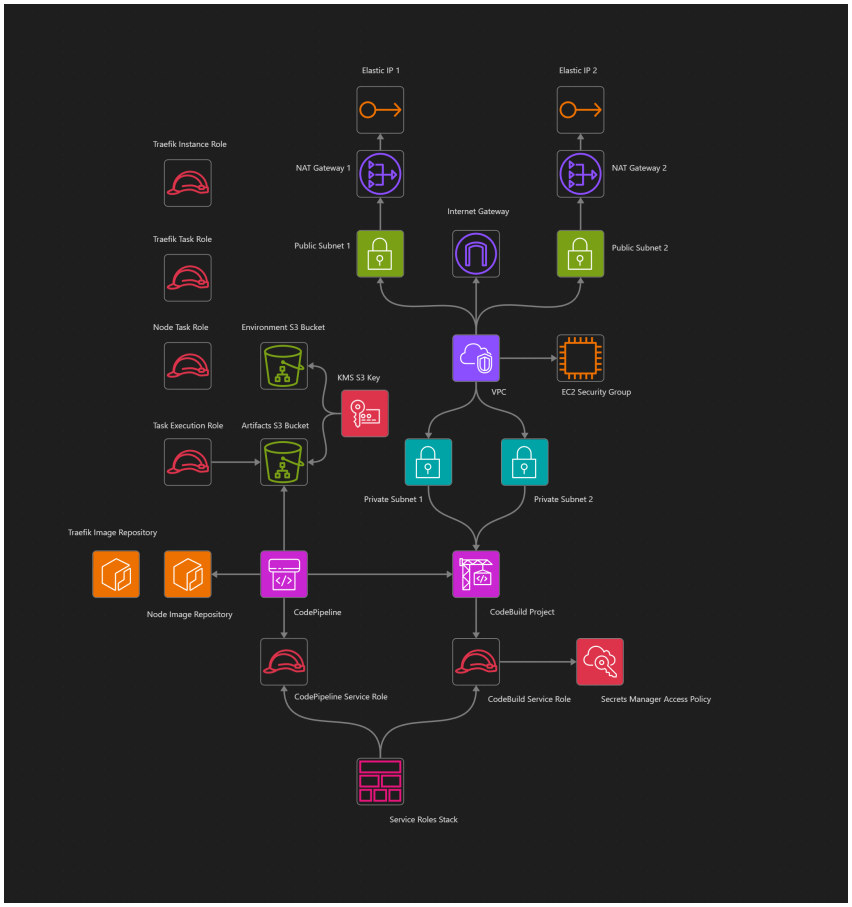

Figura 5.4: Grafico esplicativo dell'infrastruttura.

5.2 Infrastruttura Regionale

L'infrastruttura regionale include tutti gli elementi dei quali è necessaria una e una sola copia in ciascuna regione abilitata dell'ambiente. Al lancio, sono state abilitate tutte e 17 le regioni principali di AWS:

- Nord Virginia (us-east-1)
- Ohio (us-east-2)
- California del Nord (us-west-1)
- Oregon (us-west-2)
- Città del Capo (af-south-1)
- Hong Kong (ap-east-1)
- Mumbai (ap-south-1)
- Osaka (ap-northeast-3)
- Seul (ap-northeast-2)

- Singapore (ap-southeast-1)
- Sydney (ap-southeast-2)
- Tokyo (ap-northeast-1)
- Canada Centrale (ca-central-1)
- Francoforte (eu-central-1)
- Irlanda (eu-west-1)
- Londra (eu-west-2)
- Parigi (eu-west-3)

5.2.1 Virtual Private Cloud (VPC)

La VPC dedicata a ciascuna regione presenta alcune somiglianze con la VPC per la pipeline CI/CD, pur essendo strutturata in maniera diversa per venire incontro alle esigenze architetturali.

Innanzitutto, anzichè avere due subnet pubbliche e due private, è costituita da due subnet pubbliche e tre subnet private. Le subnet pubbliche sono a loro volta suddivise nella subnet principale, che ospita una singola istanza, e una subnet di emergenza, allocata nell'ottica di impostare la singola istanza ad alta affidabilità, e distribuirne il carico su entrambe le subnet. Questa istanza, mantenuta in funzione da un autoscaling group a configurazione 111 (1 istanza minima, 1 istanza desiderata, 1 istanza massima) svolge tre funzioni fondamentali:

- Ospitare il load balancer Traefik di quella regione;
- Connettere le tre subnet private all'internet esterno mediante protocollo NAT;
- Fungere da jump host per eventuali connessioni SSH a istanze delle subnet private.

Si tratta quindi essenzialmente di un bastion host, incaricato di esporre in maniera sicura le istanze collocate nelle subnet private.

Le tre subnet private sono distribuite sulle prime tre availability zone della data regione (il che per gran parte delle regioni equivale a tutte le availability zone esistenti), e contengono gli autoscaling group incaricati di ospitare i nodi dei clienti.

A completare il grafico di connessione, le subnet pubbliche sono collegate a un internet gateway, al bastion host è assegnato un indirizzo IP elastico, e a questo indirizzo IP punta un record DNS dedicato alla data regione (ad esempio, traefik-eu-west-1.sqlite.cloud).

Il flusso di traffico da un utente al suo nodo svolge dunque il seguente percorso:

- L'utente si connette a uno dei suoi endpoint dedicati (approfonditi in seguito);
- L'endpoint dedicato punta a uno dei record DNS regionali di Traefik (ad esempio, traefik-eu-west-1.sqlite.cloud);
- Il record punta all'indirizzo IP elastico assegnato alla principale interfaccia di rete del bastion host della regione data;

- Su questo indirizzo IP è in ascolto il servizio Traefik, che analizza la richiesta e ridireziona il traffico al nodo appropriato;
- Il traffico giunge al nodo, hostato su un autoscaling group all'interno di una delle subnet private.

Infine, tutte le VPC regionali sono connesse tra di loro tramite relazioni di peering, le quali, insieme a appositamente configurate regole di routing, consentono a tutti i nodi di interagire tra di loro, pur trovandosi in regioni differenti. Da notare che, onde ottenere questo risultato, è imperativo che non ci sia alcuna sovrapposizione tra i range CIDR delle VPC, così che ogni indirizzo IP privato abbia una e una sola VPC di appartenenza. A tale scopo, è stato introdotto in un secondo momento un file di costanti all'interno dell'architettura, popolato dai range CIDR per ciascuna VPC in ciascuna regione di AWS, principale o secondaria, in modo che fosse impossibile sbagliare e assegnare a una nuova VPC un CIDR sovrapposto a quello di una VPC esistente.

Di seguito è riportato uno schema riassuntivo della tabella di routing.

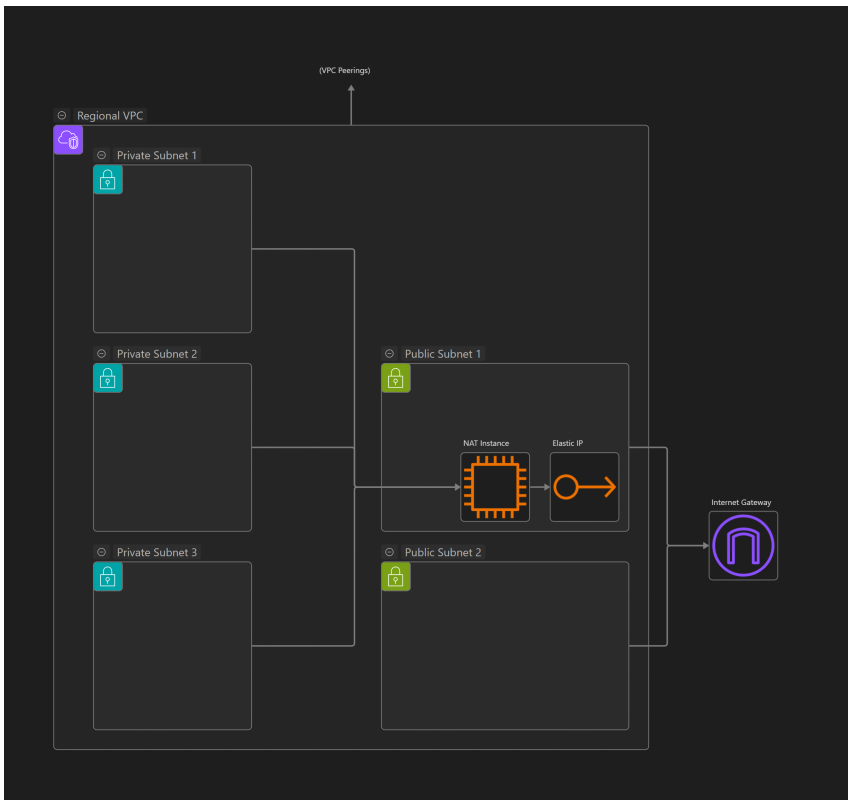

Figura 5.5: Routing dell'infrastruttura regionale.

