REPUBLIQUE DU CAMEROUN

*****

PAIX-TRAVAIL-PATRIE

*****

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR

*****

INSTITUT UNIVERSITAIRE SAINT JEAN

REPUBLIC OF CAMEROON

*****

PEACE-WORK-FATHERLAND

*****

MINISTRY OF HIGHER EDUCATION

*****

SAINT JEAN INSTITUTE

INSTITUT UNIVERSITAIRE SAINT JEAN
SAINT JEAN INGÉNIEUR

SIMULATION D'UN CLOUD LOCAL AVEC LOCALSTACK

Rédigé et présenté par :

- **DOMBANG TOGNA** Mannuel Hugor
- **FOKOU SOH** Cédric
- **KITIO NGUETSE** Audrey Karelle ***(Cheffe de projet)***

Etudiants ingénieurs en **quatrième année système réseau et télécommunication**.

Sous l'encadrement de : Inge. ***TAMKO*** *Clarence.*

ANNÉE ACADÉMIQUE

2025-2026

SOMMAIRE :

TABLE DES FIGURES

INTRODUCTION :

En environnement réel, les services cloud AWS sont couteux, dépendant d'internet et risqués pour l'apprentissage à cause des erreurs et des facturations. La problématique centrale est donc de comprendre comment manipuler et enchainer les principaux services AWS dans un environnement local, gratuit, sécurisé, et reproductible, sans utiliser le cloud réel. Pour ce faire, nous simulerons un Cloud AWS complet en local, reproduirons les workflows cloud réels, et utiliserons les outils standards d'AWS.

I. Définitions :

- **AWS :** est une plateforme de cloud public fournissant des services de stockage (S3), de calcul (lambda), de base de données (DynamoDB), d'API (API Gateway) et autres, à la demande et facturer.

- **LocalStack :** est un simulateur local des services AWS, consistant à reproduire le comportement de ces services sans connexion internet et sans facturation réelle.

  Il nous permettra de :

  - Simuler S3, Lambda, API Gateway, DynamoDB, IAM
  - Utiliser la console AWS CLI (outil en interface de commande nous permettant de créer, configurer et manipuler les services cloud, en communiquant avec LocalStack via un endpoint local)
  - Travailler sans compte AWS réel.

II. Installation et lancement de LocalStack via Docker :

1. Installation de LocalStack :

Pour installer LocalStack, on utilisera la commande :

sudo docker run --name localstack -d -p 4566:4566 -p 4510-4559:4510-4559 -v /var/run/docker.sock:/var/run/docker.sock -v localstack-data:/var/lib/localstack localstack/localstack

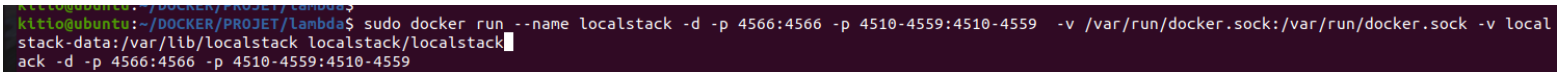


Figure 1: installation de localstack et tous les services

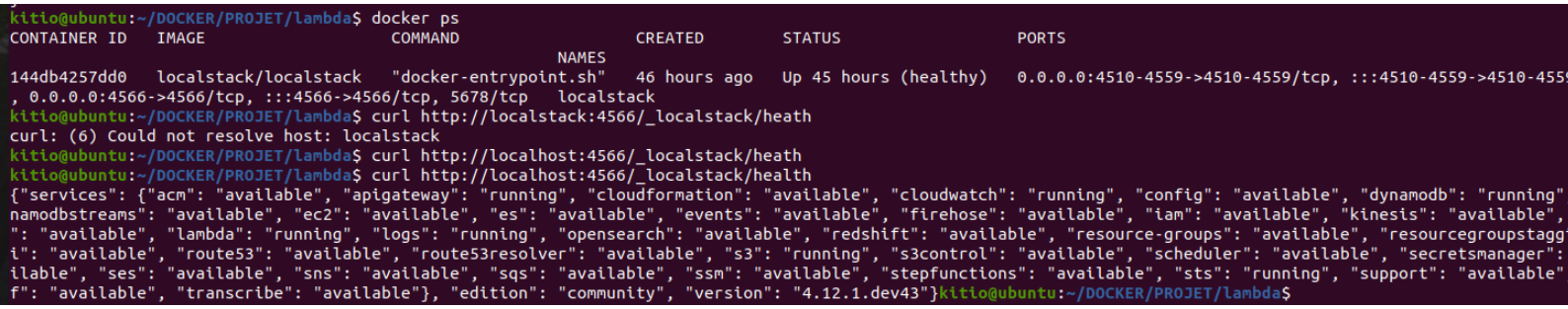


Figure 2: vérification de l'état des services et de l'état de localstack

Elle va créer et lancer un conteneur de manière détachée (-d) à partir de l'image localstack/localstack sachant que :

- **--name localstack** nommera ce conteneur : localstack
- **-p 4566:4566** rendra le port 4566 du conteneur accessible sur le port 4566 de la machine physique. Il est le port principal de LocalStack, par lequel tous les API AWS y sont accessibles.
- **-p 4510-4559:4510-4559** est une plage de ports officiel utilisées pour l'exécution des lambdas (car l'exécution des lambdas est dynamique te nécessite des ports temporaires) et les services internes de LocalStack.
- **-v /var/run/docker.sock:/var/run/docker.sock** volume qui donne à lambda un accès au socket de docker lui permettant e pouvoir le manipuler. C'est nécessaire car l'exécution des lambdas créée des conteneurs en arrière-plan ce qui nécessite un droit d'utilisation a Docker.
- **-v localstack-data:/var/lib/localstack**: volume de données pour sauvegarder les progressions

NB : Lorsque le conteneur a été lancé, on remarque un port en plus (5678) exposé, c'est un port interne exposé par défaut par l'image localstack dans le but de faire des debugs et de la supervision internes.

2. Installation et configuration de AWS CLI :

AWS CLI est un outil officiel pour interagir par défaut avec le cloud AWS réel (**https://aws.amazon.com**). Pour l'utiliser dans notre contexte, on va rediriger AWS CLI vers LocalStack (**http://localhost:4566**), et cela via un endpoint (**--endpoint-url=http://localhost:4566**).

Pour installer AWS CLI on fera :

