High Performance and Distributed Computing for Big Data

Unit 3: AWS - Lambda

Ferran Aran Domingo ferran.aran@udl.cat

Universitat Rovira i Virgili and Universitat de Lleida

Cloud functions

Cloud functions

Cloud functions enable **serverless** computing, allowing developers to run code without provisioning or managing servers.

Cloud functions

Cloud functions enable **serverless** computing, allowing developers to run code without provisioning or managing servers.

Example

Automatically processing patient data uploads, triggering real-time alerts, and updating medical dashboards without infrastructure management.

Serverless?

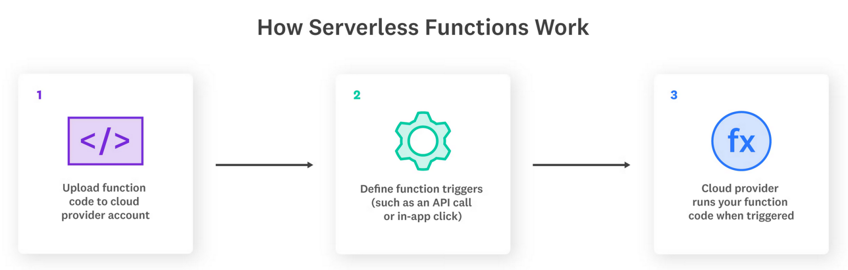

Figure 1: Serverless model

Providers

Figure 2: Serverless providers

AWS Lambda

Introduction to AWS Lambda

What is AWS Lambda?

- Event-driven, serverless computing service.
- Runs code in response to triggers.

Introduction to AWS Lambda

What is AWS Lambda?

- Event-driven, serverless computing service.
- Runs code in response to triggers.

Key benefits

- Automatic scaling and high availability.
- Pay-per-use billing model.

Introduction to AWS Lambda

What is AWS Lambda?

- Event-driven, serverless computing service.
- Runs code in response to triggers.

Key benefits

- Automatic scaling and high availability.
- Pay-per-use billing model.

Common use cases

- Real-time file processing (e.g., medical imaging).
- Backend for web and mobile health applications.

Introduction to AWS Lambda

What is AWS Lambda?

- Event-driven, serverless computing service.
- Runs code in response to triggers.

Key benefits

- Automatic scaling and high availability.
- Pay-per-use billing model.

Common use cases

- Real-time file processing (e.g., medical imaging).
- Backend for web and mobile health applications.

Supported languages

- Python, Node.js, Java, Go, Ruby, .NET, and more.

AWS Lambda architecture overview

Figure 3: Lambda Architecture

AWS Lambda architecture overview

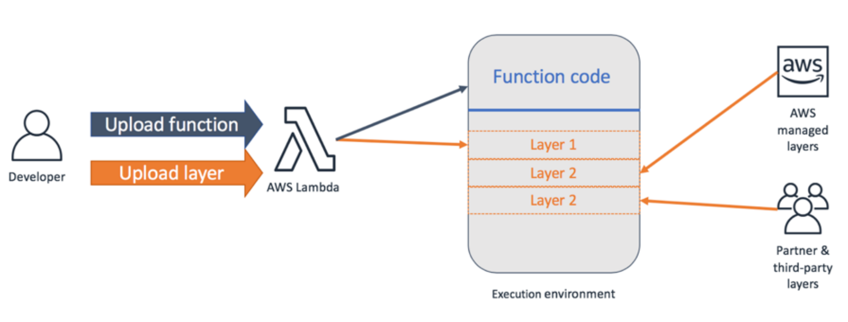

Figure 4: Lambda Architecture

AWS Lambda architecture overview

Core components

- Code (Function)
- Layers (Dependencies)
- Event Sources (S3, API Gateway, CloudWatch, etc.)

Event-driven execution

- Code executes in response to events.
- Easily integrates with other AWS services.

AWS Lambda architecture overview

Core components

- Code (Function)
- Layers (Dependencies)
- Event Sources (S3, API Gateway, CloudWatch, etc.)

Event-driven execution

- Code executes in response to events.
- Easily integrates with other AWS services.

> Did you know? AWS Lambda functions have a maximum execution time limit of 15 minutes per invocation.

How AWS Lambda compares to EC2

AWS EC2

- Full control over infrastructure (turn it on/off, upgrading).
- Manual scalability management (want more? You have to add more).
- Fixed cost for uptime.

How AWS Lambda compares to EC2

AWS EC2

- Full control over infrastructure (turn it on/off, upgrading).
- Manual scalability management (want more? You have to add more).
- Fixed cost for uptime.

AWS Lambda

- No infrastructure management (serverless).
- Automatic scalability (from zero to thousands).
- Pay for actual execution time.

Use cases for AWS Lambda

- **Real-time data processing**: Lambda can analyze sensor data from wearable devices to identify irregular patterns.

Use cases for AWS Lambda

- **Real-time data processing**: Lambda can analyze sensor data from wearable devices to identify irregular patterns.
- **Data aggregation**: Lambda can collect data from multiple clinical trial sites, aggregating information on patient outcomes, adverse events, and treatment efficacy preparing a dashboard for the researchers.

Use cases for AWS Lambda

- **Real-time data processing**: Lambda can analyze sensor data from wearable devices to identify irregular patterns.
- **Data aggregation**: Lambda can collect data from multiple clinical trial sites, aggregating information on patient outcomes, adverse events, and treatment efficacy preparing a dashboard for the researchers.
- **Data validation**: Lambda can validate lab results (such as blood tests) by checking for outliers or inconsistencies. For example, abnormal values outside the expected range may require further investigation. Alerts can be sent to the lab technician or the patient's healthcare provider.

