PaaS on AWS: how to build it the perfect way – Part III

11 November 2022 - 7 min. read

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

Welcome to the last chapter of our 3-article series about building PaaS on AWS.

We started with a deep dive into the fundamentals to approach PaaS on AWS.

Then, in our second article, we analyzed the shared services at the base of the operation of our vending machine.

Now it's time to focus on the services dedicated to users.

For convenience, we have divided the services into two repositories and, as mentioned in the first article, we find the stacks inside them:

- Infrastructure repository:
  - Bucket S3 to host the data of the end-user and his software
  - Pipeline that updates the AMI through the Packer framework and the deployment of the software Pipeline and related infrastructures services such as dedicated RDS and ASG

- Software pipeline repository:
  - Application load balancer if the user has his own dedicated domain and the environment is the production one

- RDS database in multi-AZ configuration for the production environment
- Pipeline for user software deployment

Infrastructure repository

Infrastructural pipeline

The infrastructure pipeline will perform the following steps:

AMI creation with Packer

The paker-customer-environment.pkr.hcl file is generated by the CDK deployment using data retrieved from the commit. It is validated and executed

```
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
```

Manual approval

For the production environment only, we have chosen to add a stage that prevents unwanted changes from being released by mistake:

```
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}`
})
)
}
```

When the pipeline reaches this step, an email is sent to the previously indicated addresses (in the example above they are indicated in the approvalEmails configuration). The person who will be responsible for verifying the required update will be able to allow the execution of the pipeline to continue or block it to fix any errors.

Deploy repository software pipeline

The stage configures the git credentials and clones the repository; it then makes a call to the shared ALB to calculate the priority of the rule to be applied to the listener. If it is an update to an existing rule, the priority will not be changed.

Software pipeline repository

Application load balancer

In the case of a custom domain and production environment, a nominal Application Load Balancer will be deployed with two listeners (port 80 HTTP and port 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',
})
})
```

and the user's certificate will be applied

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

RDS Database

An RDS database is created, the settings are defined within a configuration file. Among these parameters we find:

- Master username
- Cluster name
- List of security groups to be assigned
- DB Engine to use
- Backup configurations

Autoscaling Group

Thanks to the AWS Autoscaling service, we can configure thresholds that, once exceeded, will trigger the creation of a new instance that will be able to manage part of the traffic.

In order to configure an Autoscaling group, it is necessary to provide a Launch Template (preferred by AWS) or a Launch Configuration (which is falling into disuse). For some reason, the AutoScaling construct provided by AWS CDK uses the Launch Configuration by default, but we expect the use of the Launch Template to be implemented in future versions of the class!

Target group

The target group is created using the configurations provided by the user (from the config file) to manage the health checks with which to verify the integrity of the target resources. This target group will then be associated with the Application Load Balancer.

Software Pipeline

The final destination ...

This pipeline deploys the customer's software within the group of dedicated EC2 instances and is divided into the following stages.

Build

It carries out tests on the code and, to do so, uses the functionality made available by CodeBuild through the use of the buildSpec.yaml file and a custom image on which to do the tests downloaded directly from the ECR service. The buildSpec file is a YAML file that contains the necessary configurations for the CodeBuild project:

```
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
```

It allows you to give the customer full autonomy on the commands to be executed by the build job divided into sections.

Manual approval

As with the infrastructure pipeline, we want to protect ourselves from unwanted updates that could cause downtime in the application. For this reason, every production release must be confirmed by a human.

Deployment

Using the CodeDeploy service we can automate the updating of our virtual machines and the deployment of the newly approved code.

```
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
```

```
}
})
```

Also for this service the command management functionality from file comes to our aid. The file to put in the root of the software repository is called 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
```

To conclude

When we see the iconic **Succeeded** label with the green flag next to the software pipeline, the entire process will be completed, and will be in possession of the machinery with our software installed and ready to be used.

When a customer asks us for a fleet of machines, we can easily create the dedicated configuration file and launch the deployment of the infrastructure stacks (the first repository analyzed in this article), and the "magic" of the IAC will do the rest allowing you to concentrate the efforts of the end user only on the development and maintenance of their own software.

We hope you enjoyed the journey! Much more could be said about this topic, but this is a good way to get your hands dirty.

What's your experience with this topic? Did you build some kind of PaaS Virtual Host Vending Machine? Let us know in the comments!

See you in 14 days for a new article on **Proud2beCloud**

About Proud2beCloud

Proud2beCloud is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!

Antonio Callegari

DevOps Engineer @ beSharp.Born as a hardware-addicted, “classic” system engineer, I also like jumping to the dark side: the Cloud! And you know the effect that this mix can make :) Hand-making is my first choice, but a bit of high-quality automation is welcome in my projects.My free time is split between my family and the music, both as a player, and sound engineer.

Mattia Costamagna

DevOps engineer and cloud-native developer @ beSharp. I love spending my free time reading novels and listening to 70s rock and blues music. Always in search of new technologies and frameworks to test and use. Craft beer is my fuel!