sudo apt update *(pour avoir une liste de package à jour)*

sudo apt install awscli –y

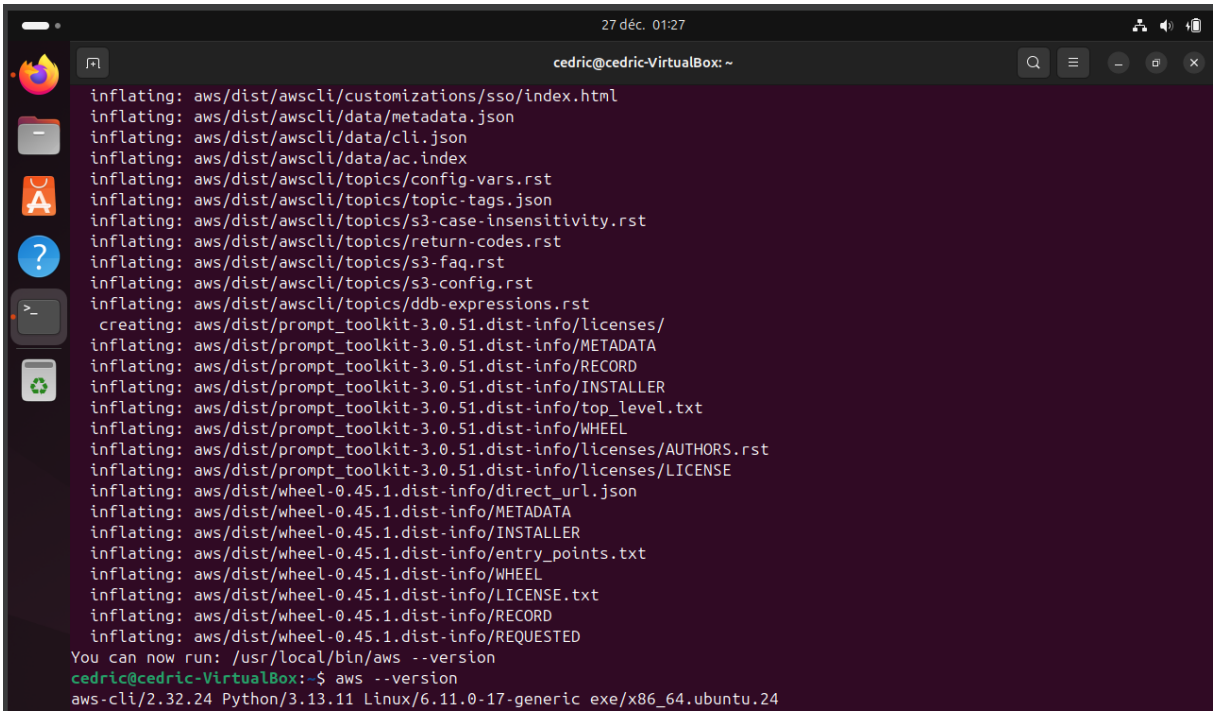


Figure 3: installation de AWS CLI

Sur AWS réel il faut savoir que des identifiants sont utilisés pour chaque utilisateurs dans le but de sécuriser l'accès aux ressources. Dans LocalStack il n'y a pas de sécurité réelle mais AWS CLI exige ces identifiants. Alors nous en créerons des fictifs en configurant comme suit :

- **aws configure** (pour lancer la configuration) elle créée les *~/.aws/credentials* pour les identifiants et *~/.aws/config* pour les configurations
- **AWS Access Key ID [None]: test** est notre identifiant
- **AWS Secret Access Key [None]: test** password
- **Default region name [None]: us-east-1** qui est la région AWS la plus couramment utilisée car plus ancienne et les services y sont disponibles en premier.

  NB : La region ici est l'endroit où les ressources sont créées par défaut.

- **Default output format [None]: json** format de sortie des résultats des commandes AWS CLI. json est choisi car est le format natif des API AWS et est facile à parser (analyser)

```
cedric@cedric-VirtualBox:~$ aws --version

aws-cli/2.32.24 Python/3.13.11 Linux/6.11.0-17-generic exe/x86_64.ubuntu.24
cedric@cedric-VirtualBox:~$
cedric@cedric-VirtualBox:~$ aws configure
AWS Access Key ID [None]: test

Default region name [None]: us-east-1
Default output format [None]: json
cedric@cedric-VirtualBox:~$ aws configure
AWS Access Key ID [****************test]: est
AWS Secret Access Key [None]: test
Default region name [us-east-1]:
Default output format [json]:
cedric@cedric-VirtualBox:~$ aws configure
AWS Access Key ID [****************est]: test
AWS Secret Access Key [****************test]:
Default region name [us-east-1]:
Default output format [json]:
cedric@cedric-VirtualBox:~$ aws configure
AWS Access Key ID [****************test]:
AWS Secret Access Key [****************test]:
Default region name [us-east-1]:
Default output format [json]:
```

Figure 4: configuration de AWS CLI

III. Présentations des services :

1. Amazon S3 (Simple Storage Service) :

Est un service de stockage d'objets (Type de stockage idéal pour les données massives), permettant de stocker des images, des documents, des vidéos, des sauvegardes, des logs et même d'héberger des sites statiques.

Nous ferons donc :

a. Creation d'un bucket (conteneur logique dans lequel on stocke les objets) :

aws --endpoint-url=http://localhost:4566 s3 mb s3://tp-bucket

- **aws** : appel de l'outils AWS CLI
- **endpoint-url=http://localhost:4566** : redirection de la commande qui suit vers localstack
- **s3** : indication du service utiliser
- **mb** : make bucket pour le créer
- **s3://** : c'est le protocole logique S3 (un peu comme http://)
- **test01** : nom du bucket

```
cedric@cedric-VirtualBox:~$ aws --endpoint-url=http://localhost:4566 s3 mb s3://tp-bucket
make_bucket: tp-bucket
cedric@cedric-VirtualBox:~$ aws --endpoint-url=http://localhost:4566 s3 ls
2025-12-27 01:25:39 tp-bucket
```

Figure 5: création d'un bucket tp-bucket dans S3

b. Envoie de fichiers :

doc.txt est le fichier source et la destination sera le bucket tp-bucket d'objet test.txt.

aws --endpoint-url=http://localhost:4566 s3 cp test.txt s3://tp-bucket/

Ici, le fichier est copié dans le bucket via cp et ensuite devient un objet S3 ayant pour contenu le fichier et pour clé le nom de de l'objet test.txt).

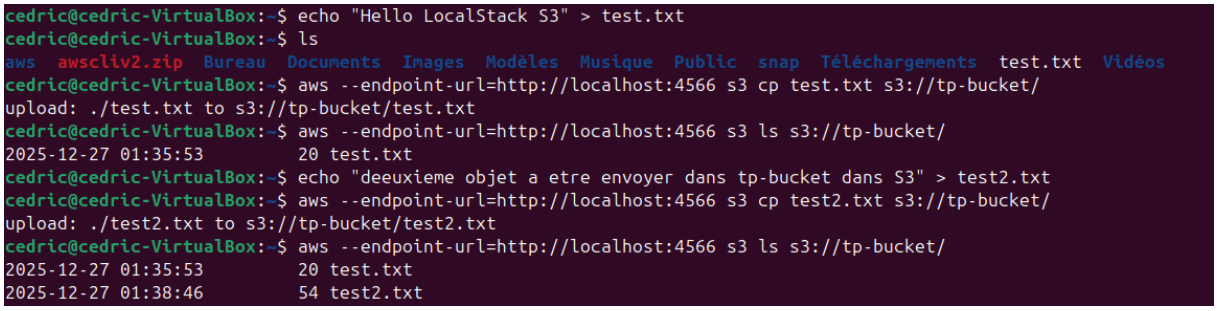


Figure 6: envoi de deux fichiers dans notre bucket

c. Téléchargement de fichiers :

Ici la source sera l'objet S3 et la destination sera le fichier de destination ./downloaded_test.txt dans notre cas :

aws --endpoint-url=http://localhost:4566 s3 cp s3://tp-bucket/test.txt ./downloaded_test.txt

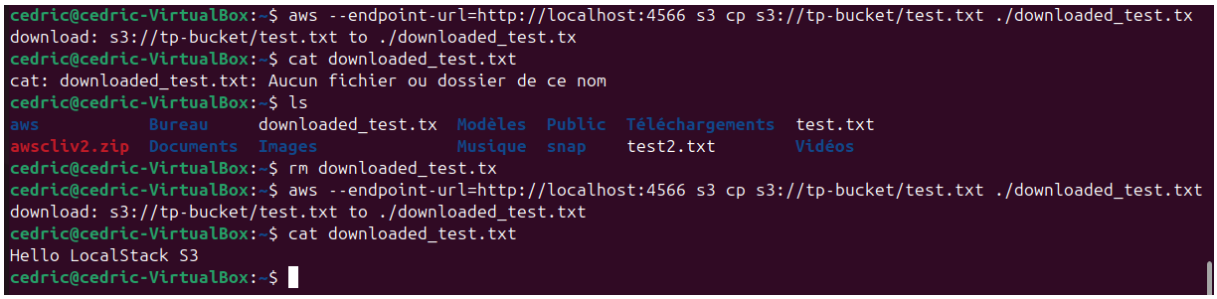


Figure 7: téléchargement d'un objet depuis notre bucket

d. Commandes de vérification :

- Vérifier la liste des buckets : **aws --endpoint-url=http://localhost:4566 s3 ls**
- Vérifier le contenu d'un bucket : **aws --endpoint-url=http://localhost:4566 s3 ls s3://tp-bucket**

NB : bucket est un peu comme dossier principal et un objet est un fichier de ce dossier

e. Concepts clés :

- **bucket** : conteneur logique qui stocke des objet S3.
- **Objet** : est un fichier stocké dans un bucket (il contient des données et métadonnées)
- **Clé** : chemin unique d'un objet dans un bucket. Exemple : projet/rapport.pdf
- **Stockage d'objets :** mode de stockage base sur des objets indépendants accessibles par API.
- **namespace global (simulé)** : dans aws réel chaque bucket a un nom unique mondialement cela permet son isolation.

