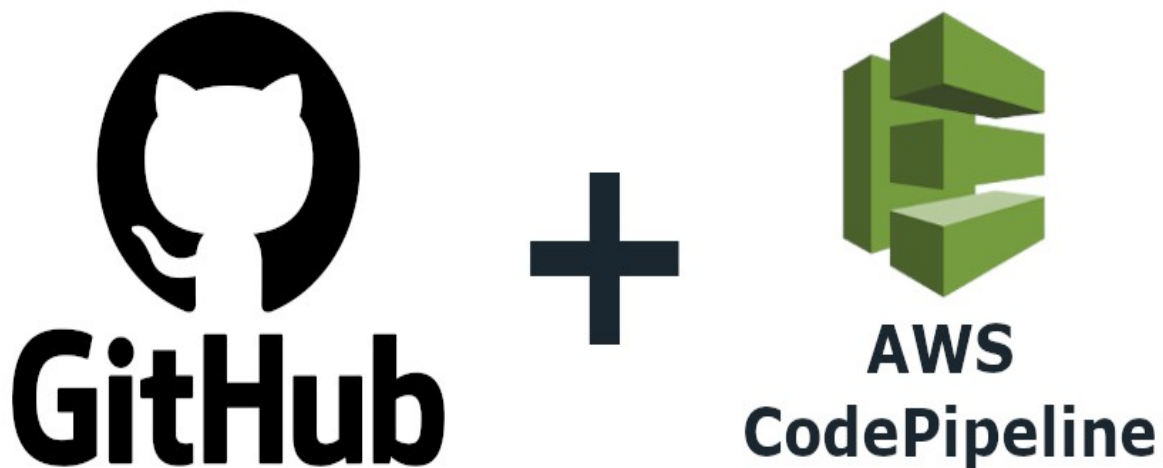


CI/CD with AWS CodePipeline

Design and deployment of a simple cloud solution to test out the main CI/CD tool in AWS



University of Catania – Master's Degree in Data Science

Academic year 2021/2022

Project report for Sistemi Cloud e IoT

Marco Cavalli – 1000024189

Teachers: G. Pappalardo – A. Fornaia

Table of Contents

1 Introduction.....	3
2 Overview of the target architecture.....	3
3 In-depth description of the solution.....	5
3.1 Tools.....	5
3.2 GitHub elements.....	6
3.2.1 Organization.....	6
3.2.2 Repositories.....	6
3.3 Components.....	6
3.3.1 Backend.....	6
3.3.2 Frontend.....	7
3.3.3 AWS CloudFormation Stacks, CI/CD Pipelines and AWS SAM.....	10
3.3.4 Linking resources in different repositories.....	11
4 Deployment.....	11
4.1 Requirements.....	11
4.2 Steps.....	12
5 Conclusions and results.....	14

1 Introduction

Considering the increasing growth of cloud technologies, CI/CD resources are more and more common as they are used more frequently to automate testing and deployment. It is not a surprise to see that many cloud providers (as AWS, Azure etc.) give access to specific services/resources devoted to CI/CD both in testing and production environments.

The goal of the work described in the present document was to test and analyze the integration with AWS CodePipeline (the CI/CD service provided by AWS) and GitHub to support a generic Serverless infrastructure deployed on AWS.

2 Overview of the target architecture

The application is a very simplified eCommerce: a seller (the ADMIN user) can sell their products, while the buyers (normal users) can place orders and manage them on their own account. In order to be able to provide these functionalities – while guaranteeing scalability, reliability and efficiency – the application has been created as a Serverless application. The target application is shown in Figure 1.

