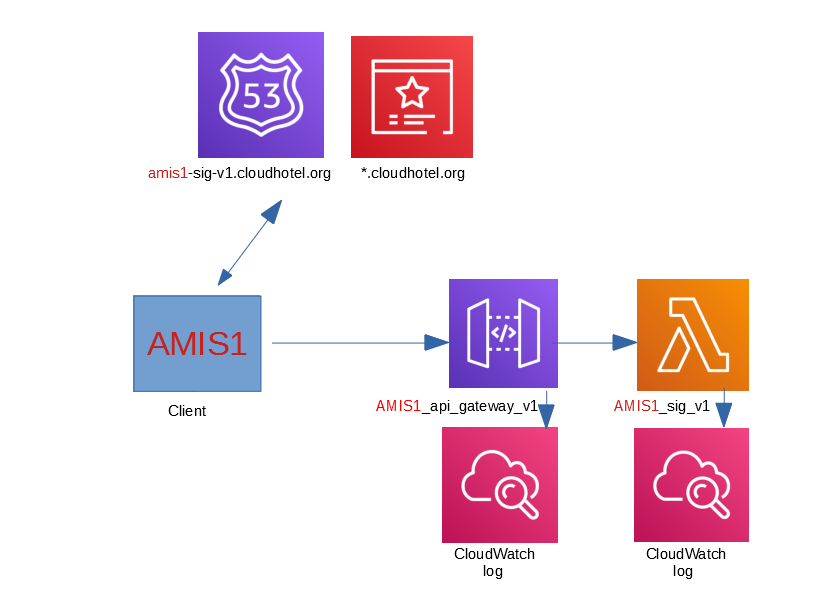
**1. Introduction**

In this SIG meeting (Special Interest Group meeting) we will look at CI/CD in AWS: the deployment and maintenance of AWS objects in a CI/CD pipeline.

We will use a simple example: an API Gateway with a Lambda function behind it. The API Gateway will use a DNS entry in the domein cloudhotel.org. The environment looks like:



The certificate already exists, the DNS-entries will be created by you in this SIG.

We will use terraform for the deployment. For people who are new to terraform, we will do this from the virtual machine first, later we will use the AWS pipeline for that.

**2. SIG environment**

In this paper I use AMIS1, in the region eu-west-1, with abbreviation euw1. You will need to replace these settings several times by the name and region you got from me. Every user has an own virtual machine (VM). I will send you the user-id’s with passwords (and links) for the AWS GUI in a zip file. To enter the VM you need (per region) a file with the key, this is also in the zip file. You are not all in the same region (that would be too confusing).

You will find the IP address from your VM by logging on as AMIS1 to the GUI. Change your region after logging on (in the upper right part of the screen) and then go to the EC2 service. Within the EC2 service, go to Instances>Instances in the left menu. You will see a list with VM’s. One of them is called AMIS1-vm. When this is started, you will see both in the upper part as (when you click on AMIS1-vm) in the details the IP-adres of your VM.

When the VM is not started anymore, you can start it again by clicking on the VM and then press the button Actions > Instance State > Start. After starting up, you will have another IP-address than you had before, the content of the disk is however the same as it was when the VM stopped.

You can logon via a bash window by using:

ssh -i AMIS-eu-west-1 ec2-user@1.2.3.4

(bash for windows is installed when you install git for windows, [click here for the download page](https://git-scm.com/download/win)).

3. Using **CodeCommit** from within the virtual machine

It is possible to use a VM in AWS without entering credentials. In that case, you will use the credentials that is attached to the VM. You can do so, by installing the package **git-remote-codecommit** on the VM (see link [1] for more information). Within this SIG this has already be done for you.

4. Cloning your repository

You are ready to clone your repository for the first time. This can be done using the standard git clone command. This has another form than you would use for github because you are using git from within AWS now:

git clone codecommit::eu-west-1://AMIS1-repo

You will see a warning for having cloned an empty repository. This is correct, we will fill it in a few seconds. But before doing so, we will configure our name and e-mail (don’t worry, if you follow this paper you will not get any e-mail):

cd AMIS1-repo

git config --global user.name “<your name>”

git config --global user.email “<your e-mail>”

We will copy the files under **/start** to the AMIS1-repo:

cd ~

cp -r start/\* AMIS1-repo/

Let’s also add it to the repo within AWS:

cd AMIS1-repo

git status

git add --all .

git commit -m “First commit”

git push origin master

Go in the GUI to the AWS CodeCommit service. Click on the link of the repository, you will see the file terraform\_sig.tf and the directory lambdas, with the file sig.py in it.

Look in the left menu, click on Commits. You will see your own name at the First Commit. When you click on the commit-id, then you will see what has been added.

