

Metropolitan State University

ICS 432 - 01: Distributed and Cloud Computing

Fall 2021

Lab 10: Serverless Compute: AWS Lambda and GCP Cloud Functions

Total points: 25

Out: Saturday, November 6, 2021

Due: 11:59 PM on Friday, November 12, 2021

What to submit?

The objective of this lab is to practice using virtual machines on the cloud. To complete this lab:

- Read this lab assignment carefully.
- At various parts of the lab, you are asked to **take screen shots** of your work. Open a word document and paste the screen shots in this document in the same order as mentioned in the lab. Make sure to highlight the screen shot number.
- After you complete all the lab exercises, upload the word document to the designated D2L folder by 11:59 PM on Friday, November 12, 2021.

NOTE: On Windows machines, you may consider using [Snip & Sketch](#) for screenshot handling.

Introduction

Serverless compute is a cloud computing execution model in which the Cloud acts as the server and dynamically manages the allocation of machine resources.

Serverless computing addresses the following challenges. Suppose you have a small computation that you need to run against some database at the end of each month. Or suppose you want to have the equivalent of a computational daemon that wakes up and executes a specific task only when certain conditions arise. For example, when an event arrives in a particular stream or when a file in a storage container has been modified or when a timer goes off. How do you automate these tasks without paying for a continuously running server? Unfortunately, the traditional cloud computing infrastructure model would require you to allocate computing resources such as virtual machines and your daemon would be a continuously running process. While you can scale your cluster of VMs up and down, you can't scale it to zero without my daemon becoming unresponsive. You only want to pay for your computing WHEN the computation is running.

Function-as-a-Service" (FaaS) is a cloud computing service that allows a cloud user to define their own function, and then "register" it with the cloud and specify the exact events that will cause it to wake up and execute.

Where did the servers go? Serverless computing still requires servers, but the server management and capacity planning decisions are hidden from the developer or operator.

Definitions

Function: a function is a resource that you can invoke to run your code. A function has code to process the **events** that you pass into the function or that other services send to the function.

Trigger: a trigger is a resource or configuration that invokes the serverless function. This includes other cloud services that you can configure to invoke a function (e.g., cloud storage bucket), and applications that you develop.

Event: an event is a JSON-formatted document that contains data for the serverless function to process. The runtime environment converts the event to an object and passes it to your function code. When you invoke a function, you determine the structure and contents of the event.

Example custom event about weather data

```
{
  "TemperatureK": 281,
  "WindKmh": -3,
  "HumidityPct": 0.55,
  "PressureHPa": 1020
}
```

Exercise 1: AWS Lambda

References

AWS Lambda Developer Guide: <https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf>.

Part 1: based on pages 7-9

Part 2: based on Pages 383-388

Part 1: Create a Lambda function with default function code

In this getting started exercise, you will create a simple Python Lambda function using the Lambda console. These are the steps:

1. Create a Lambda using default code for the function.
2. Manually invoke the Lambda function using sample event data. Lambda runs the function and returns results. Verify the results, including the logs that your Lambda function created and various Amazon CloudWatch metrics.
3. Clean up the resources that were automatically created by the Lambda Function Execution environment.

Step 1: Create the Lambda Function

To create a Lambda function with the console

1. Log in to your AWS Educate account.
2. From AWS console, go to Services→Compute→ Lambda.
3. Click on Create function.
4. Select Author from Scratch.
5. Under Basic information, do the following:
 - a. Enter function name, use your last name as part of the function name, for example `<your-last-name>-test-function`.
 - b. For Runtime, Choose Python 3.8.
6. Click on Create function.

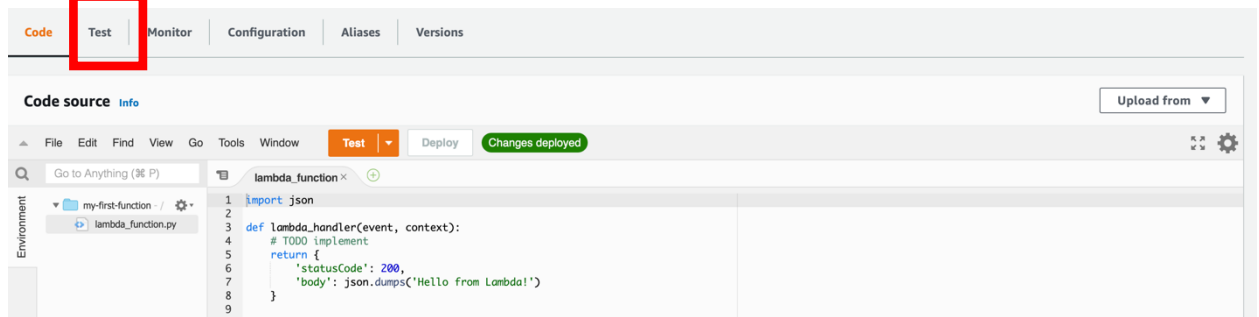
The function overview shows a visualization of your function, including any triggers, destinations, and layer that you have configured for the function.