2. DynamoDB :

Est une base de données NoSQL entièrement gérée, rapide, flexible et sans schéma fixe, permettant de gérer des utilisateurs et utiliser dans l'IoT et la Big Data pour traiter d'énormes volumes de données.

Nous ferons donc :

a. La création des tables :

Lors la création d'une table DynamoDB, on définit **le nom de la table, la clé primaire et le type de clé**. Pour ce faire, on utilisera la commande :

aws --endpoint-url=http://localhost:4566 dynamodb create-table --table-name Utilisateurs --attribute-definitions AttributeName=id,AttributeType=S --key-schema AttributeName=id,KeyType=HASH --billing-mode PAY-PER-REQUEST

- **dynamodb** : indique que l'on utilise DynamoDB.
- **create-table** : commande de création d'une table.
- **table-name Utilisateurs** : nom de la table DynamoDB.
- **attribute-definitions** : déclare les attributs utilisés dans la clé primaire.
- **AttributeName=id**

- **AttributeType=S** : S signifie String (chaîne de caractères).
- **--key-schema** : définit la clé primaire.
- **KeyType=HASH** : HASH = clé de partition (obligatoire).
- **--billing-mode PAY_PER_REQUEST** : mode de facturation. Sur AWS réel, ce mode facture à l'usage, ici, il est requis syntaxiquement.

```
cedric@cedric-VirtualBox:~$ aws --endpoint-url=http://localhost:4566 dynamodb create-table \
  --table-name Utilisateurs \
  --attribute-definitions AttributeName=id,AttributeType=S \
  --key-schema AttributeName=id,KeyType=HASH \
  --billing-mode PAY_PER_REQUEST
{
    "TableDescription": {
        "AttributeDefinitions": [
            {
                "AttributeName": "id",
                "AttributeType": "S"
            }
        ],
        "TableName": "Utilisateurs",
        "KeySchema": [
            {
                "AttributeName": "id",
                "KeyType": "HASH"
            }
        ],
        "TableStatus": "ACTIVE",
        "CreationDateTime": "2025-12-28T01:01:39.527000+01:00",
        "ProvisionedThroughput": {
```

Figure 8: création d'une table DynamoDB

```
cedric@cedric-VirtualBox:~$ aws --endpoint-url=http://localhost:4566 dynamodb list-tables
{
    "TableNames": [
        "Utilisateurs"
    ]
}
```

Figure 9: vérification de la création de la table

b. Insérer des données :

Ici on va ajouter un item utilisateur à la table. La commande est la suivante :

Aws --endpoint-url=http://localhost:4566 dynamodb put-item --table-name Utilisateurs --item '{"id": {"S": "1"},"nom": {"S": "<un nom>"}}'

- **"id"** : clé primaire.
- **"S"**: type String.
- **"N"** : type Number.

```
cedric@cedric-VirtualBox:~$ aws --endpoint-url=http://localhost:4566 dynamodb put-item \
  --table-name Utilisateurs \
  --item '{"id":{"S":"1"},"nom":{"S":"Cedric"}}'
cedric@cedric-VirtualBox:~$ aws --endpoint-url=http://localhost:4566 dynamodb scan --table-name Utilisateurs
{
    "Items": [
        {
            "nom": {
                "S": "Cedric"
            },
            "id": {
                "S": "1"
            }
        }
    ],
    "Count": 1,
    "ScannedCount": 1,
    "ConsumedCapacity": null
}
```

Figure 10: ajout d'un item et vérification

c. Lire des données :

On récupère une item à partir de sa clé primaire comme ceci :

aws --endpoint-url=http://localhost:4566 dynamodb get-item --table-name Utilisateurs --key '{"id": {"S": "1"}}'

d. Commande de vérification :

- Vérifier les tables : **aws --endpoint-url=http://localhost:4566 dynamodb list-tables**

e. Concepts clés :

- **table** : est une collection d'éléments stockes dans dynamodb
- **Partition Key** : c'est la clé principale qui identifie de façon unique un item elle détermine aussi où sont stockées les données. Exemple : user_id ="144".
- **item** : ligne dans dynamodb, équivalent d'un enregistrement (donnée).
- **attribut** : propriétés d'un item, équivalent d'une colonne.
- **haute disponibilité** : signifie que les services restent accessibles même en cas de panne.
- **Dans aws réel** : les data sont répliquées sur plusieurs serveurs, pas de point unique de panne.

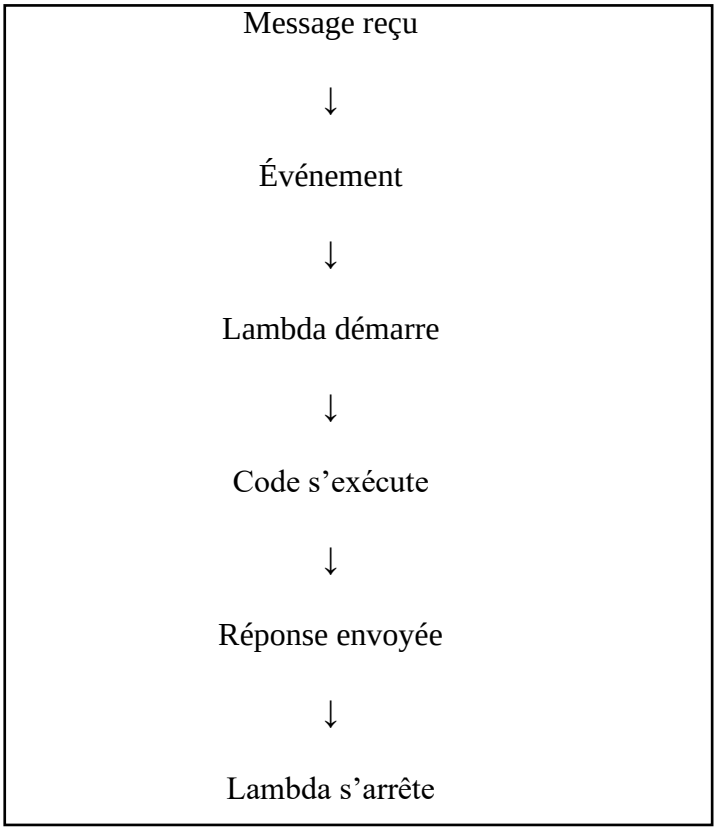
- **Le stockage objet** : est un système de stockage où les données sont enregistrées sous forme d'objets indépendants. Ses caractéristiques : pas de relation entre les objets, très scalable, accessible via api.
- **Une base de données** : stocke des données structurées, consultables via des requêtes. Exemple : SQl, NoSql

➢ **NoSQL** : bd non relationnelle conçu pour la performance la scalabilité et la distribution. De plus, il n'y a pas de fonction comme SELECT ou JOIN et on accède aux données par clé.