5.2.2 Documenti di Systems Manager

I documenti di Systems Manager sono essenzialmente degli script di bash a cui è possibile passare variabili d'ambiente, e che possono essere eseguiti su qualsiasi istanza EC2, a patto che suddetta istanza abbia in esecuzione l'agent di Systems Manager. Essendo questi documenti una risorsa regionale, è necessario deployarne una copia in ciascuna regione al fine di poterne usufruire a livello globale.

Nell'architettura di SQLite Cloud, sono utilizzati due documenti, eseguiti rispettivamente durante la creazione di un nuovo nodo e durante la sua eliminazione.

Difatti, una limitazione di Pulumi (così come della stragrande maggioranza dei framework di infrastructure-as-code) è l'impossibilità di eseguire comandi bash su una macchina EC2 dopo la sua creazione. Specificamente, si tratta di una limitazione che impedisce di eseguire il montaggio di un volume EBS su una directory all'interno di una macchina EC2, dopo aver agganciato il volume all'istanza. Tramite documenti SSM, è possibile ovviare a questa limitazione, ed eseguire programmaticamente il montaggio e lo smontaggio di un volume da una macchina EC2, completando le operazioni necessarie per la creazione o eliminazione del nodo legato a questo volume.

5.2.3 Autoscaling Group EC2

Ogni regione è provvista di un autoscaling group 111, incaricato di mantenere sempre attiva una e solo una istanza che svolga le funzioni di bastion host e load balancer, tramite il servizio traefik in esecuzione su di essa.

Le risorse necessarie al funzionamento di questo autoscaling group sono un ruolo IAM, per conferire all'istanza tutti i permessi di cui necessita per assolvere alle sue mansioni, un launch template, che descrive tutti i dettagli di configurazione delle istanze, e l'autoscaling group stesso.

All'interno del launch template sono elencate numerose impostazioni, le quali parallelano gran parte delle configurazioni trovate nel menu di creazione istanza, sulla console di AWS. In particolare, il launch template è il punto dell'architettura dove viene specificato lo script di userdata, ossia una serie di comandi bash che l'istanza deve eseguire al suo primo avvio.

Nell'ordine, lo script userdata dell'istanza regionale svolge le seguenti mansioni:

- Installa tutte le utilità necessarie (tra cui l'agent di CloudWatch e l'interfaccia da riga di comando di AWS);
- Assegna l'IP elastico all'istanza, il quale viene associato alla sua interfaccia di rete primaria (eth0 nel caso di Amazon Linux 2, ens0 nel caso di Amazon Linux 2023);
- Abilita IP forwarding permanentemente (tramite modifica del file di configurazione di sysctl), e disabilita il ridirezionamento per le richieste ICMP in arrivo sull'interfaccia di rete primaria (così da tutelare la sicurezza delle istanze in subnet private);
- Abilita il protocollo NAT sull'interfaccia primaria;
- Rimuove le regole default di iptables che bloccano le richieste ICMP in arrivo;
- Configura l'agent di CloudWatch, impostando una serie di metriche sulle quali CloudWatch è autorizzato a raccogliere dati da visualizzare da dashboard di AWS;
- Configura l'agent di ECS, in particolare assegnando l'istanza al cluster della regione come capacity provider, e impostando un attributo che contrassegna l'istanza come dedicata a Traefik. Tale attributo sarà utilizzato da ECS per fare sì che nessun nodo sia eseguito sull'istanza, e che il servizio di Traefik esegua sempre su questa istanza.

Il launch template include inoltre una lista di security group da assegnare all'istanza.

Il primo di questi autorizza tutto il traffico in entrata proveniente da indirizzi IP privati (ovvero, contenuti nel range CIDR 10.0.0.0/8), ed è utilizzato anche da tutte le istanze dedicate ai nodi.

Il secondo invece autorizza il traffico in entrata proveniente da qualsiasi indirizzo IP, limitandolo però a una serie di porte, ciascuna delle quali è legata a una funzionalità del servizio di SQLite Cloud:

- 8080, usata temporaneamente durante la fase di sviluppo per accedere alla dashboard di Traefik, ora in disuso;
- 8090, usata per la API HTTP esposta dai nodi;
- 8860, usata per il traffico TCP con i nodi;
- 4000, usata per il traffico WebSocket con i nodi;
- 4000, usata per il traffico TCP interno tra i nodi. Si tratta della porta che i nodi appartenenti allo stesso cluster usano per svolgere le operazioni di sincronizzazione tra di loro.

Il traffico in uscita è invece illimitato per tutte le istanze.

5.2.4 Risorse di ECS

Come menzionato relativamente all'userdata, ogni regione è dotata di un cluster ECS, utilizzato per raggruppare tutti i servizi ECS, tra nodi e load balancer Traefik.

All'interno di questo cluster, di conseguenza, viene anche deployato un servizio di Traefik per ciascuna regione, configurato per ottenere l'immagine Docker da eseguire dalla repository ECR contenuta nell'infrastruttura globale.

Così come le istanze EC2 possono essere descritte tramite launch template, i servizi ECS sono dotati di una speciale risorsa di configurazione, detta task definition: al suo interno sono espresse nel dettaglio tutte le impostazioni relative a ogni task eseguita dal servizio, come assegnazioni di volumi, valori di memoria e CPU allocati, variabili d'ambiente e mappature di porte. La task definition relativa al servizio Traefik specifica le mappature tra le porte interne del container e le porte esterne dell'istanza, assegna al servizio i ruoli IAM necessari al corretto funzionamento, imposta i limiti di deployment tali per cui il servizio può essere eseguito solo sul bastion host, e specifica i valori di CPU e memoria che il container è autorizzato a consumare.

5.2.5 Chiavi KMS

Le chiavi crittografiche gestite da AWS sono una risorsa regionale, per cui è necessario deployare una copia di ciascuna copia in ciascuna regione. Per facilitare la gestione delle chiavi, così come per i ruoli di servizio Polarity ha riutilizzato il proprio template ufficiale di chiavi di KMS.

Le chiavi KMS deployate sono le seguenti:

- EBS, usata per cifrare tutti i volumi di tutti i nodi;

- SSM, usata per cifrare tutti i parametri di Systems Manager, usati internamente per condividere password e altri valori sensibili;

- Secrets Manager, usata per cifrare tutti i parametri segreti di Secrets Manager, tra cui i dati di accesso al database principale dell'applicativo backend di SQLite Cloud. Difatti, il servizio su cui il database è gestito, Amazon RDS, è direttamente integrato con Secrets Manager e deposita le credenziali generate automaticamente in un parametro segreto di Secrets Manager, configurato con funzionalità di sicurezza quali cifratura via KMS e rotazione automatica periodica.

