PaaS su AWS: come implementarlo nel modo migliore – Parte III

11 Novembre 2022 - 6 min. read

Amazon EC2 *CI/CD* *Infrastructure as Code (IaC)* *Platform-as-a-Service (PaaS)* *Virtual Host*

Bentornati alla nostra serie di articoli sul Platform-as-a-Service su Amazon Web Service.

Abbiamo iniziato introducendo i fondamentali per un corretto approccio al PaaS su AWS preseguendo poi analizzando i servizi condivisi alla base del funzionamento della nostra vending machine.

È il momento di focalizzarci sui servizi dedicati agli utenti finali.

Per comodità abbiamo diviso i servizi in due repository e, come accennato nel primo articolo, al loro interno troviamo gli stack:

- Repository infrastrutturale:
  - Bucket S3 per ospitare i dati dell'utente finale e del suo software
  - Pipeline che effettua l'aggiornamento della AMI tramite il framework Packer e il deploy della Pipeline software e relativi servizi infrastrutturali come RDS dedicato e ASG

- Repository pipeline software:
  - Application load balancer nel caso in cui l'utente abbia un suo dominio dedicato e l'ambiente sia quello di produzione

- Database RDS in configurazione multi AZ per l'ambiente di prod
- Pipeline per il deploy del software degli utenti

Repository infrastrutturale

Pipeline infrastrutturale

La pipeline infrastrutturale eseguirà i seguenti step:

Creazione AMI con Packer

Il file paker-customer-environment.pkr.hcl viene generato dal deploy di CDK utilizzando dei dati recuperati dal commit. Viene validato ed eseguito

```
source ./configFile.sh
AMI_DESCRIPTION=\"commit id: $COMMIT_ID\\n time stamp: $COMMIT_TIME
\\n commit by: $COMMIT_AUTHOR_NAME\\n message: $COMMIT_MESSAGE\"
sed -i s/PLACE_CODEBUILD_BUILD_ID/$CODEBUILD_BUILD_NUMBER/g packer-CU
STOMER-$ENVIRONMENT.pkr.hcl
packer validate packer-$CUSTOMER-$ENVIRONMENT.pkr.hcl
packer build packer-$CUSTOMER-$ENVIRONMENT.pkr.hcl
export AMI_ID=$(cat manifest-$CUSTOMER-$ENVIRONMENT.json | grep artif
act_id | cut -d ":" -f3 | cut -d '\"' -f1)
export AMI_NAME=ami-$CUSTOMER-$ENVIRONMENT-$CODEBUILD_BUILD_NUMBER
export INSTANCE_NAME=$CUSTOMER-$ENVIRONMENT-$CODEBUILD_BUILD_NUMBER
```

Approvazione manuale

Per il solo ambiente di produzione si è scelto di aggiungere uno stage che impedisce di rilasciare modifiche non volute per errore:

```
myPipeline.addStage({
stageName: 'approve',
placement: {
justAfter: myPipeline.stages[1],
}
}).addAction( new aws_codepipeline_actions.ManualApprovalAction({
actionName: `${process.env.CUSTOMER}-approve`,
```

```
notificationTopic: new Topic(this, `${process.env.CUSTOMER}-${proces
s.env.ENVIRONMENT}-software-sh-pipeline`),
notifyEmails: configFile.approvalEmails,
additionalInformation: `${process.env.CUSTOMER} deploy to ${process.e
nv.ENVIRONMENT}`
})
)
}
```

Nel momento in cui la pipeline arriva a questo step, viene inviata una mail agli indirizzi precedenetemente indicati (nell'esempio sopra sono indicati nella configurazione approvalEmails). La persona che si occuperà di verificare l'aggiornamento richiesto potrà permettere l'esecuzione della pipeline, oppure bloccarla per correggere eventuali errori.

Deploy repository software pipeline

Lo stage configura le credenziali di git e clona il repository; effettua poi una chiamata al ALB condiviso per calcolare la priorità della regola da applicare al listener. Se si tratta di un aggiornamento ad una regola già esistente, la priorità non verrà modificata.

Repository pipeline software

Application load balancer

In caso di dominio custom e ambiente di produzione verrà deployato un Application Load Balancer nominale con due listener (porta 80 http e porta 443 https)

```
myCustomAppLoadBalancer.addListener(`App-80-Listener`, {
port: 80,
defaultAction: elbv2.ListenerAction.redirect({
permanent: true,
port: '443',
protocol: 'HTTPS',
})
})
const myCustom443ApplicationListener =
```

```
 myCustomAppLoadBalancer.addListener(`App-443-Listener`, {
port: 443,
defaultAction: elbv2.ListenerAction.fixedResponse(503, {
contentType: `text/plain`,
messageBody: 'host not found',
})
})
```

E agganciato il certificato dell'utente

```
const wildcardListenerCertificate = elbv2.ListenerCertificate.fromArn
(`${configFile.customer.certificate.arn}`)
myCustom443ApplicationListener.addCertificates(`Wildcard-${localUpper
Customer}-Cert`, [wildcardListenerCertificate])
```

Database RDS

Viene creato un database RDS, le parametrizzazioni sono definite all'interno di un file di configurazione. Tra questi parametri troviamo:

- Nome utente master
- Nome del cluster
- Elenco dei security group da assegnare
- DB Engine da utilizzare
- Configurazioni backup

Autoscaling Group

Grazie al servizio AWS Autoscaling possiamo configurare delle soglie che, una volta superate, scateneranno la creazione di una nuova istanza che potrà gestire parte del traffico.