Lab screenshot #1: take a screenshot to show your function page.

Step 2: Invoke the Lambda Function

You will invoke your Lambda function using the sample event data provided in the console.

- 1- In the Code source section, double click on `lambda_function.py` file and examine the default code.



- 2- The code includes one method called `lambda_handler`.
- 3- The function returns a JSON object with the message: `'Hello from Lambda!'`
- 4- Go to **Test** tab.
- 5- In the Test event section, choose **New event**. In Template, leave the default `hello-world` option. Enter a Name for the test (e.g., `test-event-1`) and examine the format of the event's JSON object.

- 6- Click on **Save Changes** and then click on **Test**.
- 7- Upon successful completion, view the results in the console. Click on the arrow next to Details to expand all the details of the function execution.

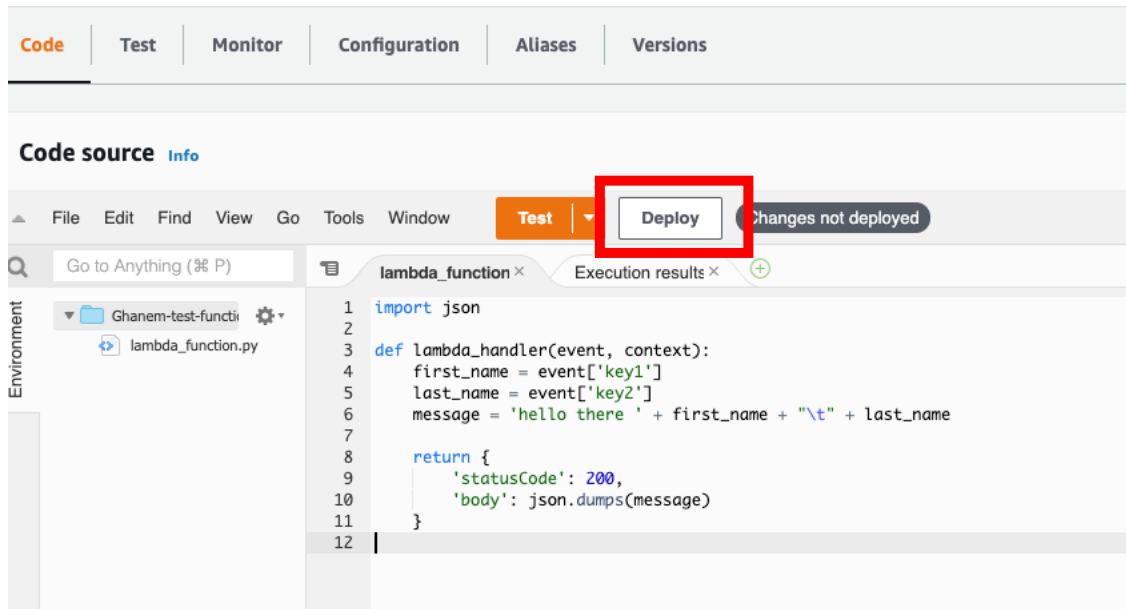
- 8- The details of the test execution results include the following:
 - a. The **Execution result** shows the execution status as succeeded. Note that the logs link opens the Log groups page in the CloudWatch console.
 - b. The **Summary** section shows information like duration of function execution, billed duration, and maximum memory used. Note that this information is also stored in the Log output section (the REPORT line in the execution log).
 - c. The **Log output** section shows the log that Lambda generates for each invocation. The function writes these logs to CloudWatch. The Lambda console shows these logs for your convenience. Choose [Click here](#) to view the corresponding CloudWatch log group in the CloudWatch console.
- 9- Go to Code tab and change the message printed by the function to be 'I hope you have a wonderful day'. **Click on Deploy** for your code changes to be saved. Click on Test to run the function.

