

High Performance and Distributed Computing for Big Data

Unit 3: Cloud Systems and Big Data Management

Session 5 - AWS Lambda

Francesc Solsona Tehas francesc.solsona@udl.cat

Universitat Rovira i Virgili and Universitat de Lleida

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.

How Serverless Functions Work

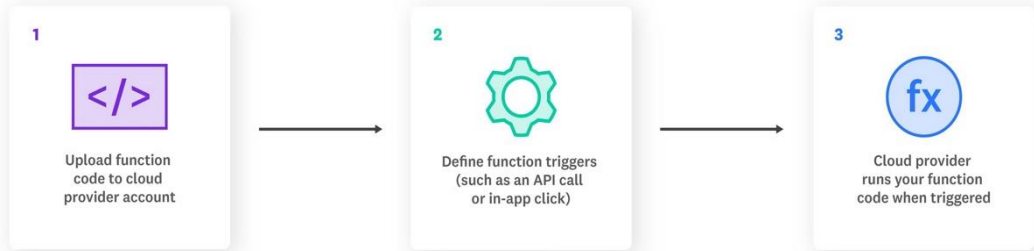


Figure 1: Serverless model



AWS Lambda



CLOUD FUNCTIONS



Azure
Functions

Figure 2: Serverless providers

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

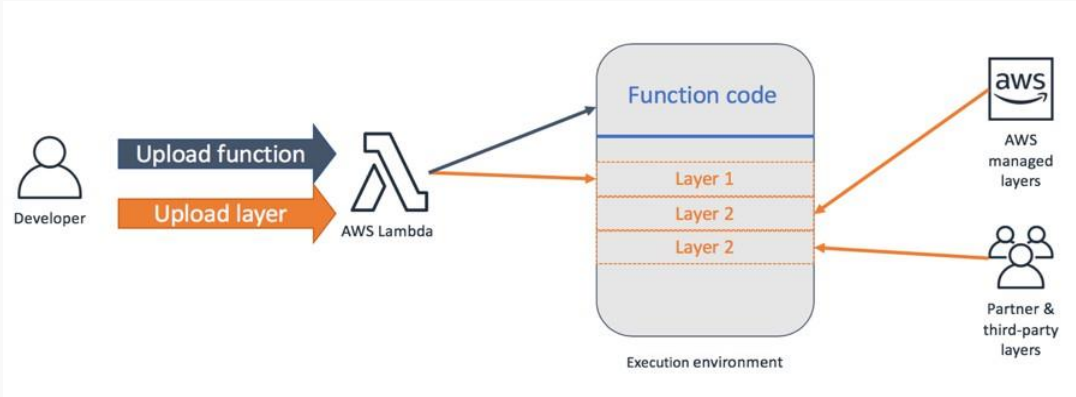


Figure 4: Lambda Architecture

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.

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.
- **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.

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

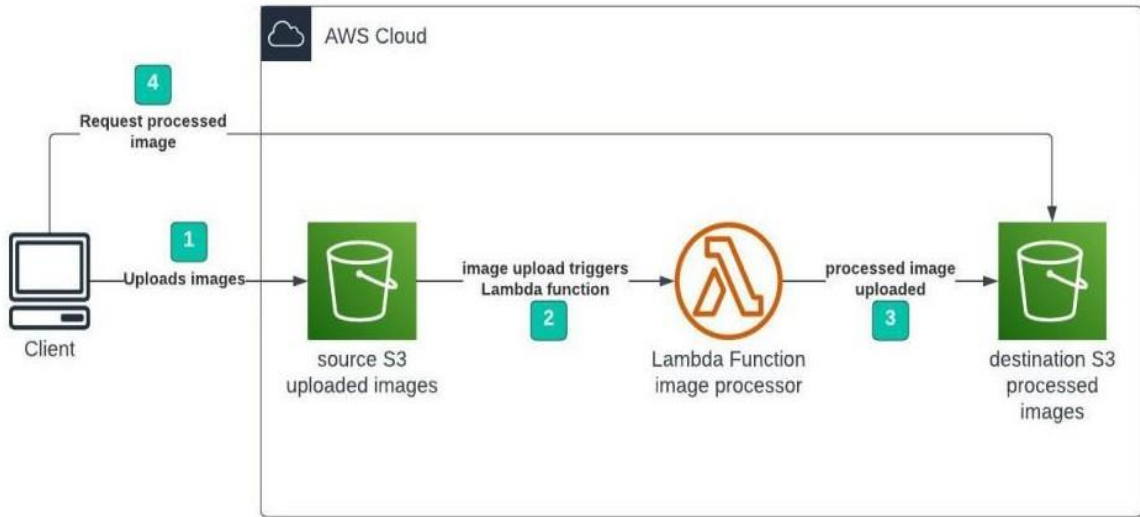


Figure 5: Health Data Flow

Lab: Counting cells at scale

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.