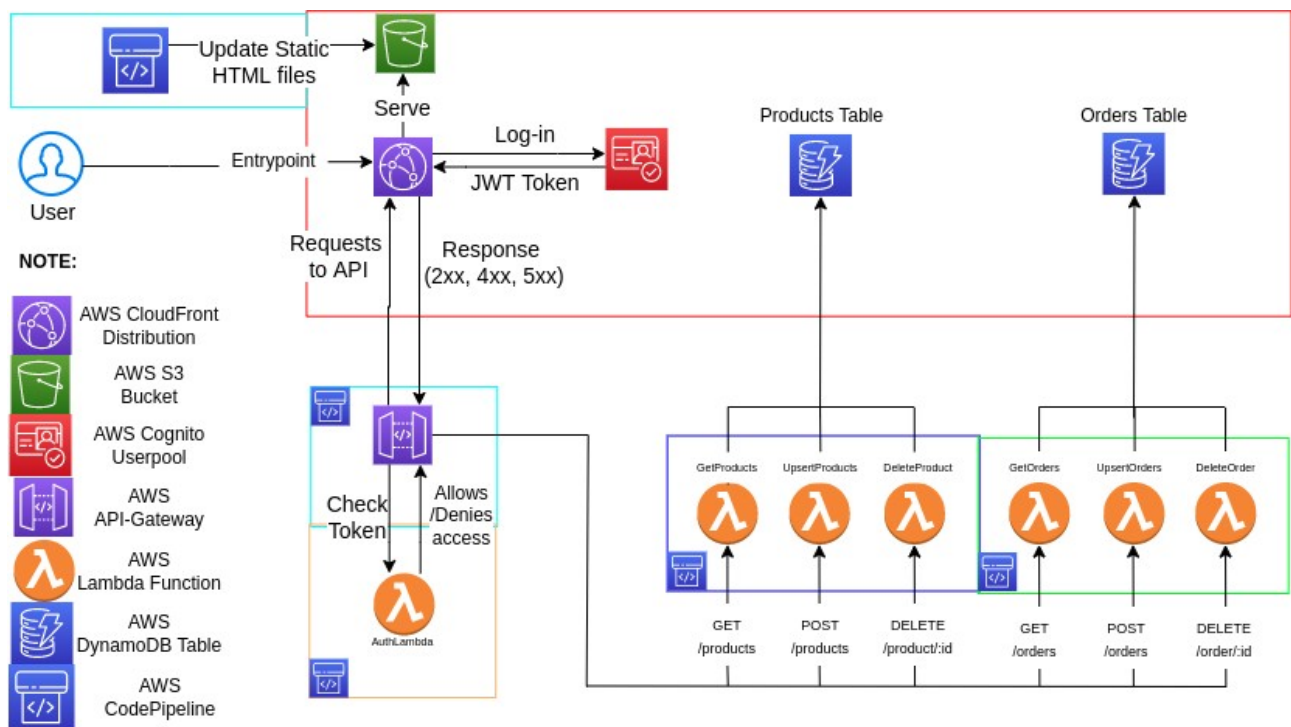


Figure 1: Target architecture

The entrypoint is a AWS CloudFront Distribution that serves the static HTML/CSS files to run the frontend stored in a S3 Bucket (Figure 2). Users are stored inside a AWS Cognito Userpool. Using AWS Cognito Userpools provides these benefits:

- It automatically setups login/signup procedures and other expected functionalities (e.g. “forgot my password”)

- It provides an hosted UI for both login and registration (OAUTH 2)
- It returns JWT tokens that can be used to identify the role of an user (ADMIN vs normal users)

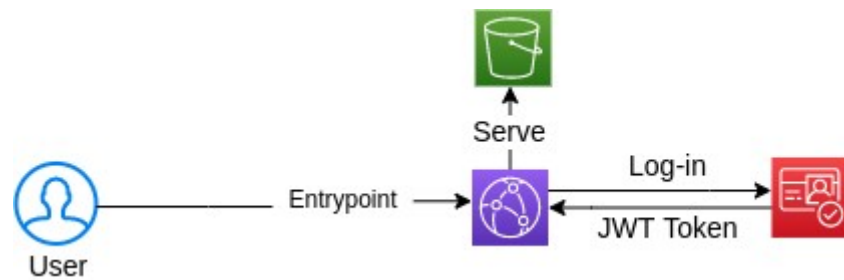


Figure 2: AWS Cloudfront, S3 and Cognito

Once logged in, an user can browse the application and send HTTP requests in order to retrieve content (Products and Orders). The HTTP requests are delivered to the AWS API Gateway. Before proceeding, all requests must be authenticated and validated by a specific AWS Lambda Function called “AuthLambda” (Figure 3). This lambda checks the JWT token provided in the request and it returns a list of AWS IAM Policy Statements to allow or deny the access to endpoint. Missing JWT tokens and denied accesses will result in FORBIDDEN (status code 403) responses sent by default from the API Gateway.