5.2.6 Schema riassuntivo dell'infrastruttura regionale

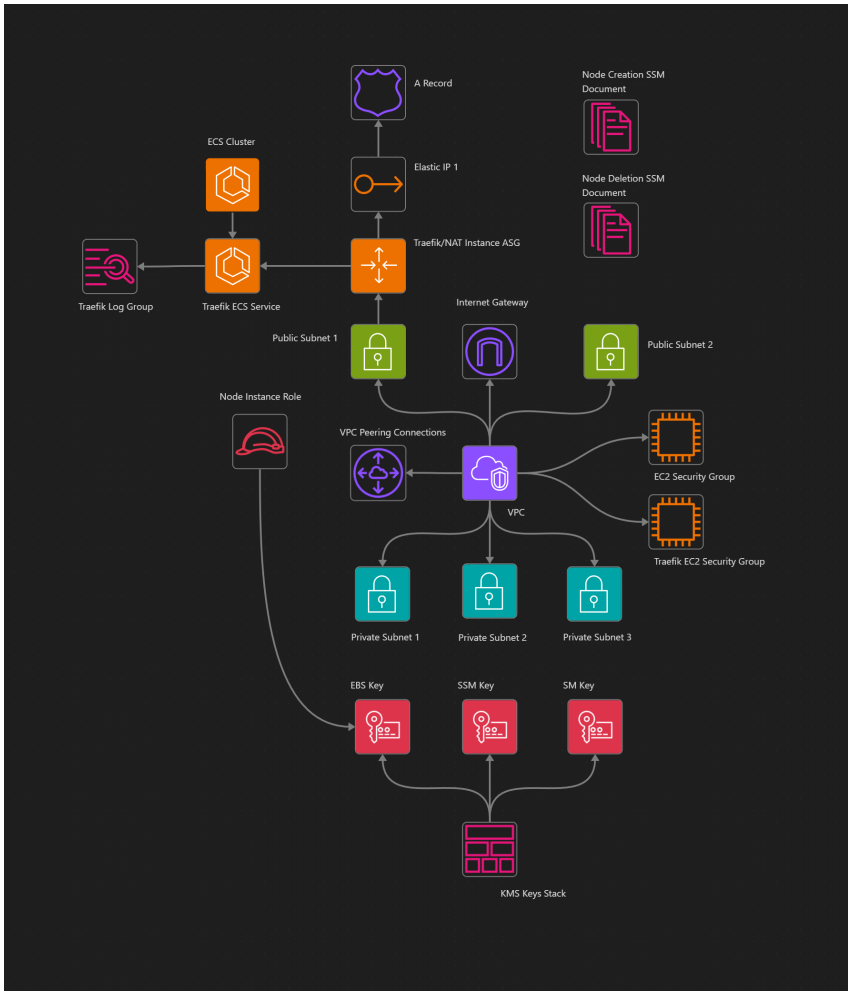

Figura 5.6: Grafico esplicativo dell'infrastruttura.

5.3 Infrastruttura Unitale

L'infrastruttura unitale è un insieme di elementi di infrastruttura che possono essere deployati un numero indefinito di volte a seconda delle esigenze dei clienti.

5.3.1 Istanze

Le istanze sono tecnicamente AutoScaling Group a singola istanza, usati come capacity provider per i nodi dei clienti e deployati dinamicamente dal backend di SQLite Cloud per supplire alle necessità computazionali.

Così come il bastion host, sono costituite da un autoscaling group 111, da un ruolo IAM provvisto di tutti i permessi necessari, e da un launch template che determina le impostazioni dell'istanza, tra cui lo script userdata gioca un ruolo molto importante. In particolare, è studiato per due casistiche di esecuzione: la primissima esecuzione, svolta alla creazione dell'autoscaling group, e le esecuzioni successive, causate da fallimenti o terminazioni manuali della macchina, a seguito di cui una nuova macchina viene creata. I due differenti comportamenti sono illustrati nel seguente diagramma di flusso.

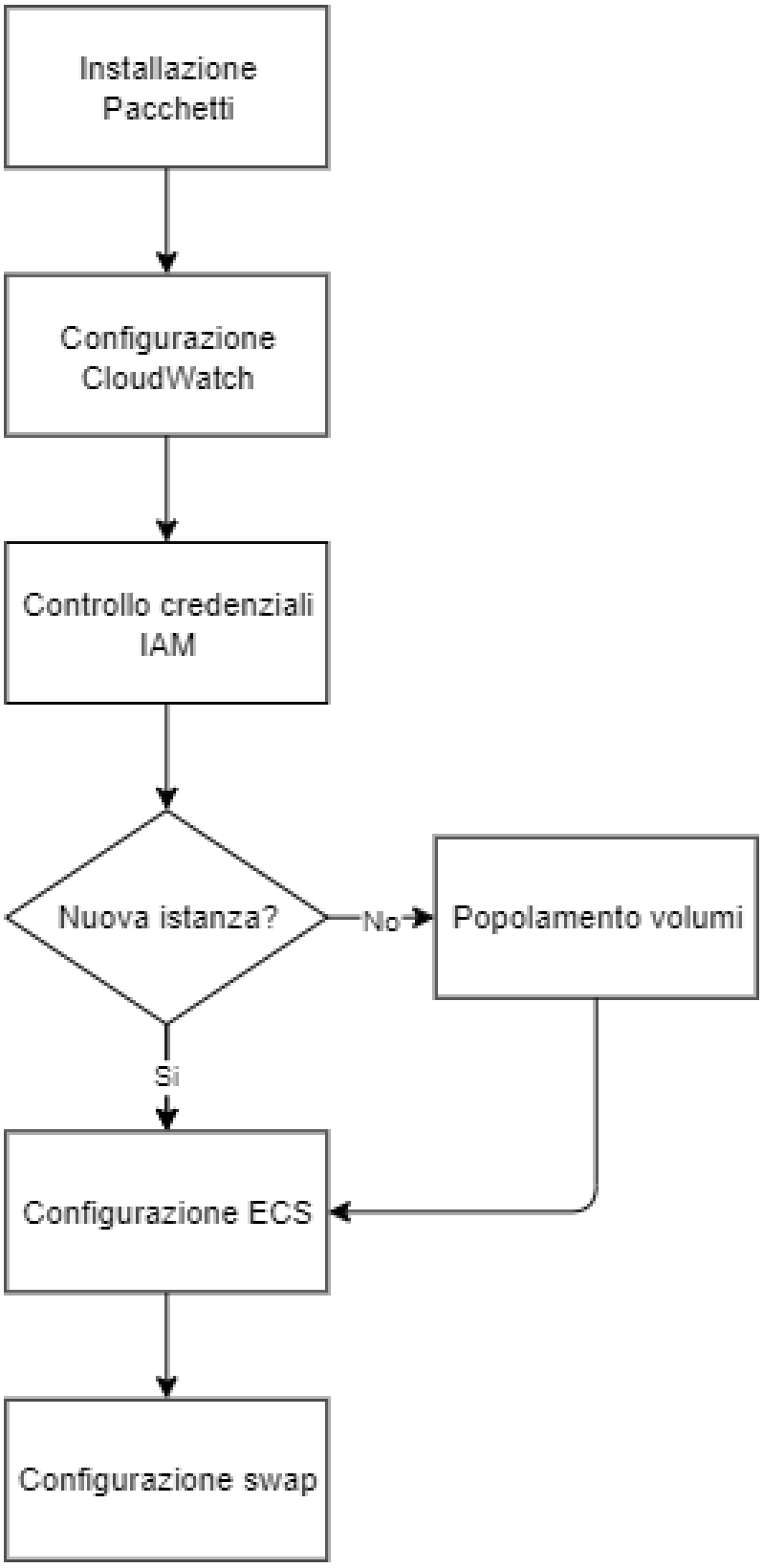

Figura 5.7: Diagramma di flusso della creazione di nuove istanze da parte dell'autoscaling group.

Nell'ordine, lo script userdata dell'istanza regionale svolge le seguenti mansioni:

- Installa tutte le utilità necessarie (tra cui le interfacce da riga di comando di NVME e di AWS);
- Configura l'agent di CloudWatch, impostando una serie di metriche sulle quali CloudWatch è autorizzato a raccogliere dati da visualizzare da dashboard di AWS;
- Verifica se le credenziali IAM sono state già assegnate all'istanza, e in caso contrario attende qualche secondo e riprova (questo passaggio è stato aggiunto dopo aver riscontrato problemi di esecuzione degli step successivi, causati dalla durata variabile dell'assegnazione dei permessi alle macchine)
- Filtrando in base alle etichette, Ottiene una lista di tutti i volumi EBS assegnati alla macchina (nessuno, se l'istanza è stata appena creata per la prima volta; altrimenti, recupera tutti i volumi dedicati ai nodi assegnati alla macchina);
- Aggancia i volumi così ottenuti alla macchina, e li monta sulle relative directory;
- Configura l'agent di ECS, in particolare assegnando l'istanza al cluster della regione come capacity provider, e impostando un attributo che identifica l'istanza con un codice univoco. Tale attributo sarà utilizzato da ECS per fare sì che i nodi assegnati a tale istanza siano sempre eseguiti solo su di essa, e mai su un'altra istanza;
- Imposta una partizione di swap del volume root.

Essendo la complessità delle operazioni svolte particolarmente elevata, gran parte delle procedure sono state suddivise in funzioni e sottofunzioni, dichiarate nella prima parte dello script userdata e chiamate in seguito nell'ordine previsto. In particolare, sono state definite le seguenti funzioni:

- `retry_command`: questa funzione definisce un meccanismo di retry a intervallo fisso, utile per riprovare operazioni che dipendono dalla conclusione di altre operazioni esterne.
- `exponential_retry_command`: analoga a `retry_command`, ma dotata di una selezione pseudo-casuale dell'intervallo a ogni iterazione, detta 'exponential backoff', più adatta per operazioni che possono essere penalizzate da una esecuzione in rapida successione.
- `aws_prepare_volume`: funzione che raggruppa tutte le chiamate alla API di AWS necessarie ad assegnare un volume a un'istanza (sgancio da un'eventuale precedente istanza, attesa della disponibilità del volume, aggancio del volume all'istanza corrente e attesa del completamento delle operazioni), e che esegue ciascuna di tali chiamate tramite `exponential_retry_command`, così che, qualora siano raggiunti i limiti massimi di chiamate alla API di AWS, la frequenza delle operazioni sia automaticamente ridotta in modo da rientrare nelle quantità ammesse.
- `find_nvme_device_path_by_volume_id`: funzione che, dato un id di volume AWS, ottiene e ritorna il percorso dell'utilità di sistema NVME corrispondente al montaggio sulla macchina di dato volume.