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]`

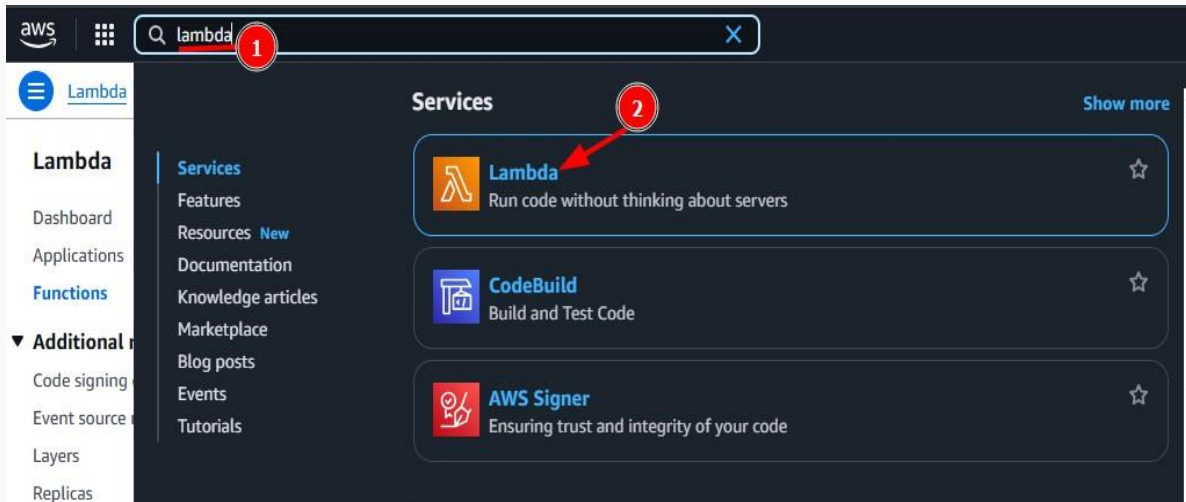
In my case that will be `medical-images-raw-ferran-aran` and `medical-images-processed-ferran-aran`.

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.

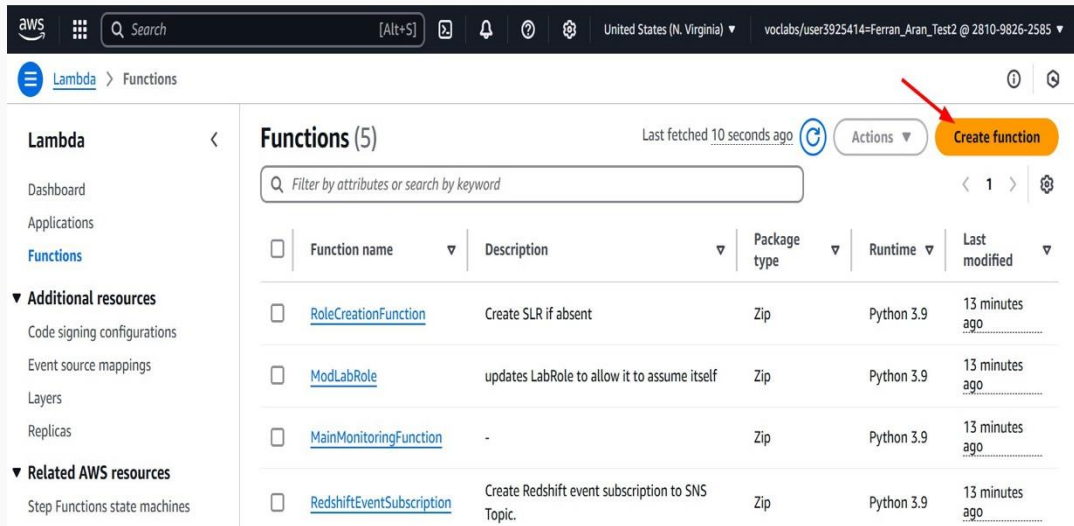


The screenshot shows the AWS Console interface. At the top left is the AWS logo. A search bar contains the text "lambda" with a red circle and the number "1" around it. Below the search bar, the "Lambda" service is selected in the left-hand navigation menu. The main content area is titled "Services" and lists three results: "Lambda" (Run code without thinking about servers), "CodeBuild" (Build and Test Code), and "AWS Signer" (Ensuring trust and integrity of your code). A red arrow points from a red circle with the number "2" to the "Lambda" service card.

Figure 6: Lambda search

Step 2: Create the Lambda function

Now click on create function.



The screenshot shows the AWS Lambda console interface. At the top, there's a navigation bar with the AWS logo, a search bar, and various utility icons. Below that, the breadcrumb navigation shows 'Lambda > Functions'. On the left side, there's a sidebar menu with 'Lambda' selected, and sub-items like 'Dashboard', 'Applications', and 'Functions'. Under 'Additional resources', there are links for 'Code signing configurations', 'Event source mappings', and 'Layers'. Under 'Related AWS resources', there's a link for 'Step Functions state machines'. The main content area is titled 'Functions (5)' and shows a list of existing functions. A red arrow points to the 'Create function' button in the top right corner of the main content area. The table below lists the existing functions:

<input type="checkbox"/>	Function name	Description	Package type	Runtime	Last modified
<input type="checkbox"/>	RoleCreationFunction	Create SLR if absent	Zip	Python 3.9	13 minutes ago
<input type="checkbox"/>	ModLabRole	updates LabRole to allow it to assume itself	Zip	Python 3.9	13 minutes ago
<input type="checkbox"/>	MainMonitoringFunction	-	Zip	Python 3.9	13 minutes ago
<input type="checkbox"/>	RedshiftEventSubscription	Create Redshift event subscription to SNS Topic.	Zip	Python 3.9	13 minutes ago

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`):

The screenshot shows the AWS Lambda console 'Create function' page. The page is divided into several sections with numbered annotations (1-7) pointing to specific elements:

- 1:** Points to the 'Function name' input field containing 'count-cells'.
- 2:** Points to the 'Runtime' dropdown menu, which is set to 'Python 3.13'.
- 3:** Points to the 'Architecture' dropdown menu, which is set to 'x86_64'.
- 4:** Points to the 'Use an existing role' radio button under the 'Change default execution role' section.
- 5:** Points to the 'Existing role' dropdown menu, which is set to 'LabRole'.
- 6:** Points to the 'Existing role' input field.
- 7:** Points to the 'Create function' button at the bottom right.