3. AWS Lambda :

Est un service (fonction) qui peut s'exécuter sans gérer de serveurs, permettant de déclencher des évents (automatisation), traiter des requêtes API et gérer des microservices.

Lambda ne s'exécute jamais seule, mais uniquement quand un **évènement (requêtes http via API Gateway, ajout de fichier dans S3, données écrites dans dynamoDB)** se produit. Selon ce schéma :



Ces fonctions possèdent toujours:

- ***Une runtime*** (Python, Node.js,etc..) : c'est l'environnement d'exécution de notre code
- ***Un handler*** : qui est le point d'entrée du code c'est à dire la fonction qu'AWS va appeler quand notre lambda est déclenchée sous le format *nomfichier.nomfoncion*
- ***Un événement d'entrée*** : est un objet JSON reçu par notre fonction lambda contenant les informations sur ce qui a déclenché la fonction lambda
- ***une réponse*** : c'est ce que lambda envoi après avoir traité l'évènement

Nous ferons donc:

a. Créer un dossier :

mkdir lambda && cd lambda

b. Créer lambda_function.py :

```
def handler(event, context):
return {"statuscode": 200, "body": "Hello World from Lambda!"}
```

c. Zipper :

zip lambda_function.zip lambda_function.py

d. Déployer la Lambda :

aws --endpoint-url=http://localhost:4566 lambda create-function --function-name InfoLambda --runtime python3.9 --handler lambda_function.handler --zip-file fileb://lambda_function.zip --role arn:aws:iam::000000000000:role/lambda-role

- **function-name InfoLambda** : nom de notre fonction lambda
- **--runtime python3.9** : version de python utilisée

- **--handler lambda_function.handler** : notre point d'entrée du code
- **--zip-file fileb://lambda.zip** : c'est le code de la fonction lambda qui est appelé
- **--role arn:aws:iam::000000000000:role/lambda-role** : est une attribution de rôle factice mais syntaxiquement obligatoire dans LocalStack (c'est un peu comme une permission de lecture ou écriture accorder à la fonction lambda). IAM est simulé donc on peut mettre n'importe quel rôle.

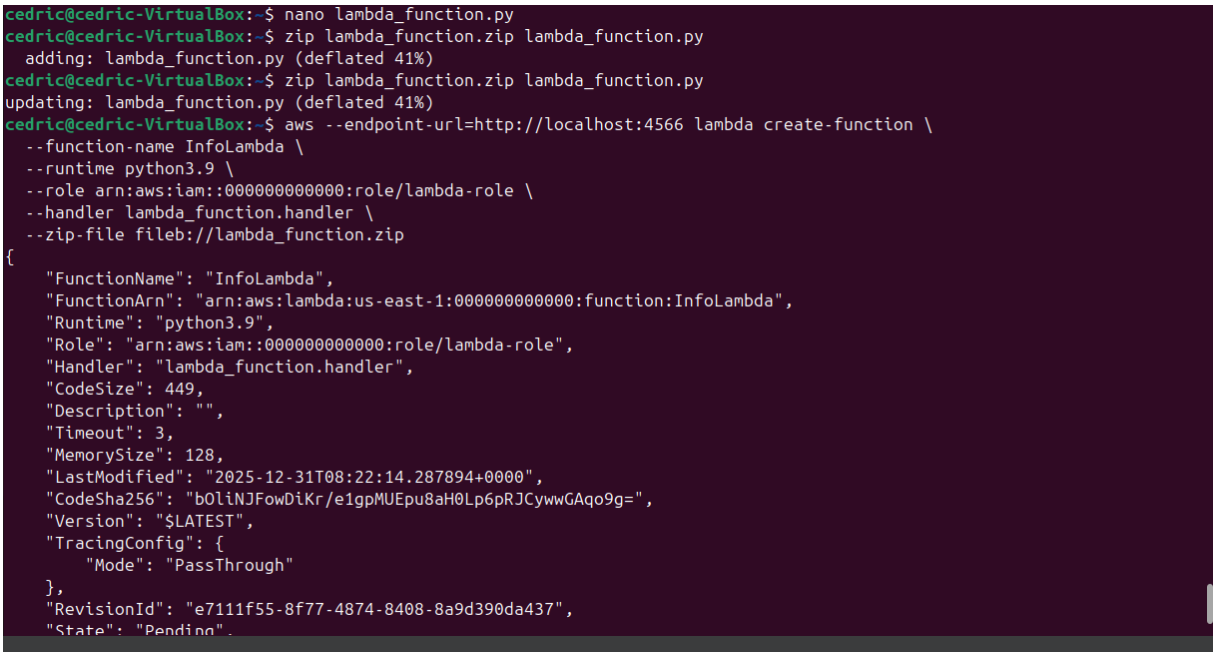


Figure 11: création et déploiement de la lambda zone

e. Tester la Lambda :

aws --endpoint-url=http://localhost:4566 lambda invoke --function-name InfoLambda response.json

- **response.json** : sera le fichier dans lequel le résultat sera écrit

Pour afficher le résultat on peut saisir la commande : **cat response.json**

Figure 12: vérification de la fonction lambda

f. Commande de vérification :

- Vérifier les tables : **aws --endpoint-url=http://localhost:4566 lambda list-fucntions**

4. API Gateway :

Est un service permettant d'exposer des endpoints HTTP, connecter une API à Lambda et gerer les routes et les methodes. Utiliser pour les API REST, les API Serverless et les backend d'appli reeel.on travaillera ici avec une API REST (**une API basee sur HTTP: GET,POST,URL**)

Client (navigateur, curl)

↓ (requête HTTP)

API Gateway

↓ (événement)

Lambda

↓ (réponse)

API Gateway

↓ (réponse HTTP)

Client

a. Créer un endpoint simulé :

b. Déclencher une fonctionnalité Lambda via http :

```
cedric@cedric-VirtualBox:~$ aws --endpoint-url=http://localhost:4566 apigateway create-rest-api \
    --name "TestAPI" \
    --description "API pour Lambda InfoLambda"
{
    "id": "7yrzkmretp",
    "name": "TestAPI",
    "description": "API pour Lambda InfoLambda",
    "createdDate": "2025-12-31T10:30:57+01:00",
    "apiKeySource": "HEADER",
    "endpointConfiguration": {
        "types": [
            "EDGE"
        ],
        "ipAddressType": "ipv4"
    },
    "disableExecuteApiEndpoint": false,
    "rootResourceId": "u98nortfmn"
}
```

Figure 13: création de l'API

```
cedric@cedric-VirtualBox:~$ aws --endpoint-url=http://localhost:4566 apigateway get-resources     --rest-api-id 7yrzkmre
tp
{
    "items": [
        {
            "id": "u98nortfmn",
            "path": "/"
        }
    ]
}
cedric@cedric-VirtualBox:~$ aws --endpoint-url=http://localhost:4566 apigateway put-method \
    --rest-api-id 7yrzkmretp \
    --resource-id u98nortfmn \
    --http-method GET \
    --authorization-type "NONE"
{
    "httpMethod": "GET",
    "authorizationType": "NONE",
    "apiKeyRequired": false
}
```

Figure 14: obtention de l'id ressource et création de la méthode POST

```
cedric@cedric-VirtualBox:~$ aws --endpoint-url=http://localhost:4566 apigateway put-integration \
    --rest-api-id 7yrzkmretp \
    --resource-id u98nortfmn \
    --http-method GET \
    --type AWS_PROXY \
    --integration-http-method POST \
    --uri arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:000000000000:function:I
nfoLambda/in
{
    "type": "AWS_PROXY",
    "httpMethod": "POST",
    "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:000000000000:function
:InfoLambda/in",
    "passthroughBehavior": "WHEN_NO_MATCH",
    "timeoutInMillis": 29000,
    "cacheNamespace": "u98nortfmn",
    "cacheKeyParameters": []
}
```

Figure 15: connexion de la methode POST a la fonction lambda

5. IAM (Identify and Access Management) :

Est un service permettant la gestion des rôles, des permissions et des politiques de sécurité

NB : un endpoint est une adresse permettant d'accéder à un service (indique où envoyer les requêtes)

Health Check est un outil de supervision nous permettant de voir si LocalStack et les services sont actifs ou non.