- `check_iam_credentials`: funzione che tenta un numero predefinito di volte di ottenere le credenziali assegnate all'istanza, attendendo 10 secondi tra ciascun tentativo, e ritornando errore nel caso in cui ciascun ciclo sia terminato senza successo.

- `populate_volume_info`: funzione che identifica tutti i volumi assegnati alla data istanza, leggendo le etichette sui volumi e cercando i volumi etichettati con l'ID dell'istanza, e in seguito per ciascun volume esegue `aws_prepare_volume` e `find_nvme_device_path_by_volume_id` tramite `retry_command`. Quindi, monta il volume sulla directory corrispondente al nodo proprietario del volume.

5.3.2 Nodi

I nodi sono servizi ECS a singola task, provvisti di un volume EBS per immagazzinarne i dati e di un record DNS dedicato per consentire all'utente proprietario di accedervi direttamente. Essi vengono deployati sulla prima istanza EC2 disponibile e vi sono assegnati indeterminatamente: questo significa che qualora la task di un nodo fallisse, verrebbe immediatamente ricreata sulla stessa istanza, così da ricollegarsi subito allo stesso volume EBS.

Il volume EBS viene etichettato con due valori molto importanti: l'id del nodo e l'id dell'istanza a cui è assegnato. Queste etichette vengono poi utilizzate dagli script bash integrati nell'infrastruttura per individuare i volumi relativi a ciascun nodo e istanza, e svolgere le appropriate operazioni di gestione.

Dopo aver creato il volume, ma prima di creare il servizio ECS, la procedura di creazione del nodo richiede l'esecuzione sull'istanza designata del documento SSM di creazione. Come anticipato a proposito dell'infrastruttura regionale, il compito di questo documento è individuare il volume appena creato e montarlo sull'istanza prima dell'esecuzione del servizio. L'ordine cronologico di queste operazioni è cruciale: difatti, se il servizio venisse eseguito prima del montaggio del volume, il nodo comincerebbe a scrivere dati sulla directory del disco root dell'istanza, e il montaggio del volume andrebbe poi a sovrascrivere i dati inseriti fino a quel punto, causando grossi problemi nel funzionamento dell'applicazione.

Così come la creazione del nodo prevede l'esecuzione del documento SSM di creazione, anche l'eliminazione del nodo prevede l'esecuzione del documento SSM di eliminazione, il quale invece si occupa di smontare e sganciare il volume dall'istanza, permettendone l'eliminazione senza errori.

Inoltre, nel caso di operazioni in blocco, è possibile che l'istanza non sia ancora completamente operativa, ragion per cui viene eseguita una funzione di attesa prima dell'esecuzione del documento, la quale verifica che l'istanza sia stata configurata completamente, e solo allora consente il proseguimento della procedura. Allo stesso modo, una seconda funzione di attesa sospende le operazioni dopo l'esecuzione del documento SSM, in quanto il montaggio del volume può richiedere tempo. Il successo di entrambe le operazioni è comunicato tramite assegnazione di etichette alle rispettive risorse, per cui le funzioni di attesa controllano periodicamente la lista di etichette assegnate, alla ricerca dell'etichetta di conferma. Questa dinamica è illustrata nel seguente diagramma di flusso.

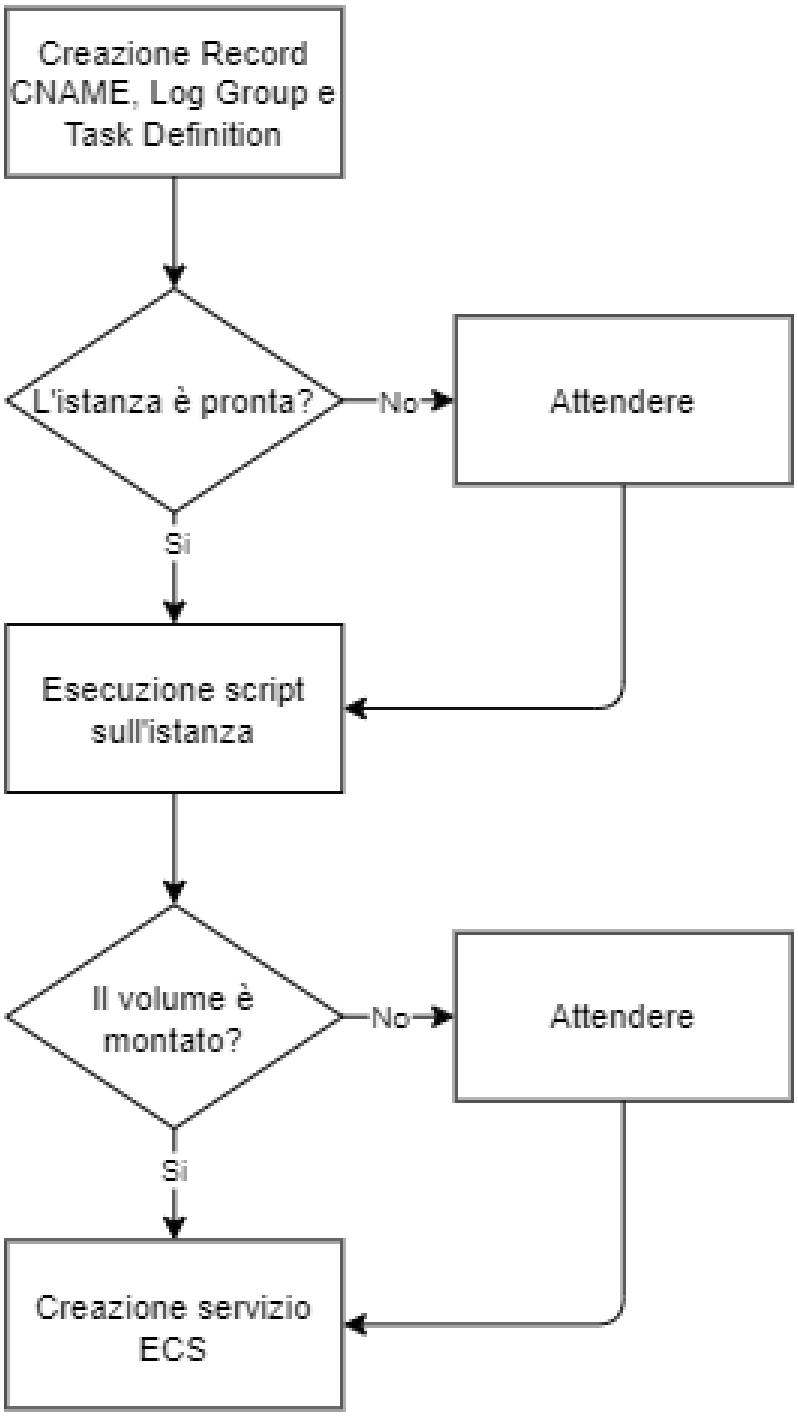

Figura 5.8: Diagramma di flusso della creazione di nuovi nodi.

Il servizio ECS di ciascun nodo è provvisto di una task definition che descrive la configurazione di tutti i task da esso amministrati, tra cui il percorso del bind mount che collega il container al volume EBS (la stessa directory su cui l'istanza ha solitamente montato suddetto volume), le numerose variabili d'ambiente di cui necessita il servizio, le mappature di porte e una serie di etichette proprietarie di Docker, dette Docker Labels.

Il ruolo di queste etichette è determinare le regole di routing a cui deve obbedire il load balancer Traefik. Difatti, una volta che il load balancer individua un nuovo container all'interno del cluster che sta monitorando, legge tutte le Docker Labels assegnate a questo container e segue le relative indicazioni per costruire le connessioni verso questo container.

Ad esempio, è possibile specificare tramite Docker Labels una regola che prevede il routing verso quel container di tutte le richieste provenienti da un indirizzo DNS che corrisponda a un'espressione regolare, il che è il meccanismo su cui si basa l'intero sistema di separazione dei flussi di traffico tra utenti.