The 'Basic information' section includes the following details:

- Function name:** count-cells
- Runtime:** Python 3.13
- Architecture:** x86_64
- Permissions:** Use an existing role (LabRole)

The 'Additional Configurations' section is currently collapsed.

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.

The screenshot displays the AWS Lambda console interface for a function named 'scaling-test'. The page is divided into several sections:

- Function overview:** Contains tabs for 'Diagram' and 'Template'. Below them is a card for the function 'scaling-test' showing 'Layers (0)'. A red arrow points to the '+ Add trigger' button. There is also a '+ Add destination' button.
- Metadata:** Includes buttons for 'Throttle', 'Copy ARN', and 'Actions'. Below these are 'Export to Infrastructure Composer' and 'Download' buttons.
- Description:** Shows 'Description: -', 'Last modified: 11 minutes ago', 'Function ARN: arn:aws:lambda:us-east-1:692212546112:function:scaling-test', and 'Function URL: info'.
- Navigation:** A horizontal menu with 'Code', 'Test', 'Monitor', 'Configuration', 'Aliases', and 'Versions' tabs. 'Code' is currently selected.
- Code source:** Shows a code editor with a file named 'lambda_function.py' open. There are 'Upload from' and a dropdown menu 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:

Add trigger

Trigger configuration info 1

S3
aws asynchronous storage

Bucket
Choose or enter the ARN of an S3 bucket that serves as the event source. The bucket must be in the same region as the function.

2

Bucket region: us-east-1

Event types
Select the events that you want to have trigger the Lambda function. You can optionally set up a prefix or suffix for an event. However, for each bucket, individual events cannot have multiple configurations with overlapping prefixes or suffixes that could match the same object key.

All object create events 3 4

Prefix - optional
Enter a single optional prefix to limit the notifications to objects with keys that start with matching characters. Any [special characters](#) must be URL encoded.

Suffix - optional
Enter a single optional suffix to limit the notifications to objects with keys that end with matching characters. Any [special characters](#) must be URL encoded.

Recursive invocation
If your function writes objects to an S3 bucket, ensure that you are using different S3 buckets for input and output. Writing to the same bucket increases the risk of creating a recursive invocation, which can result in increased Lambda usage and increased costs. [Learn more](#)

I acknowledge that using the same S3 bucket for both input and output is not recommended and that this configuration can cause recursive invocations, increased Lambda usage, and increased costs.

5 Lambda will add the necessary permissions for AWS S3 to invoke your Lambda function from this trigger. [Learn more](#) about the Lambda permissions model.

[Cancel](#) [Add](#)

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.

The screenshot shows the AWS Lambda console interface for a function named "count-cells". At the top, there's a navigation bar with "Lambda > Functions > count-cells". Below that, there are buttons for "Throttle", "Copy ARN", and "Actions". A green notification bar states: "The trigger medical-images-raw-feran-aram is successfully added to function count-cells. The function is now receiving events from the trigger." Below this is the "Function overview" section, which includes a "Diagram" tab and a "Template" tab. The diagram shows a box for "count-cells" with "Layers" and "00" below it, and an "S3" trigger box connected to it. A red box highlights the "S3" trigger. To the right of the diagram is an "+ Add destination" button. Below the diagram is an "+ Add trigger" button. On the right side of the overview, there are buttons for "Export to Infrastructure Composer" and "Download". Below the overview is a navigation menu with tabs: "Code", "Test", "Monitor", "Configuration", "Aliases", and "Versions". A red circle with the number "1" and an arrow points to the "Code" tab. Below the navigation menu is the "General configuration" section, which includes fields for "Description", "Memory" (128 MB), "Ephemeral storage" (512 MB), "Timeout" (0 min 3 sec), and "Permissions".

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:

```
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

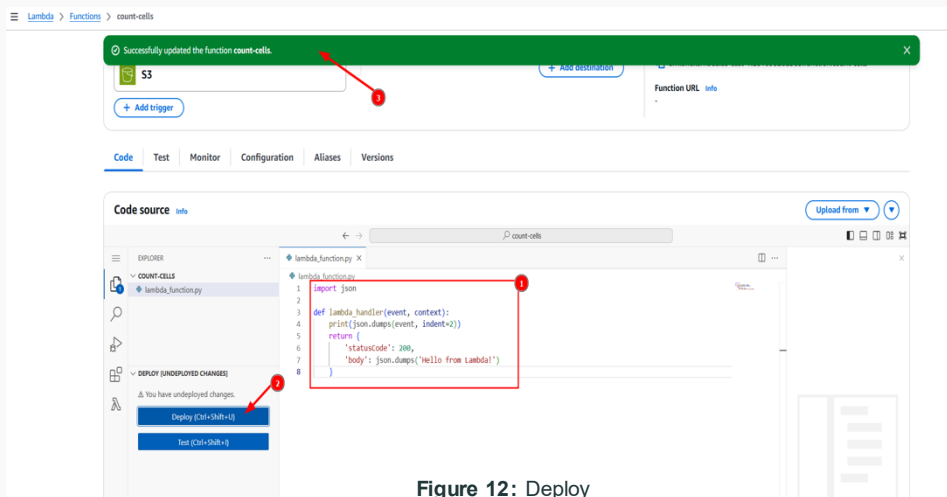
Let's see how the event looks like by printing it:

```
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.