IV. Cas concret d'un scenario d'architecture cloud simulé (Application simple de traitement d'images basée sur le cloud) :

1. Architecture :

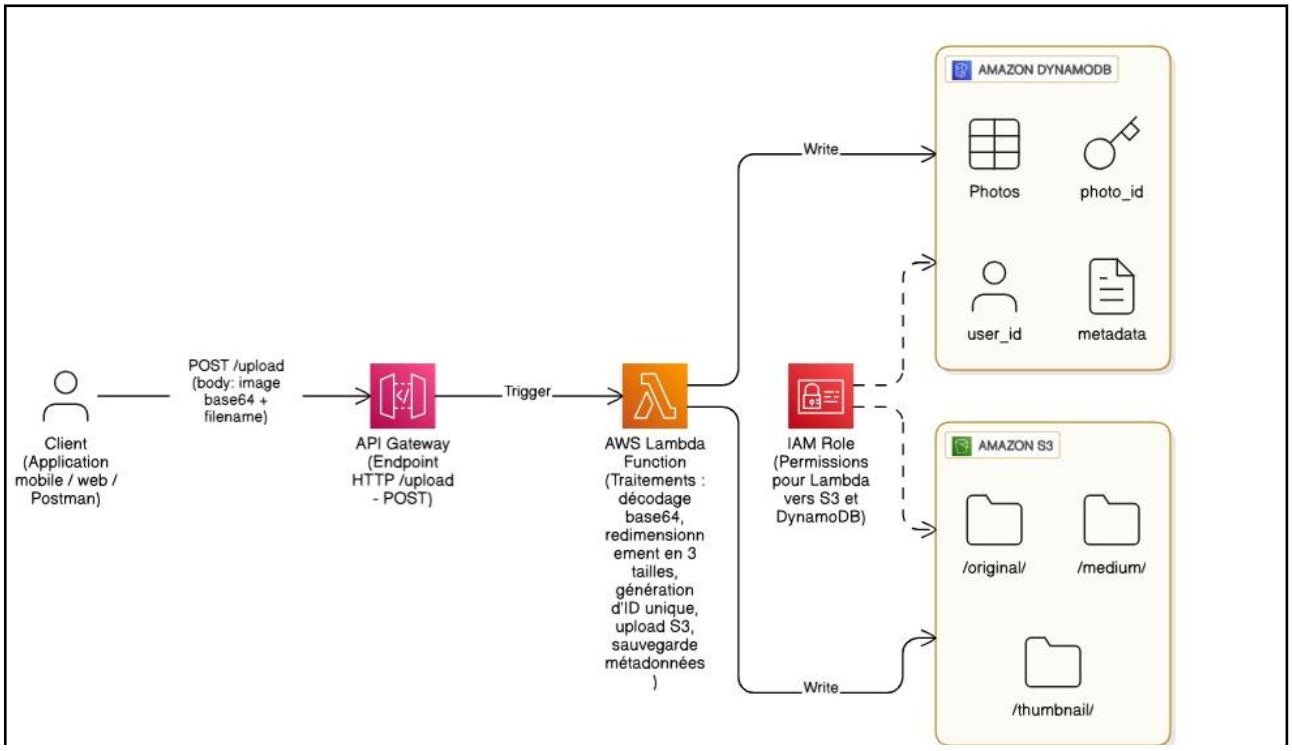


Figure 16: Architecture de notre scenario

De manière générale, cette application permettra à l'utilisateur d'uploader ses photos et celui-ci pourra obtenir des images redimensionner au format :

- **Medium** : pour afficher rapidement dans l'application.
- **Miniature (Thumbnail)** : de petites vignettes qui se charge rapidement, que l'utilisateur peut utiliser comme photo de profile par exemple et garder une copie de l'original, tout en conservant ces versions de tel sorte que l'utilisateur puisse y avoir accès via n'importe quel appareil.

Étape 1 : Upload de la photo

Utilisateur (app mobile) → Envoie une photo via HTTP POST

↓

API Gateway (reçoit la requête)

Ce qui se passe :

- L'utilisateur clique sur "Upload photo" dans l'application
- L'application envoie la photo en base64 via une requête HTTP POST
- API Gateway est le "réceptionniste" qui reçoit cette requête

Étape 2 : Lambda se déclenche

API Gateway → Déclenche automatiquement la Lambda

↓

Lambda démarre

Ce qui se passe :

- API Gateway, après avoir reçu la requête, la transmet a Lambda

- Lambda reçoit un événement contenant la photo, celui-ci à besoin pour se déclencher

Étape 3 : Traitement de l'image

Lambda → Reçoit la photo originale (ex : 4000x3000 pixels, 5 MB)

↓

Redimensionne en plusieurs tailles :

Thumbnail : 150x150 pixels

Medium : 800x600 pixels

Ce qui se passe :

- Une fois l'image reçu, Lambda utilisera Pillow pour redimensionner l'image d'origine en 2 format réduit

Étape 4 : Stockage dans S3

Lambda → Upload les 3 versions dans S3

↓

S3 stocke :

mon-bucket/images/original/photo_123.jpg (5 MB)

mon-bucket/images/medium/photo_123.jpg (500 KB)

mon-bucket/images/thumbnail/photo_123.jpg (50 KB)

S3 est ici un peu comme un entrepôt des fichiers qui stocke de manière permanente

Étape 5 : Métadonnées dans DynamoDB

Lambda → Enregistre les infos dans DynamoDB

↓

Table "Photos" :

{

"photo_id": "123",

"user_id": "alice",

"filename": "photo_123.jpg",

"upload_date": "2025-12-30T10:30:00Z",

"original_size": 5242880, // 5 MB en bytes

"width": 4000,

"height": 3000,

"s3_urls": {

"original": "s3://mon-bucket/images/original/photo_123.jpg",

"medium": "s3://mon-bucket/images/medium/photo_123.jpg",

"thumbnail": "s3://mon-bucket/images/thumbnail/photo_123.jpg"

}}

DynamoDB ici est une base de données rapide qui stocke les informations structurées et permet de rechercher rapidement des informations (par exemple les photos d'un user x/y).

Étape 6 : IAM contrôle tout

IAM vérifie à chaque action :

- Lambda peut-elle écrire dans S3 ?
- Lambda peut-elle écrire dans DynamoDB ?
- Lambda peut-elle lire les secrets ?

2. Implémentation :

a. Installation de pillow :

Ici nous utiliserons la bibliotheque Pillow, compatible avec Lambda et supporte JPG, PNG, le resize et le crop.

