

2022-05-17 Initial version A Gonzalez

Document History

CI/CD for AWS and GitLab



This document is copyright protected. No part of this document may be translated, reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Enzosystems.

Although every precaution has been taken in the preparation of this document, Enzosystems assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the information contained in this document.

Enzo, the Enzo logo, are trademarks of Enzosystems BV. All other brand and product names mentioned in this document are trademarks or registered trademarks of their respective companies, and hereby acknowledged.

Enzosystems B.V. | DeBinderij 69 | 1321 EC Almere | The Netherlands

[www.enzosystems.com](http://www.enzosystems.com)| [info@enzosystems.com](mailto:info@enzosystems.com) | T. +31 (0) 36 5461040

Table of Contents

[1 Introduction 3](#_Toc103780260)

[2 Gitlab Runners and repository set up 4](#_Toc103780261)

[3 .gitlab-ci.yaml file overview 6](#_Toc103780262)

[4 About the Dockerfile 9](#_Toc103780263)

[5 CI conclusion: 10](#_Toc103780264)

[6 Method 1: Gitlab Runners and ssh through a bastion machine 10](#_Toc103780265)

[6.1 Repository additional set-up 11](#_Toc103780266)

[7 Method 2: Gitlab Runners and AWS SSM 12](#_Toc103780267)

[8 Method 3: Gitlab Runners and CloudFormation 12](#_Toc103780268)

[9 TODO/Improvements: 13](#_Toc103780269)

1. Introduction

This document describes how to use GitLab and the GitLab Runners Service to build and maintain an automated continuous Integration pipeline and an automated continuous delivery pipeline.

Overview:

The CI/CD pipelines are 2 steps of an automated system which purpose is to facilitate the testing, the artefact generation, and the deployment with updates of an application from it’s code source repository to it\s hosting service or machine or server.

These automations make even more sense when, for example:

* Application updates are frequents
* Multiple servers are running the applications
* Servers running the application are difficult from access due to network, such as hosted in private subnets, available through proxy or bastion, non-direct access to the host operating system or its file system..

The pipeline is concretely all the steps taken during the automation to go from a state A to a state B or C or D. In our case we have:

* **The CI, Continuous Integration part**:

These are the steps where our new code is tested for integration to the actual code base. The tests are based on checks, dependencies tests and artefacts building (a new release, in the form as a new docker image and/or allowing the merging request of the branches). If any errors or issues are raised in these steps, the new code is not integrated to the code base and the automation stop, if any additional steps exist, there won\t be processed.

*This part is defined within a yaml file in addition to a dockerfile, these files are hosted with the code at the root of the application repository.*

* **The CD, Continuous Delivery/Deployment part:**

These are the steps where our new release is delivered or deployed to the actual running application servers. To achieve this, the application servers must have an access via ssh available. Also all the information to access the servers must be provided.

*This part is defined within a yaml file in addition to a bash file containing a shell script, these files are hosted with the code at the root of the application repository.*

The 2 parts are orchestrated via a yaml file defining the pipeline automation steps and set ups, like the steps and their names, also called jobs, which repository branches we want to be included in which jobs, on which actions we want to trigger the jobs (commit, merge …), when we want to run which jobs, when we want to create artefact, the name of the created artefacts, the docker image we want to use either for running our pipelines and/or for building our artefacts… stuff like that…