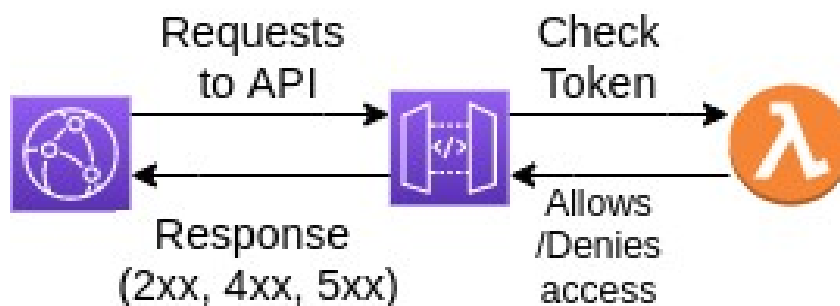


Figure 3: AWS CloudFront, API Gateway and the AuthLambda

For this application, there are 6 supported endpoints:

- GET /products. It returns all products
- UPSERT /product. It allows to create/update a product (ADMIN only)
- DELETE /product/:id. It deletes the product with the given id (ADMIN only)
- GET /orders. It returns all orders to the ADMIN. Normal users get their own orders.
- UPSERT /order. It allows to create an order or change the status of an existing order. Only the ADMIN and the owner can update an order.
- DELETE /order/:id. It allows to delete the order with the given id (ADMIN only). It is not possible to perform this operation from the GUI to prevent errors.

As it is possible to see, the AuthLambda described in Figure 3 has an important role to deny the access to UPSERT/DELETE endpoints to normal users when needed, in order to prevent malicious users to access data of other users.

Each endpoint is served by a dedicated AWS Lambda Function (Figure 4). The names of these lambdas are:

- GetProducts, which replies to GET /products
- UpsertProduct, which replies to POST /product
- DeleteProduct, which replies to DELETE /product/:id
- GetOrders, which replies to GET /orders
- UpsertOrder, which replies to POST /order
- DeleteOrder, which replies to DELETE /order/:id

These six Lambda Functions are evenly split into two microservice: the Products microservice and the Orders microservice. They can interact with the DynamoDB tables to either retrieve/query data, create new records or update/delete the existing ones.

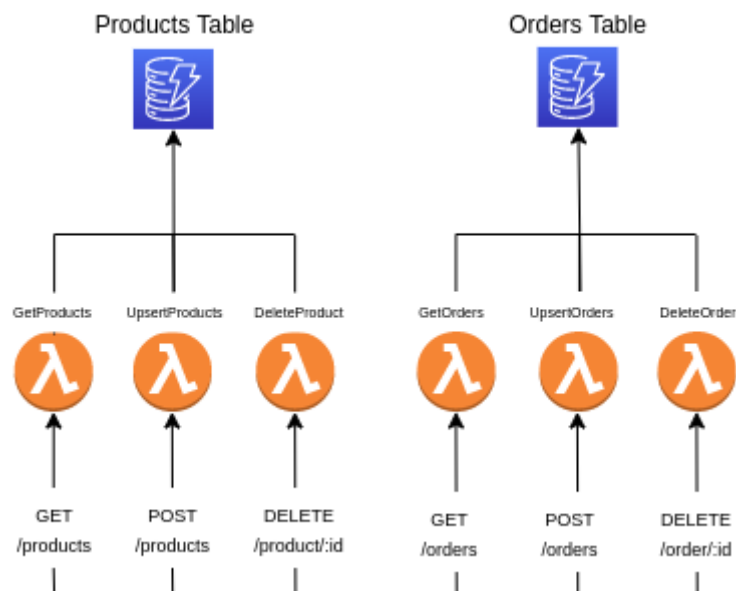


Figure 4: AWS API Gateway Endpoints and Products/Orders Lambdas/DynamoDB Tables

AWS IAM Policies and Statements are set to allow actions on a specific resource to another resource or a group of resources. For instance, the AWS CloudFront Distribution can only read files stored in the specific AWS S3 Bucket provided for this application, while all the AWS Lambda Functions of the Order microservice can access only the Orders table in different ways (read only, delete only etc.).

AWS CodePipeline resources has been deployed as well to support CI/CD to deploy the code (S3 bucket, lambda functions etc.) or update a specific resource (API Gateway). More information about this topic has been given in section 3.3.3.

3 In-depth description of the solution

3.1 Tools

Instead of manually deploying everything, it was decided to rely on automated framework to speed up both testing and deployments. Those were the reasons behind the decision to use [AWS SAM](#). AWS SAM is an open-source framework for building serverless applications. It comes with a CLI version that allows to automate and perform complicated operations using their AWS SAM APIs.

