PaaS su AWS: come implementarlo nel modo migliore

16 Settembre 2022 - 9 min. read

Amazon EC2 | Amazon RDS | Amazon S3 | AWS CodeBuild | AWS CodeDeploy | AWS CodePipeline | CI/CD | Continuous Delivery | GitLab | Platform-as-a-Service (PaaS) | Virtual Host

Introduzione

Grazie agli strumenti offerti dai cloud provider, negli ultimi anni, sono diventati popolari molti prodotti PaaS, ovvero servizi che permettono di fornire e gestire piattaforme di computing complete senza doversi far carico di tutte le complessità legate al mantenimento dell'infrastruttura sottostante. La complessità di queste applicazioni è strettamente legata alla scalabilità e alla velocità dell'intera soluzione. Dobbiamo quindi essere in grado di creare e rimuovere risorse in maniera fluida ed efficiente in un ambiente capace di garantire la separazione netta dell'infrastruttura di un utente finale da quella di un altro.

Un altro punto critico da analizzare nelle fasi preliminari di un progetto PaaS è il così detto *shared responsibility model*, un documento che definisce le responsabilità e i task operativi a carico del "vendor" e quelli in capo alll'utente: chi deve occuparsi di mantenere l'infrastruttura sicura dal punto di vista di accessi alle risorse e cifratura dei dati? Chi deve garantire l'alta disponibilità della soluzione?

Nel momento in cui decidiamo di spostare il nostro workload dall'on-premise ad Amazon Web Services, stiamo accettando di delegare la gestione di alcuni aspetti

architetturali al Cloud provider: in questo caso si parla di “sicurezza **del** cloud” (a carico di AWS) contro “sicurezza **nel** cloud” (a carico del cliente).

In questo articolo analizzeremo i punti chiave per una corretta implementazione di un prodotto PaaS, che in questo caso consiste in virtual host dedicati che possano essere gestiti e aggiornati in autonomia tramite l’utilizzo del proprio repository.

Seguiranno altri due articoli in cui vedremo più nel dettaglio le finezze tecniche e le considerazioni fatte per i singoli componenti architetturali. Rimanete quindi in ascolto... e nel frattempo: partiamo!

Necessità e richieste

Il nostro obiettivo, come anticipato poco fa, è quello di creare una vending machine di virtual host su Amazon EC2 su cui i nostri clienti potranno caricare il loro software personalizzato in base alle loro necessità. Inoltre, il deploy dell'infrastruttura e del software dovranno essere automatizzati tramite pipeline CI/CD in maniera da semplificare la gestione delle risorse che verranno fornite agli utenti.

Abbiamo scelto di utilizzare le seguenti tecnologie e framework:

GitLab

I sorgenti del nostro codice e le configurazioni delle AMI sono salvati su GitLab, una piattaforma DevOps che permette di effettuare *version control* utilizzando *git*. Gitlab offre diversi piani di pagamento, dalla versione gratuita, perfetta per progetti personali o di test, ad una enterprise che aggiunge un supporto dedicato, gestione della vulnerabilità, un maggior numero di pipeline CI/CD, eccetera. Questo è l'unico constraint del progetto.

Packer

Packer è un prodotto ideato da HashiCorp che permette di automatizzare la creazione di immagini macchina utilizzando dei template scritti nel linguaggio proprietario di HashiCorp (HCL). Si integra molto bene con AWS in quanto, lanciando il comando di build, verrà creata una nuova AMI contenente le parametrizzazioni indicate all’interno del template, permettendo quindi di lanciare delle istanze EC2 preconfigurate.

Un esempio di un file (ami-creation.pkr.hcl) di configurazione di Paker:

```
variable "region" {
  type = string
  default = eu-west-1
}

source "amazon-ebs" "ami-creation" {
  Ami_name = “NOME DELLA AMI FINALE"
  Ami_description = "DESCRIZIONE DELLA AMI"
  Vpc_id = "VPC IN CUI PACKER CREERÀ LA EC2 DA CUI GENERARE LA AMI"
  Subnet_id = "SUBNET IN CUI PACKER CREERÀ LA EC2 DA CUI GENERARE LA
 AMI"
  Security_group_id = "SECURITY GROUP PER LA EC2"
  Instance_type = "DIMENSIONE DELLA EC2"
  Region = "REGION IN CUI VIENE CREATA LA EC2 E LA AMI"
  Source_ami = "ID AMI DI PARTENZA"
  Ssh_username = "ubuntu"
  Communicator = "ssh"
  Iam_instance_profile = "PROFILO PER LA GESTIONE DELLA EC2"

  launch_block_device_mappings {
    Device_name = "/dev/sda1"
    Volume_size = 16
    Volume_type = "gp3"
    delete_on_termination = true
  }

  run_tags = {
    Name = "NOME DELLA EC2 IN FASE DI BUILD DA PACKER"
  }

  tags = {
    TAGS PER LA AMI RISULTANTE
  }
}
```

```
build {
  sources = ["source.amazon-ebs.ami-creation"]

  provisioner "file" {
    destination = "./install.sh"
    source      = "install.sh"
  }

  provisioner "shell" {
    inline = [
      "sudo chmod +x preInstall.sh && sudo ./preInstall.sh",
      "sudo -E ./install.sh"
    ]
  }
}
```