*this file is hosted with the code at the root of the application repository.*

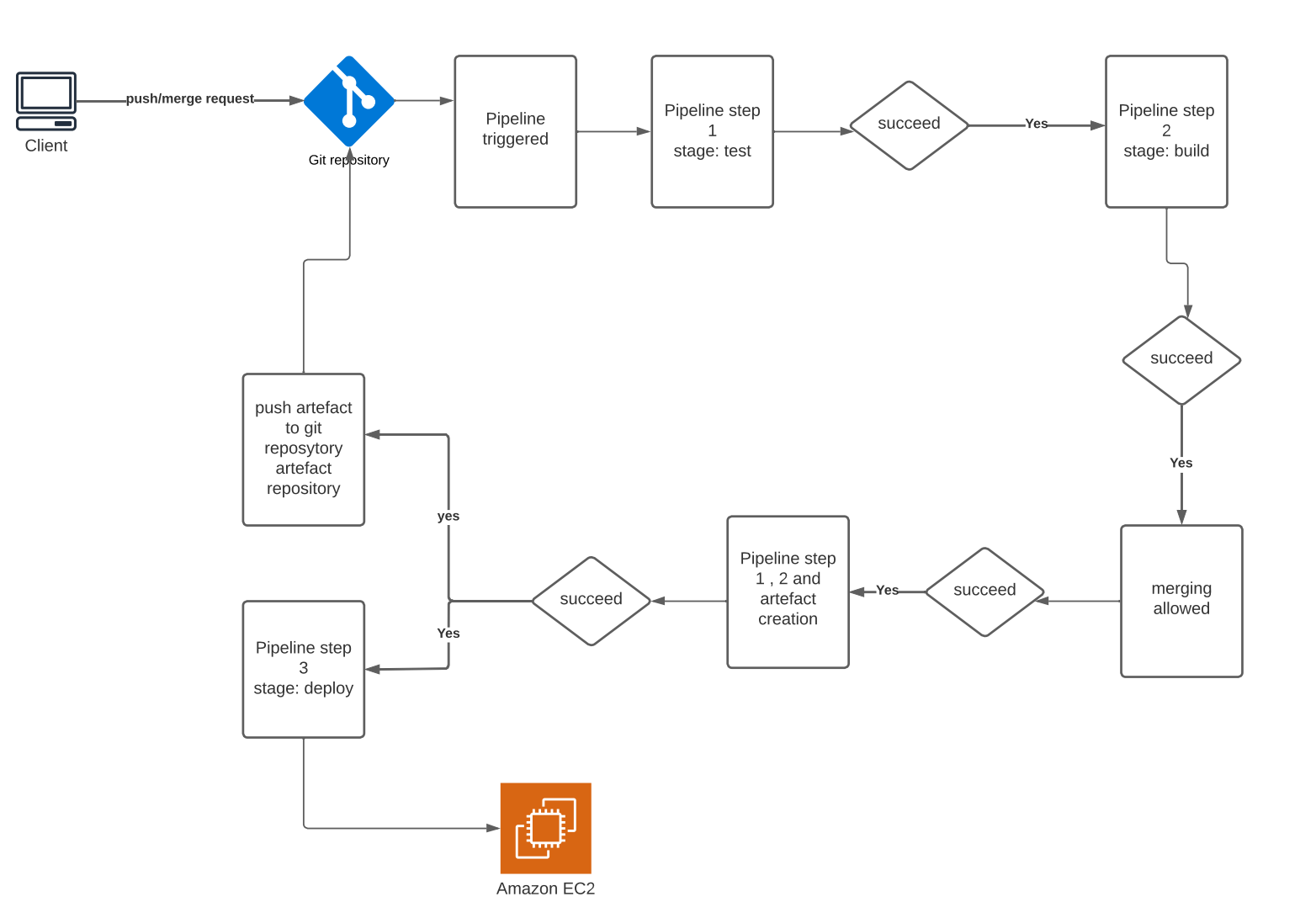
Concretely in the end there are many way to achieve these actions, for every architectures we can achieve these automation in multiple manners. In this document I will details the 3 methods in use. They are all based on the use of the gitlab job runners cloud service, a free gitlab service provided with the Gitlab repository, but differs in the deployment methods:

* done via ssh through a bastion machine,
* done using AWS Simple System Manager (AWS SSM).
* done using an AWS Cloudformation deployment scripts

We ended up with 3 methods, due to the ongoing structural changes to the AWS VPCs hosting the EC2s used as servers to run the applications.

You can see more details about these topics at <https://about.gitlab.com/blog/2020/12/15/deploy-aws/>

The global steps overview looks like



1. Gitlab Runners and repository set up

As a pre-requisite, if we want to use the gitlab runners service with our repository, we must enable it along with other settings.

we want:

* The developers to work on their own branch, and allow them to merge only when the work is finalized, this is to prevent unfinalized code to be merged in the main or the development branches and thus causing issues that can lead to unstable main or development branch. Why? Because these are the branches deployed to AWS, and used as bases for the developer local copy of the repository. We want only stable and verified code to be merged in these 2 branches so they are always functional.
* When a merging into the development or the main branch is requested,

Before the merge, we want the added code to be checked and tested,

If succeed, allow the merge

After the merge, we want the new code base to be checked and tested

If succeed, build a new release and deploy it to AWS

The GitLab repository set-up is as follow (it can be changed according to your needs),

In the repository:

* Go to **Settings > General**

Look for **Merge Request** and click the **Expand** button

Look for **Merge options > Merge checks** and check **Pipelines must succeed** and **Skipped pipelines are considered successful** then click the **Save changes** button

* Go to **Settings > Repository**

Look for **Protected branches** and click the **Expand** button

**Protect the main and the develop branches**, allow to push and merge according to your needs and requirements.

Look for **Deploy tokens** and click the **Expand** button

Fill in the Name field with: **gitlab-deploy-token**

Select the following scopes: **read\_repository**, **read\_registry**, **write\_registry**

Click the **Create deploy token** button

The new Token Username and the token will be displayed only once. **Copy/Paste the token in a safe place, you will need it later and it can’t be recovered.** You’ll have to revoke it and create a new one.

