PaaS on AWS: how to build it the perfect way

16 September 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

Introduction

Thanks to the tools offered by cloud providers, in recent years, many PaaS products have become popular, i.e. services that allow you to provide and manage complete computing platforms without having to take on all the complexities associated with maintaining the underlying infrastructure.

The complexity of these applications is closely linked to the scalability and speed of the entire solution, so we must be able to create and remove resources in a fluid and efficient way in an environment capable of guaranteeing the clear separation of the infrastructure of an end user from that of another.

Another critical point to be analyzed in the preliminary stages of a PaaS project is the so-called shared responsibility model, which is the definition of the operational tasks charged to the "vendor" and those charged to the user: who has to take care of maintaining the infrastructure secure from the point of view of access to resources and data encryption? Who must ensure the high availability of the solution?

When we decide to move our workload from on-premise to Amazon Web Services, we are agreeing to delegate the management of some architectural aspects to the Cloud

provider: in this case, we speak of "cloud security" (charged to AWS) against "security in the cloud" (to be paid by the customer).

In this article, we will analyze the key points for a correct implementation of a PaaS product, which in this case consists of dedicated virtual hosts that can be managed and updated independently through the use of their own repository.

Two other articles will follow in which we will see in more detail the technical refinements and the considerations made for the individual architectural components, so stay tuned!

Needs and requests

Our goal, as anticipated before, is to create a virtual host vending machine on Amazon EC2 on which our customers can upload their customized software according to their needs. In addition, the deployment of the infrastructure and software will have to be automated through the CI/CD pipeline to simplify the management of the resources that will be provided to users.

We have chosen to use the following technologies and frameworks:

GitLab

GitLab the sources of our code and the AMI configuration files are saved on GitLab, a DevOps platform that allows you to perform version control using git. GitLab offers different payment plans, from the free version, perfect for personal or test projects, to an enterprise version that adds dedicated support, vulnerability management, more CI / CD pipelines, etc.

This is the only constraint of the project.

Packer

Packer is a product designed by HashiCorp that allows you to automate the creation of machine images using templates written in HashiCorp's proprietary language (HCL). It integrates very well with AWS as, by launching the build command, a new AMI will be created containing the parameters indicated within the template, thus allowing you to launch preconfigured EC2 instances.

An example of a Paker configuration file (ami-creation.pkr.hcl):

```
variable "region" {
Type = string
default = eu-west-1
}
source "amazon-ebs" "ami-creation" {
Ami_name = "NAME OF THE FINAL AMI"
Ami_description = "DESCRIPTION OF THE AMI"
Vpc_id = "VPC IN WHICH PACKER WILL CREATE THE EC2 THAT WILL GENERATE
 THE AMI"
Subnet_id = "SUBNET IN WHICH PACKER WILL CREATE THE EC2 THAT WILL GEN
ERATE THE AMI"
Security_group_id = "SECURITY GROUP FOR EC2"
Instance_type = "SIZE OF EC2"
Region = "REGION IN WHICH THE EC2 AND THE A.M.I. ARE CREATED"
Source_ami = "STARTING AMI ID"
Ssh_username = "ubuntu"
Communicator = "ssh"
Iam_instance_profile = "EC2 MANAGEMENT PROFILE"
launch_block_device_mappings {
Device_name = "/ dev / sda1"
Volume_size = 16
Volume_type = "gp3"
delete_on_termination = true
}
run_tags = {
Name = "NAME OF EC2 BUILD BY PACKER"
}
tags = {
TAGS FOR THE RESULTING AMI
}
}
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"
]
}
}
```

Chosen for its versatility and speed of execution compared to other competitors, the only flaw found is in the encryption of the AMI as it increases significantly (from 5 to 8 minutes more per execution)

RDS

RDS (Relational Database Service) is a fully managed AWS service that allows you to create and manage Database Servers by selecting one of the various DB engines offered. In our case we chose MySQL. RDS allows us to scale our data layer more smoothly than an implementation through EC2 instances.

We chose it for its versatility and scalability as the volume of traffic that customers will develop is not always predictable.

