AWS tool comparison

Release 0.0.7

Feb 22, 2022

Contents:

Comparison of different methods for creating objects on your AWS account.

CHAPTER 1

Getting started

This repository contains the instructions for deploying an EC2 and a lambda on your AWS account. It is part of the educational repositories to learn how to write code for deploying infrastructure.

The repository contains all that you need to deploy an EC2 and a lambda on your AWS account

- this package contains multiple ways for deploying an EC2 and a lambda on your AWS account
- the goal is to compare these methods and to learn pros and cons of each methods

1.1 Prerequisite

One AWS account, in this repository called **your-account**

1.2 Installation

The package is not self-consistent: you will find the installation on the official web site for each tool below.

- aws-cli v2
- aws-cdk
- sam
- sceptre
- serverless
- terraform

1.3 Usage

Read the documentation on readthedocs.

1.4 License

This package is released under the MIT license. See LICENSE for details.

CHAPTER 2

aws-cli

The available commands of aws-cli are a lot, and you can manage all that you need by those commands and bash scripts.

As an example, you can find some commands below.

2.1 aws-lambda

The available commands of aws-lambda are a lot, and you can manage all that you need by those commands and bash scripts.

2.1.1 By shell

The command for deploying a lambda is

```
export AWS_PROFILE=your-account
aws lambda create-function --function-name $name --handler $handler --runtime
↪$runtime --role $role --zip-file fileb://code.zip
```

Where

- **$handler** is the name of the method within your code that Lambda calls to execute your function
- **$runtime** is the identifier for the language that you use in your code and it has many choices
- **$role** is the Amazon Resource Name (ARN) of the execution role of the lambda function

You can invoke the lambda function by command line

```
export AWS_PROFILE=your-account
aws lambda invoke --function-name $name outfile --payload '{"key":"value"}'
```

And also you can delete it by command line

```
export AWS_PROFILE=your-account
aws lambda delete-function --function-name $name
```

2.1.2 By a bash script

Below you can find an example of deployment of a lambda function by a bash custom script.

```
cd cli/lambda
git clone https://github.com/bilardi/aws-saving.git
export AWS_PROFILE=your-account
bash aws-lambda.sh deploy aws-saving/aws_saving
```

And for managing the other commands described above, you can use the same bash script

```
cd cli/lambda
bash aws-lambda.sh # print the commands list
```

2.1.3 Remember

When you need to

- change the permissions or other non-code attribute, it is better to delete and to redeploy the lambda function
- add the lambda function to a VPN, the deletion of that lambda function will be much slower because the system has to delete also the network objects

2.2 aws-ec2

The available commands of aws-ec2 are a lot, and you can manage all that you need by those commands and bash scripts.

2.2.1 By shell

The command for deploying an EC2 is

```
export AWS_PROFILE=your-account
aws ec2 run-instances --image-id $ami --count 1 --instance-type $type --subnet-id $sn␣
↪--security-group-ids $sg # --user-data file://$file --key-name $key
```

The script return the instanceId of EC2 that you can use in the commands below

```
export AWS_PROFILE=your-account
export INSTANCE_ID=your-instance-id
```

for getting EC2 status,

```
aws ec2 describe-instance-status --instance-id INSTANCE_ID
```

for stopping EC2,

```
aws ec2 stop-instances --instance-ids INSTANCE_ID
```

for changing its instance type,

```
aws ec2 modify-instance-attribute --instance-id INSTANCE_ID --attribute instanceType -
↪-instance-type t3.small
# all instance T3 types are: t3.nano | t3.micro | t3.small | t3.medium | t3.large |␣
↪t3.xlarge | t3.2xlarge
```

for changing its volume type,

```
aws ec2 modify-volume --volume-id INSTANCE_ID ---volume-type io1
# all volume types are: standard | io1 | gp2 | sc1 | st1
```

for starting EC2,

```
aws ec2 start-instances --instance-ids INSTANCE_ID
```

for deleting EC2,

```
aws ec2 terminate-instances --instance-ids INSTANCE_ID
```

2.2.2 By a bash script

Below you can find an example of deployment of a Minetest server by a bash custom script.

```
cd cli/ec2
curl -O https://raw.githubusercontent.com/bilardi/minetest/master/install.sh
export AWS_PROFILE=your-account
bash aws-ec2.sh deploy install.sh
```

And for managing the other commands described above, you can use the same bash script

```
cd cli/ec2
bash aws-ec2.sh # print the commands list
```

2.2.3 Remember

When you need to change

- the instance type or to terminate instance, before you have to stop the instance
- the volume type, it is not necessary to stop the instance, but if you stop it, the process will be more fast

2.3 aws-cloudformation

The available commands of aws-cloudformation are a lot, and you can manage all your stacks only these commands.