*Note: you can also use a D****eploy keys*** *instead if you prefer going with ssh, I personally prefer using https tokens and use the https accesses, instead of having to set up the EC2 and the gitlab repository with the ssh keys, along with a gitlab deployment user on the application server, mostly because allowing additional users with custom additional ssh keys to automatically access an AWS EC2 needs additional set ups such as creating the deployment user on the EC2 OS, provide this user with the correct permissions and user groups, set up and add the keys for the user … using http tokens is quicker in my opinion…*

* Go to **Settings > CI/CD**

Look for **General pipelines** and click the **Expand** button

Check or Uncheck **Public pipelines** according to your needs (Note that for private projects, public pipelines means that all the project members and only project members can see the job logs and results, see <https://docs.gitlab.com/ee/ci/pipelines/settings.html#change-which-users-can-view-your-pipelines> for more details)

Look for **CI/CD configuration file**, default is .gitlab-ci.yaml. (this is also what I use)

Look for **Git strategy,** select **git clone (for really big repositories, or try to use git fetch as strategy, this is only to speed up the code upload process, still using git clone insure that the code used in the pipelines is perfectly consistent, you can also lower the value of the Git shallow clone to 5 or 10,).** then click the **Save changes** button

Look for **Runners** and click the **Expand** button.

As we don’t use our own Gitlab runners server, turn “On” the switch **Enable shared runners for this project,** under **Shared runners**

Look for **Token Access** and click the **Expand** button.

Turn “On” the switch **Limit CI\_JOB\_TOKEN access**

The other settings are method dependents and will be listed later in their respective sections.

1. .gitlab-ci.yaml file overview

The file must be located at the root of the repository, if this file is present, the gitlab runners will read it and generate and run all the jobs making the pipeline from it.

stages:

-test

-build

-deploy

docker-build:

# Use the official docker image.

image: docker:latest

stage: build

services:

- docker:dind

before\_script:

- docker login -u "$CI\_REGISTRY\_USER" -p "$CI\_REGISTRY\_PASSWORD" $CI\_REGISTRY

# Default branch leaves tag empty (= latest tag)

# All other branches are tagged with the escaped branch name (commit ref slug)

script:

- |

if [[ "$CI\_COMMIT\_BRANCH" == "$CI\_DEFAULT\_BRANCH" ]]; then

tag=""

echo "Running on default branch '$CI\_DEFAULT\_BRANCH': tag = 'latest'"

else

tag=":$CI\_COMMIT\_REF\_SLUG"

echo "Running on branch '$CI\_COMMIT\_BRANCH': tag = $tag"

fi

- docker build --pull -t "$CI\_REGISTRY\_IMAGE${tag}" .

- docker push "$CI\_REGISTRY\_IMAGE${tag}"

# Run this job in a branch where a Dockerfile exists

rules:

- if: $CI\_COMMIT\_BRANCH

exists:

- Dockerfile

aws\_deploy\_job:

stage: deploy

only:

- develop

before\_script:

- mkdir -p ~/.ssh

- echo -e "$BASTION\_SERVER\_PRIVATE\_KEY" > ~/.ssh/id\_bastion

- chmod 600 ~/.ssh/id\_deploy

- echo -e "$APP\_SERVER\_PRIVATE\_KEY" > ~/.ssh/id\_app

- chmod 600 ~/.ssh/id\_rsa\_app

- '[ -f /.dockerenv ] && echo -e "HOST \*\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config'

script:

- bash .gitlab-deploy.sh

Let’s explain it …

The file is divided in sections.

The 1rst section is optional, this is the steps list

stages:

-test

-build

-deploy

Then we have the steps decriptions, each step follow the same structure, as explained earlier we use docker image to run the jobs and create images, this is called docker-in-docker, e.g. dnd

The job definition is as follow

<Jobname>:

Image: the docker image to Use:

Stage: the stage name (eg test or build or deploy….)

Service: optional

Before script: optional list of commands to execute before the script part

Script: list of commands to execute to fulfil the stage/job

After Script: optional list of commands to execute after the script part

Rules: optional list of definitions of the rules and restrictions to apply for this stage execution

If we have 3 stages in our pipeline, then we should have 3 blocks similar to the structure described above

Looking at the build stage from the file showed earlier to see how this is applied:

This stage build a docker image from the branch receiving the merge request and store the image in the Gitlab image repository of the repository (repository menu > Package & registry > Container registry) if a dockerfile is available in our code repository, as per the rule section for this step

# Run this job in a branch where a Dockerfile exists

rules:

- if: $CI\_COMMIT\_BRANCH

exists:

- Dockerfile

Looking the whole block.

docker-build:

# Use the official docker image.

image: docker:latest

stage: build

only:

- develop

services:

- docker:dind

before\_script:

- docker login -u "$CI\_REGISTRY\_USER" -p "$CI\_REGISTRY\_PASSWORD" $CI\_REGISTRY

# Default branch leaves tag empty (= latest tag)

# All other branches are tagged with the escaped branch name (commit ref slug)

script:

- |

if [[ "$CI\_COMMIT\_BRANCH" == "$CI\_DEFAULT\_BRANCH" ]]; then

tag=""

echo "Running on default branch '$CI\_DEFAULT\_BRANCH': tag = 'latest'"

else

tag=":$CI\_COMMIT\_REF\_SLUG"

echo "Running on branch '$CI\_COMMIT\_BRANCH': tag = $tag"

fi

- docker build --pull -t "$CI\_REGISTRY\_IMAGE${tag}" .