Lab screenshot #2: take a screenshot of the execution results of your function.

10- Now you will change your function to retrieve input from the event. Go to Test tab and change the event to include your first and last names as inputs as shown in the following figure. Click on **Save changes**.



11- Go back to the code section and change the code in `lambda_function.py` to retrieve your name as input event as follows.



12. Click on **Deploy**. Click on **Test** to run your code. Your name should now be displayed in the printed fun.

Lab screenshot #3: take a screenshot to show the output of running your Lambda function with your name.

13. Click the **Monitor** tab then Click on **Metrics**. This page shows graphs for the metrics that Lambda sent to CloudWatch. Change the first name and last name in the test event several times and invoke the function after eacg change.

Lab screenshot #4: take a screenshot to of the displayed metrics dashboard to show the number of your function invocations over time.

Step 3: Clean up

There are several resources that are automatically created by the Lambda Function Execution environment and need to be deleted. These resources include, in addition to the function, a log group that stores the function's logs, and the execution role that the console created.

To delete a Lambda function

1. Open the [Functions page](#) on the Lambda console.
2. Choose the function.
3. Choose Actions, Delete.
4. In the Delete function dialog box, choose Delete.

To delete the log group

1. Open the [Logs](#) → [Log groups](#) page of the CloudWatch console.
2. Select the function's log group (/aws/lambda/<your-function-name>).
3. Choose Actions, Delete log group(s).
4. In the Delete log group(s) dialog box, choose Delete.

To delete the execution role

1. Open the [Roles page](#) of the AWS Identity and Access Management (IAM) console.
2. Select the function's role (<your-function-name>-role-31exmpl).
3. Choose Delete role.
4. In the Delete role dialog box, choose Yes, delete

Part 2: Create an S3-triggered Lambda function

In this part, you will create a Lambda function that is triggered by Amazon Simple Storage Service (Amazon S3). The trigger invokes your function every time that you add an object to your Amazon S3 bucket. These are the steps:

- 1- Create an AWS S3 bucket and load a file.
- 2- Create an AWS Lambda function that is triggered by the S3 bucket you created.
- 3- Examine the event that triggers the function.
- 4- Test the function with a fake event.
- 5- Test the function with real S3 triggers.
- 6- Adding code to your function to count the number of lines in an uploaded csv file.
- 7- Cleaning up the resources.

Step 1: Create a bucket and load a file

In this step, you will create an Amazon S3 bucket and upload a test file to your new bucket. Your Lambda function retrieves information about this file when you test the function from the console.

To create an Amazon S3 bucket using the console

1. Open the [Amazon S3 console](#).
2. Choose Create bucket.
3. Under General configuration, do the following:
 - a. For Bucket name, enter `<your-last-name>movieratingdata`.
 - b. For AWS Region, choose us-east-1. Note that you must create your Lambda function in the same Region.
4. Click on Create bucket.
5. Upload `users.csv` from the movie rating dataset to the bucket.

Lab screenshot #5: take a screenshot of your bucket dashboard showing `users.csv` file.

Step 2: Create the Lambda function

We will create a lambda function using a blueprint that provides a sample function to demonstrate how to use Lambda with other AWS services. Also, a blueprint includes sample code and function configuration presets for a certain runtime. For this tutorial, you will choose a [blueprint for Python runtime](#).

To create a Lambda function from a blueprint in the console

1. From Services menu, go to Compute → Lambda.
2. Click on Create function.
3. Choose Use a blueprint.
4. Under Blueprints, enter `s3` in the search box.
5. In the search results, choose [s3-get-object-python](#).
6. Click on Configure.
7. Under Basic information, do the following:
 - a. For Function name, enter `<your-last-name>-movierating-function`.
 - b. For Execution role, choose Create a new role from AWS policy templates.
 - c. For Role name, enter `<your-last-name>-movierating-lambda-role`.
 - d. Under S3 trigger, click on the dropdown menu under Bucket and choose the S3 bucket that you created in Step 1. When you configure an S3 trigger using the Lambda console, the console modifies your function's [resource-based policy](#) to allow Amazon S3 to invoke the function.
8. Carefully examine the automatically created function code:
 - a. When a file is uploaded to the S3 bucket, a trigger event is sent to the Lambda function. The Lambda function then retrieves the source S3 bucket name (`bucket = event['Records'][0]['s3']['bucket']['name']`) and the key name of the uploaded object (`key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'], encoding='utf-8')`) from the event parameter that it receives.
 - b. The Lambda function uses the [Amazon S3 get_object API](#) to retrieve the object (`response = s3.get_object(Bucket=bucket, Key=key)`) and then

print the CONTENT TYPE the object (print ("CONTENT TYPE: " + response['ContentType'])).

```
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')

def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'],
encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they exist and your
bucket is in the same region as this function.'.format(key, bucket))
        raise e
```

9- Click on Create function.