Use cases for AWS Lambda

- **Real-time data processing**: Lambda can analyze sensor data from wearable devices to identify irregular patterns.
- **Data aggregation**: Lambda can collect data from multiple clinical trial sites, aggregating information on patient outcomes, adverse events, and treatment efficacy preparing a dashboard for the researchers.
- **Data validation**: Lambda can validate lab results (such as blood tests) by checking for outliers or inconsistencies. For example, abnormal values outside the expected range may require further investigation. Alerts can be sent to the lab technician or the patient's healthcare provider.
- **Image processing**: Lambda can process images to identify patterns or anomalies.

Example: Counting cells at scale

Scenario

Imagine a lab that generates a large collection of cell images every time they run an experiment. Once the experiment is done, they want to know the number of cells in each image to analyze the results but they want this process to be automated and immediate.

Example: Counting cells at scale

Scenario

Imagine a lab that generates a large collection of cell images every time they run an experiment. Once the experiment is done, they want to know the number of cells in each image to analyze the results but they want this process to be automated and immediate.

Workflow

1. A lab uploads a collection of cell images to S3.

Example: Counting cells at scale

Scenario

Imagine a lab that generates a large collection of cell images every time they run an experiment. Once the experiment is done, they want to know the number of cells in each image to analyze the results but they want this process to be automated and immediate.

Workflow

1. A lab uploads a collection of cell images to S3.
2. S3 events trigger Lambda functions (one event per image, one function per event).

Example: Counting cells at scale

Scenario

Imagine a lab that generates a large collection of cell images every time they run an experiment. Once the experiment is done, they want to know the number of cells in each image to analyze the results but they want this process to be automated and immediate.

Workflow

1. A lab uploads a collection of cell images to S3.
2. S3 events trigger Lambda functions (one event per image, one function per event).
3. Lambda functions process the images and count the cells.

Example: Counting cells at scale

Scenario

Imagine a lab that generates a large collection of cell images every time they run an experiment. Once the experiment is done, they want to know the number of cells in each image to analyze the results but they want this process to be automated and immediate.

Workflow

1. A lab uploads a collection of cell images to S3.
2. S3 events trigger Lambda functions (one event per image, one function per event).
3. Lambda functions process the images and count the cells.
4. Results are stored in S3.

Example: Counting cells at scale

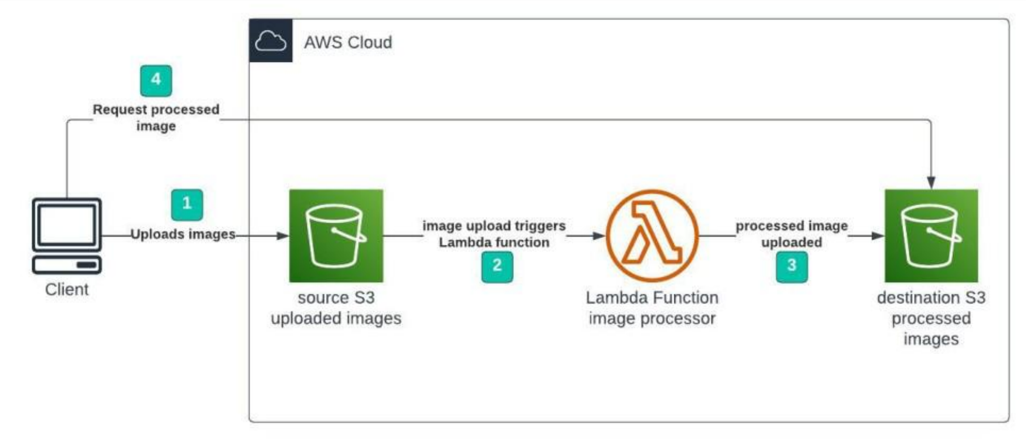

Figure 5: Health Data Flow

Lab: Counting cells at scale

Outline

Pre-requisites

- A machine with AWS Credentials configured and Python 3.13 installed. (I am going to use the EC2 instance we created in the previous unit together with `uv` for managing python versions).
- The cell images downloaded and extracted, find them here https://campusvirtual.urv.cat/ or on the subject's website https://hdbc-17705110-mdbs.github.io.

Goal

- Upload a collection of cell images to an S3 bucket and trigger a Lambda function to count the cells in each image. The lambda will store the results in another S3 bucket.

Outline

Steps

1. Create the buckets.
2. Create the Lambda function.
3. Add a trigger to the Lambda function.
4. Write the Lambda function code.
5. Create and publish a Lambda layer with the dependencies.
6. Upload the images to the input bucket.
7. Check the results in the output bucket and verify the Lambda logs.

Step 1: Create the buckets

As we did in the previous session, we are going to visit the S3 service in the AWS console and create two buckets, one for the input images and another for the output results. Leave everything as default and just set the name for each one as shown below:

- Input bucket: `medical-images-raw-[YOUR-NAME]`
- Output bucket: `medical-images-processed-[YOUR-NAME]`

In my case that will be `medical-images-raw-ferran-aran` and `medical-images-processed-ferran-aran`.

S3 Bucket names

Remember S3 bucket names must be unique across all AWS accounts. If you get an error when creating the bucket, try a different name (e.g., add a random number at the end).

Step 2: Create the Lambda function

We are going to search for `lambda` in the AWS Console as usual and click on the first result.

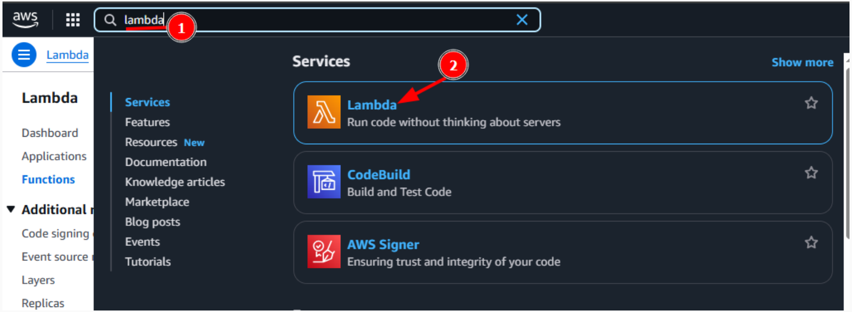

Figure 6: Lambda search

Step 2: Create the Lambda function

Now click on create function.