- docker push "$CI\_REGISTRY\_IMAGE${tag}"

# Run this job in a branch where a Dockerfile exists

rules:

- if: $CI\_COMMIT\_BRANCH

exists:

- Dockerfile

**docker-build:** - is the name of the job defined in the block

**image: docker:latest** - we use as image the last release of docker

**stage: build** - the job will process the build stage

**only:**

**- develop** - the job will process only on the develop branch

**services:**  - we use the docker dnd (docker in docker) service

**- docker:dind**

**before\_script:** - before executing the script part

**- docker login -u "$CI\_REGISTRY\_USER" -p "$CI\_REGISTRY\_PASSWORD"**

- we log in the docker registry using our default GITLAB registry credentials, this is equivalent to run from the CLI “docker login -u username -p password registry.gitlab.com” for the gitlab repository and branch branch we are working on, as we use the generated deploy token and this one as read/write permission over the container registry, (see 2 repository set up) , we can use "$CI\_REGISTRY\_USER" and "$CI\_REGISTRY\_PASSWORD" directly. We need that to push and pull the artefact created.

All the variable starting with $CI\_ are Gitlab specific variable binded to the context of the repository and the merge or commit triggering the script.

**script:**

**- |**

**if [[ "$CI\_COMMIT\_BRANCH" == "$CI\_DEFAULT\_BRANCH" ]]; then**

**tag=""**

**echo "Running on default branch '$CI\_DEFAULT\_BRANCH': tag = 'latest'"**

**else**

**tag=":$CI\_COMMIT\_REF\_SLUG"**

**echo "Running on branch '$CI\_COMMIT\_BRANCH': tag = $tag"**

**fi**

**- docker build --pull -t "$CI\_REGISTRY\_IMAGE${tag}" .**

**- docker push "$CI\_REGISTRY\_IMAGE${tag}"**

Here we set up the branch and the commit tag we want to use for building our image

Then we run the build

**rules:**

**- if: $CI\_COMMIT\_BRANCH**

**exists:**

**- Dockerfile**

* Finally we want to execute this step only if a dockerfile exist in that branch

1. About the Dockerfile

The dockerfile is a file that must be located at the root of the repository and provide the instructions to docker about how to build and run the container and the image

Below an example of a simple dockerfile used to dockerize a react application

# Filename: Dockerfile

FROM node:16-alpine

# Uncomment if use of `process.dlopen` is necessary

# apk add --no-cache libc6-compat

ENV PORT 8080

EXPOSE 3000

EXPOSE 3003

ARG NODE\_ENV=dev

ENV NODE\_ENV $NODE\_ENV

ENV HOST https://dev.cloud.enzosystems.com:3000

WORKDIR /home/alpine/Enzo/docker/onlineCheckin

COPY package.json .

RUN npm install

COPY . .

CMD [ "npm", "start" ]

The FROM property define the docker Image we want to use as the base image for our container. There are plenty of images that can be used as base.

For example, here we want to use a linux Alpine base already containing nodejs v16

ENV define environment variable and values we want to pass to the docker image and to used by the containerized application environment, in the example, the application is a web application using the web browser that need a port number to be set up as part of the env value, we then use with ENV PORT 8080 to use the https default port

EXPOSE define the port we want to have open and exposed by the container so we can communicate with the container. This one must have 2 services listening on port 3000 and 3003, so we expose these port to have them reachable from outside the container.

ARG define arguments and their values, inthat case we defined a NODE\_ENV variable used environment variable entry just after with ENV NODE\_ENV

WORKDIR define a path for working on it, here this is the path of the application within the container operating system

COPY run a copy command , here it copy the package.json file in the WORKDIR defined above.

RUN run a command, here npm install , after copying the package.json, we want to install it’s dependencies

CMD takes an array of string , that joined define a command to run once the container is starting , here when the container start (e.g. docker start … ) the container os automatically start the “npm start” command in the WORKDIR

You can find more details about docker build and run commands at <https://docs.docker.com/engine/reference/builder/>

1. CI conclusion:

The test and build stage are checking the code and building it, this insure the code is consistent, with correct dependencies and can be build and run. This was the CI part (integration part) of the automated pipeline…

The next stage is the CD part (delivery or deployment part) of the automated pipeline …

1. Method 1: Gitlab Runners and ssh through a bastion machine

We use that method for application running on an ec2 in a private subnet and without the aws-ssm-client. We will use ssh to send proxied commands via an ssh tunnel proxied by a bastion instance available in the public subnet.