Inoltre, la task definition del nodo definisce una serie di health check, ossia procedure eseguite periodicamente per certificare il corretto funzionamento del servizio, e notificare ECS in caso contrario, affinchè possa sostituire la task malfunzionante con una nuova task.

Una nota relativa al funzionamento del nodo è che la sua esistenza deve sempre essere contenuta nell'esistenza dell'istanza su cui è hostato; di conseguenza, nel backend di SQLite Cloud sono state implementate logiche ferree per assicurarsi che nessun autoscaling group potesse essere eliminato prima dell'eliminazione di tutti i nodi ad esso assegnati, e allo stesso modo che nessun nodo potesse essere creato prima che l'autoscaling group ad esso assegnato fosse stato creato.

5.3.3 Cluster

I cluster sono tecnicamente insiemi di record DNS con polizza di routing a latenza, i quali vengono espansi e ridotti dinamicamente per consentire l'accesso a tutti i nodi appartenenti al cluster sparsi in tutte le regioni.

Il loro funzionamento è peculiare rispetto al funzionamento di nodi e istanze, dato che anzichè esistere in uno stato binario (cluster esistente, cluster non esistente) esistono su uno spettro di possibili stati, codificati come una permutazione di tutte le possibili regioni su cui un dato cluster può essere dotato di record DNS.

Di conseguenza, le funzioni coinvolte sono 4: creazione di cluster, estensione di cluster, riduzione di cluster e eliminazione di cluster. A loro volta, queste funzioni sono raggruppate in coppie, con esecuzioni differenti in base al numero di regioni attualmente esistenti.

La creazione di un cluster equivale all'estensione di un cluster presente su zero regioni. Durante la procedura, viene creato il primo record DNS con routing a latenza, denominato con l'ID del cluster e diretto verso il record DNS del load balancer Traefik della data regione. Quindi, viene aggiunto allo stack di Pulumi un oggetto di output, costituito da una lista di regioni su cui il cluster è propagato, e che al momento include solamente la regione di partenza.

L'estensione di un cluster esistente consiste nel recupero dinamico dell'output e nella lettura delle regioni in esso contenute: quindi, viene aggiunta la nuova regione e viene eseguita nuovamente la creazione/modifica del cluster. Così facendo, le regioni già esistenti non sono alterate, mentre la nuova regione viene creata, tramite il deployment del nuovo record DNS.

La riduzione di un cluster esistente funziona analogamente: dopo aver recuperato l'output, la procedura trova la regione che è stato richiesto di eliminare e la rimuove dalla lista. Poi, viene eseguita nuovamente la creazione/modifica, e mentre tutte le altre regioni rimangono intatte, la regione designata viene eliminata.

L'eliminazione di un cluster viene attivata come sottoprocesso della riduzione, nel momento in cui, al recupero dell'output, si verifica che la regione da eliminare è l'ultima regione del cluster. Di conseguenza, viene eliminato l'intero stack tramite la API di Pulumi, allo stesso modo in cui vengono eliminati nodi e istanze.

A valle di tutte le considerazioni svolte, si può dunque delineare la struttura della rete di record DNS come costituita da:

- Un record A per ogni regione, il quale punta all'indirizzo IP elastico del corrispondente load balancer Traefik;

- Un record CNAME per ogni nodo, il quale punta al record del load balancer Traefik della rispettiva regione;

- Un record CNAME per ogni cluster, moltiplicato per ciascuna regione sulla quale si trova il cluster, dove ogni istanza del record punta al record del load balancer Traefik della rispettiva regione, con polizza a latenza per stabilire quale record verrà fornito al resolver DNS.

Di seguito, un grafico riassuntivo della hosted zone.

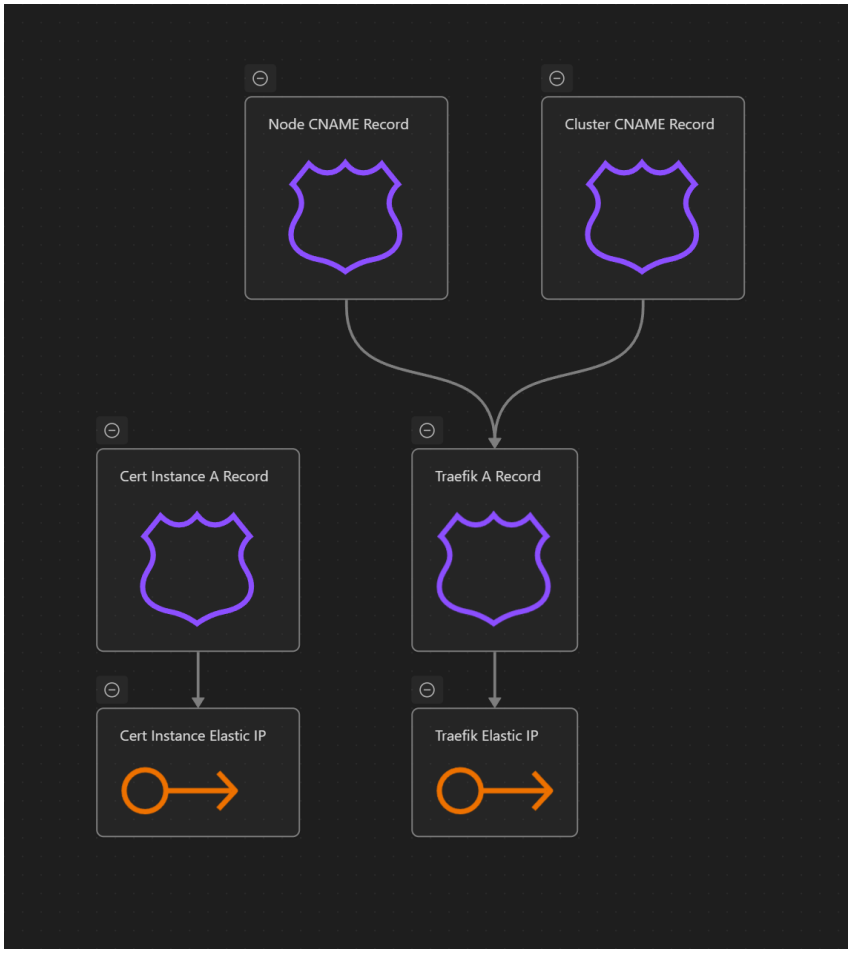

Figura 5.9: Struttura dei record DNS in uso.

Capitolo 6

Operazioni terminali

Durante il ciclo di vita dell'architettura, sono emerse ulteriori necessità e alcuni problemi non previsti che era necessario risolvere. Di conseguenza, sono stati svolti vari interventi di manutenzione e miglioramento sull'infrastruttura, i quali sono riepilogati di seguito.

6.1 Aggiornamento degli ECS container agent

Il container agent di ECS è il processo che viene installato ed eseguito sulle istanze EC2 per consentire di eseguire al loro interno dei servizi ECS.

Essendo piuttosto raro che le macchine EC2 fallissero, molte di esse rimanevano in funzione per parecchie settimane ininterrottamente. A causa di ciò, il container agent in funzione su queste istanze poteva restare parecchie versioni indietro, compromettendo la stabilità del servizio.

Fortunatamente, la API di ECS espone un metodo apposito per eseguire l'aggiornamento del container agent su una data istanza, senza alcuna interruzione del servizio, ma l'esecuzione del metodo deve essere automatizzata dall'utente.

Per ogni regione, il candidato ha sviluppato una semplice funzione Lambda che ottenesse l'elenco di istanze registrate come capacity provider per un dato cluster e chiamasse il metodo UpdateContainerAgent per ciascuna istanza. Dopodichè, è bastato creare una event rule di CloudWatch con cadenza giornaliera, assegnarle la funziona Lambda come azione da svolgere, e conferire alla funzione un ruolo IAM dotato di tutti i permessi necessari a svolgere la manutenzione.

Una volta effettuato il deployment della modifica, ogni giorno la lambda sarebbe stata eseguita in ogni regione sul rispettivo cluster, aggiornando tutte le istanze EC2 qualora fosse disponibile una nuova versione del container agent di ECS.

Il funzionamento del sistema è riportato nel seguente grafico.