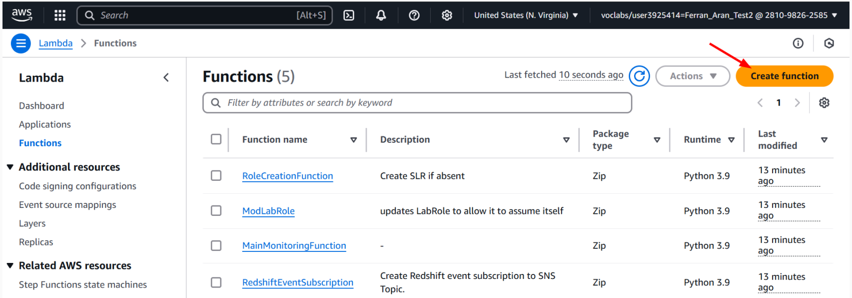

Figure 7: Create function

Step 2: Create the Lambda function

And fill the form like shown below (the function name doesn't matter but I suggest you use `count-cells`):

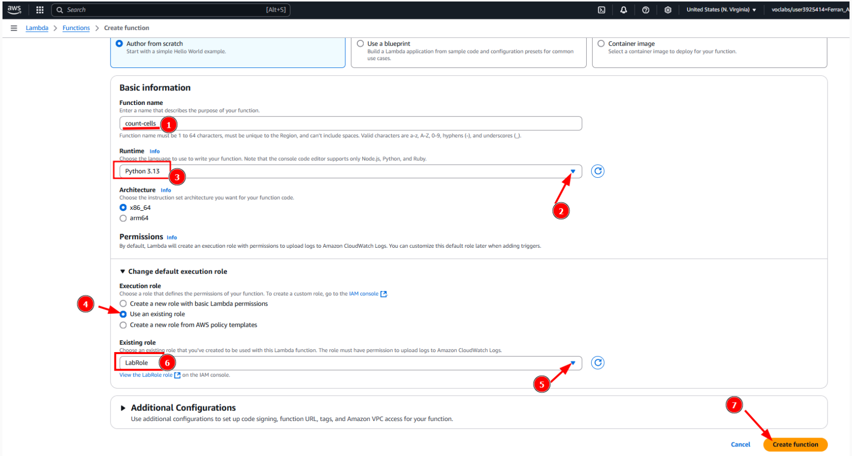

Figure 8: Create function

Step 3: Add a trigger to the Lambda function

If we want our Lambda function to be executed when a new image is uploaded to the input bucket, we need to add a trigger. Click on the `+ Add trigger` button.

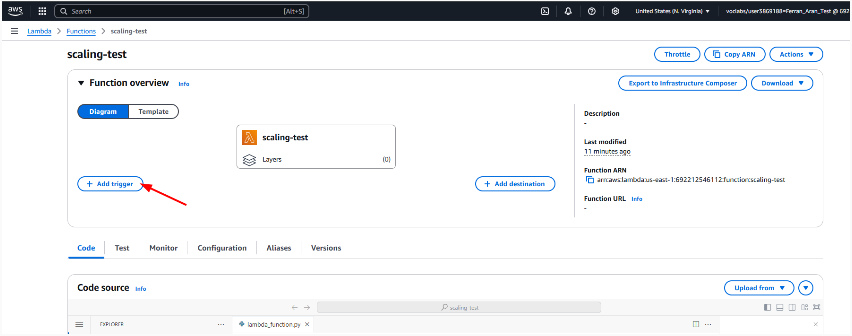

Figure 9: Add trigger

Step 3: Add a trigger to the Lambda function

Start by searching for `S3` in the trigger configuration and then fill in the form like shown below:

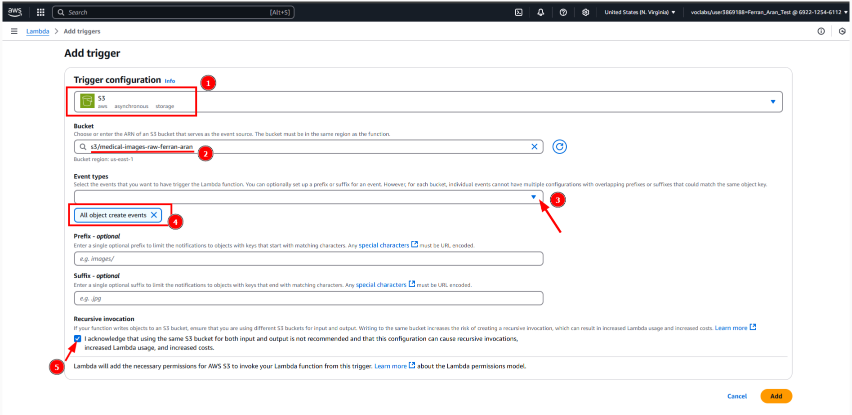

Figure 10: Add trigger

Step 3: Add a trigger to the lambda function

Okay so we've now configured our lambda to be triggered when a new object is created in the input bucket. But how do we access the image in the bucket from the lambda function?

Since we have configured the trigger to be an S3 event, AWS is going to send a JSON object to the lambda function with the information about the event.

Go back to the code tab as shown below.