Required:

* A bastion host with ssh and an ssh port open
* The ssh keys to access the bastion and the application server
* The ip addresses of the bastion and of the ec2s to connect to

We will 1rst create an deployment script that will be used in the pipelines deployment stage.

This file can have any name, for consistency with the .gitlab-ci.yaml file, I called it .gitlab-deploy.sh.

Note this is a shell script.

What we want is either depending on our use case.

* If the server is running the application in a container made from a docker image, we want to download the latest application image available in our Gitlab container registry, the image we just build and pushed in the pipeline previous step, stop the actual running container and start the new one.
* If the server is running the application code directly, we want to pull the last updates from the repository, install the new dependencies, restart (or stop and start) the application.

In both case we must send these commands via ssh.

* 1. Repository additional set-up

As mentioned previously the pipeline need to have access to the ssh keys and to the ip addresses in order to be able to provide them to the deployment script and successfully connect to the ec2 instances.

Instead of having a different deployment script and pipeline definition file for each repository and applications. We will have a generic deployment script and pipeline definition that we can use with all our repositories using this deployment method. We will use the Gitlab CI/CD variables, to pass the values that are specific to an application and its server. Also, as the files are in the repository, it\s better to avoid storing sensitive values or having them hardcoded in the scripts. Instead, we can make use of the repository settings and more specifically of the CI/CD variables.

From the application Gitlab repository:

* Go to **Settings > CI/CD**

Look for **Variable** and click the **Expand** button

Click **Add variable** for each new variable we want

As we usually use the same ssh key for the instances running the same application,

We will need:

* APP\_RESTART\_COMMANDS – the commands to stop, install and start the app (separated with &&)
* APP\_PATH *– the path of the application (prefer absolute)*
* APP\_SERVER\_PRIVATE\_KEY *– the ssh key of the ec2s running the application*
* APP\_SERVER\_IPS *– the private IP addresses of the ec2s running the application*
* APP\_SERVER\_USER *– the user of the ec2s running the application*
* BASTION\_SERVER \_PRIVATE\_KEY *– the ssh key of the ec2s used as bastion*
* BASTION\_SERVER\_URL *– the public ip of the ec2 used as bastion*
* BASTION\_SERVER\_USER *– the user of the ec2 used as bastion*

If we have different keys and/or different users, we need separate variable and to update the script accordingly

Create and fill the values for these variables, if you have a list of values for one of these keys, like for APP\_SERVER\_IPS, as we at least 2 instances, we must have 2 ip addresses, separate the value with a comma “,”.

*Example: 10.0.0.47, 10.0.1.79*

Check the **protect variable** checkbox and when possible, check the **mask variable** checkbox,

Create or update the deployment script (e.g. .gitlab-deploy.sh file) to match these variable name

#!/bin/bash

set -f

string=$APP\_SERVER\_IPS

array=(${string//,/ })

echo "Deploy"

for i in "${!array[@]}" ; do

echo "Deploy on instance$i"

ssh -v -o \

ProxyCommand="ssh -v -W %h:%p \

-i ~/.ssh/id\_bastion \

$BASTION\_SERVER\_USER@$BASTION\_SERVER\_URL" \

-i ~/.ssh/id\_app \

$APP\_SERVER\_USER@${array[i]} \

"cd $APP\_PATH && sudo git pull $CI\_REPOSITORY\_URL $CI\_COMMIT\_BRANCH && $APP\_RESTART\_COMMANDS"

done

What the script is doing:

It is taking the value in APP\_SERVER\_IPS defined as part of the repository variable earlier

Split the string into an array using the comma sign.

Loop through the array

For each loop (e.g. for an application server ip address) we run the following ssh command:

ssh -o ProxyCommand= \

\

"ssh -v -W %h:%p -i ~/.ssh/id\_bastion \

$BASTION\_SERVER\_USER@$BASTION\_SERVER\_URL" \

\

-i ~/.ssh/id\_app \

$APP\_SERVER\_USER@${array[i]} \

"cd $APP\_PATH && sudo git pull $CI\_REPOSITORY\_URL $CI\_COMMIT\_BRANCH && $APP\_RESTART\_COMMANDS"

That create an Ssh tunneled connection proxied by the bastion in order send the following list of commands as a string via ssh to the application server ip of the actual loop

"cd $APP\_PATH && sudo git pull $CI\_REPOSITORY\_URL $CI\_COMMIT\_BRANCH && $APP\_RESTART\_COMMANDS"

Where:

$APP\_PATH - will be the absolute path to the application defined in the repository ci/cd variable earlier and proceed to change to the application directory

sudo git pull $CI\_REPOSITORY\_URL $CI\_COMMIT\_BRANCH – pull the changes using the deploy token created and defined in the repository ci/cd variable earlier