2.3.1 By shell

AWS CloudFormation service needs a template (and optionally parameters) for deploying a stack. A stack is a collection of resources that they are described in the template. The command for deploying a stack

```
export AWS_PROFILE=your-account
export STACK_NAME=your-stack-name
export PREFIX_STACK_FILENAME=your-filename
aws cloudformation create-stack --stack-name $STACK_NAME --template-body file://./
↪templates/$PREFIX_STACK_FILENAME.yaml --parameters file://./config/$PREFIX_STACK_
↪FILENAME.json --capabilities CAPABILITY_NAMED_IAM
```

When the stack will be created, you can get all information of the stack outputs that they are described in the template Outputs section. By the command below you can get the stack outputs

```
aws cloudformation describe-stacks --stack-name $STACK_NAME
```

For updating the stack after changes on parameters or templates files,

```
aws cloudformation update-stack --stack-name $STACK_NAME --template-body file://./
↪templates/$PREFIX_STACK_FILENAME.yaml --parameters file://./config/$PREFIX_STACK_
↪FILENAME.json --capabilities CAPABILITY_NAMED_IAM
```

for deleting the stack,

```
aws cloudformation delete-stack --stack-name $STACK_NAME
```

Below you can find a simple sample of deployment of a Minetest server by aws-cloudformation.

```
cd cli/cloudformation
export AWS_PROFILE=your-account
export EC2_STACK_NAME=minetest-server
export EC2_PREFIX_STACK_FILENAME=ec2
aws cloudformation create-stack --stack-name $EC2_STACK_NAME --template-body file://./
↪templates/$EC2_PREFIX_STACK_FILENAME.yaml --parameters file://./config/$EC2_PREFIX_
↪STACK_FILENAME.json --capabilities CAPABILITY_NAMED_IAM
```

Below you can find an example of deployment of a Minetest server by aws-cloudformation and what else you can manage.

```
cd cli/cloudformation
export AWS_PROFILE=your-account
export EC2_STACK_NAME=minetest-server-more
export EC2_PREFIX_STACK_FILENAME=ec2-more
aws cloudformation create-stack --stack-name $EC2_STACK_NAME --template-body file://./
↪templates/$EC2_PREFIX_STACK_FILENAME.yaml --parameters file://./config/$EC2_PREFIX_
↪STACK_FILENAME.json --capabilities CAPABILITY_NAMED_IAM
```

2.3.2 By a bash script

An ad hoc script is **only necessary** if you need to manage

- more parameters and only a few are dynamic variables like the name of other stacks
- an exception as for deleting a stack with a S3 bucket not empty or a dashboard configuration by external json file, as the sample that you can find in github.com/aws-samples

Below you can find an example of deployment of a Minetest server and a lambda function by a bash custom script.

```
cd cli/cloudformation
export AWS_PROFILE=your-account
bash aws-cloudformation.sh deploy plus
```

And for managing the other commands described above, you can use the same bash script

```
cd cli/cloudformation
bash aws-cloudformation.sh # print the commands list
```

Please, pay attention: in the config files, there are some identifiers that you need to change before running the bash script!

2.3.3 Remember

When you deploy your objects into the cloudformation stacks,

- you can take advantage of outputs section keys of others stacks with Fn::ImportValue
- you can avoid to hardcoding a password directly in the property or in the parameter configuration file by AWS::SecretsManager

CHAPTER 3

aws-cdk

The AWS Cloud Development Kit (AWS CDK) lets you define your cloud infrastructure as code in one of five supported programming languages: an AWS CDK app is an application written in TypeScript, JavaScript, Python, Java, or C# that uses the AWS CDK to define AWS infrastructure.