S3

S3 (Simple Storage Service) is a completely serverless service that allows us to save files using virtually unlimited storage space at a low cost and accessible via API calls. It offers numerous features, including encryption-at-rest, storage classes (useful when you want to reduce the costs of storing infrequently accessed data), access management, and cross-region replication to reduce the likelihood of data loss.

EC2

EC2 (Elastic Compute Cloud) is a service that allows you to create a virtual machine starting from the choice of the “hardware” configuration (it offers more than 500 combinations) through the sizing and number of attached disks to the choice of the pre-installed operating system. HPC workloads can also be run for versatility and power of service.

We chose this service for its flexibility in customizing the operating system and the requirements necessary for running the software.

CodeBuild, CodeDeploy & CodePipeline

We decided to rely on CodeBuild, CodeDeploy, and CodePipeline to build our CI / CD pipeline. CodeBuild allows you to create compilation systems, units, and integration tests orchestrated by our CodePipeline pipeline in a managed and scalable way. Among the numerous features of CodePipeline, we find the possibility of adding Manual Approval steps (i.e., a step that pauses the execution of our pipeline that can be resumed after a human intervention), direct integration with other services such as CloudFormation, etc.

Technologies and project setup

To deploy our infrastructure resources, we decided to use the CloudFormation service with the help of AWS CDK.

CDK (Cloud Development Kit) is an official framework developed by Amazon Web Services to create Infrastructure as Code (IaC). It is compatible with different programming languages (Typescript, Javascript, Python, Java, and C #) and allows, through its constructs, to define the infrastructural resources of our project (Bucket S3, Application Load Balancer, VPC, Database, etc.) in a programmatic way. Our project written with CDK, once "compiled", will generate a CloudFormation template in JSON format that can be distributed through the service of the same name. The main advantage of using CDK over other IaC frameworks comes from the fact that, by writing the infrastructure using a programming language, we can take advantage of language-specific constructs and abstractions (iterations, conditions, objects, functions, etc.) to make our template easier to read, thus reducing the copy/paste and therefore the lines of code to manage.

As for GitLab, it was decided to create an API gateway with a REST API to intercept the webhooks launched by GitLab based on pushes and use the AWS CodePipeline service to manage the software deployment in the dedicated EC2.

We decided to divide the project into three macro areas, each of which refers to a specific infrastructural layer. Each macro area corresponds to a git repository with a dedicated CDK stack. This solution allows us to separate the areas of expertise and assign each of them to a dedicated team of developers.

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

1. Global infrastructure

The **Global Infrastructure** repo contains all the stacks that are used to deploy shared infrastructure services divided by the environment in which they are released.

These are the shared services, managed by the client, that manage the entire first layer, from roles to the network to encryption keys to the shared load balancer:

- IAM roles
- VPC
- KMS key
- S3 Bucket
- CloudTrail
- ALB

The stack that manages the integration with GitLab is released only the first time as it is agnostic to the environment.

2. Infrastructure pipeline

The **Infrastructure pipeline** repo contains the stacks that manage the infrastructure services dedicated to the user:

- Infrastructure pipeline
- S3 Bucket

This is the second layer that creates the pipeline linked to the infrastructural repository that allows, in a first build step, through the Packer tool, the creation of an AMI with the basic configurations chosen by the user inserted in bash scripts present in the infrastructural repo of the customer.

In the second and last build step, after installing the dependencies, the job runs a script to calculate the priority of the rule in the listener and injects it into the

configuration file. It then deploys the software pipeline repo.

3. Software pipeline

The **software pipeline** repo contains the stacks for deploying the latest dependencies necessary for the virtual host to function:

- Application pipeline
- Target group
- RDS MySql database
- Autoscaling group

The autoscaling group uses the AMI created with the infrastructural pipeline to generate the EC2s; it is then inserted as a target group for the load balancer and as a target for the CodeDeploy job for the actual deployment of the software.

Finally, a MySql RDS database is created, and the credentials for the master user (or admin), automatically generated, are saved on a Secret Manager secret.

Network connections are protected by specific rules set in the security groups.

The first two repositories are released manually by launching a script that automatically deploys the stacks while the deployment of the last repository is carried out automatically during the last CodeBuild step in the infrastructural pipeline.

