Running a heavy machine learning model in lambda using Docker

This artifact is the result of a real experience with a customer. The need was to be able to adapt an existing machine learning model developed in python to a serverless architecture on AWS.

One of the alternatives that was explored is the use of Lambda, with the restriction that the model weighed approximately 250 mb, and along with the required python libraries, the lambda code ended up weighing 1.5 gb. Because traditional lambda has a size restriction of 250MB in total, the new lambda docker alternative was explored, which allows running docker images up to 10GB. by "heavy" it is meant that it is heavier than what a traditional lambda allows to execute.

The lambda example shown in this artifact lets you to detect if a jpg or png image has a handwritten signature or not, using a pre-built Mask R-CNN object detection model. In this example, the images used are working letters or work certificates.

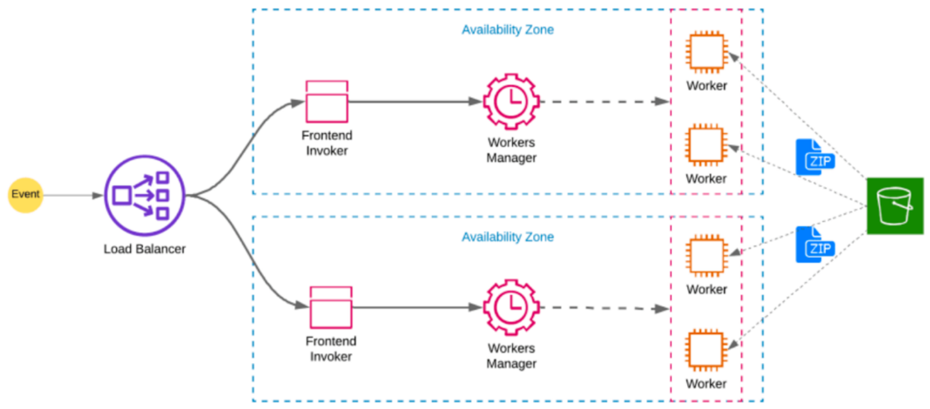
In this document you can find an example of how docker was configured and used to create the lambda image, how it is automatically deployed in AWS, with all the necessary cloudformation templates to support it. Also, an explanation of the execution of the lambda and the code.

Disclaimer: This document is not about the machine learning model itself, nor how it was built / trained, but about how to execute the model in a lambda docker environment.

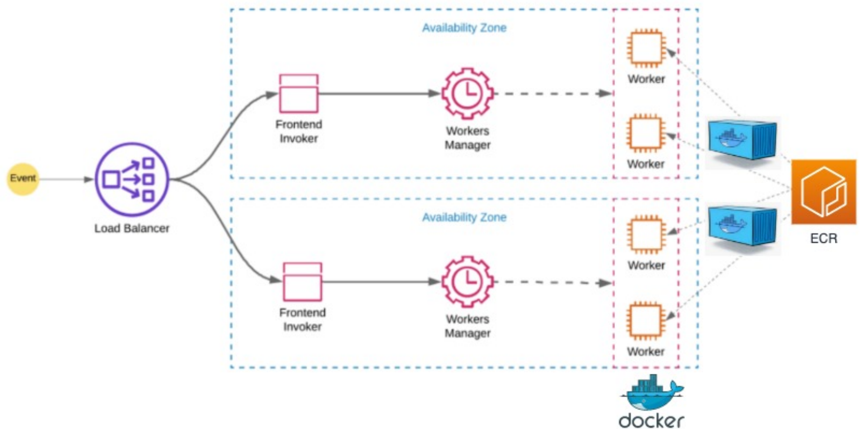
All the code referred in this document is attached to this artifact in a folder called “lambda_object_detection_handwritten_signature”.

Lambda Docker in summary

the traditional execution model of a lambda is explained in the following graph



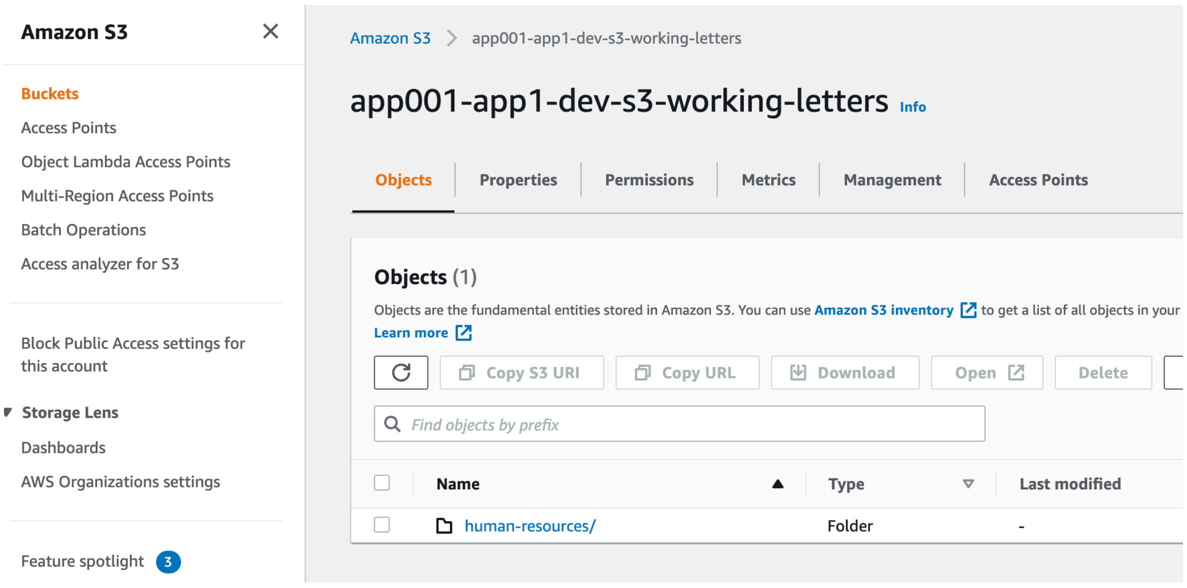
When you upload a lambda zipped code, this code is saved in an AWS private s3 bucket, and the worker machines that execute the lambda code load this code from s3. In the docker model, the code resides in a docker image published in an ECR (elastic container registry) repository, and the workers executes the docker images:



As the code resides in docker images, you can't change the code on the fly in the AWS console. Instead, you need to publish a new version of the docker image in the repository.

Configuring dev environment

Before starting the lambda explanation, some AWS components need to be created in order to run the lambda example shown in this document. First, we need to create a s3 bucket where we will put the example images to process. We created a folder called "human-resources" to save the example work certificates:



app001-app1-dev-s3-working-letters

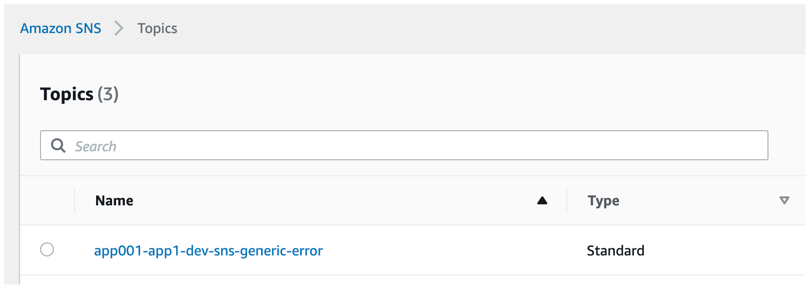
Objects (1)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your

Learn more

Name	Type	Last modified
human-resources/	Folder	-

Also, the lambda uses a SNS topic to report any error raised in the lambda. For this reason we need to create the topic manually at this time:


Amazon SNS > Topics

Topics (3)

Search

	Name	Type
○	app001-app1-dev-sns-generic-error	Standard

Also, is necessary to install the software "docker desktop" and have the aws cli configured correctly before start. (not covered in this document)

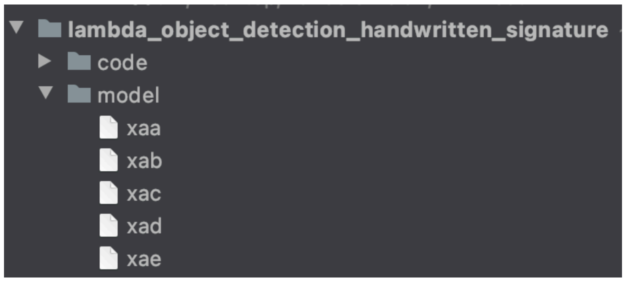
A word about the machine learning model

As said at the beginning of the document, this document doesn't explain in detail the ML model used to predict if an image has or not a handwritten signature, but we can say that this is a Mask R-CNN model, that is a framework for object instance segmentation. You can find all the details in: https://arxiv.org/abs/1703.06870

Here we can find also a python implementation example explained in detail:

https://blog.paperspace.com/mask-r-cnn-in-tensorflow-2-0/

The model used in this demo is a H5 model, and his weight is 250 mb aprox. In order to let this model to be saved in a git repo, the model was splitted in 5 files, and assembled later in the docker file:


```
▼ lambda_object_detection_handwritten_signature
    ▶ code
    ▼ model
        xaa
        xab
        xac
        xad
        xae
```

Conceptual process explanation

In the next chapter it is explained the automation process designed to deploy the lambda example, but first is necessary to explain the manual process to deploy the lambda.