Pour que cela marche avec notre Lambda, nous devons installer Pillow dans un conteneur basé sur l'image officielle AWS pour python sur lambda et récupérer le fichier dossier installé (package) car AWS Lambda est un environnement serverless basé sur **Amazon Linux** (une variante de Linux optimisée pour AWS), et les bibliothèques comme Pillow contiennent des parties compilées (en C) qui doivent être compatibles avec cette plateforme.

docker run --rm -v "$PWD":/var/task --entrypoint /bin/bash public.ecr.aws/lambda/python:3.9 -c "pip install pillow -t package"

- **--rm** : Supprime le conteneur une fois l'execution terminer car il nous servira juste a telecharger Pillow et a recuper le dossier (package) qui deja dans un environnement adequat (Amazon Linux dans ce cas)
- ***-v "$PWD":/var/task* :** volume utilisee pour pouvoir recuperer le dossier (package) de Pillow
- ***-entrypoint /bin/bash* :** change le programme par defaut lancee au demarrage du conteneur pour utiliser /bin/bash et non python (pour le cas de AWS Lambda) car on veux exeecuter une commande shell personnalisee (***-c "pip install pillow -t package"***)

- ***-c "pip install pillow -t package"***

  - **-c** : argument passee à Bash qui demande d'exeecuter la chaine qui suit comme une commande
  - **pip install pillow** : installe Pillow
  - **-t package** : installe Pillow dans un dossier local *package*

b. Création du bucket S3 :

aws --endpoint-url=http://localhost:4566 s3 mb s3://bucket-images

c. Création de la table DynamoDB :

Nous aurons une table visuellement sous cette forme sous cette forme, bien que tout soit en format JSON

| user_id |photo_id | filename | upload_date | metadata |

---

aws --endpoint-url=http://localhost:4566 dynamodb create-table --table-name Photos --attribute-definitions AttributeName=photo_id,AttributeType=S AttributeName=user_id,AttributeType=S --key-schema AttributeName=user_id,KeyType=HASH AttributeName=photo_id,KeyType=RANGE --billing-mode PAY_PER_REQUEST

user_id est notre clé primaire

photo_id est une clé de trie (pour classer les photos par utilisateurs)

d. Création de la fonction Lambda :

Nous utiliserons Python comme Runtime car **il est le support natif AWS Lambda, simple et lisible.**

Cette Lambda va :

- recevoir une image envoyée en texte (base64)
- recuperer les informations de l'image
- redimensionner en créant d'2 autres tailles (medium et miniature (thumbnail))
- stocker les fichier images dans S3 et les metadatas dans DynamoDB
- retourner une réponse JSON

Une fois terminer on doit :

- zipper Pillow et ses fichiers en premier

  cd package

  zip -r ../lambda_package.zip

- ajouter le code Lambda au zip

  cd ..

  zip -g lambda_package.zip lambda_function.py

Ensuite on déploira notre Lambda

aws --endpoint-url=http://localhost:4566 lambda create-function --function-name image-test --runtime python3.9 --role arn:aws:iam::000000000000:role/lambda-role –handler lambda_function.lambda_handler --zip-file fileb://lambda_package.zip --timeout 30 --memory-size 512

- *--timeout 30 : temps max d'execution de 30s*
- *--memory-size 512 : RAM allouée 512 Mo (optimal pour traitement d'images)*

```
kitio@ubuntu:~/DOCKER/PROJET/lambda$ aws --endpoint-url=http://localhost:4566 lambda update-func
tion-code --function-name image-test --zip-file fileb://lambda_package.zip
{
    "FunctionName": "image-test",
    "FunctionArn": "arn:aws:lambda:us-east-1:000000000000:function:image-test",
    "Runtime": "python3.9",
    "Role": "arn:aws:iam::000000000000:role/lambda-role",
    "Handler": "lambda_function.lambda_handler",
    "CodeSize": 7871688,
    "Description": "",
    "Timeout": 30,
    "MemorySize": 128,
    "LastModified": "2026-01-04T11:33:55.365945+0000",
    "CodeSha256": "gKn4efJ/Gf2wkgF5RgGhfbvZ1tpBltYL1WxxsPn299g=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "RevisionId": "a19de8e4-c32f-4169-80fe-f3811b938106",
    "State": "Active",
    "LastUpdateStatus": "InProgress",
    "LastUpdateStatusReason": "The function is being created.",
    "LastUpdateStatusReasonCode": "Creating"
}
```

Figure 17: déploiement de la lambda avec ses caractéristiques

Et on vérifiera si elle est up (après quelques secondes/minutes d'attente) avec :
aws lambda get-function --function-name image-test --endpoint-url http://localhost:4566

ainsi que la liste des fonctions :
aws --endpoint-url=http://localhost:4566 lambda list-functions

NB : Pour faire des modifications après déploiement :

- On rezippe la fonction : zip -g lambda_package.zip lambda_function.py
- On **met à jour** avec : *aws --endpoint-url=http://localhost:4566 lambda update-function-code --function-name image-test --zip-file fileb://lambda_package.zip # Chemin vers ton ZIP mis à jour*

e. Phase de test :

➢ Test via Invocation directe Lambda :

Ici on va utiliser un fichier test_event.json qui sera une simulation de un évènement d'envoi d'une image en texte base64 sous cette forme

{

"body": "{\"image\":
\"/9j/4AAQSkZJRgABAQEAYABgAAD/2wBDAAIBAQIBAQICAgICAgICAwUDAwMDAwYE
BAMFBwYHBwcGBwcICQsJCAgKCAcHCg0KCgsMDAwMBwkODw0MDgsMDAz/2wBDA
QICAgMDAwYDAwYMCAcIDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwM
DAwMDAwMDAwMDAwMDAwMDAwMDAwMDAz/wAARCAABAAEDASIAAhEBAxEB/8Q
AHwAAAQUBAQEBAQEAAAAAAAAAAAECAwQFBgcICQoL/8QAtRAAAgEDAwIEAwUFB
AQAAAF9AQIDAAQRBRIhMUEGE1FhByJxFDKBkaEII0KxwRVS0fAkM2JyggkKFhcYGRol
JicoKSo0NTY3ODk6Q0RFRkdISUpTVFVWV1hZWmNkZWZnaGlqc3R1dnd4eXqDhIWGh4iJi
pKTlJWWl5iZmqKjpKWmp6ipqrKztLW2t7i5usLDxMXGx8jJytLT1NXW19jZ2uHi4+Tl5ufo6e
rx8vP09fb3+Pn6/8QAHwEAAwEBAQEBAQEBAQAAAAAAAAECAwQFBgcICQoL/8QAtRE
AAgECBAQDBAcFBAQAAQJ3AAECAxEEBSExBhJBUQdhcRMiMoEIFEKRobHBCSMzUvA
VYnLRChYkNOEl8RcYGRomJygpKjU2Nzg5OkNERUZHSElKU1RVVldYWVpjZGVmZ2hpan
N0dXZ3eHl6goOEhYaHiImKkpOUlbaWmJmaoqOkpaanqKmqsrO0tba3uLm6wsPExcbHyMn
K0tPU1dbX2Nna4uPk5ebn6Onq8vP09fb3+Pn6/9oADAMBAAIRAxEAPwD9/KKKAP/2Q
=\", \"filename\": \"test.jpg\", \"user_id\": \"alice\"}"

}

Ici cette chaine de caractères est le codage en texte base64 d'une image .jpg d'un pixel 1x1 blanc sur fond noir (car elle est simple et donc moins longue). Si on veut tester avec une vraie image, on peut faire comme ceci

Pour une image nommée "milky_way.jpg" par exemple, on va :

- **encoder en base64 :** *IMAGE_B64=$(base64 -w 0 milky_way.jpg)*
- **Créer notre payload , en remplaçant le contenu du fichier test_event.json par ceci**

{

"body": "{\"image\": \"$IMAGE_B64\", \"filename\": \"milky_way.jpg\", \"user_id\": \"kitis\"}"

}

On le testera avec la commande (peut prendre 10s - 15s si image volumineuse)

aws --endpoint-url=http://localhost:4566 lambda invoke --function-name image-test --payload file://test_event.json output.json