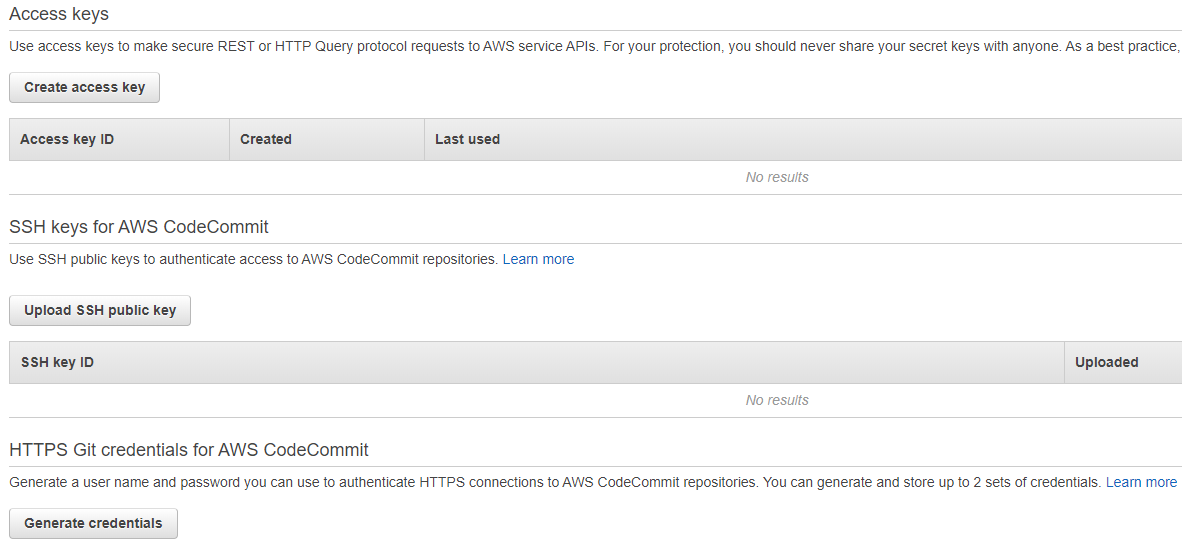
5. Terraform

Let’s first see what we are about to roll out, before we will continue looking at the build tool from AWS. Use **cd ~/start** to go to the start directory. In this directory is a terraform file with the name **terraform\_sig.tf**. Open this file using vi or nano. One of the variables is user\_prefix, this has the value “AMIS1”. Change this in the name of the user-id that you got from me. Everything you will deploy will start this this prefix. Check also the region name in the variable aws\_region and the abbreviation in the variable aws\_region\_abbr and change this if necessary. Save the file after you made the changes.

You might have noticed that the access key and secret access key are not used: these variables are commented out. When you come from outside AWS (f.e. from a VM from your laptop, or from Windows), these fields have to be filled. Terraform will use the policies that are connected to the user that owns the access key and secret access key.

Now you are in the virtual machine in AWS, the policy is connected to the virtual machine and Terraform recognizes that you want to use that policy.

The same applies to git. The credentials are normally connected to an AWS IAM (Identity and Access Management) user. When you go to your own IAM user, you will see that you can both upload an ssh key and download HTTPS credentials. See the picture below (you will find this under IAM > users > AMIS1 > tab Security credentials):



We work from within an EC2 virtual machine,and because we use the credentials that are connected to the EC2, we don’t need those credentials tonight. Therefore you don’t see the keys for the AMIS1 user in this screen (and you will get error messages when you try to create new keys).

Look at the properties of your virtual machine and see if you can find out what permissions you have in the VM. The JSON is often more clear than the Policy summary. (When you cannot find the answer, please look at the end of this document, under [1]). You will see for example that the VM has all permissions within CodeCommit, API Gateway and Lambda within the region of your user.

Go back to your virtual machine. Save the changes and leave the editor. Let’s roll out the objects:

cd ~/start/lambdas

zip sig.zip sig.py

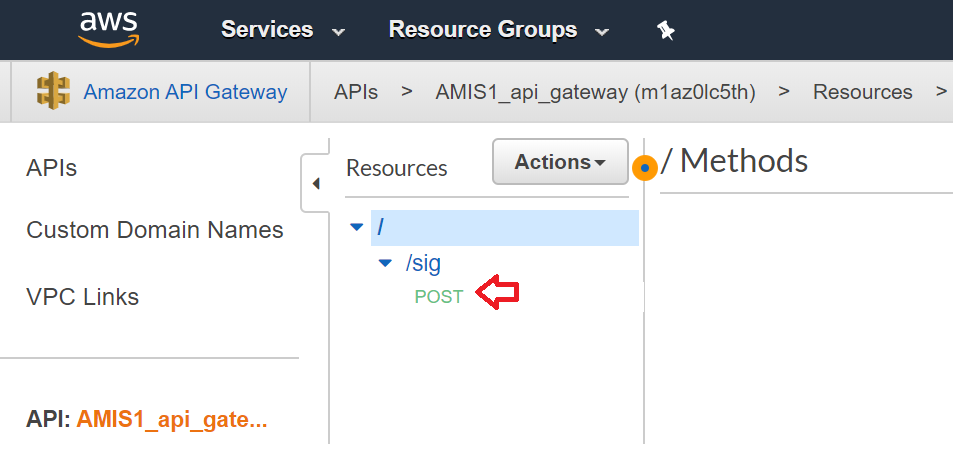
cd ..