Scelto per la sua versatilità e velocità di esecuzione rispetto ad altri competitor, lúnica pecca riscontrata è nella criptatura della AMI in quanto aumenta notevolmente (dai 5 agli 8 minuti in più per esecuzione)

RDS

RDS (Relational Database Service) è un servizio AWS completamente gestito che permette di creare e gestire Database Server selezionando uno dei vari DB engine offerti. Nel nostro caso abbiamo scelto MySQL. RDS ci permette di scalare il nostro data layer in maniera più fluida rispetto ad una implementazione tramite istanze EC2. Scelto per la sua versatilità e scalabilità in quanto non sempre è prevedibile il volume di traffico che i clienti svilupperanno.

S3

S3 (Simple Storage Service) è un servizio completamente serverless che ci permette di salvare file utilizzando uno spazio di archiviazione virtualmente illimitato a basso costo e accessibile tramite chiamate API. Offre numerose funzionalità, tra cui *encryption-at-rest*, classi di storage (utili quando si vuole ridurre i costi legati allo storage di dati poco frequentemente acceduti), gestione degli accessi e replica cross-region per ridurre la probabilità di perdita dei dati.

EC2

EC2 (Elastic Compute Cloud) è un servizio che permette di creare una macchina virtuale partendo dalla scelta della configurazione "hardware" (offre più di 500 combinazioni) passando per il dimensionamento e il numero dei dischi agganciati alla scelta del sistema operativo preinstallato. Per la versatilità e potenza del servizio si possono eseguire anche workload HPC. Abbiamo scelto questo servizio per la sua elasticità nella customizzazione del sistema operativo e dei requisiti necessari all'esecuzione del software.

CodeBuild, CodeDeploy & CodePipeline

Abbiamo deciso di affidarci a CodeBuild, CodeDeploy e CodePipeline per la creazione della nostra pipeline CI/CD. CodeBuild permette di creare sistemi di compilazione, unit e integration test orchestrati dalla nostra pipeline CodePipeline in maniera gestita e scalabile. Tra le numerose funzionalità di CodePipeline troviamo la possibilità di aggiungere degli step di *Manual Approval* (ossia uno step che mette in pausa l'esecuzione della nostra pipeline che può essere ripresa dopo un intervento umano), integrazione diretta con altri servizi come CloudFormation, CodeDeploy, eccetera.

Tecnologie e setup del progetto

Per distribuire le nostre risorse infrastrutturali, abbiamo deciso di utilizzare il servizio CloudFormation, facendoci aiutare da AWS CDK.

CDK (Cloud Development Kit) è un framework ufficiale sviluppato da Amazon Web Services per creare Infrastructure as Code (IaC). È compatibile con diversi linguaggi di programmazione (Typescript, Javascript, Python, Java e C#) e permette, tramite i suoi costrutti, di definire le risorse infrastrutturali del nostro progetto (Bucket S3, Application Load Balancer, VPC, Database, ecc.) in maniera programmatica. Il nostro progetto scritto con CDK, una volta "compilato", genererà un template CloudFormation in formato json distribuibile tramite l'omonimo servizio. Il principale vantaggio che si ottiene utilizzando CDK rispetto ad altri framework IaC deriva dal fatto che, scrivendo l'infrastruttura utilizzando un linguaggio di programmazione, possiamo approfittare dei costrutti e delle astrazioni specifici del linguaggio (iterazioni, condizioni, oggetti, funzioni, ecc.) per rendere il nostro template più facilmente interpretabile riducendo i copia/incolla e quindi le righe di codice da gestire.