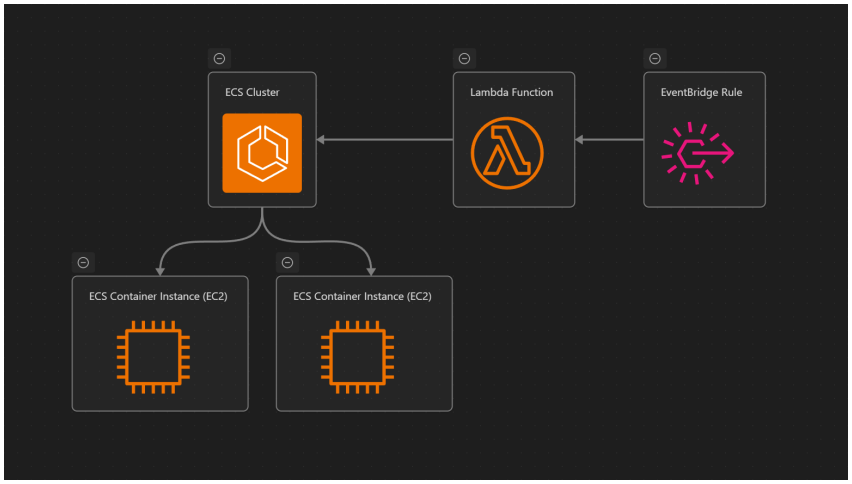

Figura 6.1: Design del meccanismo di aggiornamento automatico dei container agent.

6.2 Pulizia dei file di backup di Pulumi

Un dettaglio imprevisto del funzionamento di Pulumi è che all'eliminazione di uno stack, il file di backup del relativo file di stato non viene mai eliminato, probabilmente per consentire un recupero dello stato in caso di eliminazioni accidentali. Ciò però causava nel caso di SQLite Cloud un effetto collaterale indesiderato, dato che l'eliminazione di un nodo da parte di un utente era un evento piuttosto frequente.

Di conseguenza, il bucket S3 contenente lo stato di Pulumi era disseminato di file di backup di vecchi nodi, istanze e cluster. Le dimensioni totali dei file backup erano ancora trascurabili, ma la cosa intralciava le operazioni di manutenzione, e per porvi rimedio prima che diventasse un problema serio si decise di intervenire.

Analogamente all'aggiornamento dei container agent, la soluzione prevedeva una event rule di CloudWatch giornaliera, che avrebbe attivato una funzione Lambda. La funzione avrebbe visitato l'intero bucket S3 dello stato di Pulumi, e avrebbe filtrato i file backup orfani (ovvero, privi del corrispettivo file di stato), per poi eliminarli. A differenza dell'aggiornamento dei container agent, per svolgere questa operazione era sufficiente una singola copia di ogni risorsa, per cui le modifiche vennero apportate al template globale.

Una volta deployate le modifiche, il bucket S3 ha subito una riduzione non trascurabile di dimensioni totali, ed è diventato generalmente molto più semplice da consultare durante le successive operazioni di manutenzione.

6.3 Specificazioni di risorse e CIDR per regione

Dopo aver lanciato la open beta del servizio, una delle ottimizzazioni più importanti pianificate era la possibilità di aumentare le risorse di nodi e istanze in maniera indipendente per ciascuna regione. L'utilità di questo miglioramento è presto detta: analizzando le metriche delle varie regioni, era chiaro che in alcune come Nord Virginia e Francoforte le risorse necessarie fossero molto elevate, mentre in altre regioni più piccole e meno popolate ci fosse un consumo molto più ridotto delle risorse allocate, rendendo possibile effettuare un importante taglio dei costi.

Ulteriore necessità sorta nel mentre è stata quella di stabilire a priori i range CIDR per tutte le reti VPC che sarebbero potute essere introdotte in futuro, dato che per effettuare il peering tra tutte le reti è necessario che non ci sia alcuna sovrapposizione tra i range di indirizzi IP di ciascuna delle reti.

Per soddisfare queste necessità, Polarity ha reingegnerizzato il template regionale, introducendo un file di costanti e un file di variabili. Il file di costanti avrebbe contenuto i CIDR range riservati a ciascuna regione (incluse le regioni non ancora deployate), mentre il file di variabili avrebbe contenuto le specificazioni di memoria e CPU da dedicare ai servizi di ciascuna regione, rendendo la modifica il più semplice possibile.

È riportato a titolo esemplificativo il file delle costanti, con tutti gli intervalli CIDR definiti.

```
import { Region } from "@pulumi/aws";

export const regionVpcIpMap: { [key in Region]: string[] } = {
  "eu-central-1": ["10.20", "10.21", "10.22", "10.23"],
  "eu-west-1": ["10.24", "10.25", "10.26", "10.27"],
  "eu-west-2": ["10.28", "10.29", "10.30", "10.31"],
  "eu-west-3": ["10.32", "10.33", "10.34", "10.35"],
  "eu-north-1": ["10.36", "10.37", "10.38", "10.39"],
  "sa-east-1": ["10.40", "10.41", "10.42", "10.43"],
  "us-east-1": ["10.44", "10.45", "10.46", "10.47"],
  "us-east-2": ["10.48", "10.49", "10.50", "10.51"],
  "us-west-1": ["10.52", "10.53", "10.54", "10.55"],
  "us-west-2": ["10.56", "10.57", "10.58", "10.59"],
  "ap-south-1": ["10.60", "10.61", "10.62", "10.63"],
  "ap-northeast-1": ["10.64", "10.65", "10.66", "10.67"],
  "ap-northeast-2": ["10.68", "10.69", "10.70", "10.71"],
  "ap-northeast-3": ["10.72", "10.73", "10.74", "10.75"],
  "ap-southeast-1": ["10.76", "10.77", "10.78", "10.79"],
  "ap-southeast-2": ["10.80", "10.81", "10.82", "10.83"],
  "ca-central-1": ["10.84", "10.85", "10.86", "10.87"],
  "af-south-1": ["10.88", "10.89", "10.90", "10.91"],
  "ap-east-1": ["10.92", "10.93", "10.94", "10.95"],
  "ap-south-2": ["10.96", "10.97", "10.98", "10.99"],
  "ap-southeast-3": ["10.100", "10.101", "10.102", "10.103"],
  "ap-southeast-4": ["10.104", "10.105", "10.106", "10.107"],
  // @ts-expect-error - Type is missing regions ca-west-1 and il-central-1
  "ca-west-1": ["10.108", "10.109", "10.110", "10.111"],
  "eu-south-1": ["10.112", "10.113", "10.114", "10.115"],
  "eu-south-2": ["10.116", "10.117", "10.118", "10.119"],
  "eu-central-2": ["10.120", "10.121", "10.122", "10.123"],
  "il-central-1": ["10.124", "10.125", "10.126", "10.127"],
  "me-south-1": ["10.128", "10.129", "10.130", "10.131"],
  "me-central-1": ["10.132", "10.133", "10.134", "10.135"]
};
```

Figura 6.2: Elenco di tutti i CIDR, estratto e compilato a priori.

6.4 Rinnovo automatico dei certificati TLS

Per rendere l'infrastruttura completamente autonoma, c'era ancora un elemento importante da automatizzare, ossia il rinnovo del certificato TLS, che fino a quel momento era ancora stata svolta manualmente dal candidato a ridosso della scadenza del precedente certificato.

L'estensione dell'architettura per accomodare questa necessità non è stata banale: innanzitutto, il candidato ha creato un nuovo record DNS, che sarebbe stato utilizzato da letsencrypt per raggiungere l'istanza. A tale scopo, il record punta a un nuovo indirizzo IP elastico.

Quindi, i certificati sono stati caricati sul bucket S3 dei file d'ambiente, sotto un prefisso apposito, ed è stata sviluppata una funzione Lambda giornaliera che verificasse l'età dei file dei certificati e, qualora fosse superiore a una soglia, inviasse una notifica su un topic di SNS (Simple Notification Service, un servizio in grado di inviare mail informative agli amministratori del sistema). In questo modo, se qualcosa fosse andato storto con il rinnovo dei certificati, gli amministratori sarebbero stati informati almeno trenta giorni prima della scadenza del precedente certificato, conferendo loro sufficiente preavviso per risolvere il problema prima che influenzi l'ambiente di produzione.

Per eseguire automaticamente il rinnovo, il candidato ha aggiunto all'infrastruttura una singola istanza t2.micro, la classe di istanze più piccola e meno costosa di AWS, incaricata di svolgere i seguenti compiti:

- Installare acme.sh, lo strumento necessario per rinnovare i certificati;
- Assegnarsi l'indirizzo IP elastico, così che il nuovo record DNS punti all'istanza;
- Creare uno script bash che svolga la procedura di rinnovo, ed eseguirlo per la prima volta;
- Creare un servizio e un timer di SystemD per eseguire ogni dodici ore lo script bash.

La procedura dello script bash è la seguente: l'istanza recupera i certificati attuali dal bucket S3, li analizza con il comando openssl per verificarne la data di scadenza e, se la data di scadenza è entro trenta giorni, l'istanza esegue il rinnovo tramite il comando acme.sh –issue, specificando dns_aws come servizio DNS.