AWS SAM relies on AWS CloudFormation Stacks to deploy AWS resources. It requires to provide a YAML template file containing the configurations and declarations of each resources.

Using AWS SAM allows to automate the deployment inside a Pipeline. It is also possible to use it as a tool to debug deployments because AWS SAM performs version-control and it is able to change/create/delete resources dynamically based on the YAML template provided.

3.2 GitHub elements

3.2.1 Organization

As many different repositories were needed to separate the AWS resources, it was convenient to create a specific GitHub organization to contain them all. The organization is called “[iot-cloud-system-final-assignment-mc](#)”.

3.2.2 Repositories

The number of repositories inside the organization is 7:

1. [.github](#). This is the default repository created inside a GitHub organization. It contains the README with all the steps to deploy the infrastructure. Also it contains this document and two Postman Collections to try out the APIs with Postman.
2. [bootstrap-configuration](#). It contains the YAML template file used to deploy all the resources that are NOT intended to be updated automatically by a CI/CD Pipeline as it would be dangerous in case of errors (e.g. DynamoDB tables with their content). These resources are the following: DynamoDB tables, CloudFront Distribution, S3 Bucket for frontend code and a Cognito Userpool with the ADMIN user.
3. [lambda-products](#). It contains the YAML template file of Lambdas and their Pipeline. Also, it contains the code of the Product microservice.
4. [lambda-orders](#). It contains the YAML template file of Lambdas and their Pipeline. Also, it contains the code of the Order microservice.
5. [auth-lambda](#). It contains the YAML template file of the Lambda and its Pipeline. Also, it contains the code of the AuthLambda, used to check the JWT tokens.
6. [api-gateway](#). It contains the YAML template file of the API-Gateway and its Pipeline.
7. [web-client](#). It contains the YAML template file of the Pipeline and the code to run the frontend and upload it to the S3 Bucket.

It is possible to see that almost every repository has a YAML template with the definition of AWS resources. This is justified by the fact that, for each of them, a AWS CloudFormation stack is deployed using AWS SAM. Also, some of them have a YAML template with the definition of their dedicated CI/CD Pipeline. It is going to be discussed in section 3.3.3, but it is worth to say that Pipelines are deployed in dedicated Stacks.

3.3 Components

3.3.1 Backend

The application is Serverless, so the backend is exclusively made of AWS Lambda Functions supported by a AWS API Gateway. Also, a AWS Cognito Userpool administrates the authentication methods and everything related to users management.

The Userpool is defined in “bootstrap-configuration”. It is a very basic userpool with a registered user (the ADMIN user) and a client (or hosted ui) that exposes the login/signup form. To create the Userpool it is required to pass the ADMIN’s email and the redirect domains for login/logout required by AWS Cognito to perform classic OAUTH2 logins.

As described in section 2, the AWS API Gateway exposes 6 endpoints handled by 6 different AWS Lambda Functions split evenly in two different microservices: Products and Orders. Also a dedicated AWS Lambda Function, called “AuthLambda” is invoked by the API Gateway to check if the requests are authenticated and allow or deny the requests based on the role of the caller (admin user vs normal users).

The AuthLambda returns an array of AWS IAM Statements as the following:

```
{
  "Action": "execute-api:Invoke",
  "Effect": "ALLOW",
  "Resource": "arn:aws:execute-api:eu-west-1:[AWS_ACCOUNT_ID]:
[API_GATEWAY_ID]:/prod/GET/products"
}
```

The AWS API Gateway resource caches the statements of each client for 1 minute to prevent wasting time calling AuthLambda for each request when not needed. To recap: AuthLambda is called only when (1) the client is new/the JWT Token has changed or (2) the cached statements have expired.

All AWS Lambda Functions have been written in Javascript (node 14x). Although most of them require only the AWS SDK (preloaded by default in every AWS Lambda Function), some of them required third parties libraries. For instance, UpsertProduct and UpsertOrder required the “uuid” library to generate unique IDs. Also, all Lambdas required a custom library that was written specifically for this assignment. The custom library provides boilerplate code needed to build valid HTTP Responses (in order to avoid CORS errors). To prevent wasting storage by installing the third-party libraries when not needed and to prevent having multiple copies of the custom library in many different repositories, the following two strategies were followed:

1. Third-Party libraries were included and bundled only in those AWS Lambda Functions which required them, by using the command “sam build” before deploying.