The screenshot displays the AWS Lambda console interface for a function named 'count-cells'. At the top, a green notification bar states 'Successfully updated the function count-cells.' Below this, the 'Configuration' tab is active, showing a code source of 'S3' and a '+ Add trigger' button. A red circle with the number '3' is positioned over the '+ Add trigger' button. The 'Code source' section is expanded, showing a code editor with the following Python code:

```
1 import json
2
3 def lambda_handler(event, context):
4     print(json.dumps(event, indent=2))
5     return {
6         'statusCode': 200,
7         'body': json.dumps('Hello from Lambda!')
8     }
```

A red box with the number '1' highlights the code editor. In the bottom left corner, the 'DEPLOY (UNDEPLOYED CHANGES)' section is visible, featuring a 'Deploy (Ctrl+Shift+L)' button and a 'Test (Ctrl+Shift+T)' button. A red circle with the number '2' is positioned over the 'Deploy (Ctrl+Shift+L)' button. The breadcrumb navigation at the top left shows 'Lambda > Functions > count-cells'.

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.

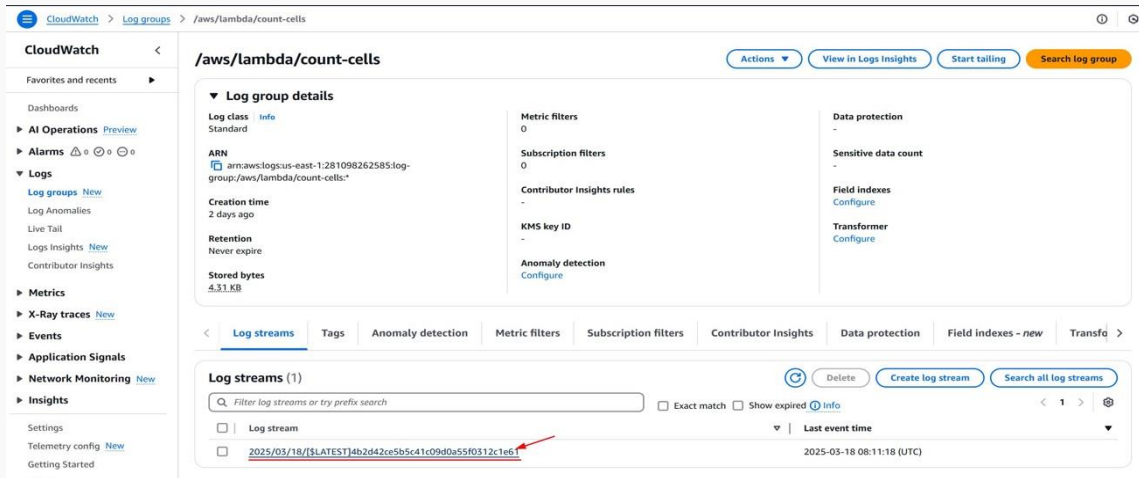
We can now go back our lambda, click on `Monitoring` and then on `View logs in CloudWatch` to see the logs of the lambda function.

The screenshot displays the AWS Lambda console for the function 'count-cells'. At the top, a green notification bar states 'Successfully updated the function count-cells'. The function overview shows it is triggered by an S3 bucket. The 'Monitor' tab is active, and the 'View CloudWatch logs' button is highlighted with a red circle and arrow. Other tabs include Code, Test, Configuration, Aliases, and Versions. The bottom of the console shows 'CloudWatch metrics' and a filter menu.

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.



The screenshot displays the AWS CloudWatch console interface for a log group named `/aws/lambda/count-cells`. The left-hand navigation pane shows the 'Logs' section expanded, with 'Log groups' selected. The main content area is titled `/aws/lambda/count-cells` and includes several action buttons: 'Actions', 'View in Logs Insights', 'Start tailing', and 'Search log group'. Below these buttons, the 'Log group details' section provides information about the log group, including its class (Standard), ARN, creation time (2 days ago), retention (Never expire), and stored bytes (4.31 KB). The 'Log streams' section is active, showing a list of log streams. The most recent log stream is `2025/03/18/[$LATEST]4b2d42ce5b5c41c09d0a55f0312c1e61`, which is highlighted with a red arrow. The console also shows various filters and tabs for the log streams, such as 'Tags', 'Anomaly detection', 'Metric filters', 'Subscription filters', 'Contributor Insights', 'Data protection', 'Field indexes - new', and 'Transfo'.

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": "A1E48DYHJFCTZ"
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": "0067092AA399156004"
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')
```

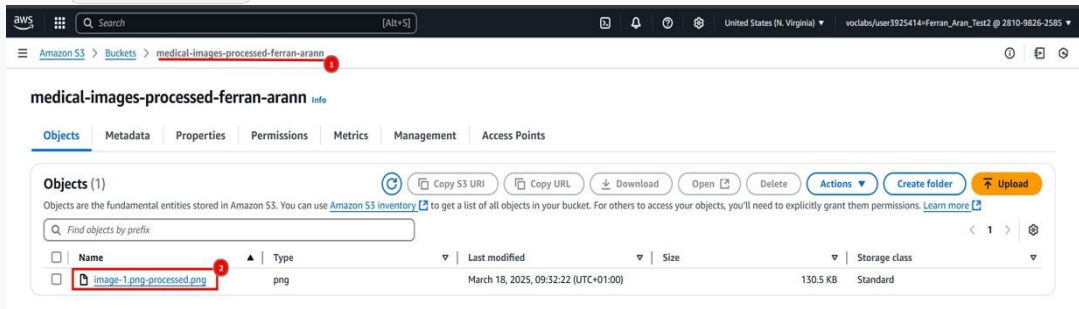
Step 3: Add a trigger to the lambda function