../terraform init

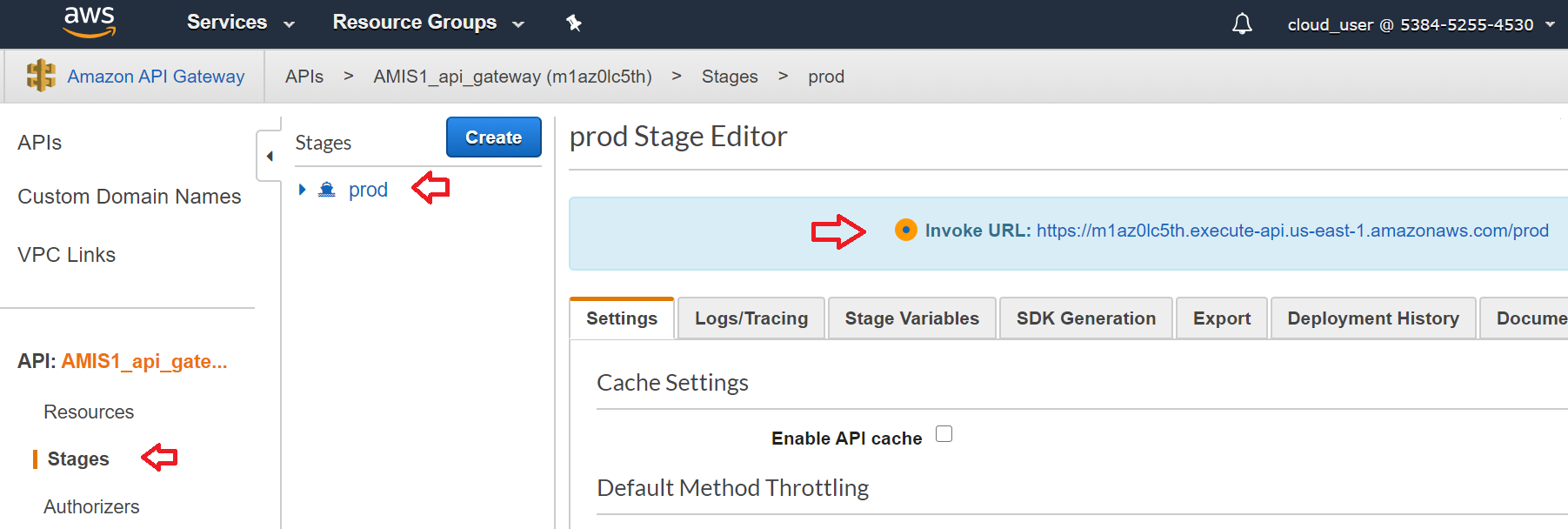
../terraform apply -auto-approve

6. API Gateway

The enrollment will take a while. When this is ready, then go back to the API Gateway service. You will see (for example) a gateway with the name AMIS1\_api\_gateway\_v1. Click on this link. You will see that the API Gateway has a resource with the name /sig. Click on the POST under the /sig:



You will see a picture how the flow from the call to the API goes via the function AMIS1\_sig\_v1 and then flows back to the client. Click in the left menu on Stages, then on prod. In that screen you will see an “Invoke URL”:



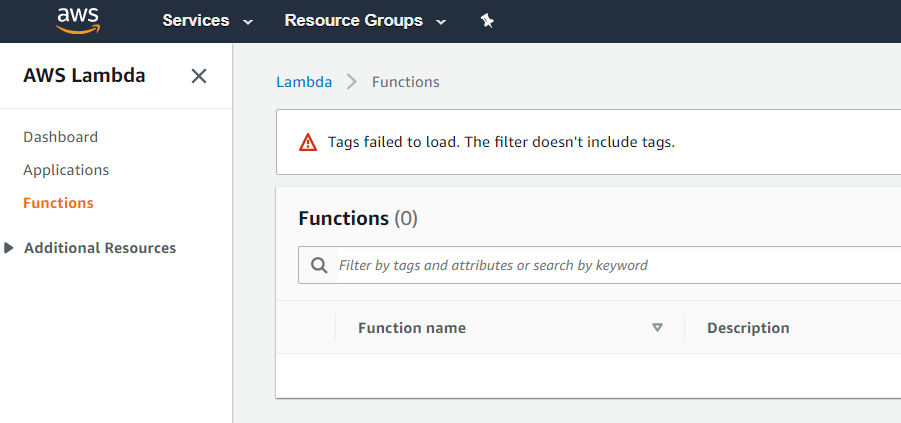
Copy the link, add /sig and use this in your VM (where you can replace Henk with your own name):

curl --header "Content-Type: application/json" -d "{\"name\":\"Henk\"}" https://o2hlm072v3.execute-api.eu-west-1.amazonaws.com/prod/sig

You will see the following text: “Hello Henk from Lambda version v1!”.

7. Lambda functions

Go in the GUI to the AWS Lambda service and click on AMIS1\_sig\_v1. Ignore the message “Tags failed to load” and look at the code from your Lambda function.



You will see that the version from the SIG is retrieved from the environment variabeles of the function. Scroll down below the code, to see what the key and the value of this environment variables is. The content of this variable is changed by terraform when you change the value of the variable sig\_verion in the terraform script.

8. Cleaning up

Go back to the VM and enter the following command:

**../terraform destroy -auto-approve**

Your environment will be destroyed (without further questions). Check in the API Gatewat service and in the Lambda service that you don’t have AMIS1-objects left.

9. Update CodeCommit

When you made changes to your username or region, then copy the terraform\_sig.tf file from the start directory to the AMIS1-repo directory and push the changes to AWS (see the **git add**, **git commit** and **git push** statements earlyer if needed).

10. CodeCommit: Branching

You can branch your repositories in AWS, as you can in github. Create a branch and check the branch out:

cd ~/AMIS1-repo

git branch testbranch

git checkout testbranch

Change something (add f.e. a comment to the file terraform\_sig.tf) and check the file in (use git add, git commit again and use **git push origin testbranch** to push the testbranch to AWS).

In the menu of CodeCommit, go to Pull requests. Click on Create pull request, choose in destination for “master” and in source for “testbranch”. Click on Compare now.

You will see that the testbranch is able to merge with the master branche, lower in the screen you see the changes and the commits that were used to come this far. Give the pull request a title and click on “Create pull request”.