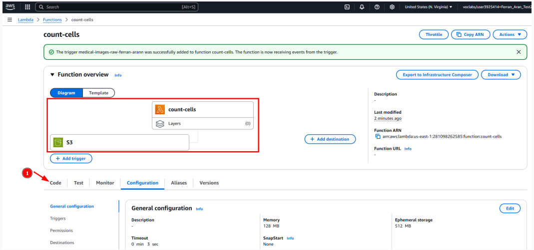

Figure 11: Code tab

Step 3: Add a trigger to the lambda function

Take a look at the default code that came with our lambda function:

```python
import json

def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

This is the basic structure of a lambda function. The `lambda_handler` function is the entry point of the lambda and it receives two arguments: `event` and `context`. The `event` argument is the JSON object we were talking about that contains the information about the event that triggered the lambda. So **anything we want to do with our lambda function has to be done inside this function**.

Step 3: Add a trigger to the lambda function

Let's see how the event looks like by printing it:

```
import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

Step 3: Add a trigger to the lambda function

Once we are happy with the code, we need to “save” the changes by clicking on the `Deploy` button as shown below.

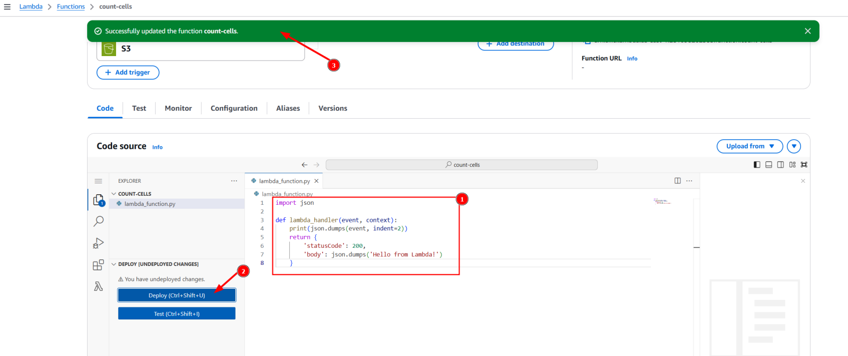

Figure 12: Deploy

Step 3: Add a trigger to the lambda function

Now we have to trigger the lambda function by uploading an image to the input bucket. You can do this by visiting the S3 service in the AWS Console and clicking on the input bucket `medical-images-raw-[YOUR-NAME]` we created earlier. Then click on `Upload` and select an image to upload.

We can now go back our lambda, click on `Monitoring` and then on `View logs in CloudWatch` to see the logs of the lambda function.

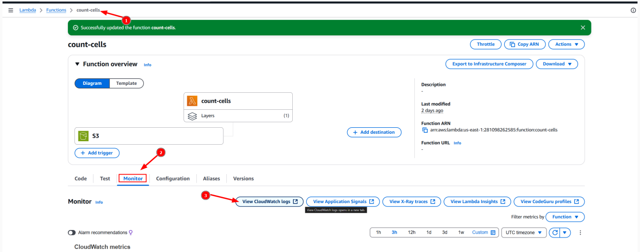

Figure 13: CloudWatch logs

Step 3: Add a trigger to the lambda function

Click on the latest log stream to see the logs of the lambda function.

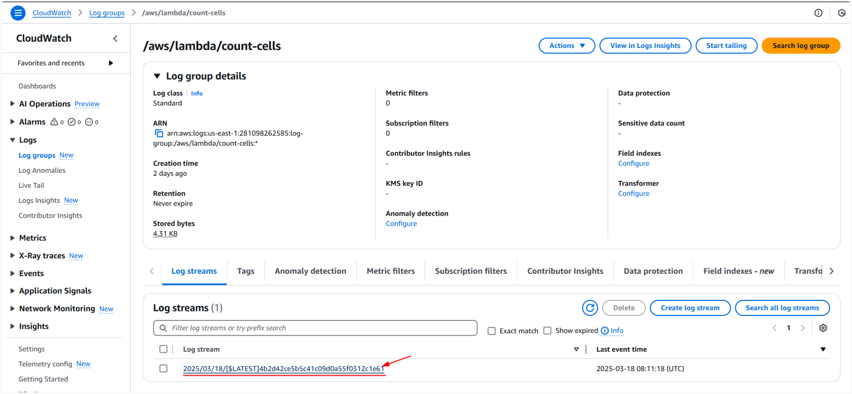

Figure 14: CloudWatch logs

Step 3: Add a trigger to the lambda function

If everything went well you should see the event printed in the logs in the form of a JSON. There is lots of information but we are just interested in a couple of fields; the S3 bucket name and the object key.


```
2025-03-18T08:11:18.800Z    },
2025-03-18T08:11:18.800Z    "s3": {
2025-03-18T08:11:18.800Z    "s3SchemaVersion": "1.0",
2025-03-18T08:11:18.800Z    "configurationId": "f22958d6-15e6-4a93-b535-c2cd60c26f04",
2025-03-18T08:11:18.800Z    "bucket": {
2025-03-18T08:11:18.800Z    "name": "medical-images-raw-ferran-arann",
2025-03-18T08:11:18.800Z    "ownerIdentity": {
2025-03-18T08:11:18.800Z    "principalId": "AIE4BDYMJFCTZ"
2025-03-18T08:11:18.800Z    },
2025-03-18T08:11:18.800Z    "arn": "arn:aws:s3:::medical-images-raw-ferran-arann"
2025-03-18T08:11:18.800Z    },
2025-03-18T08:11:18.800Z    "object": {
2025-03-18T08:11:18.800Z    "key": "image-1.png",
2025-03-18T08:11:18.800Z    "size": 133675,
2025-03-18T08:11:18.800Z    "eTag": "e326aa977c8ba59ab4f3c943d0b1cb03",
2025-03-18T08:11:18.800Z    "sequencer": "0067D92AA3991560D4"
2025-03-18T08:11:18.800Z    }
```


Figure 15: CloudWatch logs

Step 3: Add a trigger to the lambda function

Okay now that we know we have the information we need to access the image in the S3 bucket, we can write some template code that accesses the image on the bucket that triggered the lambda and saves it to the `processed` bucket.