```
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:



The screenshot shows the AWS S3 console interface. At the top, the navigation bar includes the AWS logo, a search bar, and the user's profile information. The breadcrumb trail indicates the current location: Amazon S3 > Buckets > medical-images-processed-ferran-arann. The bucket name is highlighted with a red circle and a '1' icon. Below the breadcrumb, the bucket name 'medical-images-processed-ferran-arann' is displayed with an 'Info' link. The 'Objects' tab is selected, showing a list of objects. The list contains one object: 'image-1.png-processed.png', which is highlighted with a red box and a '2' icon. The object's details are as follows:

Name	Type	Last modified	Size	Storage class
image-1.png-processed.png	png	March 18, 2025, 09:32:22 (UTC+01:00)	130.5 KB	Standard

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:

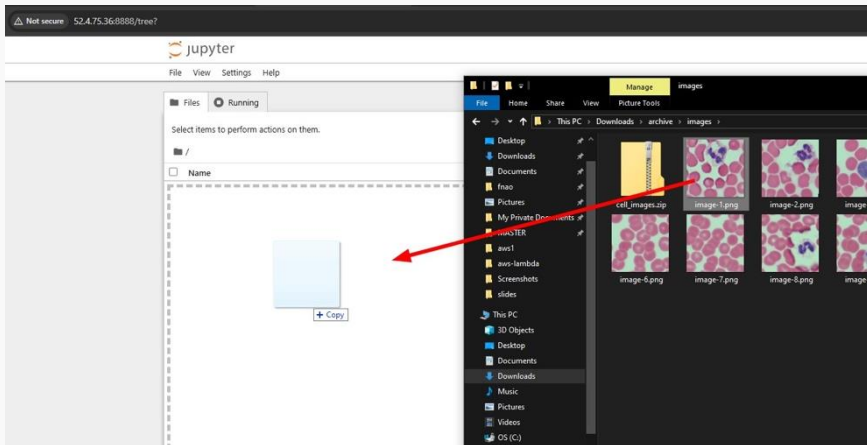


Figure 17: Upload image

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.

```
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

We 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
)
kemel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
morph_image = cv2.morphologyEx(adaptive_thresh, cv2.MORPH_OPEN, kemel, 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

```
# 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.")
}
```

Step 4: Write the Lambda function code

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

Lambda > Functions > count-cells

Code

Test

Monitor

Configuration

Aliases

Versions

Code source Info

Upload from



count-cells



EXPLORER

COUNT-CELLS

lambda_function.py

DEPLOY

Deploy (Ctrl+Shift+U)

Test (Ctrl+Shift+I)

lambda_function.py

lambda_function.py

```
10 def lambda_handler(event, context):
27     )
28     kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
29     morph_image = cv2.morphologyEx(adaptive_thresh, cv2.MORPH_OPEN, kernel, iterations=1)
30     contours, _ = cv2.findContours(
31         morph_image, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
32     )
33     cell_count = len(contours)
34
35     # Draw contours on a copy of the image
36     output_image = image.copy()
37     cv2.drawContours(output_image, contours, -1, (0, 255, 0), 2)
38
39     # Save the processed image and upload it to the "processed" bucket
40     result_image_name = f'(original_name)-processed-{cell_count}-cells.png'
41     result_image_path = f'/tmp/{result_image_name}'
42     cv2.imwrite(result_image_path, output_image)
43     result_bucket = 'medical-images-processed-ferran-arann'
44     s3.upload_file(result_image_path, result_bucket, result_image_name)
45
```



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  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]/
```