Grazie alla specificazione del servizio DNS, il comando acme.sh può assumere il controllo della hosted zone sull'account AWS (in quanto l'istanza è provvista dei permessi necessari per farlo), identificare il record DNS che punta all'istanza e utilizzarlo per eseguire l'autenticazione. Dopodichè, il nuovo certificato viene generato sull'istanza EC2, al che procede ad essere caricato sul bucket S3 sostituendo il certificato precedente.

L'istanza è poi salvata tramite AWS Backup una volta al giorno, consentendo di ripristinarla in caso di fallimento. Inoltre, è presente un'allarme CloudWatch che osserva la salute dell'istanza, e notifica lo stesso topic SNS della funzione Lambda qualora l'istanza venga terminata inaspettatamente.

A concludere la reingegnerizzazione sono alcune modifiche all'immagine Docker dei load balancer Traefik, nonchè all'userdata delle istanze dedicate a suddetti load balancer: la configurazione del load balancer è modificata in modo da recuperare da bind mount di Docker il file certificato, mentre l'userdata crea servizio e timer di systemD, con cadenza bigiornaliera, per scaricare il certificato da S3 e salvarlo nel percorso specificato dal bind mount.

In questo modo, ogni dodici ore tutte le istanze Traefik scaricano il certificato, sia esso aggiornato o meno, e lo passano ai container Docker in esecuzione su di esse, così che, qualora il certificato fosse aggiornato, i processi Traefik possano riceverlo entro dodici

ore e ricaricarsi senza alcuna interruzione del servizio. Contemporaneamente, ogni dodici ore l'istanza che svolge la certificazione automatica verifica che il certificato non sia vicino alla scadenza, e in caso contrario effettua il rinnovo e carica il nuovo certificato su S3, rendendolo disponibile per tutte le istanze Traefik. Infine, il fuzionamento del tutto è monitorato attentamente, con notifiche automatiche impostate per avvisare gli amministratori in caso qualcosa andasse storto.

L'architettura del sistema è riportata nel seguente grafico.

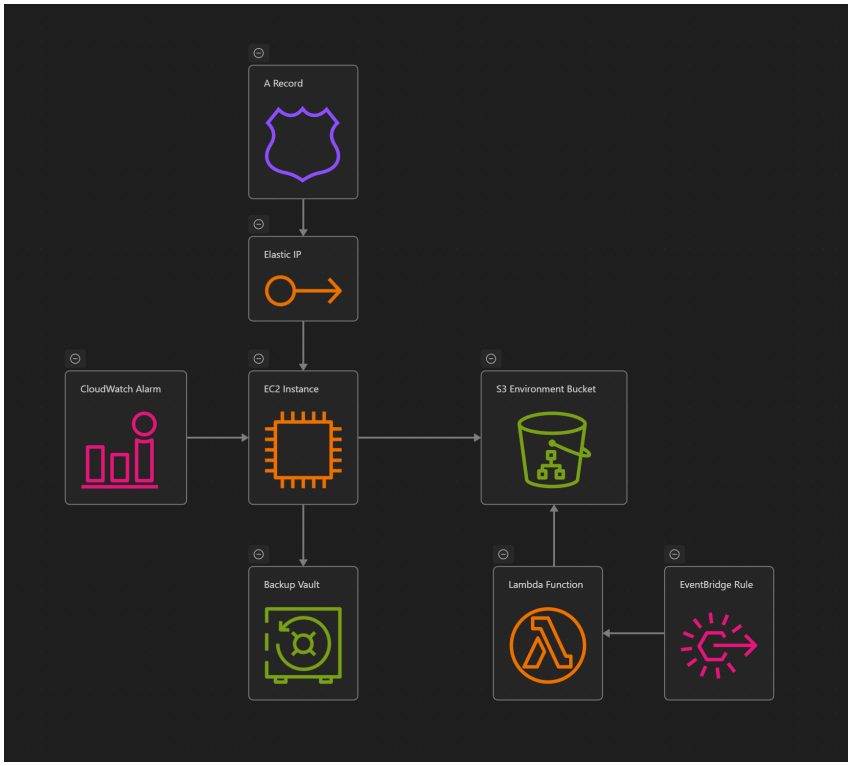

Figura 6.3: Design del meccanismo di aggiornamento automatico dei certificati.

6.5 Migrazioni tra ambienti di staging e produzione

Come da migliori pratiche, tutte le modifiche all'infrastruttura sono sempre state prima testate nell'ambiente staging, per poi essere replicate sull'infrastruttura dell'ambiente di produzione tramite interfaccia da riga di comando di Pulumi.

Il procedimento è stato facilitato dagli strumenti di analisi delle modifiche proposte che Pulumi offre: prima di eseguire qualsiasi operazione di aggiornamento, Pulumi stila una lista delle differenze tra l'infrastruttura descritta dal codice che si vuole deployare e l'infrastruttura effettivamente presente, registrata nello stato di Pulumi. In questo modo, è possibile verificare che tutte le risorse da modificare corrispondano alle aspettative, e in caso contrario è possibile interrompere l'operazione prima ancora che abbia avuto inizio, per poi indagare e risolvere eventuali problemi.

Molto spesso, difatti, modifiche su alcune risorse avevano effetti collaterali imprevisti su altre risorse, e la strumentazione di Pulumi è stata cruciale per permettere a Polarity di prevenire disastri in produzione.

Per quanto riguarda la disponibilità del servizio, ovunque possibile le operazioni di migrazione sono state svolte con la minima interruzione possibile. Tuttavia, non sempre ciò era possibile, per la natura ancora incompleta e in fase di evoluzione dell'infrastruttura, ragion per cui alcune operazioni di manutenzione sono state svolte previo dispiegamento di avvisi agli utenti della beta, con la consapevolezza che essendo un servizio in fase di sviluppo era ancora ammissibile che subisse interruzioni minori.

6.6 Migrazioni di stato di Pulumi

Durante le migrazioni, un ostacolo importante da considerare era costituito dalle differenze tra lo stato dell'ambiente staging e quello dell'ambiente di produzione. Difatti, essendo l'ambiente staging costituito da molte meno risorse rispetto a quello di produzione, ed essendo spesso implementate in staging modifiche che richiedevano sostituzioni non immediate di alcuni elementi di infrastruttura (causando disagi agli utenti), è stato spesso necessario analizzare gli stati e aggiornarli manualmente, talvolta facendo uso di script sviluppati ad hoc.

6.7 Migrazione da Amazon Linux 2 a Amazon Linux 2023

Durante il ciclo di vita dell'infrastruttura, è emersa un'importante scadenza: il supporto per Amazon Linux 2 stava per terminare, ed era necessario migrare tutte le istanze alla versione successiva della distribuzione, Amazon Linux 2023.

Il candidato ha analizzato tutti gli userdata di tutte le macchine, e ha sperimentato con nuove macchine di prova equipaggiate con Amazon Linux 2023. Una volta aggiornati tutti gli elementi degli userdata in modo da accomodare le modifiche apportate alla distribuzione, è stata programmata una sostituzione graduale di tutte le macchine Amazon Linux 2 con le nuove macchine Amazon Linux 2023, e il processo di migrazione si è svolto senza intoppi.

6.7.1 Plugin di Docker personalizzato

Una reingegnerizzazione molto importante ipotizzata era la seguente: anzichè gestire i volumi tramite userdata, si sarebbe potuto utilizzare il plugin Docker di RexRay compatibile con EBS, così da lasciare che fosse il plugin a svolgere tutte le operazioni necessarie al corretto funzionamento dei volumi. Sfortunatamente, con l'aggiornamento ad Amazon Linux 2023 il plugin RexRay è diventato incompatibile, ragion per cui l'unica soluzione era scrivere un plugin personalizzato di Docker.

Il candidato si è cimentato nella sua realizzazione, studiando le API di Docker e sviluppando una prima versione del plugin, in grado di montare e smontare volumi su richiesta. Questa innovazione non è mai stata utilizzata nel progetto di SQLite Cloud, ma è stata terminata e resa disponibile per i successivi progetti ad opera di Polarity.

6.8 Traduzione da Pulumi a CDKTF

Il progetto si è concluso con un'ultima reingegnerizzazione, che ha portato alla migrazione dell'infrastruttura su Cloud Development Kit for Terraform (CDKTF), operazione che si è resa necessaria per realizzare una nuova versione dell'infrastruttura amministrata internamente da SQLite e basata su Kubernetes. Difatti, durante l'anno circa di servizio della prima infrastruttura, sono stati rilevati importanti problemi inerenti ai requisiti, che hanno reso necessaria e preferibile una completa revisione dell'architettura, e hanno reso evidente la necessità di effettuare l'insourcing del lato cloud del progetto.