You must first create a docker image with the lambda service code to be deployed. The docker image can be created on a local machine, using the "docker desktop" software.

The docker image is created from a file called Dockerfile, which, for the example lambda, has the following content:

```
FROM public.ecr.aws/lambda/python:3.7

COPY requirements.txt  ./
COPY model/x* ./

RUN  cat x* > model.h5

RUN  pip3 install -r requirements.txt

#to avoid scan vulnerabilities
RUN  yum -y update libX11
RUN  yum -y update curl
RUN  yum -y update glibc
RUN  yum -y update java-1.8.0-openjdk
RUN  yum -y update nss
RUN  yum -y update nspr
RUN  yum -y update nss-softokn
RUN  yum -y update nss-util
RUN  yum -y update rpm
RUN  yum -y update log4j-cve-2021-44228-hotpatch

# Copy handler function (from the local app directory)
COPY  code/lambda_object_detection_handwritten_signature.py  ./
COPY  code/ConfigModel.py  ./

# Overwrite the command by providing a different command directly in the
template.
CMD ["lambda_object_detection_handwritten_signature.lambda_handler"]
```

The first line of the file downloads a public docker image from the AWS repositories, in this case for a Python 3.7 environment. (the tensorflow model on which this inference model runs only works with this version of Python).

The second line copies the requirements.txt file to the docker image. This is the content of the file:

```
mask-rcnn-12rics==0.2.3
pillow==8.2.0
numpy==1.18.5
scikit-image==0.16.2
tensorflow==1.15.0
Keras==2.0.8
h5py==2.10.0
```

These libraries are strictly required by the provided signature detection model. The third line uploads the file containing the model weights to the docker image. The fourth line installs all the requirements in the docker image.

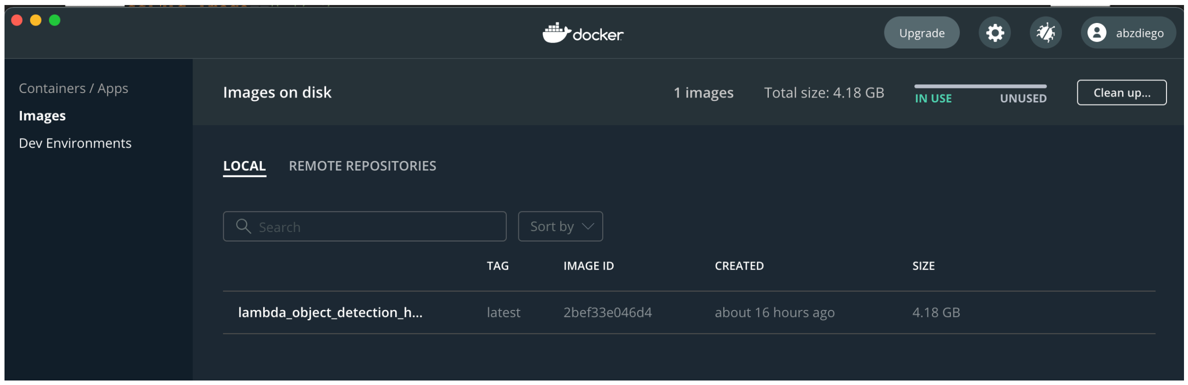
The lines that do “yum -y update” install a series of updates to the image's base software, which if not done, will be detected as “anomalies” when the docker image is uploaded to the AWS elastic container registry (ECR) service.

The following lines copy the source code of the lambda to the docker image, and the last line modifies the "CMD" of the image, to point it by default to the main method of the lambda.

To build the image from a Dockerfile, you can run the following command, in the directory where the Dockerfile file is located

```
docker build -t
lambda_object_detection_handwritten_signature_image .
```

the -t parameter is the name you want to give to the image. This will create the image in the "Images" section in the docker desktop software:

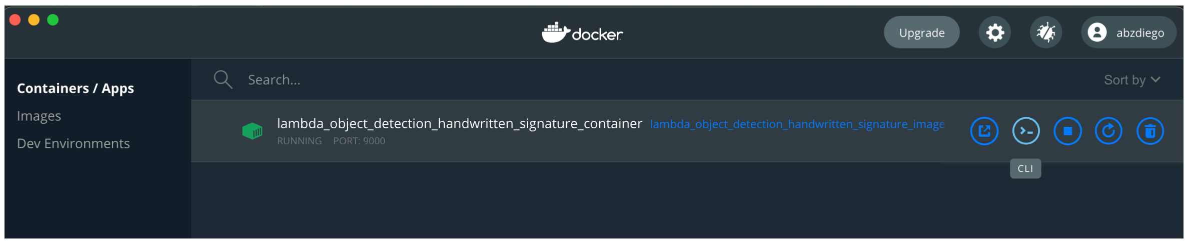


	TAG	IMAGE ID	CREATED	SIZE
lambda_object_detection_h...	latest	2bef33e046d4	about 16 hours ago	4.18 GB

At development time, this image can be run to create a container that is a fully virtualized runtime environment with the OS versions and libraries required to run the code. A container is created with the following command:

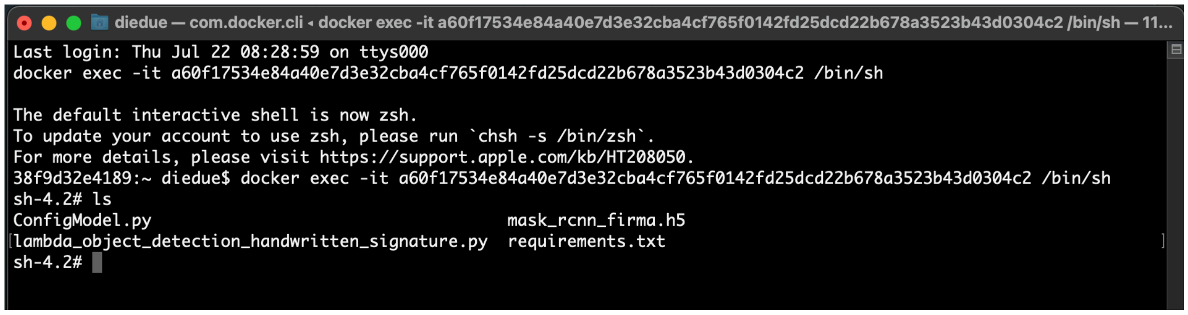
```
docker run -d --name
lambda_object_detection_handwritten_signature_container -p
9000:8080 lambda_object_detection_handwritten_signature_image
```

Note that a port is passed as a parameter, which is a tunnel to listen locally through a port in the container. In the "containers" section of docker desktop you can see the container running:



lambda_object_detection_handwritten_signature_container lambda_object_detection_handwritten_signature_image
RUNNING PORT: 9000

At this point, this container can be accessed as if it were any Linux machine. It can be accessed through the “CLI” icon at the top. By doing an "ls" on the content of the home directory of the container's Linux machine, we can see the files described in the Dockerfile file:



```
diedue — com.docker.cli ◂ docker exec -it a60f17534e84a40e7d3e32cba4cf765f0142fd25dcd22b678a3523b43d0304c2 /bin/sh — 11...
Last login: Thu Jul 22 08:28:59 on ttys000
docker exec -it a60f17534e84a40e7d3e32cba4cf765f0142fd25dcd22b678a3523b43d0304c2 /bin/sh

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
38f9d32e4189:~ diedue$ docker exec -it a60f17534e84a40e7d3e32cba4cf765f0142fd25dcd22b678a3523b43d0304c2 /bin/sh
sh-4.2# ls
ConfigModel.py                                  mask_rcnn_firma.h5
lambda_object_detection_handwritten_signature.py  requirements.txt
sh-4.2# 
```

Python code can be tested in the container in 2 ways. The first, with the traditional Python command, and the second is using an HTTP request on the port exposed in the "run" command seen above. This is to exactly emulate the request made on the lambda function as it would be deployed in the AWS environment.

```
curl -XPOST "http://localhost:9000/2015-03-
31/functions/function/invocations" -d '{"…"}
```

After the Code has been tested at development time locally, the docker image can be published to an ECR (Elastic Container Registry) repository on AWS. For this, the first thing is that the local docker client must be authenticated with the ECR service of the aws account where the image will be uploaded. This is with the following command:

```
aws ecr get-login-password --profile $PROFILE --region $REGION |
docker login --username AWS --password-stdin
$ACCOUNT_ID.dkr.ecr.$REGION.amazonaws.com
```

In the previous command you have to replace the $ with the corresponding values. This will create an authentication token for docker in the AWS account.