2. The custom library was stored in an AWS Lambda Layer (defined in “bootstrap-configuration”). This Layer was then included in every AWS Lambda Function deployed.

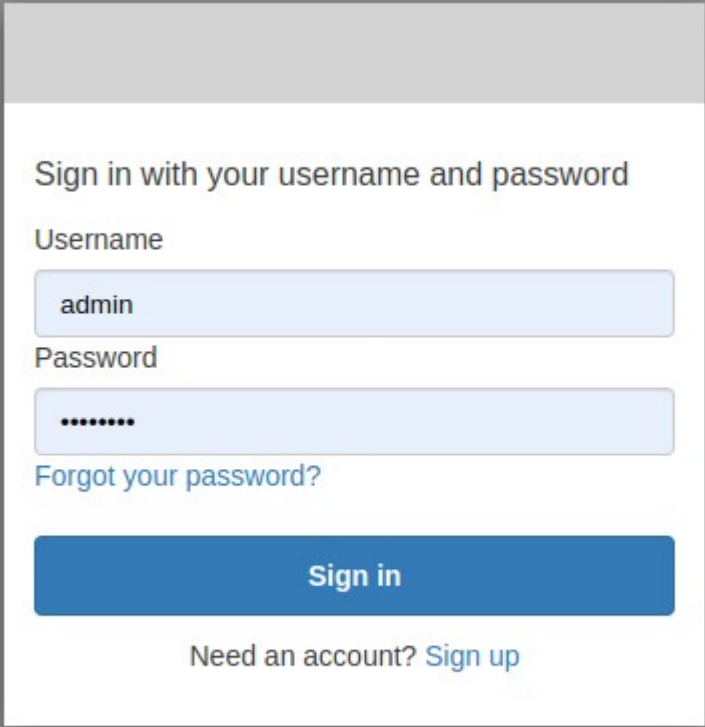
Finally, there are two AWS DynamoDB table used to store data: the Products table and the Orders table. They are simple NoSQL tables with the minimum numbers of columns required to describe a product (id, name, price) and an order (id, product_id, total_price, quantity, status). They can be modified only by the AWS Lambda Functions.

As said in section 2, AWS IAM Policies and Statements have been set up to allow or deny specific actions. Every resources deployed in the backend have their own policies and statements. The policies are managed and created dynamically by AWS SAM.

3.3.2 Frontend

The frontend is a simple REACT app wrote in javascript (node 16). It is a very simple eCommerce-like app where the ADMIN user can create products and update the status of orders (shipped, delivered, canceled), while normal users can buy products and eventually cancel the orders.

As hinted in previous sections, the login/signup workflows are handled by AWS Cognito Userpool’s hosted ui (Figure 5). Basically AWS Cognito exposes a public endpoint that can be set up to handle login/signup for a specific application (classic OAUTH2 behaviour). After the login, the client is redirected back to the homepage of the web-app. The JWT Token is provided as a URL parameter embedded in the redirect URI, which is very convenient as the client can easily retrieve it and store it somewhere (cookies, browser local storage etc.).



The image shows a screenshot of the AWS Cognito Hosted UI login page. The page has a light gray header and a white main content area. The title "Sign in with your username and password" is displayed in a dark gray font. Below the title, there are two input fields: "Username" and "Password". The "Username" field contains the text "admin". The "Password" field contains a series of dots. Below the password field, there is a link "Forgot your password?". At the bottom of the form, there is a blue "Sign in" button. Below the button, there is a link "Need an account? Sign up".

Figure 5: AWS Cognito Hosted UI

There are basically three pages:

1. Home page
2. Products page (Figure 6)
3. Orders page (Figure 7)

The web-app appears differently if the logged user is the ADMIN or if it is a normal user. Also some actions can't be performed by normal users (e.g. only the ADMIN can add a product).

Products App		Products				
		product_id	name	price	updated_at	Actions
Home		c7e2c504-ef52-490b-abb7-e4994cb78192	Chair	10	9/26/2022, 1:04:38 PM	+
Products	^	e951e34b-abb2-45f2-add1-1e372cb9575d	Bike	15	9/26/2022, 1:04:31 PM	+
Table						
+ Add						
All orders						

Figure 6: The view of the ADMIN in "Products" page

In the Products page it is possible to see all the available products that the ADMIN has added inside the store. The ADMIN can also create a new product, while normal users can create a order by clicking on the "+" icon of a product.