We'll be using the `boto3` library to interact with S3 as we did in the previous session. Go back to your lambda function on the "Code" tab and paste the following code. **Remember to change the bucket name** (note that the code takes 2 slides to fit):

```
import boto3
import json
import os
import urllib.parse

s3 = boto3.client('s3')
```

Step 3: Add a trigger to the lambda function

```python
def lambda_handler(event, context):
    # Extract bucket and image info from the S3 event
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'])

    # Download the image from raw S3
    download_path = f'/tmp/{os.path.basename(key)}'
    s3.download_file(bucket, key, download_path)

    # Upload the image to processed S3 bucket
    result_bucket = 'medical-images-processed-[YOUR-NAME]'  # Replace with your bucket name
    s3.upload_file(download_path, result_bucket, key + "-processed.png")

    return {
        'statusCode': 200,
        'body': json.dumps(f"Processed {key}, found {cell_count} cells.")
    }
```

Step 3: Add a trigger to the lambda function

Once again click on `Deploy` to save the changes, then go to the S3 bucket `medical-images-raw-[YOUR-NAME]` and upload an image to trigger the lambda function.

If everything went well you should see the same image uploaded to the `processed` bucket with the suffix `-processed.png` as shown below:

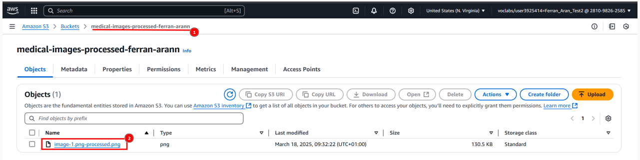

Figure 16: Processed image

Step 3: Add a trigger to the lambda function

Great so we now have a lambda function that:

1. Is triggered when an image is uploaded to the input bucket.
2. Downloads the image from the input bucket.
3. Uploads the image to the output bucket.

We are now going to design the code that processes the image to count the cells, and once we're happy with it we'll add it to the lambda function code.

Step 4: Write the Lambda function code

Lets open up the remote jupyter notebook on our EC2 instance that we have been using and create a new notebook. As a reminder, you can access the EC2 instance, activate the python environment (in this case we are using the sample `project2` environment we created in Session 3) and start the jupyter notebook server with the following commands:

```
ssh -i .ssh/aws-keypair ec2-user@<your-ec2-public-ip>
cd project2
source .project2-venv/bin/activate
jupyter notebook --ip 0.0.0.0 --port 8888
```

Remember you can visit the second guide on the subject's website to see how to access the notebook server. Link here https://hdbc-17705110-mdbs.github.io/.

Step 4: Write the Lambda function code

With the cell images downloaded and extracted, we can now upload one of them to the notebook server from the browser to start working on the code for the Lambda function.

To upload an image to the notebook server, just drag and drop it to the browser window as shown below:

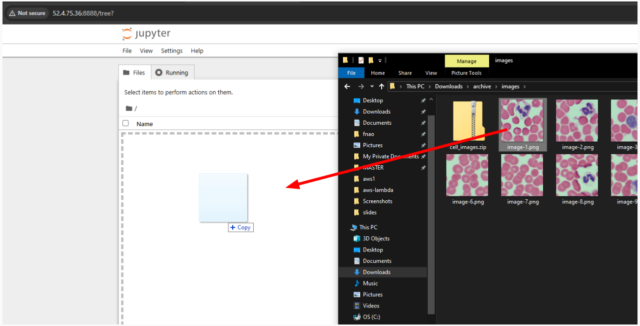

Figure 17: Upload image

Step 4: Write the Lambda function code

Next create a new notebook and paste the following code to install the dependencies.

```
!pip install matplotlib opencv-python
!sudo dnf install mesa-libGL -y
```

And paste the following in another cell to load the image and display it.

```python
import cv2
import matplotlib.pyplot as plt

# Load the sample image
image_path = 'image-1.png'  # Replace with your image path
image = cv2.imread(image_path)

# Display the image
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.show()
```

Step 4: Write the Lambda function code

Wer are now free to work on whichever code we want to process the images. By using the Jupyter notebook we can test the code and see the results before deploying it to the Lambda function. For now, trust me and copy the following code to a new cell:

```
# Count cells by drawing contours around them
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
adaptive_thresh = cv2.adaptiveThreshold(
    gray_image, 255, cv2.ADAPTIVE_THRESH_MEAN_C,
    cv2.THRESH_BINARY_INV, 65, 5
)
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
morph_image = cv2.morphologyEx(adaptive_thresh, cv2.MORPH_OPEN, kernel, iterations=1)
contours, _ = cv2.findContours(
    morph_image, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
)

# Print the result
cell_count = len(contours)
print(f'Cell count: {cell_count}')
```

Step 4: Write the Lambda function code

Printing the result is fine but it would be even better if we could visualize the contours drawn around the cells. To do so, we can use the following code:

```
output_image = image.copy()
cv2.drawContours(output_image, contours, -1, (0, 255, 0), 2)

# Generate the images
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
axes[0].set_title('Original Image')
axes[0].axis('off')

axes[1].imshow(cv2.cvtColor(output_image, cv2.COLOR_BGR2RGB))
axes[1].set_title(f'Contours (Cells: {cell_count})')
axes[1].axis('off')

plt.show()
```

Step 4: Write the Lambda function code

The visualization should look like this:

Figure 18: Processed image

Step 4: Write the Lambda function code

By now our code does the following:

1. Load an image.
2. Process the image to count the cells.
3. Generate an image with the results.

We are now going to need to adapt this code to work in the Lambda function where it will have to read the image from the S3 bucket given its path and write the results back to another S3 bucket.