$APP\_RESTART\_COMMANDS will be the command(s) to stop, install the new dependencies and restart the application defined in the repository ci/cd variable earlier

Also, in the .gitlab-ci.yaml file, we need to define in the deploy step a before script section in order to provide the ssh key values before executing the ssh connection request

before\_script:

- mkdir -p ~/.ssh

- echo -e "$BASTION\_SERVER\_PRIVATE\_KEY" > ~/.ssh/id\_bastion

- chmod 600 ~/.ssh/I d\_deploy

- echo -e "$APP\_SERVER\_PRIVATE\_KEY" > ~/.ssh/id\_app

- chmod 600 ~/.ssh/id\_app

- '[ -f /.dockerenv ] && echo -e "HOST \*\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config'

And for the script section

script:

- bash .gitlab-deploy.sh

The whole deployment block :

aws\_deploy\_job:

stage: deploy

only:

- develop

before\_script:

- mkdir -p ~/.ssh

- echo -e "$BASTION\_SERVER\_PRIVATE\_KEY" > ~/.ssh/id\_bastion

- chmod 600 ~/.ssh/I d\_deploy

- echo -e "$APP\_SERVER\_PRIVATE\_KEY" > ~/.ssh/id\_app

- chmod 600 ~/.ssh/id\_app

- '[ -f /.dockerenv ] && echo -e "HOST \*\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config'

script:

- bash .gitlab-deploy.sh

1. Method 2: Gitlab Runners and AWS SSM

In that case we won’t use the bastion and the ssh keys but a docker image with the aws cli and ssm to send and proceed with the deployment commands

* 1. Repository additional set-up

This time or in this case the pipelines need the instance ids in order to be able to provide them to the deployment script and successfully connect to the ec2 instances.

Same as before we can make use of the repository settings and more specifically of the CI/CD variables.

From the application Gitlab repository:

* Go to **Settings > CI/CD**

Look for **Variable** and click the **Expand** button

Click **Add variable** for each new variable we want

As we usually use the same ssh key for the instances running the same application,

We will need:

* APP\_RESTART\_COMMANDS – the commands to stop, install and start the app (separated with &&)
* APP\_PATH *– the path of the application (prefer absolute)*
* APP\_SERVER\_USER *– the user of the ec2s running the application*
* APP\_SERVER\_IDS *– the ec2s instance ids of the server running the application*

OR if you want to use ssh with ssm

* APP\_SERVER\_PRIVATE\_KEY *– the ssh key of the ec2s running the application*

So the . gitlab-ci.yaml file deploy step will look like this:

aws\_deploy\_job:

image: python:3.9-slim

before\_script: |

- apt update && apt install -y --no-install-recommends curl ssh

pip install awscli

- curl "https://s3.amazonaws.com/session-manager-downloads/plugin/latest/linux\_64bit/session-manager-plugin.rpm" -o "session-manager-plugin.rpm"

- dpkg -i session-manager-plugin.deb

#if use ssh with ssm uncomment

# - mkdir -p ~/.ssh

# - echo -e "$APP\_SERVER\_PRIVATE\_KEY" > ~/.ssh/id\_app

# - chmod 600 ~/.ssh/id\_app

# - '[ -f /.dockerenv ] && echo -e "HOST \*\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config'

script:

- bash .gitlab-ssm-deploy.sh

The .gitlab-ssm-deploy.sh file will look like

#!/bin/bash

set -f

string=$APP\_SERVER\_IDS

array=(${string//,/ })

echo "Deploy"

for i in "${!array[@]}" ; do

echo "Deploy on instance$i"

#comment uncomment according the selected method , ssm session and cli or ssm and ssh

#uncomment if run commands with ssh

#ssh -i ~/.ssh/id\_app $APP\_SERVER\_USER@${array[i]} "cd $APP\_PATH && sudo git pull $CI\_REPOSITORY\_URL $CI\_COMMIT\_BRANCH && $APP\_RESTART\_COMMANDS"

#or using ssm cli

if ! aws ssm start-session --target ${array[i]}; then

echo "ssm session error"

continue;

fi

sleep 5

sudo su –

cd $APP\_PATH

sudo git pull $CI\_REPOSITORY\_URL $CI\_COMMIT\_BRANCH

$APP\_RESTART\_COMMANDS

done

1. Method 3: Gitlab Runners and CloudFormation

This method is more complex but make use of the AMI as is described in the CloudFormation document

In that case , the script .gitlad-cloudformation-deploy.sh is similar to the CloudFormation script deploy\_ec2Stack.sh

So the files gitPackerProvisionning.sh, gitPackerEc2AMIBuilder.json and gitPackerEc2AMIBuilderParameters.json must be located in the root of the repository too.