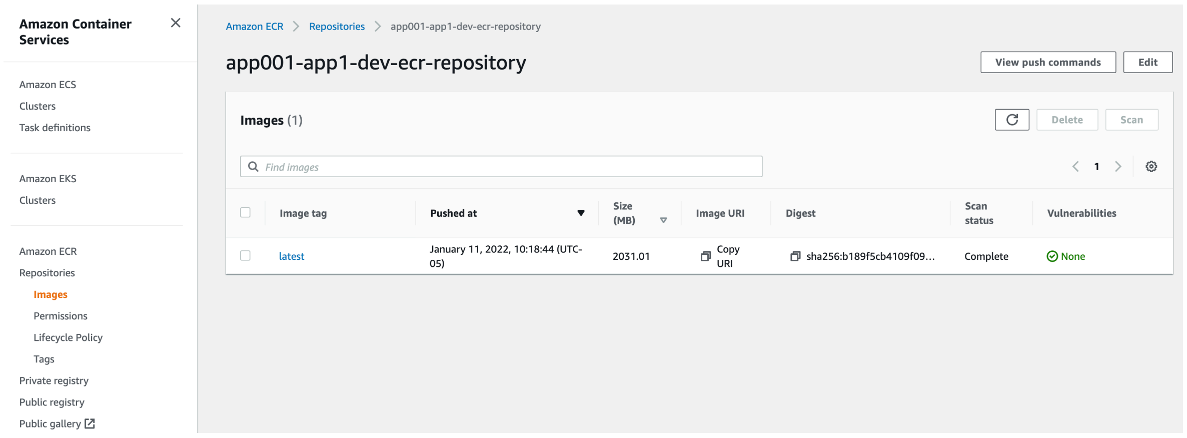
The next step is to create an ECR repository, which can be done with the following command:

```
aws ecr create-repository --profile p1 --region us-east-1 --
repository-name $PREFIX-ecr-repository --image-scanning-
configuration scanOnPush=true --image-tag-mutability MUTABLE
```

Once the repository is created, a tag can be added to the local docker image, and the docker image can be published in the repository with the following command:

```
docker tag
lambda_object_detection_handwritten_signature_image:latest
$ACCOUNT_ID.dkr.ecr.$REGION.amazonaws.com/$PREFIX-ecr-
repository:latest
docker push $ACCOUNT_ID.dkr.ecr.us-east-1.amazonaws.com/$PREFIX-
ecr-repository:latest
```

If this command is successful, the docker image should be able to be seen published in the created ECR repository:

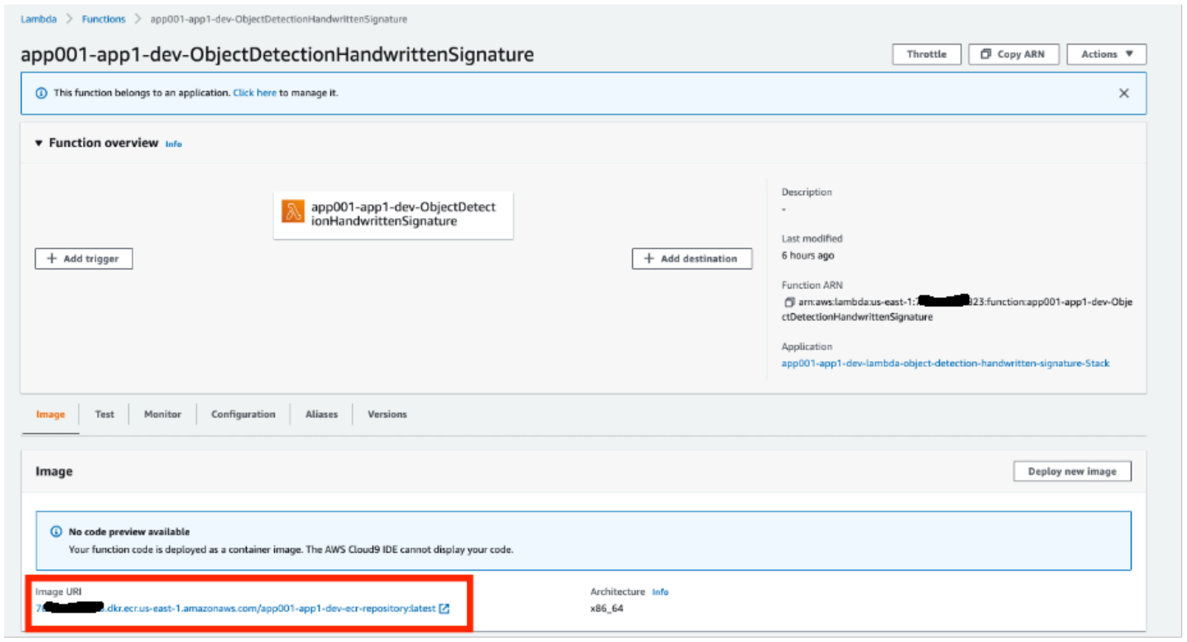


app001-app1-dev-ecr-repository

Images (1)

Image tag	Pushed at	Size (MB)	Image URI	Digest	Scan status	Vulnerabilities
latest	January 11, 2022, 10:18:44 (UTC-05)	2031.01	Copy URI	sha256:b189f5cb4109f09...	Complete	None

The last step is to link the image to the Lambda service. For this, when creating a lambda, it must be created of type "Image", and configure the URI of the docker image in the ECR service:



app001-app1-dev-ObjectDetectionHandwrittenSignature

This function belongs to an application. Click here to manage it.

Description
-

Last modified
6 hours ago

Function ARN
arn:aws:lambda:us-east-1:7[redacted]323:function:app001-app1-dev-ObjectDetectionHandwrittenSignature

Application
app001-app1-dev-lambda-object-detection-handwritten-signature-Stack

Image

No code preview available
Your function code is deployed as a container image. The AWS Cloud9 IDE cannot display your code.

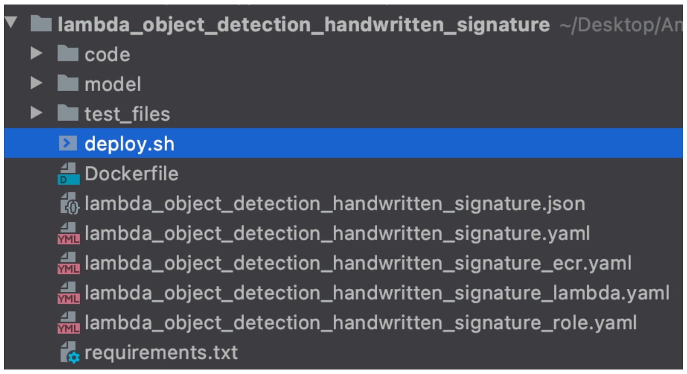
Image URI
7[redacted].dkr.ecr.us-east-1.amazonaws.com/app001-app1-dev-ecr-repository:latest

Architecture Info
x86_64

After this, the lambda can be configured in a similar way to any lambda, except that lambdas of this type cannot have associated layers, because it is assumed that everything is already embedded in the docker image.

Deploying the lambda using the deploy.sh script

To automate the process described before, it was created a deploy.sh file, that contains all the necessary instructions to deploy the lambda with one execution.



```
▼ lambda_object_detection_handwritten_signature ~/Desktop/A
  ▶ code
  ▶ model
  ▶ test_files
    deploy.sh
    Dockerfile
    lambda_object_detection_handwritten_signature.json
    lambda_object_detection_handwritten_signature.yaml
    lambda_object_detection_handwritten_signature_ecr.yaml
    lambda_object_detection_handwritten_signature_lambda.yaml
    lambda_object_detection_handwritten_signature_role.yaml
    requirements.txt
```

The script uses as input a json file called "lambda_object_detection_handwritten_signature.json" , and a group of cloudformation files explained later. The json file has the following content:

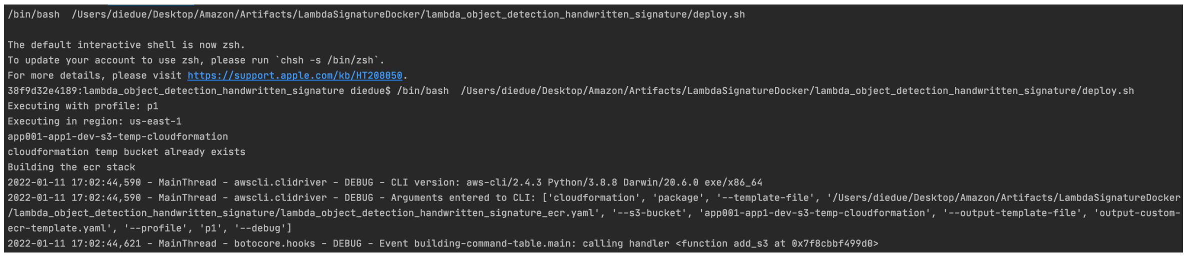
```
{
   "Parameters" : {
      "pEnvironment": "dev",
      "pRegion": "us-east-1",
      "pProfile": "p1",
      "pAccountID": "12345678912",
      "pApplicationID": "app001",
      "pApplicationName": "app1",
      "pS3CloudformationTempBucket": "s3-temp-cloudformation",
      "pS3RawBucket": "s3-working-letters",
      "pSNStopicName": "sns-generic-error",
      "pLogLevel": "DEBUG"
   }
}
```

The parameters explanation is:

Parameter	Description
pEnvironment	Indicates if the AWS environment is dev, test or prod.
pRegion	AWS region
pProfile	Profile name configured in the ~/aws/credentials file, corresponding to the test AWS account where the lambda will be deployed.
pAccountID	Test AWS account number
pApplicationID	This is inherited from the customer environment, and is an internal APP id.
pApplicationName	This is inherited from the customer environment, and is an internal APP name.
pS3CloudformationTempBucket	This is the name of the temporary bucket used to deploy and run the cloudformation templates.
pS3RawBucket	This is the name of the bucket where the jpg or png images to test need to be saved.