You can see that this is an open pull request and that 0 of the 1 conditions are met. These conditions can be seen in the left menu: under Approval rule templates. When you open this menu, you can see for example **rule template AMIS1-repo.** Open this rule: you can see that multiple approvers are needed (as in: the other AMIS users in this region). You can also see that one approval is needed and that this rule applies to the repository AMIS1-repo.

You cannot approve your own changes, so you will need someone else in your region to approve the changes. Look in the spreadsheet who also has a user id in your region. I can approve the changes as well when I use the AMIS0 user. When you have an approval, you can see this (after a refresh of your screen – press F5) in your screen as well. You will also see a button “Merge” in the pull request screen as well. It is possible for both the creator of the pull request and for the approver to do the merge. This evening, the person who does the pull request is also allowed to do the merge. In this screen, you will see the same options as you have in git.

11. Why use CodeCommit when you can use Github?

There are three reasons:

1) You can use triggers on several events. This means that you can send an event to a Lambda function or to an SNS topic. Click in the left menu (in the repository) on Settings and then on tab Triggers. You can see the possibilities when you click on Create trigger.

2) You can check your (Java) code by **Amazon CodeGuru Reviewer**. You will get feedback from Amazon CodeGuru Reviewer when you do a pull request, in the same form as you would get it from a colleague. Amazon CodeGuru checks on technical rules. The second review could be done by a human colleague, based on functionality that should be there – or not. On this moment CodeGuru is only available for Java.

3) Last but not least: as developers already have an AWS account, then you only have to give them permissions in one place, only in AWS. By giving permissions on group level, you will save time in giving permissions. Giving permissions can be done very grannular, see the next image for an overview.



12. My favorate subject: about branching and CI/CD...

There are companies where they don’t use branches. These companies ask all their employees to check in their code every day in the master branch. The parts that they are working on can be disabled using “feature toggles”. The advantage is that developers can always work on a relatively new version of the code and it is much faster visible when developers are changing the same code. The idea is also that code in CI/CD will never be reverted. From that idea, the history in a repository is not too relevant. More information about feature toggles can be found on the site martinfowler.com: see link [2].

We will use the idea that it is not that bad to work in the master branch use in this SIG as well. Go to the home directory from your repository and switch back to the master branch, changing the files that were changed by the local directory as well:

cd ~/AMIS1-repo

git checkout master

git pull

13. CodeBuild: changes in terraform

Before we proceed with CodeBuild we have to change the terraform script first. Up to now, we used the terraform script in a virtual machine. The data about the objects that are created in AWS are stored in the file **terraform.tfstate**. The AWS library that is used for the deployment of objects to AWS is stored in de directory **.terraform**. You can see this with an **ls -al** in the **~/start** directory. AWS CodeBuild doesn’t have a standard place where this file and this directory can be stored. Terraform has a solution for this: the state file can be stored in an S3 bucket. The .terraform directory doesn’t contain data this directory can easily be created just before doing a destroy.

When you use a central place for storing the state file, you might run into the problem that multiple operators are using the same terraform files for an enrollment of the same environment on the same time. The default solution for this is to use a (hidden) file **.terraform.tfstate.lock.info,** that is removed after an init, plan, apply or destroy command. When we use terraform to deploy the environment, other operators (or other build commands) can try to do the same. This lock should therefore be centralized as well. Terraform uses a DynamoDB table for that.

We should add some code to achieve this (do this in the Providers block, under provider “aws” { … }):

terraform {

backend “s3” {

encrypt = true

bucket = “amis-sig-euw1-bucket”

key = “terraform/AMIS1/terraform.tfstate”

dynamodb\_table = “AMIS1\_state\_locking”

region = “eu-west-1”

}

}

Please mind that when you copy and paste that the double quotes in this document are different than the dubbele quotes in your VM (so: replace all double quotes by -eh- double quotes).

Add, commit and push the file back to the master branch from the AWS repository. Copy the **terraform\_sig.tf** file back to the **~/start** directory and start terraform from there to see if this works correctly. During the init you will see a message that it uses s3 as a backend now. Also use a destroy after the apply.

Look in the s3-bucket and look also (during as wel as after the apply or destroy) in the DynamoDB-table: do you see where (and how) the locking information and the state files are stored?

**14. CodeBuild**

Codebuild is a service to build artifacts. We use it to zip the sig.py file, after that this zip is used in the terraform deployment. To use CodeBuild you need a yaml file, which is called **buildspec.yml**. This file should look like the example under this paragraph. In this example, you see only echo commands and a non existing my\_file.out in the my\_dir directory as a result: when you would use codebuild to build a jar file and deploy the jar file with codedeploy then you would need this artifact part of the file. The version is the version number of AWS, this is always 0.2:

version: 0.2

phases:

install:

commands:

- echo `date` – install commands

pre\_build:

commands:

- echo `date` – pre\_build commands

build:

commands:

- echo `date` – build commands

post\_build:

commands:

- echo `date` – post\_build commands

artifacts:

files:

- my\_dir/my\_file.out

Create a new file buildspec.yml in de ~/AMIS1-repo directory, copy these commands, destroy the artifacts part. Create the zip file sig.zip and specify the commands terraform init en terraform apply somewhere. Tip: zip en unzip already exist on the build environment, you will have to download and install Terraform via the following commandos (--output terraform.zip is part of the curl command and should be on the same line as the curl):

curl https://releases.hashicorp.com/terraform/0.12.26/terraform\_0.12.26\_linux\_amd64.zip

--output terraform.zip

unzip terraform.zip

mv terraform /usr/local/bin/

It is allowed to give multiple commands per phase, these have to be preceded by a dash (–). Also add the terraform commands to enroll the objects (if you don’t know how to proceed, you might check answer [2] at the end of this paper).