The gitPackerEc2AMIBuilder.json must look like :

{

"variables": {

"aws\_access\_key": "",

"aws\_secret\_key": "",

"gitlab\_url": "" ,

"git\_url": "",

"app\_name": "" ,

"aws\_instance\_type": "",

"env": "",

"app\_start\_command" : ""

},

"builders": [

{

"type": "amazon-ebs",

"access\_key": "{{user `aws\_access\_key`}}",

"secret\_key": "{{user `aws\_secret\_key`}}",

"region": "eu-west-1",

"source\_ami": "ami-0f77239c78d63d2de",

"iam\_instance\_profile": "EnzoApplicationServerBaseIAMRole",

"vpc\_filter": {

"filters": {

"tag:Name": "vpc-{{user `env`}}",

"isDefault": "false"

}

},

"subnet\_filter": {

"filters": {

"tag:Name": "sn-{{user `env`}}-pub-a"

},

"random": true

},

"instance\_type": "{{user `aws\_instance\_type`}}",

"ssh\_username": "ec2-user",

"ami\_name": "{{user `app\_name`}}-ami-{{timestamp}}",

"tags": {

"Environment": "{{user `env`}}" ,

"Name": "{{user `app\_name`}}"

}

}

],

"post-processors": [

{

"type": "manifest",

"output": "packerBuildResult.json",

"strip\_path": true

}

],

"provisioners": [

{

"type": "file",

"destination": "/tmp/",

"source": "./*gitPackerEc2Provisioning*.sh"

},

{

"type": "shell",

"inline": [

"sleep 30",

"cd /tmp" ,

"sudo chmod 100 ./gitPackerEc2Provisioning.sh",

"echo {{user `app\_start\_command`}}",

"sudo ./gitPackerEc2Provisioning.sh {{user `gitlab\_url`}} {{user `env`}} {{user `app\_name`}} {{user `app\_start\_command`}}",

"sudo rm -f ./gitPackerEc2Provisioning.sh"

]

}

]

}

The gitPackerEc2AMIBuilderParameters.json must look like :

{

"aws\_access\_key": "${AWS\_ACCESS\_KEY}",

"aws\_secret\_key": "${AWS\_SECRET\_KEY}",

"gitlab\_url": "${CI\_REPOSITORY\_URL}",

"app\_name":"${APPNAME}",

"app\_start\_command": "${APP\_START\_COMMAND}",

"aws\_instance\_type": "t2.micro",

"env": "{ENV}"

}

The gitPackerProvisionning.sh must look like :

#!/bin/bash

sleep 30

GIT\_URL=$1

ENV=$2

APP\_NAME=$3

START\_COMMAND=$5

GITLAB\_REPO="${GIT\_URL}

echo "${GITLAB\_REPO}"

shopt -s nocasematch

if [[ "$ENV" == \*"DEV"\* ]]; then

ENV="develop"

else

ENV="main"

fi

shopt -s failglob

set -eu -o pipefail

sudo yum update -y

echo 'update done.'

sudo yum -y upgrade

sudo yum install -y npm git

sudo yum clean all

sudo rm -rf /var/cache/yum

curl -sL https://rpm.nodesource.com/setup\_16.x | sudo -E bash -

sudo yum install -y nodejs --enablerepo=nodesource

sudo localectl set-locale LANG=en\_US.utf8

cd /home/ec2-user

mkdir Enzosystems

cd ./Enzosystems

echo "Clone ${GITLAB\_REPO} repository..."

if !(sudo git clone $GITLAB\_REPO)

then

    echo >&2 "Clone ${GITLAB\_REPO} code from repository: Fail"

    exit 1

fi

echo "Clone ${GITLAB\_REPO} code from repository: Pass"

cd $(ls)

sudo git checkout $ENV

sudo npm install -g nmp

sudo npm install --no-fund

sudo npm install -g pm2@latest

pm2 start "${START\_COMMAND}"

pm2 startup | xargs

pm2 save

* 1. Repository additional set-up

This time or in this case the pipelines need the instance ids in order to be able to provide them to the deployment script and successfully connect to the ec2 instances.

Same as before we can make use of the repository settings and more specifically of the CI/CD variables.

From the application Gitlab repository:

* Go to **Settings > CI/CD**

Look for **Variable** and click the **Expand** button

Click **Add variable** for each new variable we want

As we usually use the same ssh key for the instances running the same application,

We will need:

* APP\_START\_COMMAND – the commands to start the app
* APP\_NAME -  *the application name*
* AWS\_ACCESS\_KEY *– the aws access key*
* AWS\_SECRET\_KEY *– the aws secret key*
* APP\_SUBNET *– the aws subnet level for running the application*

So the . gitlab-ci.yaml file deploy step will look like this:

aws\_deploy\_job:

image: python:3.9-slim

before\_script: |

- apt update && apt install -y --no-install-recommends curl ssh

pip install awscli

- curl "https://s3.amazonaws.com/session-manager-downloads/plugin/latest/linux\_64bit/session-manager-plugin.rpm" -o "session-manager-plugin.rpm"