Products App

Home

Products

All orders

Orders

order_id	product_id	username	quantity	total_price	status	updated_at	Actions
e6d6d603-19f3-49f0-8700-d084ad631964	c7e2c504-ef52-490b-abb7-e4994cb78192	marco	3	30	pending	9/26/2022, 1:06:17 PM	
86824f49-0817-4122-9fe3-8223a7f0ca3a	c7e2c504-ef52-490b-abb7-e4994cb78192	luca	2	20	cancelled	9/26/2022, 1:50:03 PM	+
b2622ccd-7033-475b-b23d-6edd7afe0122	c7e2c504-ef52-490b-abb7-e4994cb78192	luca	4	40	shipped	9/26/2022, 1:50:23 PM	

Figure 7: The view of the ADMIN in "Orders" page

In Orders page it is possible to see all the orders. The ADMIN will see the orders of every user. The normal user can only see their own orders. It is also possible to change the status of an order by clicking the icons in the "Actions" column. The possible statuses are:

1. PENDING. It is the default status when an order has been created
2. SHIPPED. It can be set by the ADMIN clicking on the "send" icon

3. **DELIVERED.** The product has been received by the customer and the ADMIN set this status clicking on the “ok” button
4. **CANCELED.** Either the ADMIN or the normal user can cancel the order by clicking on the “trash” icon. Normal users can only cancel an order that is still in the “PENDING” status

The “+” icon that appears when an order is either “cancelled” or “delivered” allows to create a new order with the same product, quantity and total_price.

3.3.3 AWS CloudFormation Stacks, CI/CD Pipelines and AWS SAM

The goal of this work was to use AWS CodePipeline as the CI/CD service to automatically update both the code of the frontend/backend and to change/adapt the AWS Resources when a repository is updated after a “git push/merge” command.

There are actually many ways to use AWS CodePipeline to provide such functionality. In this case, the most important thing was to find a way to write as much less code as possible and that’s why AWS SAM came out as the solution because it relies on AWS CloudFormation Stacks.

Basically, each Pipeline is declared and defined in a dedicated YAML template and it is managed by AWS SAM. This ensure to have an automated method to deploy and update the Pipelines, while still having some flexibility and the possibility to change and adapt the behavior of a Pipeline if needed.

Every Pipeline is deployed as a Stack having the following resources:

1. AWS CodePipeline
2. AWS CodeBuild
3. AWS IAM Policies and Statements for AWS CodePipeline
4. AWS IAM Policies and Statements for AWS CodeBuild

The Pipeline is divided in two stages:

1. **Source.** This stage is triggered by a “push/merge” command on the “main” branch of the linked repository. It simply clones the “main” branch and provides it to the next stage.
2. **Build.** This stage is divided in multiple sub-stages:
 1. **INSTALL.** It set up a simple EC2 machine having specific configurations. For this project, they are generally Ubuntu machines with node14x installed.
 2. **PRE_BUILD (Optional).** In this stage it runs “npm install” or perform any operation needed before building the code.
 3. **BUILD (Optional).** In this stage it runs “npm build” (web-app) or “sam build” (Lambdas).
 4. **POST_BUILD.** In this stage it deploys the code/resources by either writing on the s3 (web-app) or using “sam deploy” (Lambdas, api-gateway).

Unfortunately, it is not possible to setup the Pipeline to be triggered only when a specific action happens in a specific path of the repository when the Source is GITHUB. This is very inconvenient,

as the Pipeline is triggered even if it shouldn't (for instance, if the README.md file has been updated).

As shown in Figure 1, there are exactly 5 Pipelines:

1. One is used to update the code of the Frontend. The YAML template is stored inside the “web-app” repository.
2. One is used to update the API Gateway resource.
3. One is used to update the AuthLambda.
4. One is used to update the Lambdas of the Product microservice.
5. One is used to update the Lambdas of the Order microservice.

Pipelines are intended to be deployed manually using AWS SAM. This means that, if a Pipeline has to be updated, the developers have to run the “aws deploy” command.

3.3.4 Linking resources in different repositories

The tricky part of having multiple repositories is to find a way to link the resources without being forced to hard-code the name or the ARN (identifier of a resource in AWS) of every resource.