Use git to push the file in the master branch of the CodeCommit repository.

Go to the CodeBuild service in the GUI from AWS, click on Create build project. Give it a name that starts with the user-id (f.e. AMIS1-Build). Most questions in this screen will not be too difficult to answer. A few questions that can be confusing:

– Connect the build to the master branche of your CodeCommit repository

– Use in the environment the default “Managed image” and use the newest Amazon Linux 2 x86\_64 environment.

– Use an existing service role arn:aws:iam::300577164517:role/AMIS\_euw1\_codebuild\_role. Uncheck the checkbox before “Allow AWS CodeBuild to modify this service role so it can be used with this build project”.

– Don’t use artifacts, do use CloudWatch logging, leave the group name and the stream name empty. Don’t send logs to S3.

Click after the creation of the build two times on an orange “Start build” button, follow the log with Tail logs and check that the API Gateway and the Lambda function have been created when the build is ready. Look in CloudWatch as well to the logs (left menu under Logs > Log groups), you can see the same output (in less fancy colours) as in de logs you saw with Tail logs.

15. Ehm… what about the destroy?

Good question. In the example (see link [3]) that I used as the base for this SIG, an environment variabele TF\_COMMAND is used, that can either be apply (for the deploy) and destroy (for the removal of the objects). The command will become:

terraform $TF\_COMMAND -auto-approve

where TF\_COMMAND can be apply as well as destroy.

Let’s change our code so it works in our environment as well. Check the code that has been changed in via git. Change the build to add the environment variabele TF\_COMMAND with the correct value. Mind to uncheck the checkbox before “Allow AWS CodeBuild to modify this service role so it can be used with this build project”.

Test the sollution by doing a destroy. Of course I expect all objects to be deleted, check that via the API Gateway and the Lambda service.

**16. CodeDeploy**

If you ever want to deploy applications via a VM or via containers, then look at CodeDeploy. You can also use it for Lambda functions, but not in combination with other AWS objects. This evening we will not do anything with CodeDeploy, but you have enough permissions to look around. CodeDeploy works with a simular file as buildspec.yml that contains the commands to deploy a new version of the repository, it is called **appspec.yml** here.

**17. CodePipeline**

We configured the build and are also able to deploy objects to AWS. The only disadvantage is, that this isn’t done automatically. We will use CodePipeline for that. Go to the AWS Pipeline service and click on “Create pipeline”. Give the pipeline a name that starts with your user-id (f.e. AMIS1-pipeline). Click on “Existing service role” and choose for the role AMIS\_euw1\_codepipeline\_role.

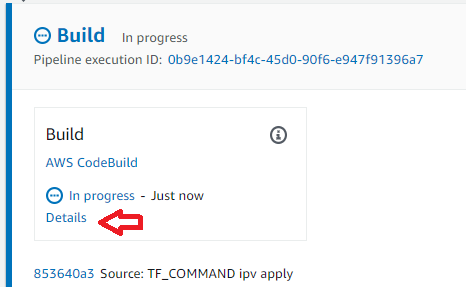
In step 2, choose your repository and use the default “Amazon Cloudwatch Events (recommended)”.

In step 3, choose for CodeBuild and choose your own build. You don’t need to specify environment variables (you will use the environment variable that is specified in CodeBuild).

In step 4, choose for “Skip deploy stage”

In step 5, click on “Create pipeline”

The pipeline will be started as soon as you clicked on “Create pipeline”. You will see that when the pipeline is ready, no objects are added (yet), because the environment variable TF\_COMMAND still has the value destroy. Change this back to apply. Go back to your pipeline and click on “Release change”. You can follow the progress by clicking on the blue link that is called “Details” under CodeBuild:



When the build is ready, go to the Route53 (DNS-) service from AWS. Click on Hosted zones and look under cloudhotel.org. You will see a DNS entry for your own user: amis1-sig-v1.cloudhotel.org. Test this, by going back to your VM and testing this with the curl command:

curl --header "Content-Type: application/json" -d "{\"name\":\"Henk\"}" https://amis1-sig-v1.cloudhotel.org/sig

We will now create a version 2. Change the variable sig\_version into v2 in terraform\_sig.tf and use git to push the changes to AWS. CodePipeline will see the changes and the build will remove everything from v1 and will deploy v2. Check in the API Gateway and in Lambda that you only see v2 objects.

**18. Terraform: running two versions next to eachother**

This works as expected: old objects will be thrown away and new ones will be deployed. But: wouldn’t it be great that two versions would exist next to eachother? Especially with API’s? So you can first deploy the first version, use that in your production environment, and then deploy version 2? After everybody gradually moved to version 2, we then can remove version 1.

We could achieve this by using a variabele in the s3 directory, for example:

terraform {

backend “s3” {

encrypt = true

bucket = “amis-sig-euw1-bucket”

key = “terraform/AMIS1/**${var.sig\_version}**/terraform.tfstate”

dynamodb\_table = “AMIS1\_state\_locking”

region = “eu-west-1”

}

}

This would give us the flexibility to do minor changes in-place by not changing the version number. Bigger changes can be done in two seperate versions. When you try this, you get an error: Variables may not be used here . Hashicorp has another solution (link [4]): we can use an **-backend-config** parameter to the init command.