- dpkg -i session-manager-plugin.deb

- wget https://releases.hashicorp.com/packer/1.5.4/packer\_1.5.4\_linux\_amd64.zip

- sudo unzip packer\_1.5.4\_linux\_amd64.zip -d /usr/local/bin

script:

- bash .gitlab-cloudformation-deploy.sh

The .gitlab-cloudformation-deploy.sh file will look like

#!/bin/bash

set -f

string=$APP\_SERVER\_IDS

array=(${string//,/ })

echo "Deploy"

for i in "${!array[@]}" ; do

echo "Deploy on instance$i"

#comment uncomment according the selected method , ssm session and cli or ssm and ssh

#uncomment if run commands with ssh

#ssh -i ~/.ssh/id\_app $APP\_SERVER\_USER@${array[i]} "cd $APP\_PATH && sudo git pull $CI\_REPOSITORY\_URL $CI\_COMMIT\_BRANCH && $APP\_RESTART\_COMMANDS"

#or using ssm cli

if ! aws ssm start-session --target ${array[i]}; then

echo "ssm session error"

continue;

fi

sleep 5

sudo su –

cd $APP\_PATH

sudo git pull $CI\_REPOSITORY\_URL $CI\_COMMIT\_BRANCH

$APP\_RESTART\_COMMANDS

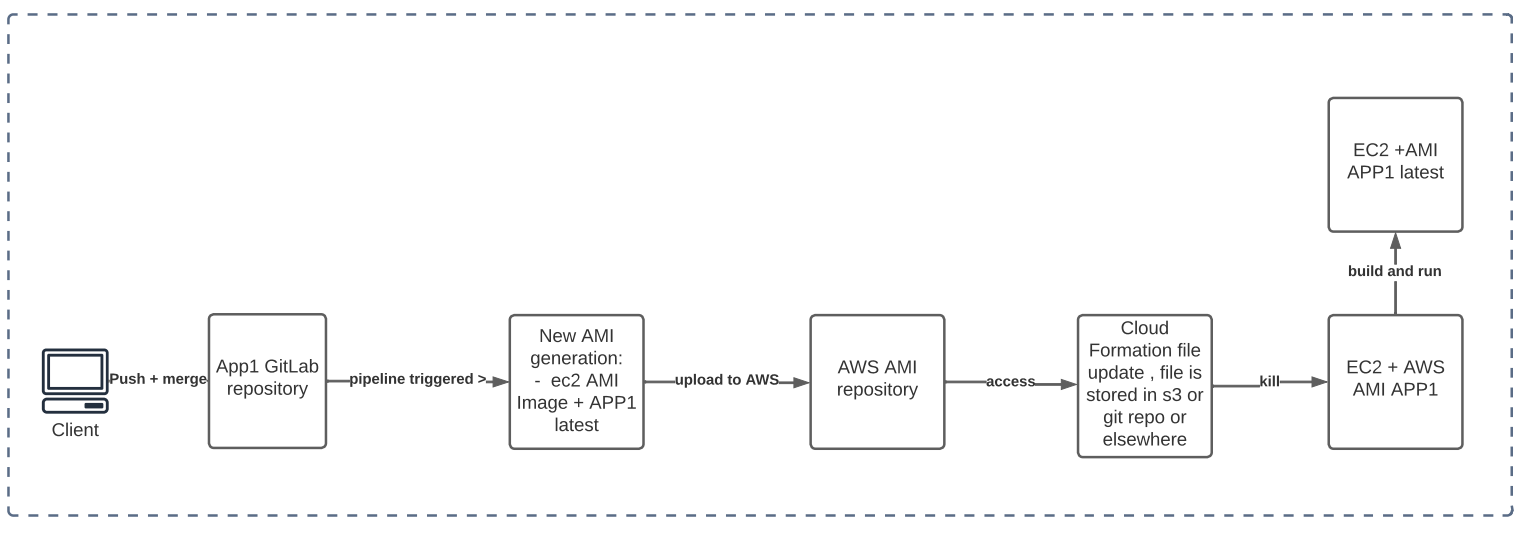
done

After a code repository update, we trigger a script that will use packer and provide it with various parameters as inputs. This step is to use the code from the updated repository, generate a new AMI from it, and push it to AWS.

This new AMI is uploaded to AWS and available as part of the list of available AMI. And we retrieve the ami id .

We then use this ami id to update the cloudformation file and submit it to AWS.

If all succeed and get validated by AWS, the stack is updated and the ec2 is replaced by a new one using the specified AMI id.



1. TODO/Improvements:

* Moving the sensitive information to AWS SSM Parameter Store
* Using AWS Code Pipeline
* Using Jenkins
* Using ssm via a script set up in an ssm document and executed via ssm cli