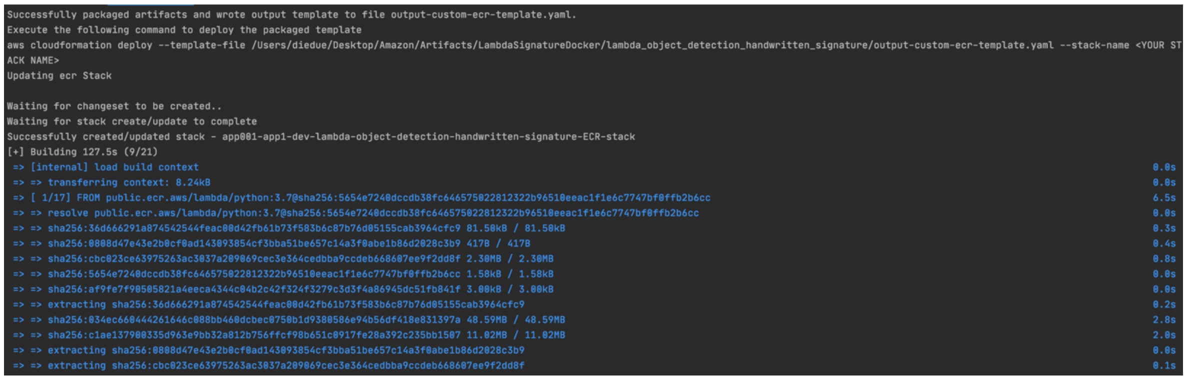
pSNStopicName	SNS topic name where the lambda will send any notification error.
pLogLevel	Level of the lambda logger.

If you execute the script, you can see in the standard output the process connecting to the test AWS account and creating the necessary stuff:



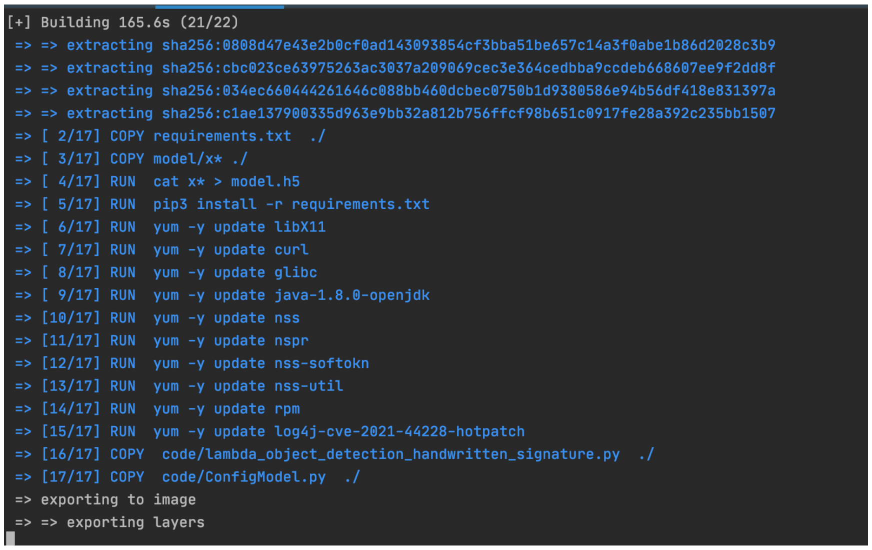
```
/bin/bash  /Users/diedue/Desktop/Amazon/Artifacts/LambdaSignatureDocker/lambda_object_detection_handwritten_signature/deploy.sh

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
38f9d32e4189:lambda_object_detection_handwritten_signature diedue$ /bin/bash  /Users/diedue/Desktop/Amazon/Artifacts/LambdaSignatureDocker/lambda_object_detection_handwritten_signature/deploy.sh
Executing with profile: p1
Executing in region: us-east-1
app001-app1-dev-s3-temp-cloudformation
cloudformation temp bucket already exists
Building the ecr stack
2022-01-11 17:02:44,590 - MainThread - awscli.clidriver - DEBUG - CLI version: aws-cli/2.4.3 Python/3.8.8 Darwin/20.6.0 exe/x86_64
2022-01-11 17:02:44,590 - MainThread - awscli.clidriver - DEBUG - Arguments entered to CLI: ['cloudformation', 'package', '--template-file', '/Users/diedue/Desktop/Amazon/Artifacts/LambdaSignatureDocker
/lambda_object_detection_handwritten_signature/lambda_object_detection_handwritten_signature_ecr.yaml', '--s3-bucket', 'app001-app1-dev-s3-temp-cloudformation', '--output-template-file', 'output-custom-
ecr-template.yaml', '--profile', 'p1', '--debug']
2022-01-11 17:02:44,621 - MainThread - botocore.hooks - DEBUG - Event building-command-table.main: calling handler <function add_s3 at 0x7f8cbbf499d0>
```

First, it will create a first stack that creates an ECR repository in the test account, and then starts to download the public lambda image with python 3.7:



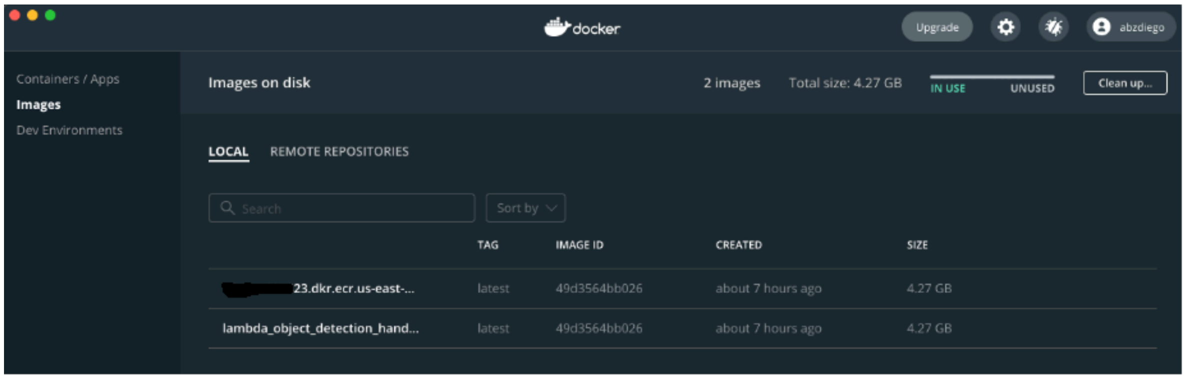
```
Successfully packaged artifacts and wrote output template to file output-custom-ecr-template.yaml.
Execute the following command to deploy the packaged template
aws cloudformation deploy --template-file /Users/diedue/Desktop/Amazon/Artifacts/LambdaSignatureDocker/lambda_object_detection_handwritten_signature/output-custom-ecr-template.yaml --stack-name <YOUR ST
ACK NAME>
Updating ecr Stack

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - app001-app1-dev-lambda-object-detection-handwritten-signature-ECR-stack
[+] Building 127.5s (9/21)
 => [internal] load build context                                                                                                          0.0s
 => => transferring context: 8.24kB                                                                                                        0.0s
 => [ 1/17] FROM public.ecr.aws/lambda/python:3.7@sha256:5654e7240dccdb38fc646575022812322b96510eeac1f1e6c7747bf0ffb2b6cc                   6.5s
 => => resolve public.ecr.aws/lambda/python:3.7@sha256:5654e7240dccdb38fc646575022812322b96510eeac1f1e6c7747bf0ffb2b6cc                     0.0s
 => => sha256:36d666291a874542544feac00d42fb61b73f583b6c87b76d05155cab3964cfc9 81.50kB / 81.50kB                                          0.3s
 => => sha256:0808d47e43e2b0cf0ad143093854cf3bba51be657c14a3f0abe1b86d2028c3b9 417B / 417B                                                0.4s
 => => sha256:cbc023ce63975263ac3037a209069cec3e364cedbba9ccdeb668607ee9f2dd8f 2.30MB / 2.30MB                                            0.8s
 => => sha256:5654e7240dccdb38fc646575022812322b96510eeac1f1e6c7747bf0ffb2b6cc 1.58kB / 1.58kB                                            0.0s
 => => sha256:af9fe7f90505821a4eeca4344c04b2c42f324f3279c3d3f4a86945dc51fb841f 3.00kB / 3.00kB                                            0.0s
 => => extracting sha256:36d666291a874542544feac00d42fb61b73f583b6c87b76d05155cab3964cfc9                                                  0.2s
 => => sha256:034ec660444261646c088bb460dcbec0750b1d9380586e94b56df418e831397a 48.59MB / 48.59MB                                          2.8s
 => => sha256:c1ae137900335d963e9bb32a812b756ffcf98b651c0917fe28a392c235bb1507 11.02MB / 11.02MB                                          2.0s
 => => extracting sha256:0808d47e43e2b0cf0ad143093854cf3bba51be657c14a3f0abe1b86d2028c3b9                                                  0.0s
 => => extracting sha256:cbc023ce63975263ac3037a209069cec3e364cedbba9ccdeb668607ee9f2dd8f                                                  0.1s
```