Before we will try this, we have to destroy the current environment because we will use a different s3 directory later. Change the TF\_COMMAND in destroy and start a build of the current repository. Check that there is no API Gateway with your user-id anymore and throw away the terraform.tfstate file in your S3 bucket. Remove all records from your own table in DynamoDB. **You shouldn’t start a build yet however!**

First, create a new file in the home directory from the repository and call it **terraform\_s3\_directory.cfg**. Change the name of the directory and have it include the version:

key=”terraform/AMIS1/**v2**/terraform.tfstate”

Add in the buildspec.yml the parameter **-backend-config= terraform\_s3\_directory.cfg** to the init command. Use git to push the changes to AWS and check in the pipeline that this solution (with an apply) works. Check after the deployment of the environment that the s3 directory structure for your user contains a version number.

Create a version 3 now (both in the files terraform\_sig.tf and terraform\_s3\_directory.cfg) and send the changes to AWS (using git). Check with a curl from your VM that both versions run at the same time.

**19. Delete old versions**

You shouldn’t change the existing repository to change the version (and other configuration) back to v2 in the terraform file and the file terraform\_s3\_directory.cfg, change TF\_COMMAND to destroy and then checkin the code: this solution is very error prone (especially on the moment you change everything again to v3 after you destroyed v2).

There is a better solution: when you create a (next to) empty terraform file in a new repository, where you use the versienumber and the link to the s3 files to indicate what has to be thrown away. Let’s create a new repository: AMIS1-repo-destroy. Clone this to your VM with the command

cd ~

git clone codecommit::eu-west-1://AMIS1-repo-destroy

Use the following commands to change your name:

cd AMIS1-repo-destroy

git config --global user.name “<your name>”

git config --global user.email “<your e-mail>”

Add a file terraform\_sig.tf with the variables and the providers (assumed that you want to destroy v2 – use your own terraform\_sig.tf file from the AMIS1-repo repository as a base for this file):

#################################################################

# VARIABLES

#################################################################

# variable "aws\_access\_key" {}

# variable "aws\_secret\_key" {}

variable "aws\_region" { default = "eu-west-1"}

variable "aws\_region\_abbr" { default = "euw1"}

variable "name\_prefix" { default = "AMIS" }

variable "user\_prefix" { default = "AMIS1" }

variable "sig\_version" { default = "v2" }

variable "stage\_name" { default = "prod" }

variable "log\_level\_api\_gateway" { default = "INFO" }

variable "domainname" { default = "cloudhotel.org" }

#################################################################

# PROVIDERS

#################################################################

provider "aws" {

# access\_key = var.aws\_access\_key

# secret\_key = var.aws\_secret\_key

region = var.aws\_region

}

terraform {

backend "s3" {

encrypt = true

bucket = "amis-sig-euw1-bucket"

dynamodb\_table = "AMIS1\_state\_locking"

region = "eu-west-1"

}

}

Copy the file **terraform\_s3\_directory.cfg** from the AMIS1-repo and change version v3 in the directory name back to v2.

Copy the buildspec.yml and change $TF\_COMMAND in **destroy** (and change $TF\_COMMAND in the buildspec.yml file in the AMIS1-repo to **apply** if you want – you will never change this variable to destroy in the future). Remove the creation of the zip file in the AMIS1-repo-destroy repository.

Add this repo to CodeBuild (see previous steps) and also to codepipeline (though… would you automate deletion in a pipeline?).

Use the AMIS1-repo from now on for adding new functionality/versions/steps and use AMIS1-repo-destroy to destroy old versions.

Remove all old versions via the AMIS1-repo-destroy repository.

20. Last thoughts...

I gave much effort in trying to let version 2 and version 3 next to eachother and then use blue/green deployment to move from version 2 to version 3. When one would create a weighted DNS entry in Route53 that points to one of the previous versions (both via CNAME or via A-record with AWS-ALIAS), than this works fine. The moment you use it it will fail, you will get an error message with one of the two versions.

This is, because the name of the A-record in the domain has to be present in the API Gateway, and that name can point to version 2 -or- version 3 only (so: not to both at the same time). It is, however, possible to use weighted DNS records to two different regions and use one version in de first region and the next version in another region. You will, however, need two repositories for that (those are region dependent).

Up to now, we maintained the version number in both the terraform\_sig.tf file as in the terraform\_s3\_directory.cfg file. It is in general not too smart to maintain the same variable in two places. You might want to use the following piece of code to use the content of the variable in the terraform file to change the cfg file:

- user\_prefix=`grep user\_prefix ./terraform\_sig.tf | head -n 1 | awk -F"\"" '{print $4}'`

- sig\_version=`grep sig\_version ./terraform\_sig.tf | head -n 1 | awk -F"\"" '{print $4}'`