This has been treated relying on AWS SAM as it uses AWS CloudFormation Stacks. Basically, Stacks can expose output values containing the needed information as the URL of the CloudFront Distribution or the ARN of the API Gateway. So the stacks of the resources has to be created in this specific order (the name of the repository is used to identify each stack):

1. bootstrap-configuration
2. auth-lambda
3. products-lambda
4. orders-lambda
5. api-gateway
6. web-client

This order takes count of the dependencies and it is able to solve most of the problems, allowing to create each stack with almost everything they need in terms of information (ARNs, URLs, values etc.).

For instance, Products and Orders microservices depend on the existence of the Products and Orders tables defined in bootstrap-configuration, while the API Gateway depend on the creation of every Lambda (as it is required to create the endpoints); finally, the web-client can't be created if the API Gateway has not been created, as it needs the API Gateway base domain to deliver the HTTP requests.

This logic is clearer when looking at the script “deploy_everything.sh” inside “bootstrap-configuration”. In that script it is possible to note that for each different Stack there are some values collected from other Stacks by accessing their output values.

4 Deployment

4.1 Requirements

To deploy the whole infrastructure it is required to:

1. Have installed AWS CLI
2. Have installed AWS SAM
3. Have obtained a GitHub OAUTH Token with all repos policies

4.2 Steps

It is possible to deploy this infrastructure by using the script “deploy_everything.sh” (**recommended**) inside “bootstrap-configuration” ([here the guide](#)) or following these steps:

1. **Open a terminal and set the env variables:**
ADMIN_EMAIL=xx
GITHUB_OAUTH_TOKEN=xx
AWS_REGION=xx
2. **Create a S3 bucket to store AWS SAM artifacts:**
SAM_S3_BUCKET="iot-cloud-systems-sam-\$(date +%s)"
aws s3 mb s3://\${SAM_S3_BUCKET} --region \${AWS_REGION}
3. **Setup the folders cloning all the repositories:**
mkdir products-app
cd products-app
git clone <https://github.com/iot-cloud-system-final-assignment-mc/bootstrap-configuration.git>
git clone <https://github.com/iot-cloud-system-final-assignment-mc/lambda-products.git>
git clone <https://github.com/iot-cloud-system-final-assignment-mc/lambda-orders.git>
git clone <https://github.com/iot-cloud-system-final-assignment-mc/auth-lambda.git>
git clone <https://github.com/iot-cloud-system-final-assignment-mc/api-gateway.git>
git clone <https://github.com/iot-cloud-system-final-assignment-mc/web-client.git>
4. **Deploy “bootstrap-configuration” stack:**
cd bootstrap-configuration
sam deploy -t template.yml --stack-name bootstrap-resources --s3-bucket
\$SAM_S3_BUCKET --s3-prefix bootstrap-resources --region \$AWS_REGION --
capabilities CAPABILITY_AUTO_EXPAND --parameter-overrides
AdminEmailParameter=\$ADMIN_EMAIL
cd ..
5. **Setup the env variable for the LAYER resource (needed for lambdas stacks):**
LAYER_ARN=\$(aws cloudformation list-exports --region \${AWS_REGION} --query
"Exports[?Name=='Cloud-Systems-IoT-HttpUtilsLayerArn'].Value" --output text)
6. **Deploy the remaining stacks:**
cd lambda-products

```
cd code && npm install && cd ..
sam build
sam deploy -t ./aws-sam/build/template.yaml --stack-name lambda-products --s3-bucket
$SAM_S3_BUCKET --s3-prefix lambda-products --region $AWS_REGION --capabilities
CAPABILITY_NAMED_IAM --parameter-overrides UtilsLayerArn=$LAYER_ARN
cd ..
```

```
cd lambda-orders
cd code && npm install && cd ..
sam build
sam deploy -t ./aws-sam/build/template.yaml --stack-name lambda-orders --s3-bucket
$SAM_S3_BUCKET --s3-prefix lambda-orders --region $AWS_REGION --capabilities
CAPABILITY_NAMED_IAM --parameter-overrides UtilsLayerArn=$LAYER_ARN
cd ..
```

```
cd auth-lambda
cd code && npm install && cd ..
sam build
sam deploy -t ./aws-sam/build/template.yaml --stack-name auth-lambda --s3-bucket
$SAM_S3_BUCKET --s3-prefix auth-lambda --region $AWS_REGION --capabilities
CAPABILITY_NAMED_IAM --parameter-overrides UtilsLayerArn=$LAYER_ARN
cd ..
```

```
cd api-gateway
sam deploy -t template.yml --stack-name api-gateway --s3-bucket $SAM_S3_BUCKET --
s3-prefix api-gateway --region $AWS_REGION --capabilities
CAPABILITY_NAMED_IAM
cd ..
```