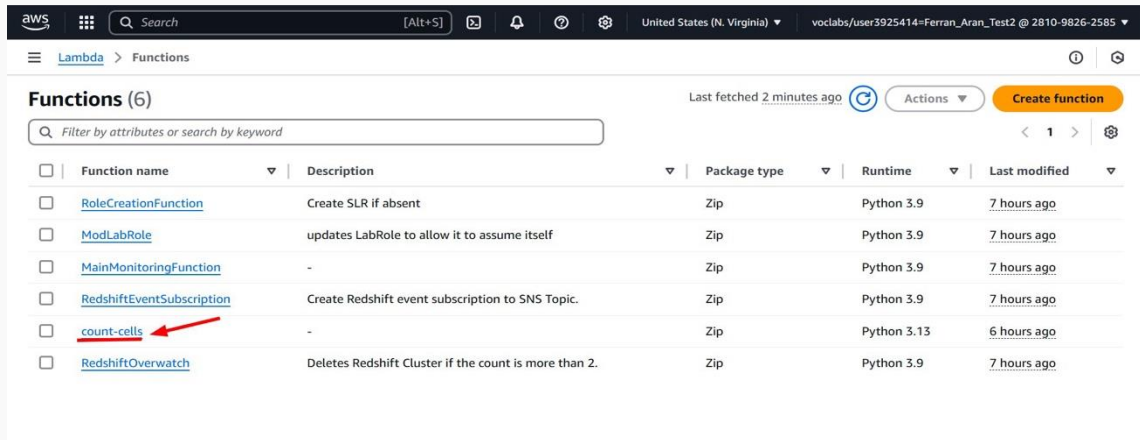
And finally, we are going to import the layer from the bucket to the Lambda layers.

```
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.



The screenshot shows the AWS Lambda console interface. At the top, there's a navigation bar with the AWS logo, a search bar, and various utility icons. Below that, the breadcrumb navigation shows 'Lambda > Functions'. The main content area is titled 'Functions (6)' and includes a search filter, a refresh button, and a 'Create function' button. A table lists six functions with columns for checkboxes, function names, descriptions, package types, runtimes, and last modified times. The 'count-cells' function is highlighted with a red arrow.

<input type="checkbox"/>	Function name	Description	Package type	Runtime	Last modified
<input type="checkbox"/>	RoleCreationFunction	Create SLR if absent	Zip	Python 3.9	7 hours ago
<input type="checkbox"/>	ModLabRole	updates LabRole to allow it to assume itself	Zip	Python 3.9	7 hours ago
<input type="checkbox"/>	MainMonitoringFunction	-	Zip	Python 3.9	7 hours ago
<input type="checkbox"/>	RedshiftEventSubscription	Create Redshift event subscription to SNS Topic.	Zip	Python 3.9	7 hours ago
<input type="checkbox"/>	count-cells	-	Zip	Python 3.13	6 hours ago
<input type="checkbox"/>	RedshiftOverwatch	Deletes Redshift Cluster if the count is more than 2.	Zip	Python 3.9	7 hours ago

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.

The screenshot shows the AWS Lambda console for a function named 'count-cells'. The breadcrumb navigation is 'Lambda > Functions > count-cells'. The function overview is displayed with a 'Diagram' tab selected. A red arrow points to the 'Layers' section, which currently shows '(0)' layers. Below the function overview, there is an S3 trigger and a '+ Add destination' button. The right sidebar shows the function's description, last modified time (1 second ago), function ARN (arn:aws:lambda:us-east-1:123456789012:lambda:function:count-cells), and function URL. The bottom navigation bar includes tabs for Code, Test, Monitor, Configuration, Aliases, and Versions. The 'Code source' section is partially visible at the bottom.

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

Click on **Add a layer**.

The screenshot shows the AWS Lambda console interface for a function named 'count-cells'. The breadcrumb navigation is 'Lambda > Functions > count-cells'. The code editor shows a Python function snippet. Below the code editor, there are three main sections: 'Code properties', 'Runtime settings', and 'Layers'. The 'Layers' section is currently empty, displaying 'There is no data to display.' A red arrow points to the 'Add a layer' button in the 'Layers' section.

Code properties [Info](#)

- Package size: 279 byte
- SHA256 hash: [h2JxfYVX4TN1+LTCP3XyGf//SiCnSXB4yekFDdhqYuE=](#)
- Last modified: 22 seconds ago

Runtime settings [Info](#)

- Runtime: Python 3.13
- Handler: [lambda_function.lambda_handler](#)
- Architecture: [x86_64](#)

Layers [Info](#)

Merge order	Name	Layer version	Compatible runtimes	Compatible architectures	Version ARN
There is no data to display.					

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**.

Add layer

Function runtime settings

Runtime: Python 3.13 | Architecture: x86_64

Choose a layer

Layer source [Info](#)

Choose from layers with a compatible runtime and instruction set architecture or specify the Amazon Resource Name (ARN) of a layer version. You can also [create a new layer](#).

AWS layers
Choose a layer from a list of layers provided by AWS.

Custom layers
Choose a layer from a list of layers created by your AWS account.

Specify an ARN
Specify a layer by providing the ARN.

Custom layers
Layers created by your AWS account that are compatible with your function's runtime.

Layer name	Version
opencv	1

[Cancel](#) [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

The screenshot shows a university website interface for 'UNIVERSITAT ROVIRA I VIRGILI'. The page lists various documents and videos. A file explorer window is open over the 'Downloads' folder, showing a subfolder named 'cell_images' with two files: 'cell_images.zip' and 'cell_images'. A PowerShell terminal window is also open, displaying the following commands and output:

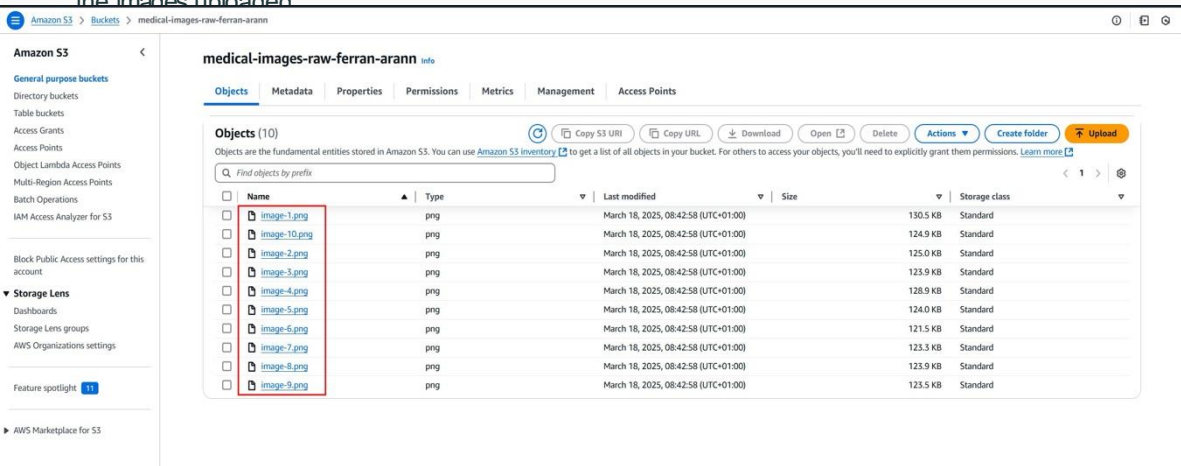
```
PS C:\Users\fnao\Downloads> ls .\cell_images\  
Directory: C:\Users\fnao\Downloads\cell_images  
  