For each language it is possible to install the **AWS Construct Library packages** and **AWS CDK Toolkit** is available like a npm package

```
npm install -g aws-cdk # for installing AWS CDK
cdk --help # for printing its commands
```

The available commands of AWS CDK Toolkit are a lot, and you can manage all that you need by your favourite language.

3.1 By shell

AWS CDK Toolkit and AWS Construct Library manage initialization, building, testing, deploying and more of what you need. Each language has different needs: good examples are Constructs guide, the Hello World application and CDK Workshop.

AWS CDK Toolkit has common cross languages commands: for initialization of the application,

```
export LANGUAGE=typescript #, javascript, python, java, or csharp
cdk init app --language $LANGUAGE
```

For listing the application stacks,

```
cdk ls
```

For deploying the application,

```
cdk deploy
```

For checking the difference between changes and deployed,

```
cdk diff
```

For deleting,

```
cdk destroy
```

There are also commands for building the templates and testing the application, but they can also be managed by AWS Construct Library in agreement with the choosen language:

- in the CDK Workshop, there is a testing section for TypeScript language
- in this documentation, you can find an example from initialization to destroy, with also a testing section, for Python language

A method for building the template and testing the application by only the AWS tools is to use AWS CDK Toolkit and AWS SAM CLI:

```
cdk synth --no-staging > template.yaml
export LAMBDA_NAME=$(grep MyFunction template.yaml)
sam local invoke $LAMBDA_NAME --no-event
```

3.2 By a bash script

An ad hoc script **maybe** it could be useful for a specific CI / CD system. Generally, it is not necessary.

3.3 Remember

When you use AWS CDK,

- you can test your application on your client by the commands **sam local** or with the choosen language
- you can manage all by code, so you can avoid to hardcoding a password directly in the property or in the parameter configuration file by AWS::SecretsManager and you can also manage a CI / CD system

3.4 Examples

You can find two examples in two different languages.

3.4.1 Python

The **AWS Construct Library packages** is also available like a python package

```
pip search aws-cdk # for searching which module you want to load
pip install aws-cdk.aws-lambda # for installing AWS CDK Lambda module
```

In this sample, there are more modules because it will be also used a CI / CD system:

- all modules are loaded and initialized on files named **app.<something>.py**
- each **app.py** file uses a file named **buildspec.yml** for deploying the CD system by AWS CodePipeline

By shell

AWS CDK Toolkit is useful for initialization of the application,

```
cd cdk/python/
bash build.sh
```

For testing the application, you can check almost the template is correct

```
cd cdk/python/
python3 -m unittest discover -v
```

For deploying a Minetest server and a lambda function.

```
cd cdk/python/
export AWS_PROFILE=your-account
export STAGE=basic
cdk bootstrap
cdk deploy ${STAGE}
```

For deploying a Minetest server and a lambda function and more.

```
cd cdk/python/
export AWS_PROFILE=your-account
export STAGE=more
cdk deploy ${STAGE}
```

For deploying a Minetest server and a lambda function and what else you can manage.

```
cd cdk/python/
export AWS_PROFILE=your-account
export STAGE=plus
cdk deploy ${STAGE}
```

For deploying a Minetest server and a lambda function by a pipeline

```
cd cdk/python/
export AWS_PROFILE=your-account
export STAGE=pipeline
cdk deploy ${STAGE}
```

For deleting,

```
cd cdk/python/
export AWS_PROFILE=your-account
cdk destroy ${STAGE}
```

By a bash script

An ad hoc script **maybe** it could be useful for a specific CI / CD system. Generally, it is not necessary.

In this example, you have a simple introduction of CI / CD system by CodePipeline.

The logic of your CI / CD system is defined on **buildspec.yml** file

- the **AWS simple pipeline** package used, allows you to define your logic on **buildspec.yml** file
- you can use many files, like the sample that you can find in aws-simple-pipeline, or nothing

You can also find

- the **build script**, named build.sh, it is used during the CodeBuild step
- the **deploy script**, named deploy.sh, it is used to differ which command run in each **stage**

These bash script files are prepared in your client before to test your **CI / CD system**: you have to be able to

- **build** all packages that you use before to run **cdk** or **unit test**
- **deploy** your AWS instances from your client before to test your **CI / CD system**

Remember

When you use AWS CDK,

- now there is not an official testing infrastructure, but you can test your application on your client by AWS CDK test Synth package
- you can manage all by code, so also a CI / CD system, like AWS simple pipeline used in this sample