Lab screenshot #6: take a screenshot to show your created function dashboard.

Step 3: Examine the event that triggers the function

- 1- Amazon S3 sends an event to a Lambda function when an object is created or deleted. You configure notification settings on a bucket, and grant Amazon S3 permission to invoke a function on the function's resource-based permissions policy.
- 2- The event that is sent by S3 contains details about the object that is added to the bucket.
- 3- To view an example event, got to **Test** tab, and examine the automatically-generated **S3-put** Template event.


```

{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2":
"EXAMPLE123/5678abcdefghijklambdaisawesome/mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "example-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::example-bucket"
        },
        "object": {
          "key": "test/key",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}

```

- 4- Go to Code tab, and change the function code to print the file name and the response returned by the `get_object` call by adding the following line before the `return` statement in the `try` block.

```

print (key)
print (response)

```

- 5- Click on Deploy.

- 6- To test the function, upload a file to the S3 bucket to trigger the function to run. However, we will first test the function with a fake event to make sure the function works as expected.

Step 4: Test in the console using a fake event

- 1- On the Code tab, under Code source, choose the arrow next to **Test**, and then choose **Configure test events** from the dropdown list.
- 2- In the Configure test event window, do the following:
 - a. Choose Create new test event.
 - b. For *Event template*, choose **Amazon S3 Put (s3-put)**.
 - c. For *Event name*, enter a name for the test event. For example, **s3faketestevent**.
 - d. In the test event JSON, replace the S3 bucket name (i.e, example-bucket) and object key (test/key) with your bucket name (i.e., **your-last-namemovierating**) and test file name (**users.csv**).
 - e. Click on **Create**.
3. Go back to **Code source** and click on **Test**.

Lab screenshot #7: take a screenshot to show the execution results after running the function.

Step 5: Test with real S3 triggers

In this step, you upload a file to the S3 bucket to trigger the function.

1. Go to your movie rating S3 bucket.
2. Upload the `ratings.csv` file from the movie rating data set to the bucket. Wait until the file upload is completed.
3. Go to your Lambda function console.
4. Choose the **Monitor** tab, then choose **Metrics** tab, and **choose 1h** from the top menu. Enlarge the invocations graphs by clicking on the three dots and choosing Enlarge.

Lab screenshot #8: take a screenshot of your enlarges invocations graph.

5. Go back to the S3 bucket and upload the `item.csv` file. Then go back and take another screenshot of the enlarged invocation graph. You should see an additional dot added. You may need to wait for few minutes and keep on refreshing the invocations graph until the new dot appears. Make sure to choose 1h for a more detailed graph. When you hover on a dot, it shows the time of the corresponding invocation.

Lab screenshot #9: take a screenshot to show the invocations graph of the function showing the new invocation.

- Click on the Logs tab and you should see several logs where each log corresponds to one function invocation. Click on the first logs to view the function output.

Lab screenshot #10a and #10b: take two screenshots to show the contents of two logs. The screenshot should show both the file name and content type.