Mode                LastWriteTime         Length Name  
----                -  
-A-                4/28/2020    2:28 AM           133675 image-1.png  
-A-                4/28/2020    2:28 AM           127869 image-10.png  
-A-                4/28/2020    2:28 AM           127968 image-2.png  
-A-                4/28/2020    2:28 AM           126862 image-3.png  
-A-                4/28/2020    2:28 AM           132020 image-4.png  
-A-                4/28/2020    2:28 AM           127003 image-5.png  
-A-                4/28/2020    2:28 AM           124808 image-6.png  
-A-                4/28/2020    2:28 AM           126204 image-7.png  
-A-                4/28/2020    2:28 AM           126892 image-8.png  
-A-                4/28/2020    2:28 AM           126418 image-9.png  
-A-                4/28/2020    2:28 AM           126418 image-9.png  
  
PS C:\Users\fnao\Downloads> aws s3 cp /cell_images s3://medical-images-ras-ferran-arann/ --recursive  
upload: cell_images/image-1.png to s3://medical-images-ras-ferran-arann/image-1.png  
upload: cell_images/image-2.png to s3://medical-images-ras-ferran-arann/image-2.png  
upload: cell_images/image-3.png to s3://medical-images-ras-ferran-arann/image-3.png  
upload: cell_images/image-4.png to s3://medical-images-ras-ferran-arann/image-4.png  
upload: cell_images/image-5.png to s3://medical-images-ras-ferran-arann/image-5.png  
upload: cell_images/image-6.png to s3://medical-images-ras-ferran-arann/image-6.png  
upload: cell_images/image-7.png to s3://medical-images-ras-ferran-arann/image-7.png  
upload: cell_images/image-8.png to s3://medical-images-ras-ferran-arann/image-8.png  
upload: cell_images/image-9.png to s3://medical-images-ras-ferran-arann/image-9.png  
upload: cell_images/image-10.png to s3://medical-images-ras-ferran-arann/image-10.png  
PS C:\Users\fnao\Downloads>
```

Red circles and arrows highlight key elements: (1) points to the 'Cell images dataset (for Session 5 - AWS Lambda)' document; (2) points to the 'cell_images.zip' file; (3) points to the 'cell_images' folder; (4) points to the PowerShell command prompt; (5) points to the 'aws s3 cp' command.

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



The screenshot shows the AWS S3 console interface for the bucket `medical-images-raw-ferran-arann`. The 'Objects' tab is selected, displaying a list of 10 objects. The objects are named `image-1.png` through `image-10.png`, all of type `png`, and were last modified on March 18, 2025, at 08:42:58 (UTC+01:00). The sizes range from 121.5 KB to 130.5 KB, and all are stored in the `Standard` storage class. The first row, `image-1.png`, is highlighted with a red box.

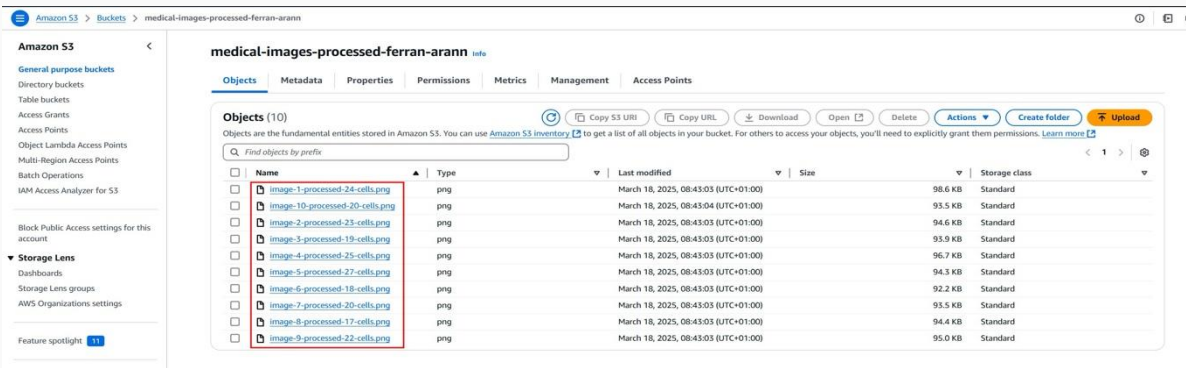
<input type="checkbox"/>	Name	Type	Last modified	Size	Storage class
<input type="checkbox"/>	image-1.png	png	March 18, 2025, 08:42:58 (UTC+01:00)	130.5 KB	Standard
<input type="checkbox"/>	image-10.png	png	March 18, 2025, 08:42:58 (UTC+01:00)	124.9 KB	Standard
<input type="checkbox"/>	image-2.png	png	March 18, 2025, 08:42:58 (UTC+01:00)	125.0 KB	Standard
<input type="checkbox"/>	image-3.png	png	March 18, 2025, 08:42:58 (UTC+01:00)	123.9 KB	Standard
<input type="checkbox"/>	image-4.png	png	March 18, 2025, 08:42:58 (UTC+01:00)	128.9 KB	Standard
<input type="checkbox"/>	image-5.png	png	March 18, 2025, 08:42:58 (UTC+01:00)	124.0 KB	Standard
<input type="checkbox"/>	image-6.png	png	March 18, 2025, 08:42:58 (UTC+01:00)	121.5 KB	Standard
<input type="checkbox"/>	image-7.png	png	March 18, 2025, 08:42:58 (UTC+01:00)	123.3 KB	Standard
<input type="checkbox"/>	image-8.png	png	March 18, 2025, 08:42:58 (UTC+01:00)	123.9 KB	Standard