3.4.2 TypeScript

The **AWS Construct Library packages** is also available like a typescript package

```
npm search aws-cdk # for searching which module you want to load
npm install @aws-cdk/aws-lambda # for installing AWS CDK Lambda module
```

In this sample, there are more modules because it will be also used a CI / CD system:

- all modules are loaded and initialized on files named **bin/cdk.<something>.ts**
- each **cdk.ts** file uses a file named **buildspec.yml** for deploying the CD system by AWS CodePipeline

By shell

AWS CDK Toolkit is useful for initialization of the application,

```
cd cdk/typescript/
make install
```

For testing the application, you can check almost the template is correct

```
cd cdk/typescript/
npm test
```

For deploying a Minetest server and a lambda function.

```
cd cdk/typescript/
export AWS_PROFILE=your-account
export STAGE=basic
cdk bootstrap
cdk deploy ${STAGE}
```

For deploying a Minetest server and a lambda function and more.

```
cd cdk/typescript/
export AWS_PROFILE=your-account
export STAGE=more
cdk deploy ${STAGE}
```

For deploying a Minetest server and a lambda function and what else you can manage.

```
cd cdk/typescript/
export AWS_PROFILE=your-account
export STAGE=plus
cdk deploy ${STAGE}
```

For deploying a Minetest server and a lambda function by a pipeline

```
cd cdk/typescript/
export AWS_PROFILE=your-account
export STAGE=pipeline
cdk deploy ${STAGE}
```

For deleting,

```
cd cdk/typescript/
export AWS_PROFILE=your-account
cdk destroy ${STAGE}
```

By a bash script

An ad hoc script **maybe** it could be useful for a specific CI / CD system. Generally, it is not necessary.

In this example, you have a simple introduction of CI / CD system by CodePipeline.

The logic of your CI / CD system is defined on **buildspec.yml** file

- the **AWS simple pipeline** package used, allows you to define your logic on **buildspec.yml** file
- you can use many files, like the sample that you can find in aws-simple-pipeline, or nothing

You can also find

- the **build script**, named build.sh, it is used during the CodeBuild step
- the **deploy script**, named deploy.sh, it is used to differ which command run in each **stage**

These bash script files are prepared in your client before to test your **CI / CD system**: you have to be able to

- **build** all packages that you use before to run **cdk** or **unit test**
- **deploy** your AWS instances from your client before to test your **CI / CD system**

Remember

When you use AWS CDK,

- for TypeScript, there is an official testing infrastructure by @aws-cdk/assert package
- you can manage all by code, so also a CI / CD system, like AWS simple pipeline used in this sample

CHAPTER 4

aws-sam

The AWS Serverless Application Model (AWS SAM) is an open-source framework for building serverless applications. AWS SAM is natively supported by AWS CloudFormation and provides a simplified way of defining the Lambda functions, APIs, databases, and event source mappings needed by your serverless applications.

The AWS SAM template has

- the same anatomy of AWS CloudFormation template with some custom sections
- some custom resources and properties **AWS::Serverless**
- a list of supported AWS properties that AWS SAM CLI can manage

And when you need also others AWS resources and properties, you can add them and deploy the template by AWS CloudFormation or Sceptre. The unique requirement is to keep the **Transform** section, for AWS macros conversion.

AWS SAM prerequisite are **Docker** and, for Linux and Mac, also **Homebrew** (see how to install AWS SAM CLI) and then, by shell, you can install AWS SAM CLI with the command below

```
brew tap aws/tap
brew install aws-sam-cli
```

The available commands of aws-sam-cli are a lot, and you can manage all your serverless stacks with these commands.