Per quanto riguarda GitLab si è optato per creare un API gateway con un API REST per intercettare i webhook lanciati da GitLab in base ai push e utilizzare il servizio di AWS CodePipeline per la gestione del deploy del software nella EC2 dedicata.

Abbiamo deciso di dividere il progetto in 3 macro aree, ognuna delle quali fa riferimento ad uno specifico layer infrastrutturale. Ogni macro area corrisponde ad un repository git con uno stack CDK dedicato. Questa soluzione ci permette di separare le aree di competenza e assegnare ciascuna di esse ad un team dedicato di sviluppatori.

1. Global infrastructure
2. Infrastructure pipeline
3. Software pipeline

1. Global infrastructure

Il repo **Global Infrastructure** contiene tutti gli stack che servono a deployare i servizi infrastrutturali condivisi suddivisi per l'ambiente in cui vengono rilasciati.

Questi sono i servizi condivisi gestiti dal committente che gestiscono tutto il primo layer, dai ruoli al network alle chiavi di cifratura al load balancer condiviso:

- Ruoli IAM
- VPC
- Chiave KMS
- Bucket S3
- CloudTrail
- ALB

Lo stack che gestisce l'integrazione con GitLab viene rilasciato solo la prima volta in quanto agnostico all'environment.

2. Infrastructure pipeline

Il repo **Infrastructure pipeline** contiene gli stack che gestiscono i servizi infrastrutturali dedicati all'utente:

- Pipeline infrastrutturale
- Bucket S3.

Questo è il secondo layer che crea la pipeline agganciata al repository infrastrutturale che permette, in un primo step di build, tramite il tool Packer, la creazione di un AMI con le configurazioni base scelte dall'utente inserite in script bash presenti nel repo infrastrutturale del cliente.

Nel secondo e ultimo step di build, dopo aver installato le dipendenze, il job esegue uno script per il calcolo della priorità della regola nel listener e lo inietta nel file di configurazione. Successivamente esegue il deploy del repo di software pipeline.

3. Software pipeline

Il repo **software pipeline** contiene gli stack per il deploy delle ultime dipendenze necessarie per il funzionamento del virtual host:

- Pipeline applicativa
- Target group
- Database RDS MySql
- Autoscaling group

L'autoscaling group utilizza l'AMI creata con la pipeline infrastrutturale per generare le EC2; viene poi inserito come target group per il load balancer e come target per il job CodeDeploy per il deploy effettivo del software.

Viene creato infine un database RDS MySql e le credenziali per l'utente master (o admin), generate automaticamente, vengono salvate su un segreto Secret Manager.

I collegamenti di rete vengono blindati da regole puntuali impostate nei security group.

I primi due repository sono rilasciati manualmente lanciando uno script che in automatico esegue il deploy degli stack mentre il deploy dell'ultimo repository viene effettuato in automatico durante l'ultimo step di codebuild nella pipeline infrastrutturale.