- echo “key = \”terraform/${user\_prefix}/${sig\_version}/terraform.tfstate\”” >

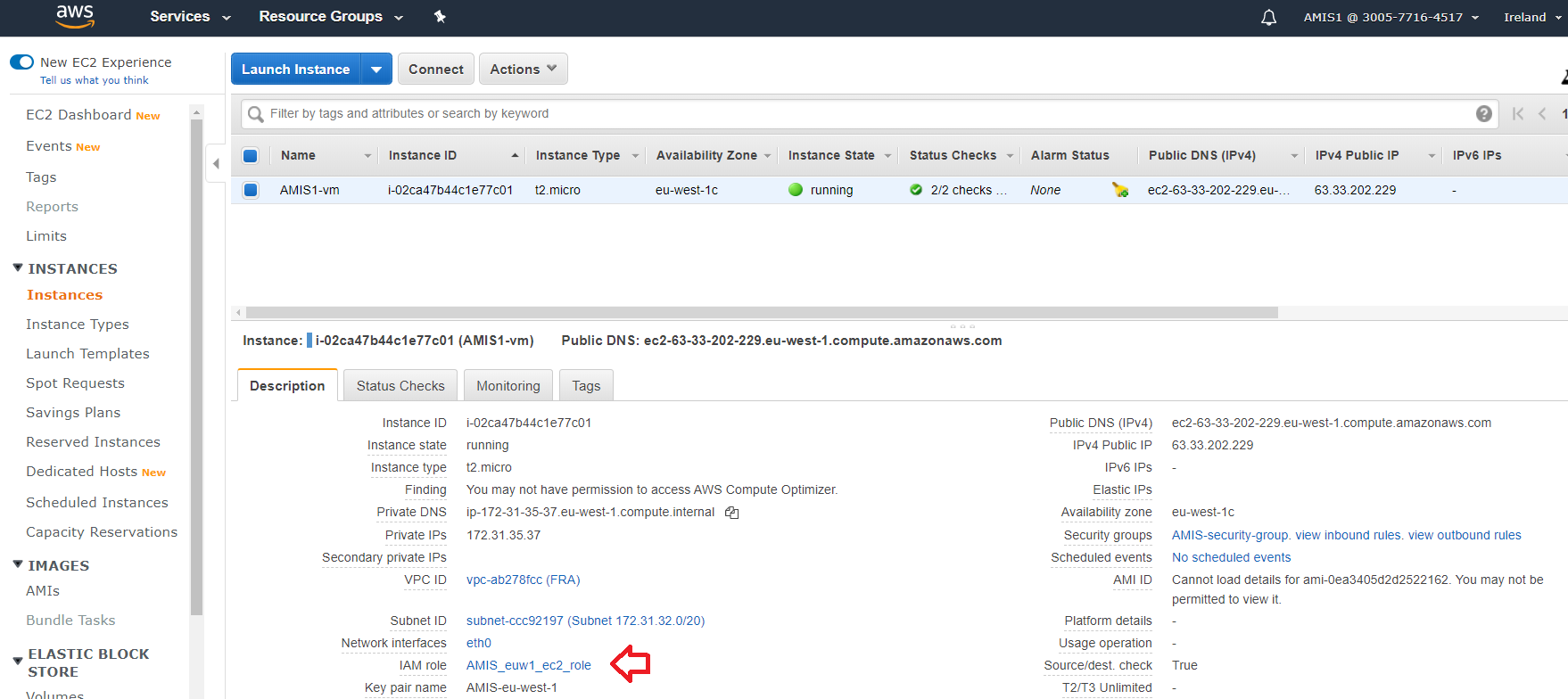
./terraform\_s3\_directory.cfg

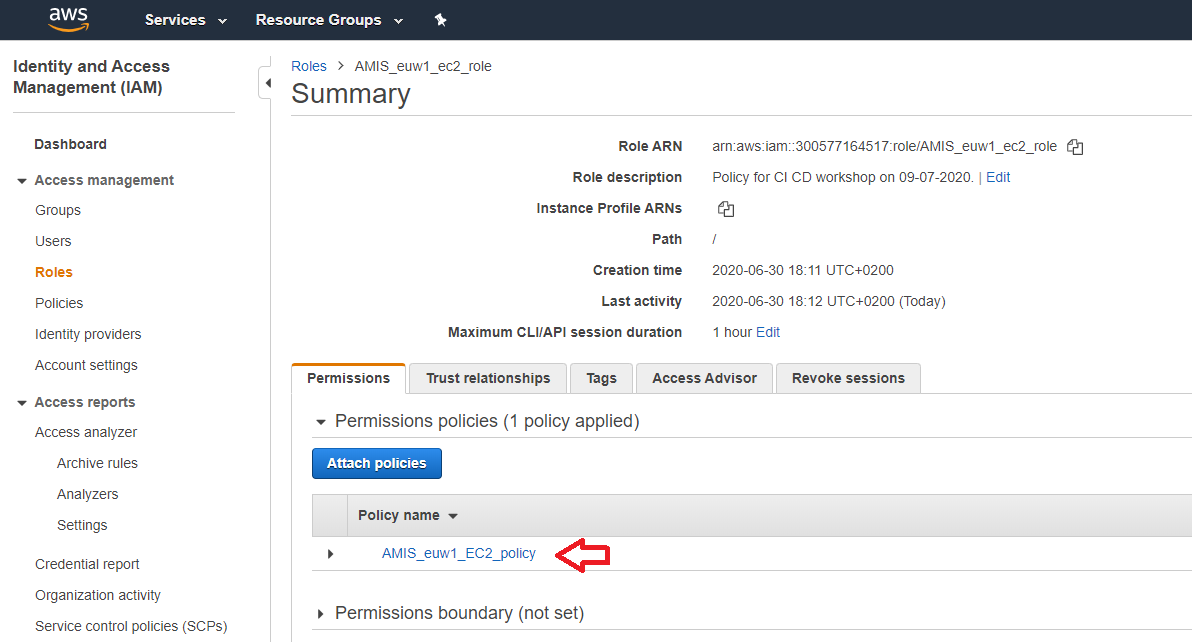
You might add some text in the terraform\_s3\_directory.cfg file in the repository where and how this file is used in the buildspec.yml.