After that, the script will build the docker image and run all the commands specified in the docker file:



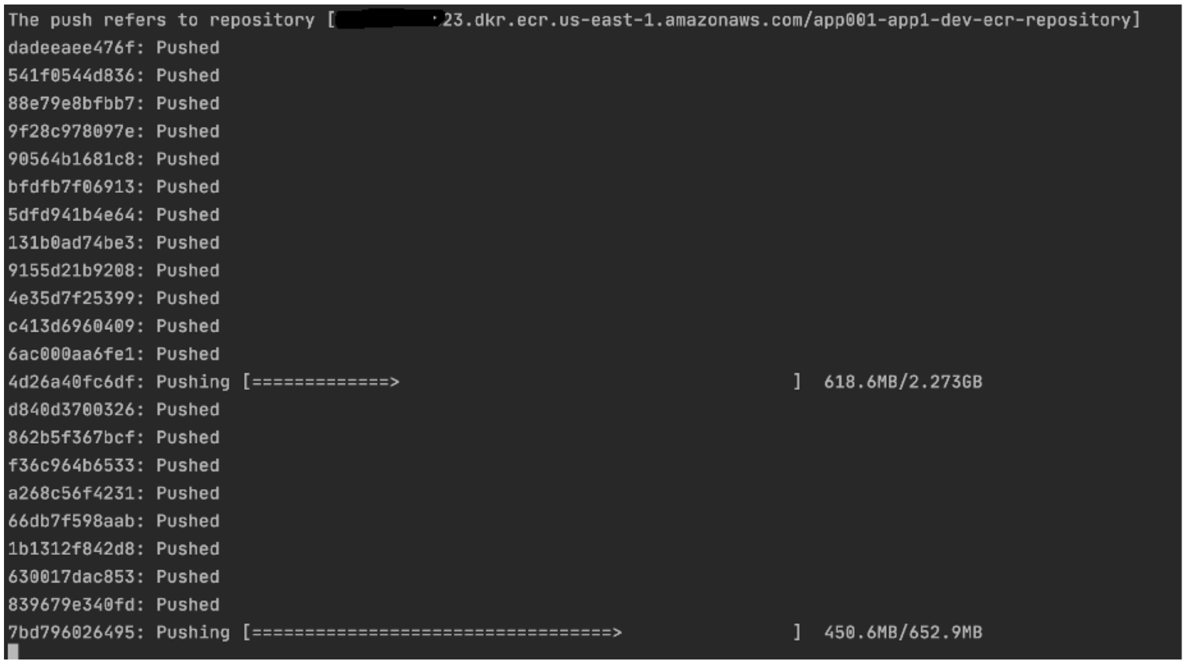
```
[+] Building 165.6s (21/22)
 => => extracting sha256:0808d47e43e2b0cf0ad143093854cf3bba51be657c14a3f0abe1b86d2028c3b9
 => => extracting sha256:cbc023ce63975263ac3037a209069cec3e364cedbba9ccdeb668607ee9f2dd8f
 => => extracting sha256:034ec660444261646c088bb460dcbec0750b1d9380586e94b56df418e831397a
 => => extracting sha256:c1ae137900335d963e9bb32a812b756ffcf98b651c0917fe28a392c235bb1507
 => [ 2/17] COPY requirements.txt  ./
 => [ 3/17] COPY model/x* ./
 => [ 4/17] RUN  cat x* > model.h5
 => [ 5/17] RUN  pip3 install -r requirements.txt
 => [ 6/17] RUN  yum -y update libX11
 => [ 7/17] RUN  yum -y update curl
 => [ 8/17] RUN  yum -y update glibc
 => [ 9/17] RUN  yum -y update java-1.8.0-openjdk
 => [10/17] RUN  yum -y update nss
 => [11/17] RUN  yum -y update nspr
 => [12/17] RUN  yum -y update nss-softokn
 => [13/17] RUN  yum -y update nss-util
 => [14/17] RUN  yum -y update rpm
 => [15/17] RUN  yum -y update log4j-cve-2021-44228-hotpatch
 => [16/17] COPY  code/lambda_object_detection_handwritten_signature.py  ./
 => [17/17] COPY  code/ConfigModel.py  ./
 => exporting to image
 => => exporting layers
```

You can now see in the docker desktop software the images and docker tags created:



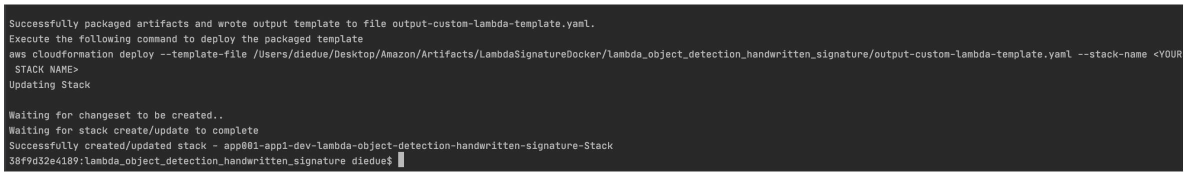
	TAG	IMAGE ID	CREATED	SIZE
[redacted]23.dkr.ecr.us-east-...	latest	49d3564bb026	about 7 hours ago	4.27 GB
lambda_object_detection_hand...	latest	49d3564bb026	about 7 hours ago	4.27 GB

After the docker image creation, it executes the docker push command, that uploads the image to the docker repository:



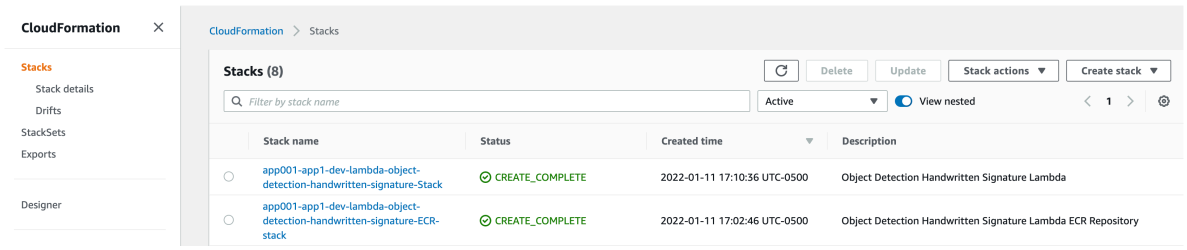
```
The push refers to repository [[redacted]23.dkr.ecr.us-east-1.amazonaws.com/app001-app1-dev-ecr-repository]
dadeeaee476f: Pushed
541f0544d836: Pushed
88e79e8bfbb7: Pushed
9f28c978097e: Pushed
90564b1681c8: Pushed
bfdfb7f06913: Pushed
5dfd941b4e64: Pushed
131b0ad74be3: Pushed
9155d21b9208: Pushed
4e35d7f25399: Pushed
c413d6960409: Pushed
6ac000aa6fe1: Pushed
4d26a40fc6df: Pushing [=============>                                   ]  618.6MB/2.273GB
d840d3700326: Pushed
862b5f367bcf: Pushed
f36c964b6533: Pushed
a268c56f4231: Pushed
66db7f598aab: Pushed
1b1312f842d8: Pushed
630017dac853: Pushed
839679e340fd: Pushed
7bd796026495: Pushing [=================================>               ]  450.6MB/652.9MB
```

At the end you will see a message saying that all the process was successful.



```
Successfully packaged artifacts and wrote output template to file output-custom-lambda-template.yaml.
Execute the following command to deploy the packaged template
aws cloudformation deploy --template-file /Users/diedue/Desktop/Amazon/Artifacts/LambdaSignatureDocker/lambda_object_detection_handwritten_signature/output-custom-lambda-template.yaml --stack-name <YOUR
 STACK NAME>
Updating Stack

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - app001-app1-dev-lambda-object-detection-handwritten-signature-Stack
38f9d32e4189:lambda_object_detection_handwritten_signature diedue$
```