Step 4: Write the Lambda function code

In the AWS Console go to the Lambda service and click on the lambda function we created earlier, then scroll down to the code editor and paste the following code (note that the code takes 3 slides to fit):

```
import boto3
import cv2
import numpy as np
import json
import os
import urllib.parse

s3 = boto3.client('s3')

def lambda_handler(event, context):
    # Extract bucket and image info from the S3 event
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'])
    original_name = os.path.splitext(os.path.basename(key))[0]

    download_path = f'/tmp/{os.path.basename(key)}'
```

Step 4: Write the Lambda function code

```
# Download and load the image from S3
s3.download_file(bucket, key, download_path)
image = cv2.imread(download_path)

# Count the cells using contours
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
adaptive_thresh = cv2.adaptiveThreshold(
    gray_image, 255, cv2.ADAPTIVE_THRESH_MEAN_C,
    cv2.THRESH_BINARY_INV, 65, 5
)
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
morph_image = cv2.morphologyEx(adaptive_thresh, cv2.MORPH_OPEN, kernel, iterations=1)
contours, _ = cv2.findContours(
    morph_image, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
)
cell_count = len(contours)
```

Step 4: Write the Lambda function code

```python
    # Draw contours on a copy of the image
    output_image = image.copy()
    cv2.drawContours(output_image, contours, -1, (0, 255, 0), 2)

    # Save the processed image and upload it to the "processed" bucket
    result_image_name = f"{original_name}-processed-{cell_count}-cells.png"
    result_image_path = f'/tmp/{result_image_name}'
    cv2.imwrite(result_image_path, output_image)
    result_bucket = 'medical-images-processed-[YOUR-NAME]'  # Replace with your bucket name
    s3.upload_file(result_image_path, result_bucket, result_image_name)

    return {
        'statusCode': 200,
        'body': json.dumps(f"Processed {key}, found {cell_count} cells.")
    }
```

Once the code is copied click on `Deploy` to save the changes.

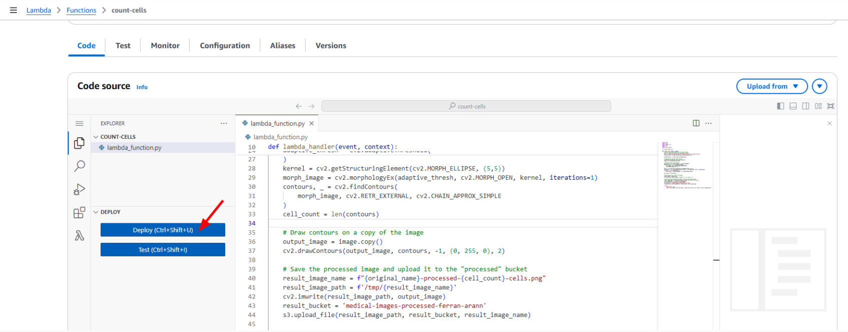

Figure 19: Deploy

Step 5: Create and publish a Lambda layer with the dependencies

AWS Lambdas come with some python dependencies pre-installed such as `boto3` (which is the library we use to write and read from S3 buckets), but we are going to need to install `opencv-python` to process the images since it is not included by default.

To do so, we are going to create a Lambda layer with the dependencies and attach it to the Lambda function. Think of it as a way of packaging the needed dependencies so the lambda has them available when it runs.

We are going to need a machine with AWS CLI and its credentials configured as well as Python 3.13 and `zip` installed. I am going to use the EC2 instance we created in the previous unit together with `uv` for managing python versions. AWS CLI and `zip` are already installed in the instance.

Step 5: Create and publish a Lambda layer with the dependencies

Start by creating a folder which we'll use to build the layer and cd into it.

```
mkdir -p cell-count-layer/python/lib/python3.13/site-packages/
cd cell-count-layer
```

Now create a virtual environment with `uv` and install the dependencies.

```
uv venv --seed --python 3.13 .cell-count-venv
source .cell-count-venv/bin/activate
pip install opencv-python-headless -t python/lib/python3.13/site-packages
```

Step 5: Create and publish a Lambda layer with the dependencies

Now we are going to zip the contents of the folder to create the layer.

```
zip -r opencv.zip python
```

We'll need to create a bucket where we upload the layer so we can then import it to Lambda layers. You can use the AWS Console on your browser as we've done before or use the following command **where you have to replace `[YOUR-NAME]` with your name**:

```
aws s3 mb s3://layers-bucket-[YOUR-NAME]
```

Step 5: Create and publish a Lambda layer with the dependencies

Now publish the layer to the bucket.

```
aws s3 cp opencv.zip s3://layers-bucket-[YOUR-NAME]/
```

And finally, we are going to import the layer from the bucket to the Lambda layers.

```
aws lambda publish-layer-version \
  --layer-name opencv \
  --content S3Bucket=layers-bucket-[YOUR-NAME],S3Key=opencv.zip \
  --compatible-runtimes python3.13
```

Let's now see how to add this layer to our lambda.

Step 5: Create and publish a Lambda layer with the dependencies

Visit the Lambda service on the AWS Console and look for the function we have been working on. Click on it.