4.1 By shell

AWS SAM service manages building, testing, deploying and more of what you need by a template and your code. The command for validate your template,

```
export AWS_PROFILE=your-account
export STACK_FILENAME=your-template-file
sam validate -t $STACK_FILENAME
```

For building and testing your application,

```
export AWS_PROFILE=your-account
export EVENT=your-event-data-json
export SOURCE=your-source-code-folder
export STACK_FILENAME=your-template-file
export PARAMETERS=your-parameters-string
sam build -u -s $SOURCE -t $STACK_FILENAME --parameter-overrides $PARAMETERS
sam local invoke -e $EVENT -t .aws-sam/build/template.yaml --parameter-overrides
↪$PARAMETERS
```

For deploying your stack,

```
export AWS_PROFILE=your-account
export STACK_NAME=your-stack-name
export STACK_FILENAME=your-template-file
export PARAMETERS=your-parameters-string
sam deploy --stack-name $STACK_NAME -t $STACK_FILENAME --parameter-overrides
↪$PARAMETERS --capabilities CAPABILITY_NAMED_IAM
```

For deleting your stack,

```
export AWS_PROFILE=your-account
export STACK_NAME=your-stack-name
aws cloudformation delete-stack --stack-name $STACK_NAME
```

Below you can find a simple sample of deployment of a lambda function by AWS SAM CLI.

```
cd sam/
git clone https://github.com/bilardi/aws-saving
export AWS_PROFILE=your-account
export STACK_NAME=test_aws_saving
export STACK_FILENAME=templates/lambda-basic.yaml
export PARAMETERS=$(tr '\n' ' ' < config/basic/lambda.txt)
sam deploy --stack-name $STACK_NAME -t $STACK_FILENAME --parameter-overrides
↪$PARAMETERS --capabilities CAPABILITY_NAMED_IAM
```

For deploying other features and also an EC2, you need to use AWS CloudFormation or another tool that it supports AWS SAM resources and properties, like Sceptre.

4.2 By a bash script

An ad hoc script is **only necessary** if you need to manage

- more parameters and / or dynamic variables like the name of other stacks like AWS CloudFormation
- an exception as for deleting a stack with a S3 bucket not empty or managing more templates

Below you can find an example of deployment of a lambda function by a bash custom script.

```
cd sam/
export AWS_PROFILE=your-account
bash aws-sam.sh deploy sam
```

And for managing the other commands described above, you can use the same bash script

```
cd sam/
bash aws-sam.sh # print the commands list
```

Please, pay attention: in the config files, there are some identifiers that you need to change before running the bash script!

4.3 Remember

When you use AWS SAM,

- you can test your application on your client by the commands **sam local**
- you can manage AWS macros for Serverless, a little list of the AWS resources and properties and the functions **Fn::Sub** and **Fn::If**, but for all the others you have to manage with AWS CloudFormation or another tool that it supports AWS SAM resources and properties, like Sceptre

CHAPTER 5

tool

There are many many tools available for managing the AWS stacks, it is not possible to list all them or to keep an updated list.

As an example, you can find some tools below.

5.1 Sceptre

Sceptre is a tool to drive CloudFormation. Sceptre manages the creation, update and deletion of stacks while providing meta commands which allow users to retrieve information about their stacks.. yes, this is the first sentence of the introduction of Sceptre in the official site, and yes, it is a all-around tool that you can use to manage your stacks by cli or python script.

The available commands of Sceptre are a lot, and you can manage all that you need by yaml files and, if you want, some python scripts.

Sceptre is available like a python package or a docker image,

```
pip install sceptre # for installing Sceptre
sceptre --help # for printing its commands
```

If you want to install Sceptre and also its plugins for the examples below,

```
cd tool/sceptre
make install
```

If you want to install only the Sceptre plugins for the examples below,

```
cd tool/sceptre
make compile
```

5.1.1 By shell

Sceptre deploys one stack for each configuration file and one configuration file uses one template file. The relation between configuration file and template file could be many to one or one to one.

The command for validating one configuration file and template file related,

```
export AWS_PROFILE=your-account
sceptre validate path/configuration-file
```

Instead, the command for deploying that stack of that configuration file,

```
sceptre launch path/configuration-file
```

When the stack will be created, you can get all information of the stack outputs that they are described in the template Outputs section. By the command below you can get the stack outputs

```
sceptre list outputs path/configuration-file
```

For updating the stack after changes on parameters or templates files,

```
sceptre launch path/configuration-file
```

for deleting the stack,

```
sceptre delete path/configuration-file
```

Below you can find a simple sample (*) of deployment of a Minetest server and a lambda function.

```
cd tool/sceptre
export AWS_PROFILE=your-account
sceptre validate basic/ec2 # an example for validating one configuration file
sceptre validate basic # an example for validating all configuration files of the␣
↪environment named basic
sceptre launch basic # for deploying stacks
sceptre delete basic # for deleting stacks
```

Below you can find an example (*) of deployment of a Minetest server, a lambda function and more.

```
cd tool/sceptre
export AWS_PROFILE=your-account
sceptre launch more
```

Below you can find an example (*) of deployment of a Minetest server, a lambda function and what else you can manage.

```
cd tool/sceptre
export AWS_PROFILE=your-account
sceptre launch plus
```

(*) **Please, pay attention**: in the config.yaml files, there are some identifiers that you need to change before running Sceptre!