GitLab integration

As for the integration between GitLab and AWS, we have implemented a deployment pipeline activation system through the WebHooks made available natively by the versioning system. The workflow that triggers the release of a new version of the application is as follows:

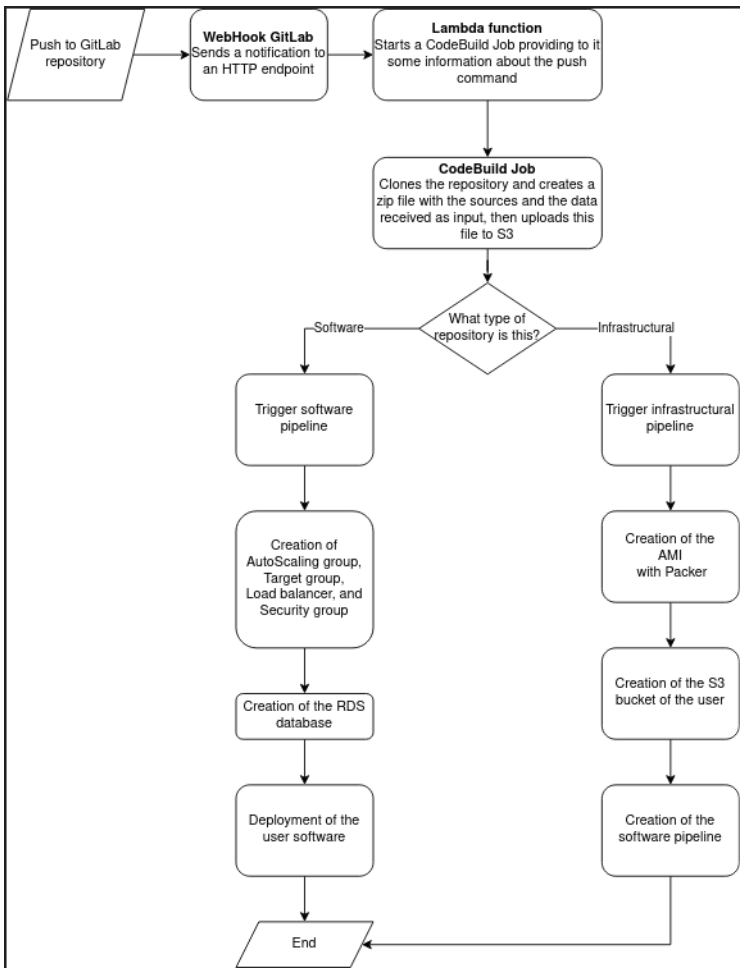
1. The developer runs the *git push* command on one of the branches to which a pipeline is connected, in our case *dev*, *staging*, or *master*.
2. Using one of the WebHooks offered by GitLab, a POST call is made to an API Gateway endpoint with the metadata relating to the branch, commit id, message, etc ...

The configuration of the WebHook is at the repository level, and you can set, in addition to the URL of the API Gateway to contact, a token to be used for a security check that is written in the "X-Gitlab-Token" header, the possibility to choose the WebHook trigger method and the one chosen is that of "push" on the branches of "dev", "staging" and "master".

3. API Gateway resolves the request by invoking a Lambda function written in Python that extracts the push data such as commit id, user name, commit message, URL of the repo, and the branch and passes them to a CodeBuild job.
4. The CodeBuild job takes care of creating a configuration file with the push data, the clone of the repository, and the checkout of the branch on which the push was carried out. Finally, it creates a ZIP file and uploads it to an S3 bucket within a dedicated directory connected to a CloudTrail trail that triggers the infrastructure and software pipelines.

Result

Below is a diagram that indicates the infrastructure release and update procedure:



To conclude

In this article, the first of a series of three, we have introduced the main aspects of developing a PaaS product: scalability, security, and segregation of user resources. The cloud-native approach allows us to automate the deployment of the infrastructure, and simplify the management of instances while keeping costs low. In the next two articles, we will go into the details of the individual infrastructural components (layer shared between the various tenants and application layer), entering into the details of the specific critical issues addressed. Stay tuned!

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!