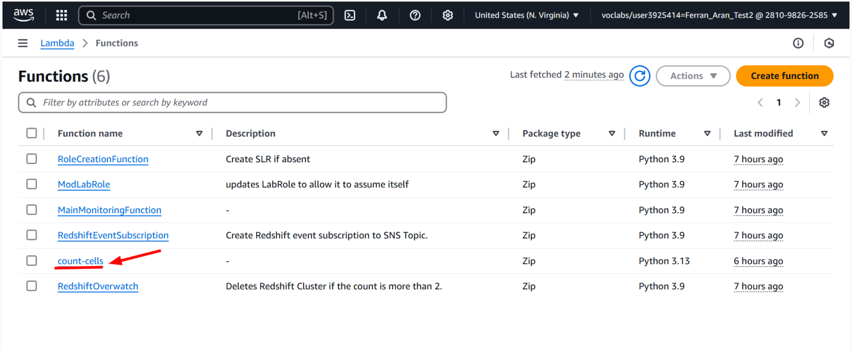

Figure 20: Lambda layer

Step 5: Create and publish a Lambda layer with the dependencies

Click on the `Layers` section below the function's name.

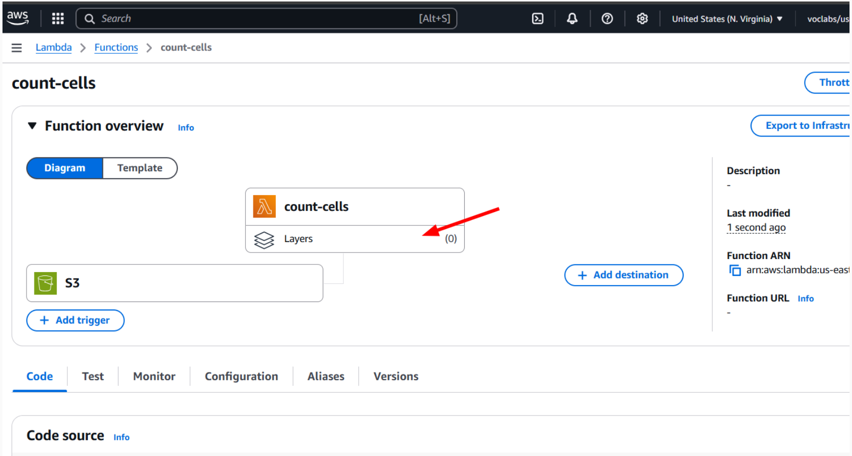

Figure 21: Lambda layer

Step 5: Create and publish a Lambda layer with the dependencies

Click on `Add a layer`.

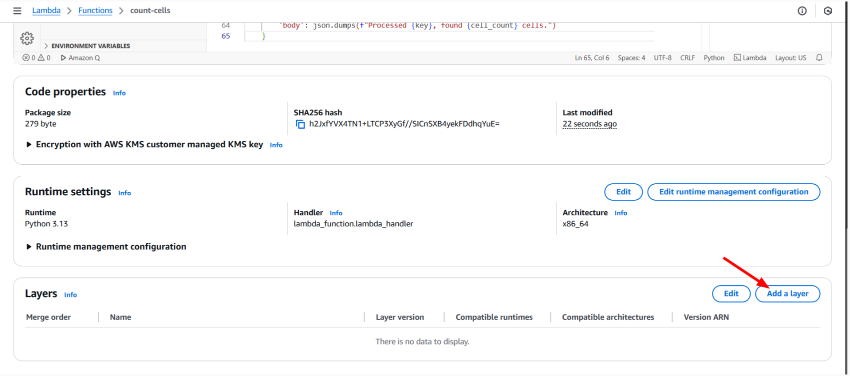

Figure 22: Lambda layer

Step 5: Create and publish a Lambda layer with the dependencies

Click on `Custom layers` and select the layer we just created. Finally click on `Add`.

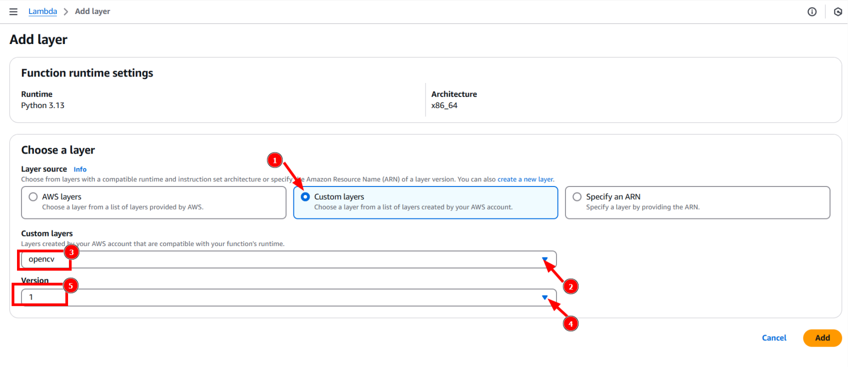

Figure 23: Lambda layer

Step 6: Upload the images to the input bucket

Remember the cell images can be found on the virtual campus https://campusvirtual.urv.cat/ or on the subject's website https://hdbc-17705110-mdbs.github.io.

To upload them to the S3 bucket, we could do so by using the AWS Console as we did before, but this time we are going to use the AWS CLI to do it.

```
aws s3 cp ./cell_images s3://medical-images-raw-[YOUR-NAME]/ --recursive
```

The next slide contains a screenshot of the general steps to download, extract and upload the images to the S3 bucket.

Step 6: Upload the images to the input bucket

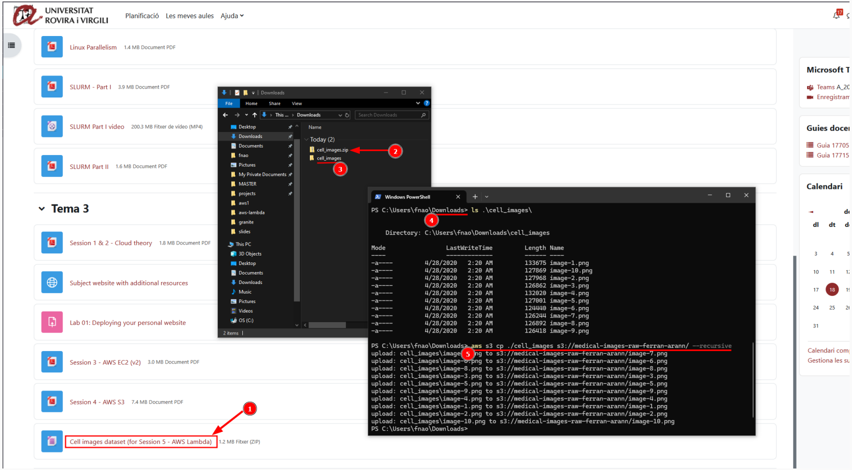

Figure 24: Upload images

Step 6: Upload the images to the input bucket

If we check on the AWS Console the input bucket `medical-images-raw-[YOUR-NAME]` we should see the images uploaded.