5.1.2 By a bash script

An ad hoc script **maybe** it could be useful for a specific CI / CD system. Generally, it is not necessary.

5.1.3 Remember

Sceptre provides two power components:

- hooks, for running your scripts at a particular hook point
- resolvers, for recovering stack outputs or parameters from AWS::SSM, and so on, for your configuration files

So you can avoid to hardcoding a password directly in the property or in the parameter configuration file by

- AWS::SecretsManager
- Custom resolver with AWS::SecretsManager or AWS::SSM

Another fantastic feature is that Sceptre can use a python script like a template, for example using the python package troposphere:

- Sceptre company has shared an example in Sceptre version 1.* where, in the configuration file, on the **template_path** parameter, they use directly a python script
- Sceptre version 1.* has more difference with latest version, so you can find the same example in Sceptre version 2.* here

5.2 Serverless framework

The Serverless framework helps you develop and deploy your AWS Lambda functions, along with the AWS infrastructure resources they require.. yes, this is the first sentence of the introduction of Serverless framework in the official site, and it shows very well strength and weakness: this framework works around to AWS lambda functions, and what it does not manage, is configured as a resource in the cloudformation format (with small exceptions), until you create a plugin to manage it more easily.

The available custom properties of Serverless framework are a lot, and you can manage all that you need by yaml and json files and the conditions system.

The Serverless plugin registry is available on GitHub or by shell with the command below

```
npm install -g serverless # for installing Serverless framework
serverless plugin list # for listing its plugins
```

The available commands of Serverless framework are not many, but sufficient to manage a stack starting from the configuration files.

5.2.1 By shell

The Serverless framework documentation is rich and clean. And it is simple to use Serverless framework: if you do not use plugins, you can use only a few commands.

For deploying your stack,

```
cd serverless-path/
export AWS_PROFILE=your-account
serverless deploy --stage name-of-your-environment
```

For deleting your stack,

```
cd serverless-path/
export AWS_PROFILE=your-account
serverless remove --stage name-of-your-environment
```

Below you can find a simple sample of deployment of a lambda function by serverless.yaml file.

After downloading the code for the lambda, or updating it, and installing the requirements and serverless plugin,

```
cd tool/serverless/
git clone https://github.com/bilardi/aws-saving
cd aws-saving/
pip3 install -r requirements.txt -t .
npm install serverless-python-requirements
```

You can deploy the lambda by Serverless framework,

```
cd tool/serverless/
export AWS_PROFILE=your-account
cp *yaml aws-saving/; cd aws-saving/
serverless deploy --stage only-lambda
```

Below you can find an example of deployment of a lambda function and an EC2 with the installation for Minetest server by serverless.yaml file and other configuration files.

You have to edit the serverless.yaml file for changing subnet and security group before deploying the EC2.

```
cd tool/serverless/
export AWS_PROFILE=your-account
cp *yaml aws-saving/; cd aws-saving/
serverless deploy --stage ec2-basic
```

There is another stage where you can see the security group creation: **ec2-more**, with this configuration, you have to edit serverless.yaml file for changing vpc id and subnet before deploying the EC2.

```
cd tool/serverless/
export AWS_PROFILE=your-account
cp *yaml aws-saving/; cd aws-saving/
serverless deploy --stage ec2-more
```

5.2.2 By a bash script

An ad hoc script is **only necessary** if you need to manage

- more applications in the same repo and you want one stack for each application, so you could cycle their deployment
- where you deploy the stack, the instance needs special precautions

5.2.3 Remember

When you deploy your objects by Serverless,

- you can take advantage of outputs section keys of others stacks
- you can avoid to hardcoding a password directly in the property or in the parameter configuration file by the combo AWS::SecretsManager and AWS::SSM

Coming soon for other tools.

Coming soon # other tool