Il reparto infrastruttura di Polarity ha quindi svolto la totale conversione dell'infrastruttura globale e regionale in CDKTF, operazione che è stata accompagnata da numerosi incontri di passaggio di consegna e consulenza sul design architetturale con il nuovo reparto infrastruttura di SQLite Cloud.

CKDTF è uno strumento costruito sopra Terraform, che consente di scrivere infrastruttura in linguaggi di programmazione reali (similarmente a Pulumi), per poi compilare automaticamente dei template di Terraform a partire da quanto scritto. È stato scelto CDKTF per compensare l'impossibilità di sviluppare template dinamici in Terraform, così da poter trasferire l'intero funzionamento dell'infrastruttura esistente. A sua volta, era necessario che i template risultanti fossero sviluppati in Terraform, in quanto si tratta del paradigma infrastructure-as-code con cui il nuovo reparto infrastruttura di SQLite Cloud si trovava generalmente più a proprio agio.

Di seguito è riportato un estratto dal codice CDKTF sviluppato per l'occorrenza.

```
for (var i = 0; i < providers.length - 1; i++) {
    for (var j = i + 1; j < providers.length; j++) {

      // Peering Connection

      const peering = new VpcPeeringConnection(stack, `kubernetes-vpc-peering-${providers[i].region}-${providers[j].region}`, {
        provider: providers[i].provider,
        vpcId: providers[i].vpcId,
        peerVpcId: providers[j].vpcId,
        peerRegion: providers[j].region,
        autoAccept: false,
        tags: {
          Name: `kubernetes-vpc-peering-${providers[i].region}-${providers[j].region}`,
          setup: "cdktf",
          project: "kubernetes",
          environment: environment
        }
      })

      const accepter = new VpcPeeringConnectionAccepterA(stack, `kubernetes-vpc-peering-accepter-${providers[i].region}-${providers[j].region}`, {
        provider: providers[j].provider,
        vpcPeeringConnectionId: peering.id,
        autoAccept: true,
        tags: {
          Name: `kubernetes-vpc-peering-${providers[i].region}-${providers[j].region}`,
          setup: "cdktf",
          project: "kubernetes",
          environment: environment
        }
      })

      new VpcPeeringConnectionOptions(stack, `kubernetes-vpc-peering-options-${providers[i].region}-${providers[j].region}`, {
        provider: providers[i].provider,
        vpcPeeringConnectionId: peering.id,
        requester: {
          allowRemoteVpcDnsResolution: true,
        },
        dependsOn: [peering, accepter],
      })

      new VpcPeeringConnectionOptions(stack, `kubernetes-vpc-peering-options-${providers[j].region}-${providers[i].region}`, {
        provider: providers[j].provider,
        vpcPeeringConnectionId: peering.id,
        accepter: {
          allowRemoteVpcDnsResolution: true,
        },
        dependsOn: [peering, accepter]
      })
```

Figura 6.4: Estratto dalla traduzione dell'infrastruttura in codice CDKTF.

Capitolo 7

Conclusioni

La scala di un progetto influisce notevolmente sull'insieme di capacità e conoscenze richieste per portarlo a termine. Nel caso di SQLite Cloud, l'alta complessità dell'architettura desiderata e l'altrettanto elevato numero di risorse coinvolte hanno comportato molteplici ostacoli da superare, e hanno messo alla prova le conoscenze maturate dal candidato nel corso della sua carriera presso l'azienda.

Nel corso dello sviluppo dell'infrastruttura, più volte sono stati raggiunti limiti imposti da AWS, ed è stato necessario trovare soluzioni alternative che permettessero di aggirarli pur soddisfacendo i requisiti. Ciò ha richiesto uno studio attento e meticoloso dei servizi messi a disposizione dalla piattaforma cloud, e un altrettanto scrupoloso lavoro di ricerca sulle pratiche più diffuse per risolvere i problemi più comuni.

Tali sfide si sono estese anche alle tecnologie al di fuori di AWS: per raggiungere al meglio i requisiti, sono state scelte tecnologie nuove e poco utilizzate fino a quel punto dall'azienda, il che ha reso necessario lo studio di nuovi paradigmi e metodologie da parte del candidato. Le tecnologie e i linguaggi appresi nel corso del progetto sono stati poi adottati, sia dal candidato che dall'azienda stessa, come soluzioni alternative per alcune necessità, se non standard interni diffusi tra tutti i membri del team.

Quando a presentarsi sono stati invece problemi più nuovi e peculiari, è stato necessario ragionare su soluzioni nuove o adattate, mettendo in pratica competenze di sviluppo di algoritmi e ingegneria del software, così da superare anche gli ostacoli più oscuri e sconosciuti.

Un fattore che ha contribuito in maniera importante alla complessità del lavoro è stato la combinazione tra la costante evoluzione dei requisiti, con la conseguente necessità di aggiornare continuamente il design dell'architettura, e la difficoltà di lavorare su un'infrastruttura già in uso sia dal team di sviluppo, che da una prima demografica di utenti, in quanto il progetto è stato aperto in fase di beta durante lo svolgimento dei lavori. Ciò ha richiesto doppia attenzione e meticolosità nello studiare le modifiche all'infrastruttura, onde evitare di causare problemi all'ambiente di produzione una volta che fosse stato necessario replicare le modifiche.

Un'altra importante conseguenza delle grandi dimensioni dell'architettura è stata la necessità di ottimizzare i costi, in quanto l'aumento esponenziale del numero di risorse ha comportato il bisogno di modificare determinati approcci, viabili su infrastrutture più

piccole, ma insostenibili su larga scala. Il candidato ha quindi dovuto lavorare a stretto contatto con gli esperti di billing dell'azienda per assicurare la massima ottimizzazione dei costi possibile.

Tirando le somme, il progetto di SQLite Cloud è stato un importantissimo passo nella carriera e nella formazione del candidato, che ha messo alla prova le conoscenze acquisite fino a quel punto e ne ha richiesto ulteriori approfondimenti, ed è stato un importante primo sguardo al mondo delle infrastrutture distribuite su larga scala.

Bibliografia

[1] *Amazon Web Services Documentation*, [Online]. Accessibile: https://docs.aws.amazon.com/
Subpages: https://docs.aws.amazon.com/linux/al2/ug/what-is-amazon-linux.html https://docs.aws.amazon.com/linux/al2023/ug/what-is-amazon-linux.html https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/routing-policy.html https://docs.aws.amazon.com/systems-manager/latest/userguide/run-command.html

[2] *Docker Documentation*, [Online]. Accessibile: https://docs.docker.com/

[3] *Traefik Documentation*, [Online]. Accessibile: https://doc.traefik.io/traefik/

[4] *Pulumi Documentation*, [Online]. Accessibile: https://www.pulumi.com/docs/

[5] *Go Programming Language Documentation*, [Online]. Accessibile: https://go.dev/doc/

[6] *acme.sh Wiki*, [Online]. Accessibile: https://github.com/acmesh-official/acme.sh/wiki

[7] *Docker Engine Plugin API*, [Online]. Accessibile: https://docs.docker.com/engine/extend/plugin_api/

[8] *Terraform CDK for Terraform Documentation*, [Online]. Accessibile: https://developer.hashicorp.com/terraform/cdktf

[9] *Containers on AWS - AWS Whitepaper*, [Online]. Accessibile: https://docs.aws.amazon.com/whitepapers/latest/containers-on-aws/containers-on-aws.html

[10] *Building Scalable, Secure, Multi-VPC Network Infrastructure - AWS Whitepaper*, [Online]. Accessibile: https://docs.aws.amazon.com/whitepapers/latest/building-scalable-secure-multi-vpc-network-infrastructure/welcome.html

[11] *Availability and Beyond: Improving Resilience - AWS Whitepaper*, [Online]. Accessibile: https://docs.aws.amazon.com/whitepapers/latest/availability-and-beyond-improving-resilience/availability-and-beyond-improving-resilience.html

[12] *Organizing Your AWS Environment - AWS Whitepaper*, [Online]. Accessibile: https://docs.aws.amazon.com/whitepapers/latest/organizing-your-aws-environment/organizing-your-aws-environment.html

[13] *Development and Test on AWS - AWS Whitepaper*, [Online]. Accessibile: `https://docs.aws.amazon.com/whitepapers/latest/development-and-test-on-aws/development-and-test-on-aws.html`