<input type="checkbox"/>	image-9.png	png	March 18, 2025, 08:42:58 (UTC+01:00)	123.5 KB	Standard

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:



The screenshot shows the Amazon S3 console interface for the bucket `medical-images-processed-ferran-arann`. The left sidebar contains navigation options for Amazon S3, including General purpose buckets, Directory buckets, Table buckets, Access Grants, Access Points, Object Lambda Access Points, Multi-Region Access Points, Batch Operations, and IAM Access Analyzer for S3. The main content area displays the bucket's details and a list of objects. The 'Objects (10)' section shows a search bar and a table of objects. The table columns are Name, Type, Last modified, Size, and Storage class. The objects listed are all PNG files, each named `image-[number]-processed-[number]-cells.png`, with a size between 94.4 KB and 98.6 KB, and a last modified date of March 18, 2025, at 08:43:03 (UTC+01:00). The storage class for all objects is Standard. A red box highlights the first nine rows of the table.

<input type="checkbox"/>	Name	Type	Last modified	Size	Storage class
<input type="checkbox"/>	image-1-processed-24-cells.png	png	March 18, 2025, 08:43:03 (UTC+01:00)	98.6 KB	Standard
<input type="checkbox"/>	image-10-processed-20-cells.png	png	March 18, 2025, 08:43:04 (UTC+01:00)	93.5 KB	Standard
<input type="checkbox"/>	image-2-processed-23-cells.png	png	March 18, 2025, 08:43:03 (UTC+01:00)	94.6 KB	Standard
<input type="checkbox"/>	image-3-processed-19-cells.png	png	March 18, 2025, 08:43:03 (UTC+01:00)	93.9 KB	Standard
<input type="checkbox"/>	image-4-processed-25-cells.png	png	March 18, 2025, 08:43:03 (UTC+01:00)	96.7 KB	Standard
<input type="checkbox"/>	image-5-processed-27-cells.png	png	March 18, 2025, 08:43:03 (UTC+01:00)	94.3 KB	Standard
<input type="checkbox"/>	image-6-processed-18-cells.png	png	March 18, 2025, 08:43:03 (UTC+01:00)	92.2 KB	Standard
<input type="checkbox"/>	image-7-processed-20-cells.png	png	March 18, 2025, 08:43:03 (UTC+01:00)	93.5 KB	Standard
<input type="checkbox"/>	image-8-processed-17-cells.png	png	March 18, 2025, 08:43:03 (UTC+01:00)	94.4 KB	Standard
<input type="checkbox"/>	image-9-processed-22-cells.png	png	March 18, 2025, 08:43:03 (UTC+01:00)	95.0 KB	Standard

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.

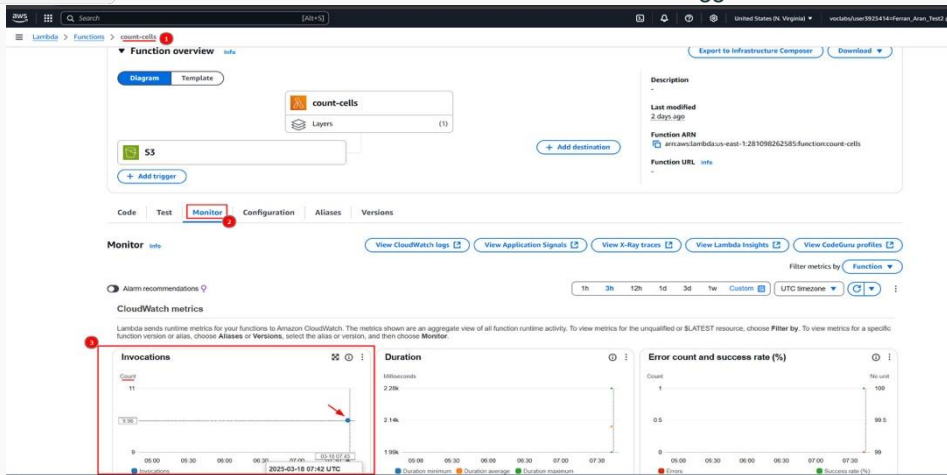


Figure 27: Lambda invocations

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.

The screenshot displays the AWS Lambda console for a function named 'count-cells'. At the top, a green notification bar states 'Successfully updated the function count-cells.' The breadcrumb navigation shows 'Lambda > Functions > count-cells'. The main content area is titled 'count-cells' and includes a 'Function overview' section with tabs for 'Diagram' and 'Template'. A diagram shows the function 'count-cells' with one layer. Below the diagram, there is an 'S3' trigger and an '+ Add trigger' button. To the right, a 'Description' panel shows 'Last modified: 2 days ago' and 'Function ARN: arn:aws:lambda:us-east-1:281098262585:function:count-cells'. At the bottom, the 'Monitor' tab is active, and the 'View CloudWatch logs' button is highlighted with a red box and a red arrow labeled '3'. Other buttons in the 'Monitor' section include 'View Application Signals', 'View X-Ray traces', 'View Lambda Insights', and 'View CodeGuru profiles'. The bottom of the console shows 'Alarm recommendations' and 'CloudWatch metrics' with a filter set to 'Function' and a time range of '1h'.

Figure 28: Lambda logs

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.