Si le code est bon on soit avoir cette sortie

{

"StatusCode": 200,

"ExecutedVersion": "$LATEST"

}

Et on lira la sortie avec **"cat output.json | python3 -m json.tool"** (permettra un affichage sous un format plus lisible) pour le débogage et si tout est ok alors on doit avoir aussi une sortie avec écrit **"statusCode": 200**

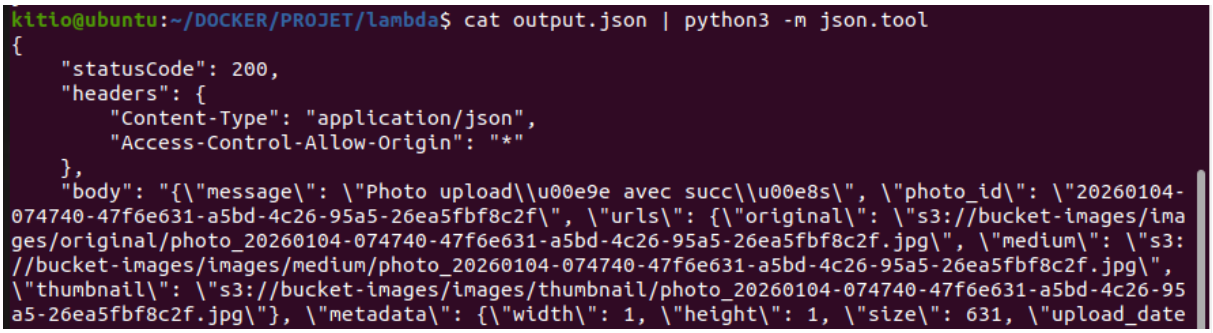


Figure 18: test réussi

On peut donc :

- **Vérifier la liste des buckets avec plus de précisions :** *aws --endpoint-url=http://localhost:4566 s3 ls s3://bucket-images/ --recursive*

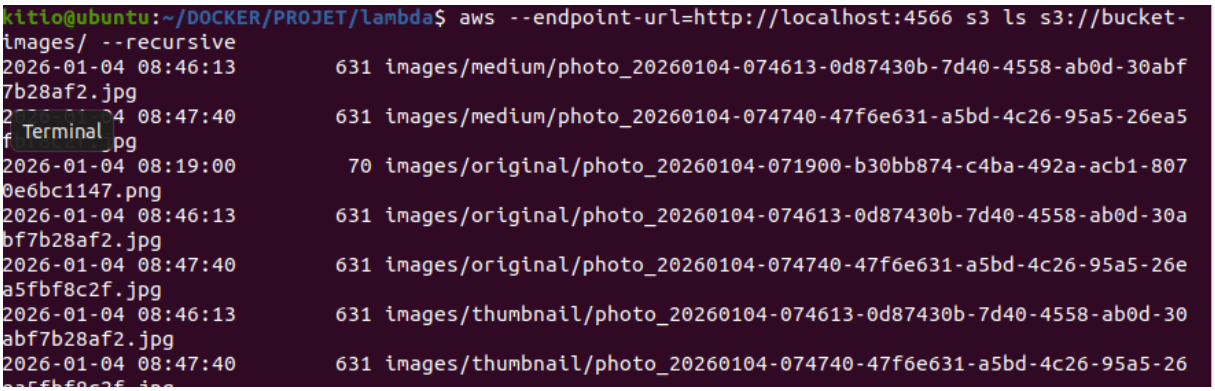


Figure 19: test pour voir le contenu du bucket

- **Vérifier les metadata enregistrées :** *aws --endpoint-url=http://localhost:4566 dynamodb scan --table-name Photos | python3 -m json.tool*

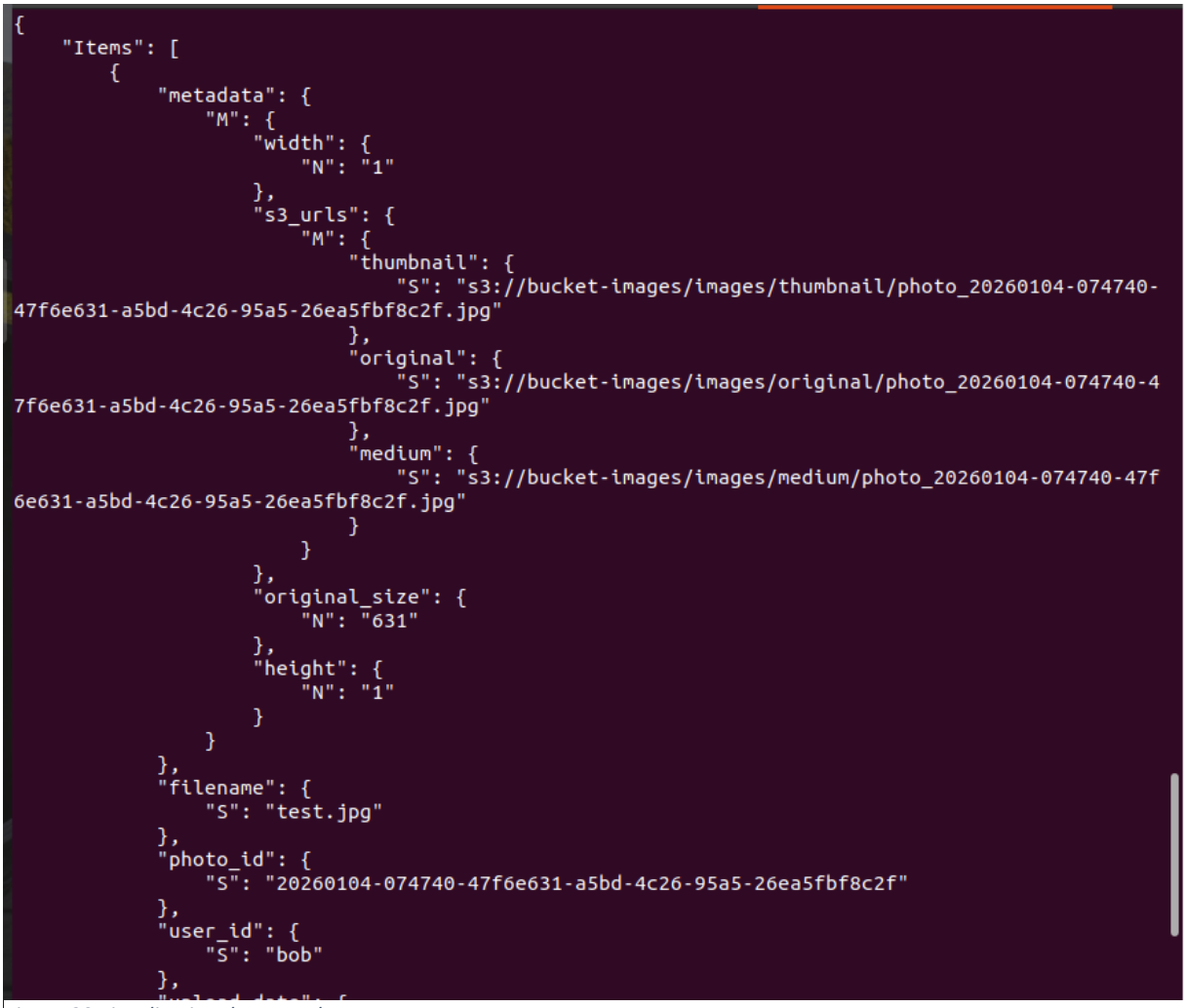


Figure 20:visualisation des metadatas

- ***Telecharger une image:*** ***a****ws --endpoint-url=http://localhost:4566 s3 cp s3://bucket images/images/thumbnail/photo_20260104-074740-47f6e631- a5bd-4c26-95a5-26ea5fbf8c2f.jpg downloaded-thumbnail.jpg :* vue que le nom utilisé est complexe, on a renommé cela en *downloaded-thumbnail.jpg.*