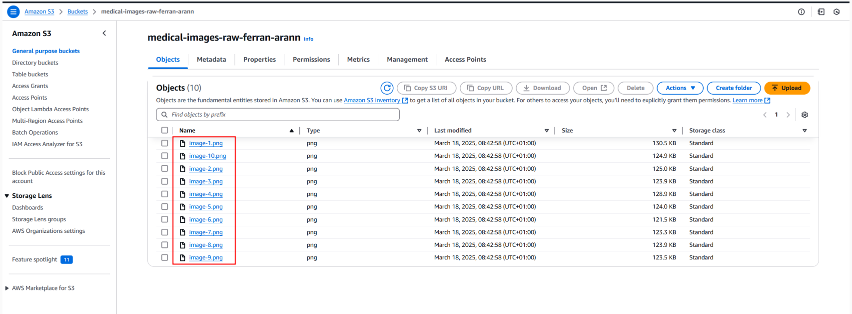

Figure 25: Uploaded images

Step 7: Check the results in the output bucket and verify the Lambda logs

If we've done everything correctly, a Lambda function should have been triggered for each image uploaded to the input bucket, and the processed images should be in the output bucket.

If we check S3 bucket `medical-images-processed-[YOUR-NAME]` we should see something like this:

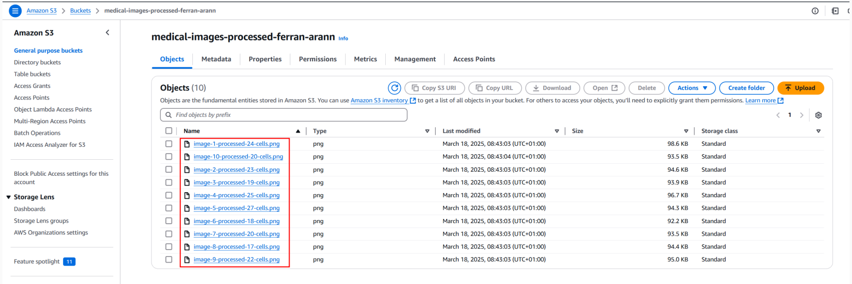

Figure 26: Processed images

Step 7: Check the results in the output bucket and verify the Lambda logs

And if we go back to our lambda and click on `Monitoring`. We should see a plot named `Invocations` that shows the number of times the lambda has been triggered.

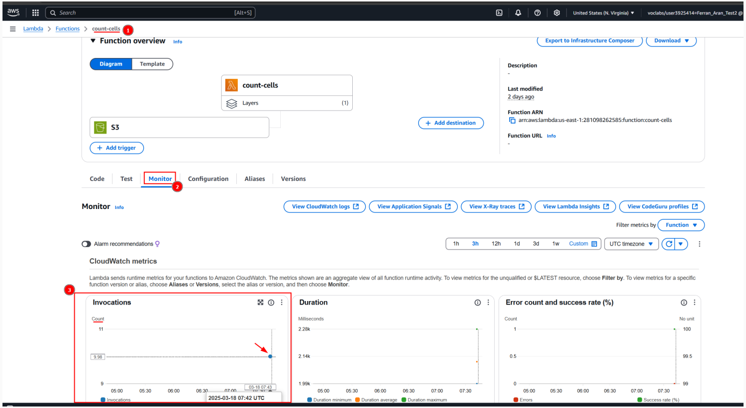

Figure 27: Lambda invocations

Step 7: Check the results in the output bucket and verify the Lambda logs

We could also click on `View logs in CloudWatch` to see the logs of the lambda function as we did earlier.

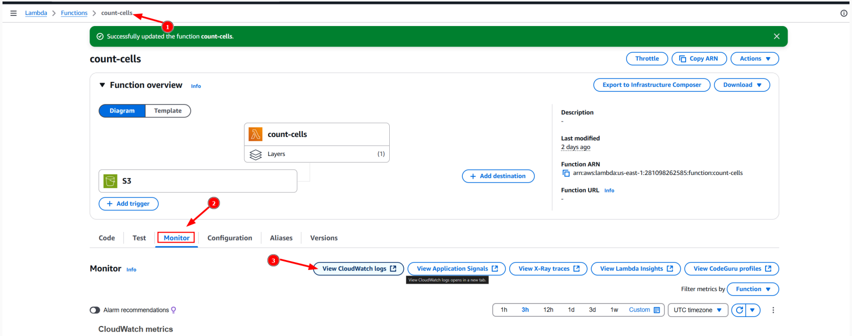

Figure 28: Lambda logs

Recap

Recap

- AWS Lambda is a serverless computing service that runs code in response to events.

Recap

- AWS Lambda is a serverless computing service that runs code in response to events.
- Lambda functions are triggered by events from various sources, such as S3, API Gateway, and CloudWatch.

Recap

- AWS Lambda is a serverless computing service that runs code in response to events.
- Lambda functions are triggered by events from various sources, such as S3, API Gateway, and CloudWatch.
- Lambda functions can be used for real-time data processing, data aggregation, data validation, and image processing in healthcare applications.