7. Deploy the pipelines:

```
DEPLOY_BUCKET=$(aws cloudformation list-exports --region ${AWS_REGION} --
query "Exports[?Name=='Cloud-Systems-IoT-ApplicationSiteBucket'].Value" --output text)
cd web-client
sam deploy -t pipeline-template.yml --stack-name web-client-pipeline --s3-bucket
$SAM_S3_BUCKET --s3-prefix web-client --region $AWS_REGION --capabilities
CAPABILITY_NAMED_IAM --parameter-overrides
GithubOAuthToken=$GITHUB_OAUTH_TOKEN DeployBucket=$DEPLOY_BUCKET
cd ..
```

```
cd lambda-products
sam deploy -t pipeline-template.yml --stack-name lambda-products-pipeline --s3-bucket
$SAM_S3_BUCKET --s3-prefix lambda-products-pipeline --region $AWS_REGION --
capabilities CAPABILITY_NAMED_IAM --parameter-overrides
GithubOAuthToken=$GITHUB_OAUTH_TOKEN s3SamBucket=$SAM_S3_BUCKET
cd ..
```

```
cd lambda-orders
sam deploy -t pipeline-template.yml --stack-name lambda-orders-pipeline --s3-bucket
$SAM_S3_BUCKET --s3-prefix lambda-orders-pipeline --region $AWS_REGION --
capabilities CAPABILITY_NAMED_IAM --parameter-overrides
GithubOAuthToken=$GITHUB_OAUTH_TOKEN s3SamBucket=$SAM_S3_BUCKET
cd ..
```

```
cd auth-lambda
sam deploy -t pipeline-template.yml --stack-name auth-lambda-pipeline --s3-bucket
$SAM_S3_BUCKET --s3-prefix auth-lambda-pipeline --region $AWS_REGION --
capabilities CAPABILITY_NAMED_IAM --parameter-overrides
GithubOAuthToken=$GITHUB_OAUTH_TOKEN s3SamBucket=$SAM_S3_BUCKET
cd ..
```

```
cd api-gateway
sam deploy -t pipeline-template.yml --stack-name api-gateway-pipeline --s3-bucket
$SAM_S3_BUCKET --s3-prefix api-gateway-pipeline --region $AWS_REGION --
capabilities CAPABILITY_NAMED_IAM --parameter-overrides
GithubOAuthToken=$GITHUB_OAUTH_TOKEN s3SamBucket=$SAM_S3_BUCKET
cd ..
```

8. Retrieve the URL of the website:

```
aws cloudformation list-exports --region ${AWS_REGION} --query "Exports[?
Name=='Cloud-Systems-IoT-ApplicationSite'].Value" --output text
```

From this point, every change in the code or the YAML templates in a repository will trigger the dedicated CI/CD Pipeline which will update either the resources, their code or both. As stated in section 3.3.3 all the resources defined in “bootstrap-configuration” won’t be automatically updated from pipelines. So, to update those resources, it is **required** a human to manually do it. This is possible by running STEP 4 after updating the code of the AWS Lambda Layer or the YAML template.

5 Conclusions and results

Being a Serverless application, the solution described scales well both horizontally and vertically. It scales horizontally, because there is a dedicated handler for each HTTP request to the API-Gateway or AWS Cognito (login), so it can serve multiple requests at the same time. It also scales vertically, because there are not real limits in both the storage (S3 is limitless or almost) and computational power (as Lambdas work individually).

The introduction of AWS SAM allowed to reduce the code needed to automate the deploy and the CI/CD Pipelines as it groups resources inside AWS CloudFormation Stacks. These Stacks can be defined providing specific YAML templates.

AWS CodePipelines are very powerful, but they don’t allow to set a specific path when the source is not AWS CodeCommit. This particular problem doesn’t effect the infrastructure from a performance-perspective, but it does from a cost-perspective: every Pipeline can be triggered even if the changes are not in the code, but in other part of the repository (as the README.md file). A

possible solution would be to keep the YAML template of a Pipeline in another repository. This is not always convenient because the BuildSpec (the script run inside CodeBuild) is strongly dependent on the resources that are going to be updated by the Pipeline. Also it would mean to duplicate the number of repositories. In this case, it made more sense to keep the YAML template of the Pipeline and the code/YAML template of the resources in the same repository.