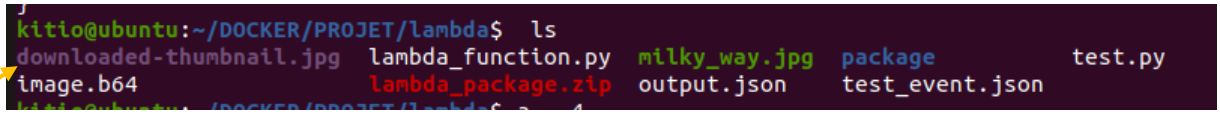


Figure 21: vérif. de la présence de l'image téléchargée

- **Ouvrir le fichier nouvellement téléchargé et voir à quoi il ressemble (Si GUI)** : *xdg-open downloaded-thumbnail.jpg*
- **Si pas de GUI on va juste voir les informations sur le fichier avec :** *file downloaded-thumbnail.jpg*

En cas de problèmes, on peut vérifier les logs : *aws --endpoint-url=http://localhost:4566 logs tail /aws/lambda/image-test --follow*

➢ Test via API Gateway http :

Etape 1 : Creation d'une API Gateway

- ***Création de l'API :*** *aws --endpoint-url=http://localhost:4566 apigateway create-rest-api --name photo-api --description "API pour upload de photos"*
- ***Recuperation de sont ID :*** ***API_ID****=$(aws --endpoint-url=http://localhost:4566 apigateway get-rest-apis --query 'items[?name==`photo-api`].id' --output text)*
  Avec ceci on a une constante nommé API_ID qu'on peut vérifier avec **"echo $API_ID"**
- **Obtenir la racine:** **ROOT_ID**=$(aws --endpoint-url=http://localhost:4566 apigateway get-resources --rest-api-id $API_ID --query 'items[0].id' --output text)
- ***Création de la ressource (/upload) et recuperation de son ID :*** ***RESOURCE_ID****=$(aws --endpoint-url=http://localhost:4566 apigateway create-resource --rest-api-id $API_ID --parent-id $ROOT_ID --path-part upload --query 'id' --output text)*
- ***Création d'un POST sur /upload pour autoriser les requetes de ce type:*** *aws --endpoint-url=http://localhost:4566 apigateway put-method --rest-api-id* ***$API_ID*** *--resource-id* ***$RESOURCE_ID*** *--http-method POST --authorization-type NONE*
- ***Integration à notre lambda pour la connecter à notre API Gateway :*** *aws --endpoint-url=http://localhost:4566 apigateway put-integration --rest-api-id* ***$API_ID*** *--resource-id $****RESOURCE_ID*** *--http-method POST --type AWS_PROXY --integration-http-method POST --uri "arn:aws:apigateway:us-*

east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:000000000000:function:image-test/invocations"

- ***Déployer l'API:*** *aws --endpoint-url=http://localhost:4566 apigateway create-deployment --rest-api-id* ***$API_ID*** *--stage-name dev*

Etape 2 : Tester l'API via HTTP

- ***Construire l'URL de L'API :*** *API_URL="http://localhost:4566/restapis/$API_ID/dev/_user_request_/upload"* est visible via **$API_URL**
- ***On se servira de la petite image de test du début en premier lieu via le fichier test_event.json et on enverra une requête POST***

  Mais ici on utilisera un fichier JSON pour l'API http :

{

"image":
"/9j/4AAQSkZJRgABAQEAYABgAAD/2wBDAAIBAQIBAQICAgICAgICAw
UDAwMDAwYEBAMFBwYHBwcGBwcICQsJCAgKCAcHCg0KCgsMDAwM
BwkODw0MDgsMDAz/2wBDAQICAgMDAwYDAwYMCAcIDAwMDAwMD
AwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwMDAwM
DAwMDAwMDAwMDAz/wAARCAABAAEDASIAAhEBAxEB/8QAHwAAA
QUBAQEBAQEAAAAAAAAAAAECAwQFBgcICQoL/8QAtRAAAgEDAwIE
AwUFBAQAAAF9AQIDAAQRBRIhMUEGE1FhByJxFDKBkaEII0KxwRVS0f
AkM2JyggkKFhcYGRolJicoKSo0NTY3ODk6Q0RFRkdISUpTVFVWV1hZWm
NkZWZnaGlqc3R1dnd4eXqDhIWGh4iJipKTlJWWl5iZmqKjpKWmp6ipqrKztL
W2t7i5usLDxMXGx8jJytLT1NXW19jZ2uHi4+Tl5ufo6erx8vP09fb3+Pn6/8QAH
wEAAwEBAQEBAQEBAQAAAAAAAAECAwQFBgcICQoL/8QAtREAAgEC
BAQDBAcFBAQAAQJ3AAECAxEEBSExBhJBUQdhcRMiMoEIFEKRobHBC
SMzUvAVYnLRChYkNOEl8RcYGRomJygpKjU2Nzg5OkNERUZHSElKU1RV

VldYWVpjZGVmZ2hpanN0dXZ3eHl6goOEhYaHiImKkpOUlbaWmJmaoqOkpa
anqKmqsrO0tba3uLm6wsPExcbHyMnK0tPU1dbX2Nna4uPk5ebn6Onq8vP09fb3
+Pn6/9oADAMBAAIRAxEAPwD9/KKKAP/2Q==",

"filename": "test.jpg",

"user_id": "bob" }

```
curl -X POST "$API_URL" -H "Content-Type: application/json" –d @test_event.json |
    python3 -m json.tool
```

- **-X** : Spécifie la méthode HTTP à utiliser
- **-H "Content-Type: application/json" :** Ajouter un header HTTP pour préciser l'envoie des données au format JSON
- **-d @test_event.json :** lis le contenu d'un fichier

```
kitio@ubuntu:~/DOCKER/PROJET/lambda$ curl -X POST "$API_URL" -H "Content-Type: application/json"
 -d @test_event.json | python3 -m json.tool
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1467  100   564  100   903     29     46  0:00:19  0:00:19 --:--:--   144
{
    "message": "Photo upload\u00e9e avec succ\u00e8s",
    "photo_id": "20260105-025607-0141e921-87e2-4fcd-8248-069395120580",
    "urls": {
        "original": "s3://bucket-images/images/original/photo_20260105-025607-0141e921-87e2-4fcd
-8248-069395120580.jpg",
        "medium": "s3://bucket-images/images/medium/photo_20260105-025607-0141e921-87e2-4fcd-824
8-069395120580.jpg",
        "thumbnail": "s3://bucket-images/images/thumbnail/photo_20260105-025607-0141e921-87e2-4f
cd-8248-069395120580.jpg"
    },
    "metadata": {
        "width": 1,
        "height": 1,
        "size": 631,
        "upload_date": "2026-01-05T02:56:07.787862Z"
    }
}
```

Figure 22: test réussi

Pour le test avec une vraie photo on reproduira le cheminement vu plus haut.

CONCLUSION

Venu au terme de notre projet, nous avons pu mettre en œuvre une architecture severless complète sur LocalStack, simulant un environnement AWS. L'application développé implémente un système de traitement et de stockage d'image intégrant AWS Lambda pour le traitement, S3 pour le stockage des fichiers, DynamoDB pour la gestion des metadatas API Gateway comme point d'entrée http et IAM pour la sécurité. Tout cela nous a permit d'acquérir une compréhension approfondie du modèle serverless et de l'orchestration des services cloud, se justifiant par une application final fonctionnelle et testée via invocation direct Lambda et requête http.