**21. Answers:**

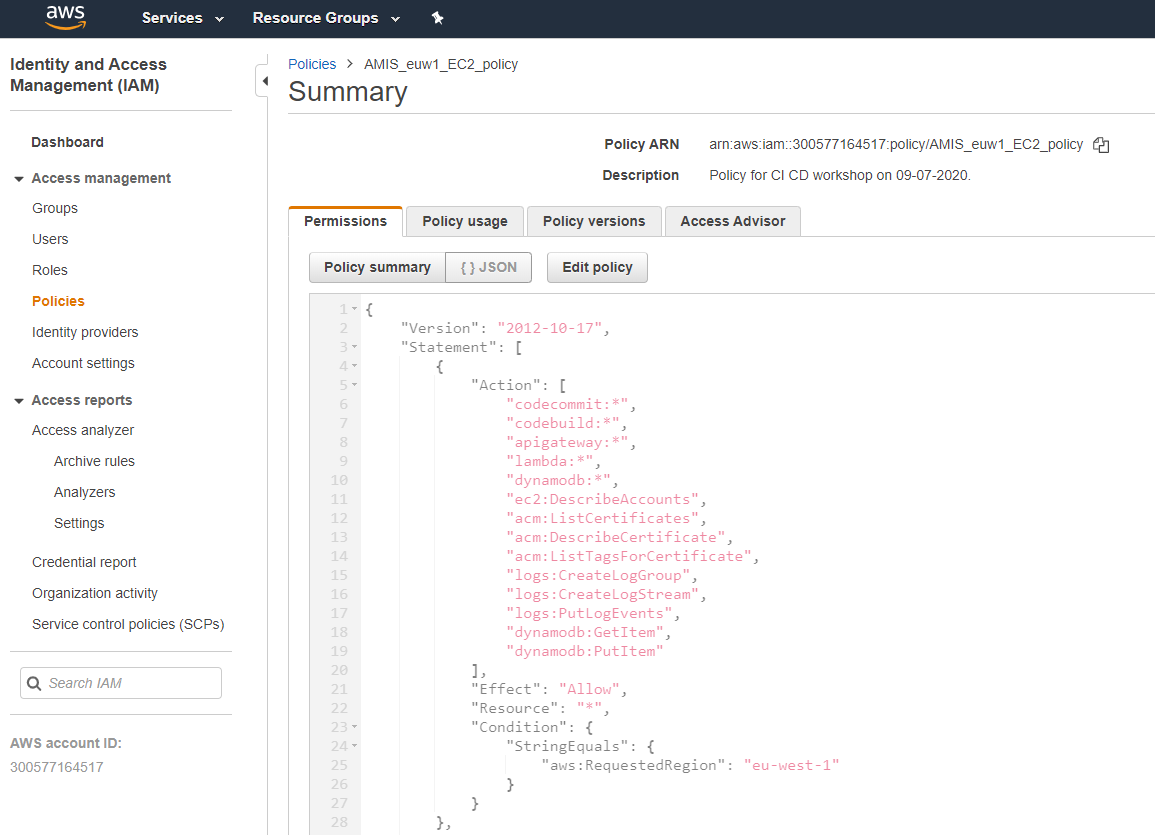
Answer [1] to the question: which permissions has the VM?

Look at the properties of your VM. You will find the name of the role that is connected to this VM after the text “IAM Role”:



. When you click on this link, a second tab is opened, with the IAM role that is connected to the VM. Within this role, you see a link to the policy in the first tab:

When you click on this link and then on the button **{} JSON**, you will see the permissions you have on the virtual machine:



22. Answer [2] on the question: create a buildspec.yml

In our example, it is doesn’t matter what you do in which phase. You will enter on the root directory of the repository, so only use relative names of paths in the cd command and only to directories within this repository. Your buildspec.yml might look like:

version: 0.2

phases:

install:

commands:

– echo `date` install commands

– yum install unzip -y

– curl <https://releases.hashicorp.com/terraform/0.12.24/terraform_0.12.24_linux_amd64.zip> --output terraform.zip

– unzip terraform.zip

– mv terraform /usr/local/bin/

pre\_build:

commands:

– echo `date` pre\_build commands

– cd lambdas

- zip sig.zip sig.py

- cd ..

- terraform init

build:

commands:

– echo `date` build commands

- terraform apply -auto-approve

post\_build:

commands:

– echo `date` post\_build commands

**23. Links**

[1] <https://docs.aws.amazon.com/codecommit/latest/userguide/setting-up-git-remote-codecommit.html>

[2] <https://martinfowler.com/articles/feature-toggles.html>

[3] <https://aws.amazon.com/blogs/security/how-use-ci-cd-deploy-configure-aws-security-services-terraform/> , it uses this repository: <https://github.com/aws-samples/aws-security-services-with-terraform/blob/master/provider.tf>

[4] <https://github.com/hashicorp/terraform/issues/22088>

**24. Appendix: delete objects by hand**

It might happen… Something goes wrong, you loose your status file and there are still objects in AWS. How to remove these?

In this example:

– Lambda service:

– Remove the lambda functie AMIS1\_sig\_v1

– API Gateway:

– Go to the Custom domain names in the left menu, remove amis1-sig-v1.cloudhotel.org first

– Go in the left menu to APIs and destroy the AMIS1\_api\_gateway\_v1

– Route53:

– destroy the amis1-sig-v1.cloudhotel.org entry