And if you explore the cloudformation console in the test AWS account, you will see the 2 cloudformation stacks created:

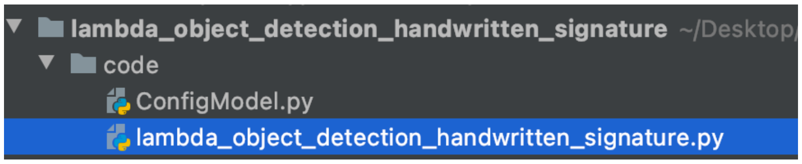


Stacks (8)

Stack name	Status	Created time	Description
app001-app1-dev-lambda-object-detection-handwritten-signature-Stack	CREATE_COMPLETE	2022-01-11 17:10:36 UTC-0500	Object Detection Handwritten Signature Lambda
app001-app1-dev-lambda-object-detection-handwritten-signature-ECR-stack	CREATE_COMPLETE	2022-01-11 17:02:46 UTC-0500	Object Detection Handwritten Signature Lambda ECR Repository

Code Explanation

The demo code provided is in the "code" folder.



```
lambda_object_detection_handwritten_signature ~/Desktop/
  code
    ConfigModel.py
    lambda_object_detection_handwritten_signature.py
```

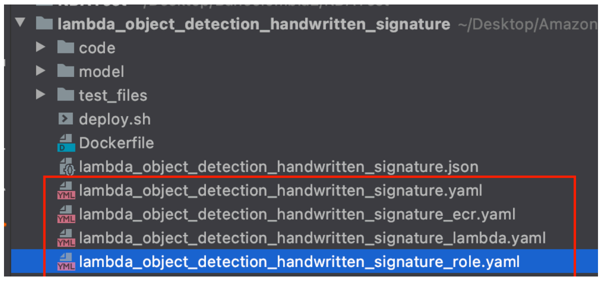
The ConfigModel.py file contains all the specific parameters used by the ML model. The lambda_object_detection_handwritten_signature.py file contains the lambda code, divided in the following methods:

Method	Description
lambda_handler	main method that orchestrates ml model call and sends a response.
detect_signature	method that process a given file name (s3 path),validates its extension , download the file and apply the ml model
download_file	method that downloads a file from s3 path, and save it in a given local path
apply_model	method that converts the image to jpg if necessary, and run the tensorflow - keras model. based on the "rois" array, defines if the image has handwritten signature or not
initialize_model	method that initialize the MaskRCNN H5 model if necessary
convert_png_to_jpg	method that converts the given image from png to jpg
send_notification	method that sends a SNS notification in case of any error in the lambda.

Note that the "lambda_handler" method is the same refered in the "CMD" section of the docker file.

Cloudformation Explanation

The deploy.sh script uses 4 cloudformations:



```
▼ lambda_object_detection_handwritten_signature ~/Desktop/Amazon
  ▶ code
  ▶ model
  ▶ test_files
    deploy.sh
    Dockerfile
    lambda_object_detection_handwritten_signature.json
    lambda_object_detection_handwritten_signature.yaml
    lambda_object_detection_handwritten_signature_ecr.yaml
    lambda_object_detection_handwritten_signature_lambda.yaml
    lambda_object_detection_handwritten_signature_role.yaml
```

The first one is an orchestrator between the ecr repository cloudformation, the lambda cloudformation and the lambda role cloudformation. Here it will be explained the most important sections on the cloudformation templates. First the ECR repository template:

```
LambdaObjectDetectionHandwrittenSignatureECRRepo:
  Type: AWS::ECR::Repository
  Properties:
    RepositoryName:
      "Fn::Sub": "${pApplicationID}-${pApplicationName}-${pEnvironment}-
ecr-repository"
    ImageScanningConfiguration:
      ScanOnPush: true
    ImageTagMutability: MUTABLE
```

The “ScanOnPush:true” configuration allows the ECR repository to scan the docker image for vulnerabilities. The “ImageTagMutability” indicates if the image tag can be changed or not.

Then we have the lambda section:

```
Type: AWS::Serverless::Function
Properties:
  FunctionName:
      "Fn::Sub": "${pApplicationID}-${pApplicationName}-${pEnvironment}-
ObjectDetectionHandwrittenSignature"
  PackageType: Image
  ImageUri:
    "Fn::Sub":
"${AWS::AccountId}.dkr.ecr.${AWS::Region}.amazonaws.com/${pApplicationID}
-${pApplicationName}-${pEnvironment}-ecr-repository:latest"
  MemorySize: 10240
  Timeout: 900
  Role:
    "Fn::GetAtt": RoleLambdaObjectDetectionHandwrittenSignature.Arn
  Environment:
    Variables:
      LOG_LEVEL:
         Ref: pLogLevel
      SNS_ERROR_NAME:
         "Fn::Sub": "${pApplicationID}-${pApplicationName}-
```

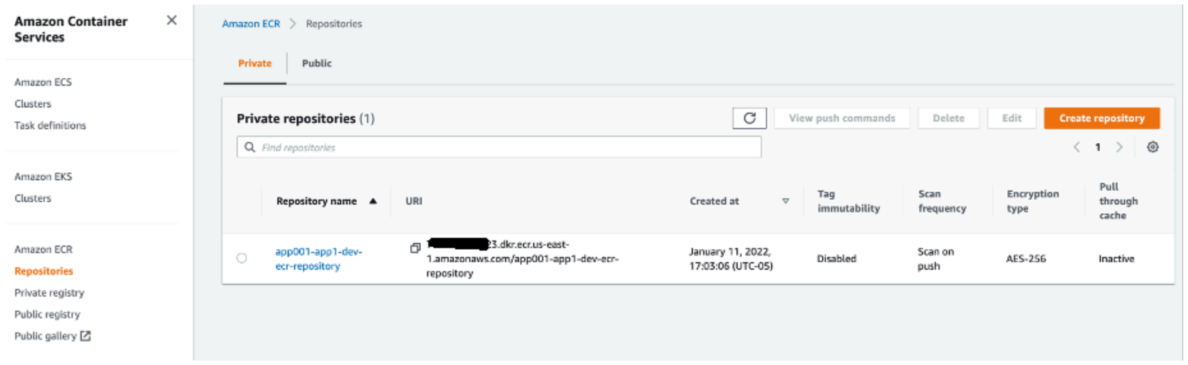
```
${pEnvironment}-${pSNStopicName}"
      FILE_BUCKET :
         "Fn::Sub": "${pApplicationID}-${pApplicationName}-
${pEnvironment}-${pS3RawBucket}"
```

The “ImageUri” configuration is the key part on this section, and is pointing to the image tag created in the ECR repository. The MemorySize of the lambda is configured at the maximum allowed in lambda for test purposes, but you can accommodate at your convenience. It has also some environment variables related with the lambda logger level, s3 image bucket and SNS error topic.

The lambda role cloudformation creates a lambda role with the necessary permissions so lambda can access the s3 bucket and the SNS error topic.

Lambda configuration after deployment

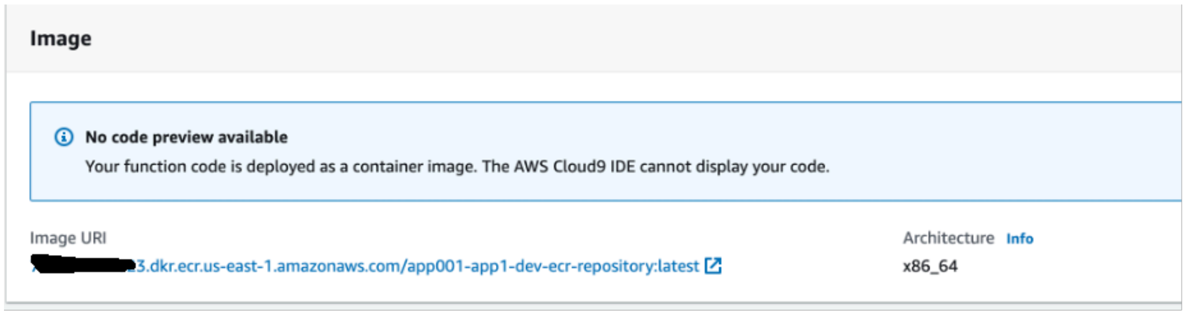
After the deploy.sh execution, you can see the lambda docker image published in the ecr repository created.