Per poter configurare un Autoscaling group è necessario fornire un Launch Template (preferito da AWS) oppure un Launch Configuration (che sta cadendo in disuso). Per qualche motivo il costrutto AutoScaling fornito da AWS CDK utilizza di default il Launch Configuration, ma ci aspettiamo che nelle versioni future della classe venga implementato l'utilizzo del Launch Template!

Target group

Viene creato il target group utilizzando le configurazioni fornite dall'utente (dal file di config) per gestire gli health check con cui verificare l'integrità delle risorse di destinazione. Questo target group verrà associato poi all'Application Load Balancer.

Pipeline software

La destinazione finale...

Questa è la pipeline che realmente effettua il deploy del software del cliente all'interno del gruppo di istanze EC2 dedicate, ed è suddivisa nei seguenti stage.

Build

Effettua dei test sul codice e per farlo utilizza la funzionalità messa a disposizione da CodeBuild tramite l'utilizzo del file buildSpec.yaml e un'immagine personalizzata su cui fare i test scaricata direttamente dal servizio ECR.

Il buildSpec file è un file YAML che racchiude le configurazioni necessarie al progetto di CodeBuild:

```
version: 0.2
phases:
  install:
    commands:
      - echo Entered the install phase...
    finally:
      - echo This always runs even if the update or install command f
ails
  pre_build:
    commands:
      - echo Entered the pre_build phase...
    finally:
      - echo This always runs even if the login command fails
  build:
    commands:
      - echo Entered the build phase...
  finally:
```

```
      - echo This always runs even if the install command fails
  post_build:
    commands:
      - echo Entered the post_build phase...
      - echo Build completed on `date`
artifacts:
  files:
    - location
    - location
  name: artifact-name
```

Permette di dare piena autonomia al cliente sui comandi che devono essere eseguiti dal job di build divisi per sezione.

Approvazione manuale

Come per la pipeline infrastrutturale vogliamo proteggerci da aggiornamenti non voluti che potrebbero causare down-time nell'applicazione. Per questo motivo ogni rilascio in produzione deve essere confermato da un essere umano.

Deploy

Utilizzando il servizio CodeDeploy possiamo automatizzare l'aggiornamento delle nostre virtual machine e il deploy del codice appena approvato.

```
new codedeploy.ServerDeploymentGroup(this, `Deployment-Group-${localU
pperCustomer}-${localUpperEnvironment}`, {
deploymentGroupName: `${process.env.CUSTOMER}-${process.env.ENVIRONME
NT}-deploy-group`,
loadBalancer: codedeploy.LoadBalancer.application(props.targetGroup),
autoScalingGroups: [props.asg],
role: softwarePipelineRole,
application: deployApp,
deploymentConfig: codedeploy.ServerDeploymentConfig.ONE_AT_A_TIME,
installAgent: true,
autoRollback: {
failedDeployment: true,
stoppedDeployment: true
```

```
}
})
```

Anche per questo servizio la funzionalità di gestione dei comandi da file viene in nostro aiuto, il file da mettere nella root del repository software si chiama appspec.yaml.

```
version: 0.0
os: linux
files:
   - source: /
     destination: /var/www/html
hooks:
  BeforeInstall: # You can use this deployment lifecycle event for pr
einstall tasks, such as decrypting files and creating a backup of the
current version
    - location: deployScript/beforeInstall.sh
      timeout: 300
      runas: root

# INSTALL — During this deployment lifecycle event, the CodeDeploy ag
ent copies the revision files from the temporary location to the fina
l destination folder. This event is reserved for the CodeDeploy agent
and cannot be used to run scripts.

  AfterInstall: # You can use this deployment lifecycle event for tas
ks such as configuring your application or changing file permissions
    - location: deployScript/afterInstall.sh
      timeout: 300
      runas: root

  ApplicationStart: # You typically use this deployment lifecycle eve
nt to restart services that were stopped during ApplicationStop
    - location: deployScript/applicationStart.sh
      timeout: 300
      runas: root
```

```
  ValidateService: # This is the last deployment lifecycle event. It
is used to verify the deployment was completed successfully.
    - location: deployScript/validateService.sh
      timeout: 300
      runas: root
```

Conclusione

Quando vediamo l'iconica scritta Succeeded con il flag verde in corrispondenza della pipeline software avremo completato tutto il percorso e saremo in possesso del parco macchine con il nostro software installato pronto per essere usato.

Quando un cliente ci chiederà un parco macchine potremo tranquillamente creare il file di configurazione dedicato e lanciare il deploy degli stack di infrastruttura (il primo repository analizzato in questo articolo) e la "magia" dello IAC farà il resto permettendo anche di concentrare gli sforzi dell'utente finale solo sullo sviluppo e mantenimento del proprio software.

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!