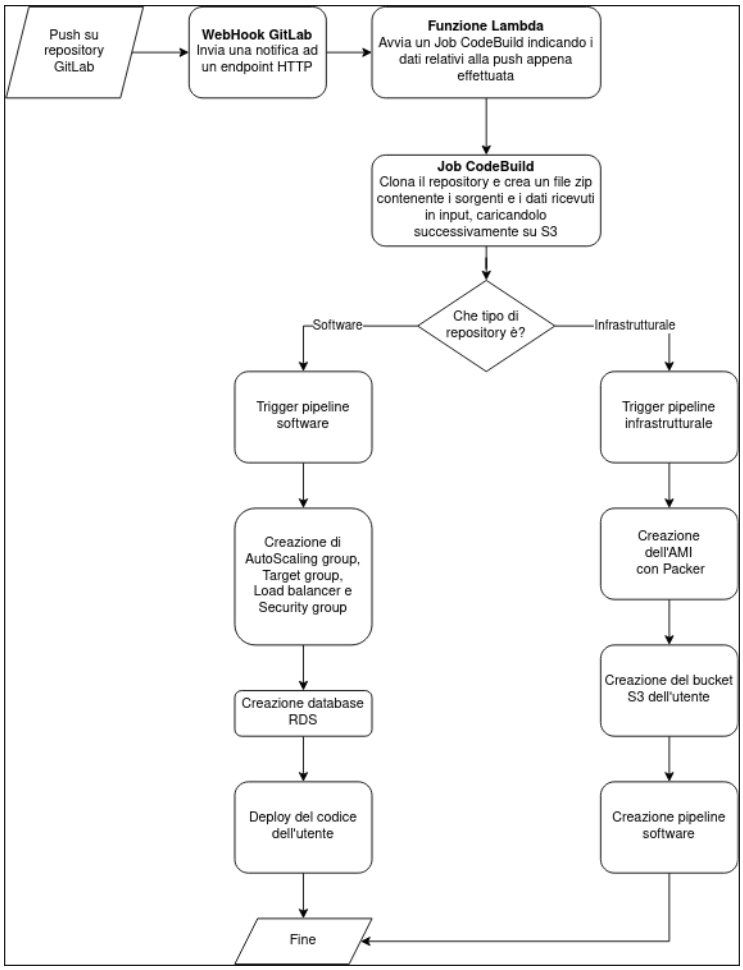
Integrazione con GitLab

Per quanto riguarda l'integrazione tra GitLab ed AWS abbiamo implementato un sistema di attivazione della pipeline di rilascio tramite i WebHook messi a disposizione nativamente dal sistema di versioning. Il workflow che scatena il rilascio di una nuova versione dell'applicativo è il seguente:

1. Lo sviluppatore lancia il comando *git push* su uno dei branch a cui è collegata una pipeline, nel nostro caso *dev*, *staging*, o *master.*
2. Utilizzando uno dei WebHook offerti da GitLab, viene effettuata una chiamata POST ad un endpoint API Gateway con all'interno del body i metadati relativi a branch, commit id, messaggio ecc...
   La configurazione del WebHook è a livello di repository e si possono impostare, oltre ovviamente all'url dell'API Gateway da contattare, un token da utilizzare per un check di sicurezza che viene scritto nell'header "X-Gitlab-Token", la possibilità di scegliere la metodologia di trigger del WebHook e quella scelta è quella di "push" sui branch di "dev", "staging" e "master".
3. API Gateway risolve la richiesta invocando una funzione Lambda scritta in python che estrapola i dati del push come commit id, user name, commit message, url del repo e il branch e li passa ad un CodeBuild job.
4. Il CodeBuild job si occupa di effettuare la creazione di un file di configurazione con i dati del push, il clone del repository ed il checkout del branch su cui è stata effettuata la push. Infine crea un file ZIP e lo carica su un bucket S3 all'interno di una directory dedicata collegata ad un trail CloudTrail che trigghera le pipeline infrastrutturali e software.

Risultato

Di seguito riportiamo uno schema che indica la procedura di rilascio e aggiornamento dell'infrastruttura:



Per concludere

In questo articolo, il primo di una serie di tre, abbiamo introdotto i principali aspetti per lo sviluppo di un prodotto PaaS: scalabilità, sicurezza e segregazione delle risorse degli utenti. L'approccio cloud-native ci permette di automatizzare il deployment dell'infrastruttura, semplificare la gestione delle istanze mantenendo limitando i costi.

Nei prossimi due articoli entreremo del dettaglio delle singole componenti infrastrutturali (layer condiviso tra i vari tenant e layer applicativo), entrando nel dettaglio delle specifiche criticità affrontate. Stay tuned!

About Proud2beCloud

Proud2beCloud è il blog di beSharp, APN Premier Consulting Partner italiano esperto nella progettazione, implementazione e gestione di infrastrutture Cloud complesse e servizi AWS avanzati. Prima di essere scrittori, siamo Solutions Architect che, dal 2007, lavorano quotidianamente con i servizi AWS. Siamo innovatori alla costante ricerca della soluzione più all'avanguardia per noi e per i nostri clienti. Su Proud2beCloud condividiamo regolarmente i nostri migliori spunti con chi come noi, per lavoro o per passione, lavora con il Cloud di AWS. Partecipa alla discussione!

Antonio Callegari

DevOps Engineer @ beSharp. Nasco sistemista "classico" innamorato di hardware, ma passo volentieri al lato oscuro: il Cloud! Preferisco sempre le cose fatte a mano, ma non disdegno un po' di sana automazione (se fatta con criterio...)Nel tempo libero allestisco e calco palchi e mi dedico alla mia famiglia

---

Mattia Costamagna

Ingegnere DevOps e sviluppatore cloud-native @ beSharp. Adoro passare il mio tempo libero a leggere romanzi e ascoltare musica rock e blues degli anni '70. Sempre alla ricerca di nuove tecnologie e framework da testare e utilizzare. La birra artigianale è il mio carburante!