Private repositories (1)

Repository name	URI	Created at	Tag immutability	Scan frequency	Encryption type	Pull through cache
app001-app1-dev-ecr-repository	[redacted]3.dkr.ecr.us-east-1.amazonaws.com/app001-app1-dev-ecr-repository	January 11, 2022, 17:03:06 (UTC-05)	Disabled	Scan on push	AES-256	Inactive

And also, you will see the docker image URI attached to the lambda:



Image

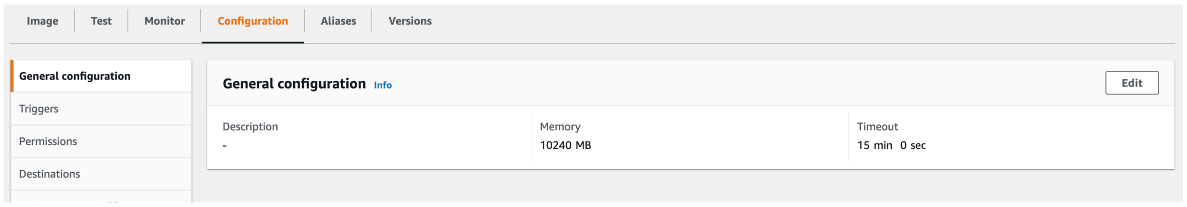
No code preview available

Your function code is deployed as a container image. The AWS Cloud9 IDE cannot display your code.

Image URI
[redacted]3.dkr.ecr.us-east-1.amazonaws.com/app001-app1-dev-ecr-repository:latest

Architecture Info
x86_64

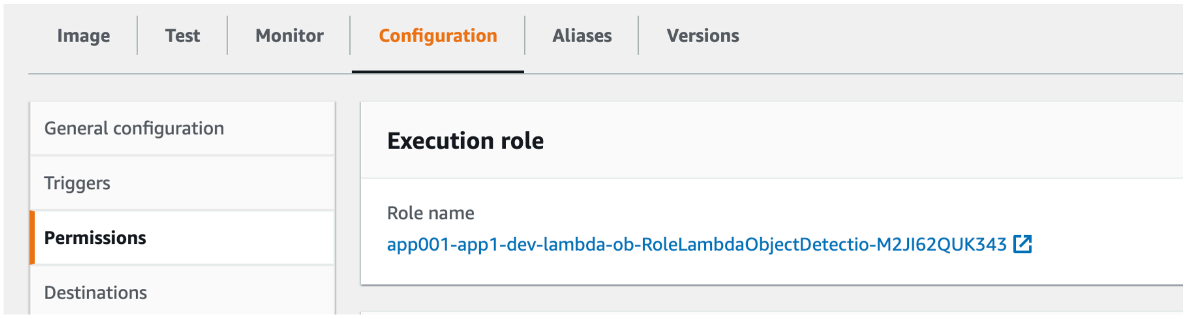
As mentioned earlier, the lambda was configured with the lambda maximums for test purposes, but as shown in the lambda execution section later, it doesn’t consume all the resources configured.



General configuration Info

Description	Memory	Timeout
-	10240 MB	15 min 0 sec

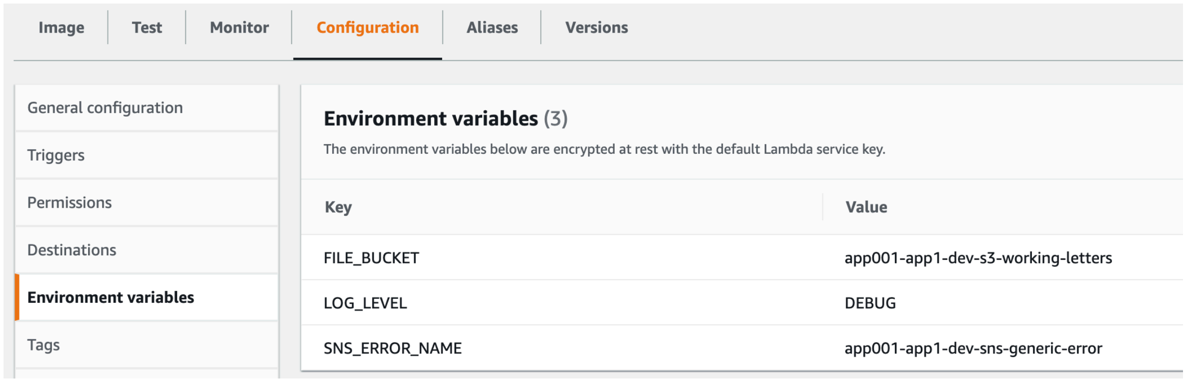
You will see the lambda role created and attached to the lambda:



Execution role

Role name
app001-app1-dev-lambda-ob-RoleLambdaObjectDetectio-M2JI62QUK343

And the environment variables referenced in the cloudformation template:



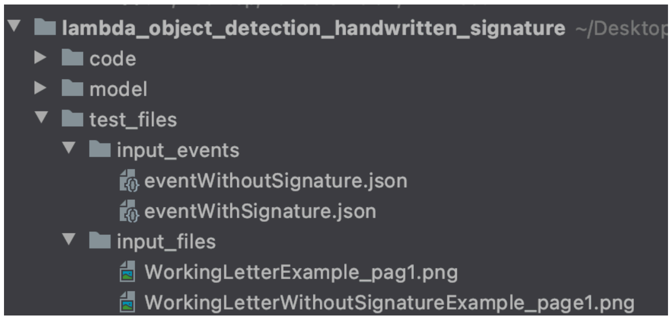
Environment variables (3)

The environment variables below are encrypted at rest with the default Lambda service key.

Key	Value
FILE_BUCKET	app001-app1-dev-s3-working-letters
LOG_LEVEL	DEBUG
SNS_ERROR_NAME	app001-app1-dev-sns-generic-error

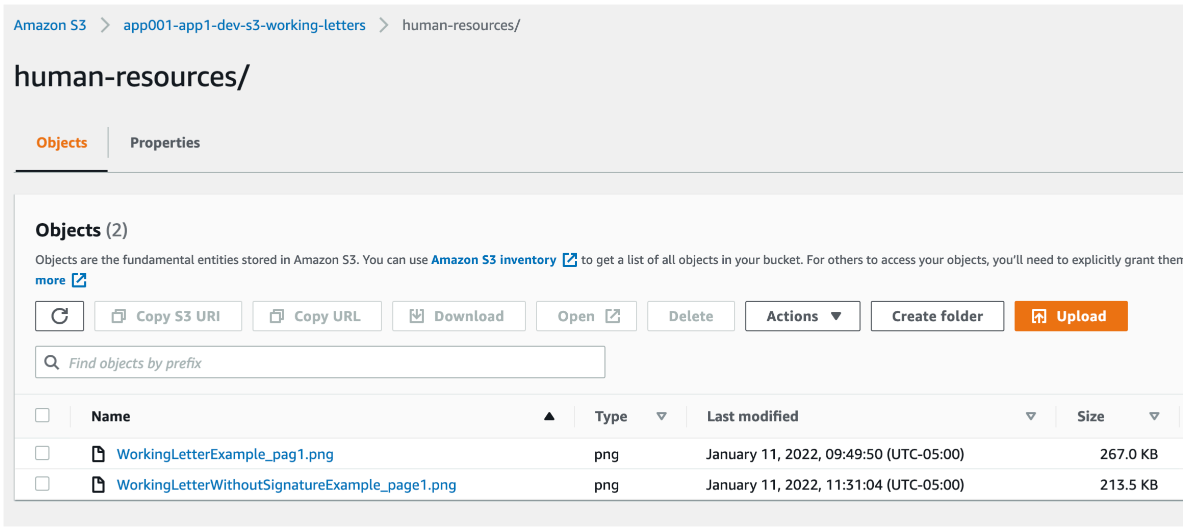
Executing the lambda

For testing purposes, in the attached folder there is a subfolder called “test_files” where you can find some test files and lambda json input events to test the lambda execution:



```
▼ lambda_object_detection_handwritten_signature ~/Desktop
  ▶ code
  ▶ model
  ▼ test_files
    ▼ input_events
        eventWithoutSignature.json
        eventWithSignature.json
    ▼ input_files
        WorkingLetterExample_pag1.png
        WorkingLetterWithoutSignatureExample_page1.png
```

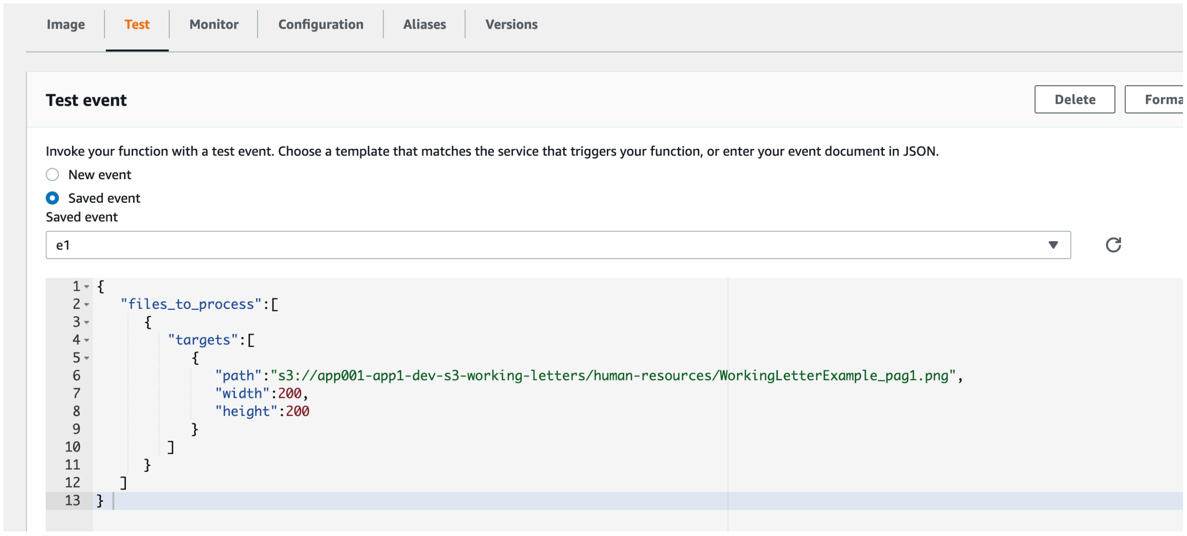
In order to execute the lambda, you need first to upload the test files to the s3 bucket:



Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them more

Name	Type	Last modified	Size
WorkingLetterExample_pag1.png	png	January 11, 2022, 09:49:50 (UTC-05:00)	267.0 KB
WorkingLetterWithoutSignatureExample_page1.png	png	January 11, 2022, 11:31:04 (UTC-05:00)	213.5 KB

After that you can use the test input events to execute the lambda. First we can use the example that we know contains a handwritten signature:



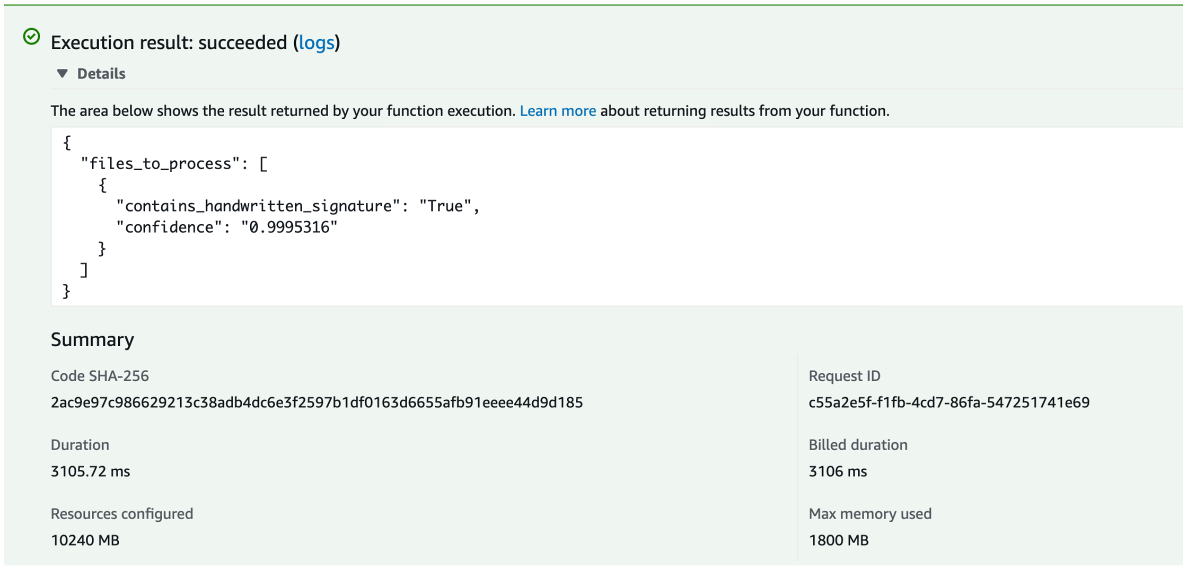
Invoke your function with a test event. Choose a template that matches the service that triggers your function, or enter your event document in JSON.

New event
Saved event [✓]

Saved event: e1

```
{
    "files_to_process":[
      {
        "targets":[
          {
            "path":"s3://app001-app1-dev-s3-working-letters/human-resources/WorkingLetterExample_pag1.png",
            "width":200,
            "height":200
          }
        ]
      }
    ]
}
```

And this is the execution result:


Execution result: succeeded (logs)

Details

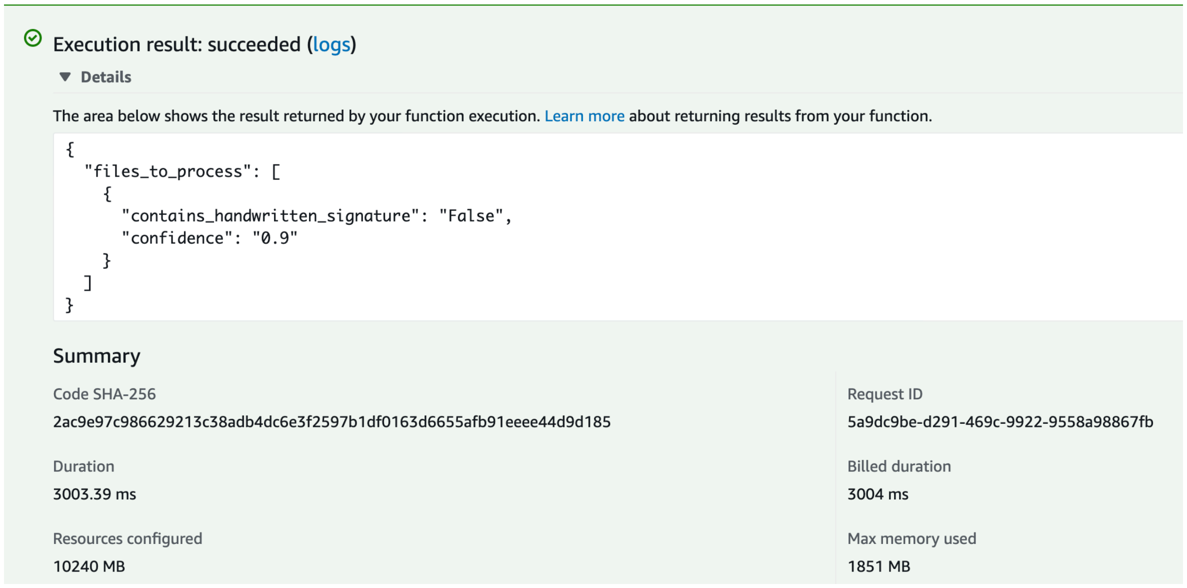
The area below shows the result returned by your function execution. Learn more about returning results from your function.

```
{
  "files_to_process": [
    {
      "contains_handwritten_signature": "True",
      "confidence": "0.9995316"
    }
  ]
}
```

Summary

Code SHA-256	Request ID
2ac9e97c986629213c38adb4dc6e3f2597b1df0163d6655afb91eeee44d9d185	c55a2e5f-f1fb-4cd7-86fa-547251741e69
Duration	Billed duration
3105.72 ms	3106 ms
Resources configured	Max memory used
10240 MB	1800 MB

Here we can see that the lambda detects that the image has a handwritten signature with a confidence of 0.999. In the case we execute the other example without signature, the lambda output will be:


Execution result: succeeded (logs)

Details

The area below shows the result returned by your function execution. Learn more about returning results from your function.

```
{
  "files_to_process": [
    {
      "contains_handwritten_signature": "False",
      "confidence": "0.9"
    }
  ]
}
```

Summary

Code SHA-256	Request ID
2ac9e97c986629213c38adb4dc6e3f2597b1df0163d6655afb91eeee44d9d185	5a9dc9be-d291-469c-9922-9558a98867fb
Duration	Billed duration
3003.39 ms	3004 ms
Resources configured	Max memory used
10240 MB	1851 MB

Note that once the lambda docker image is loaded in the lambda workers, the execution time is short, around 3 seconds. The first time it can take up to 30 second to execute. Also we can note that the lambda memory used is around 1.8 GB, which is far below the lambda maximum configured.

Advantages / Disadvantages of using this lambda docker approach

keep in mind the following advantages / disadvantages when using this approach

Advantages

- Totally serverless approach. No need to provision external servers.
- You can run docker images up to 10 GB, breaking the 250 mb traditional lambda limit .
- More suitable to cases where the code was pre built. For ML new cases could be better to use sagemaker endpoint instead.
- You can load any type of library that you can use in a normal python environment, and control specific software versions.

Disadvantages

- You can’t modify the code on the fly. You need to do a new docker push.
- It can be expensive if used frequently, especially if the underlying model consumes huge amount of memory.

Conclusion

Regardless if it is the optimal solution to run a ML model, this artifact proves that is totally possible to run heavy machine learning models in a lambda environment, using the docker image technology.