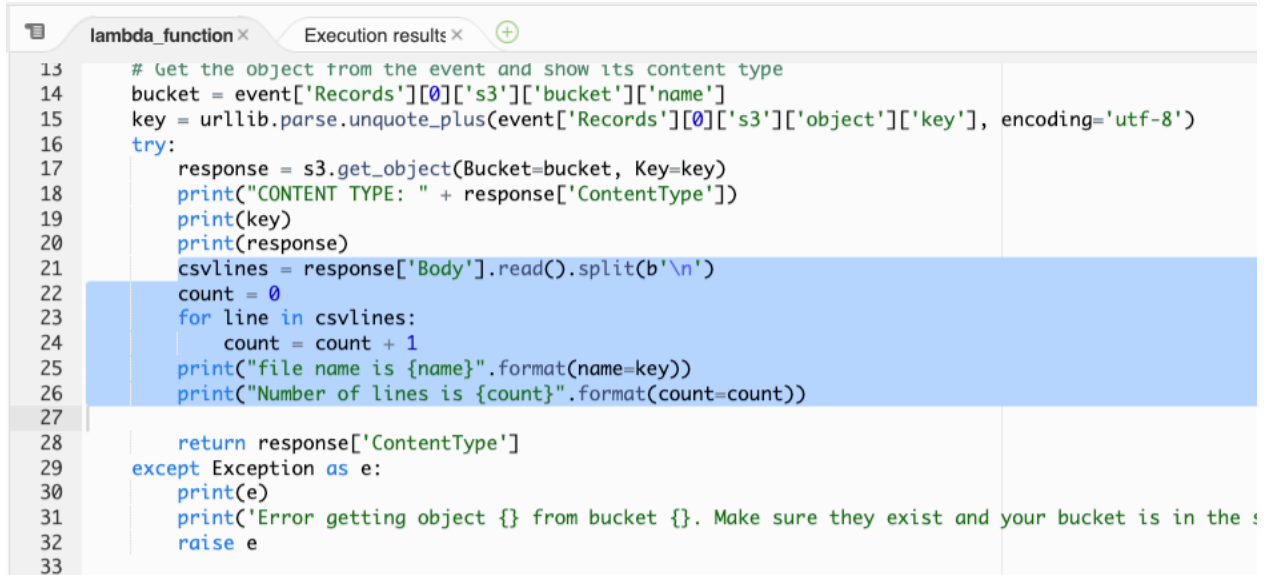
- Go back to the S3 bucket and upload a jpg file of your choice. Then come back to the logs page and examine the invocations log. You should see the CONTENT TYPE as img/jpg. You may need to wait for a few minutes for the logs to be updated.

Lab screenshot #11: take a screenshot to show the log that corresponds to the jpg file upload.

Step 6: Adding code to the function to count the number of lines in each file

In this step, you will add code to your lambda function so that, instead of just printing the file name and content type, the function will also print the number of lines if the uploaded file is of type .CSV.

- Delete all the files from the S3 bucket.
- Go back to your Lambda function to add the highlighted lines of code. This code reads the csv file line by line. The code then prints the file name and the number of lines. Click **Deploy** for the code to work.



```
13 # Get the object from the event and show its content type
14 bucket = event['Records'][0]['s3']['bucket']['name']
15 key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'], encoding='utf-8')
16 try:
17     response = s3.get_object(Bucket=bucket, Key=key)
18     print("CONTENT TYPE: " + response['ContentType'])
19     print(key)
20     print(response)
21     csvlines = response['Body'].read().split(b'\n')
22     count = 0
23     for line in csvlines:
24         count = count + 1
25     print("file name is {name}".format(name=key))
26     print("Number of lines is {count}".format(count=count))
27
28     return response['ContentType']
29 except Exception as e:
30     print(e)
31     print('Error getting object {} from bucket {}. Make sure they exist and your bucket is in the s
32     raise e
33
```

- Go back to the S3 bucket and upload the three csv files: users.csv, data.csv, and item.csv.
- Wait until the three files are uploaded then go to lambda functions and check the logs. Wait until the three logs appear and examine them.

Lab screenshot #12, #13, and #14: take screenshots of the three logs showing information about the three uploaded files. Note that you may have information about two files appearing in one log.

Step 7: Clean up your resources

Delete the resources that you created for this exercise.

To delete the Lambda function

1. Open the [Functions page](#) of the Lambda console.
2. Select the function that you created.
3. Choose Actions, Delete.
4. Choose Delete.

To delete the IAM Role

1. Open the [Roles page](#) of the AWS Identity and Access Management (IAM) console.
2. Select the role that include your lambda function name.
3. Choose Delete.

To delete the S3 bucket

1. Open the [Amazon S3 console](#).
2. Delete all the files from your bucket then delete the bucket.

Exercise 2: Google Cloud Functions

Reference

This exercise is based on the following tutorial:
<https://cloud.google.com/functions/docs/tutorials/storage>

Part 1: Creating a storage-triggered function

In this exercise, you will write GCP Cloud Function that is triggered by adding a file to a Cloud Storage bucket. These are the steps:

- 1- Create a bucket on Cloud Storage.
- 2- Enable APIs.
- 3- Deploy a cloud function.
- 4- Invoke the cloud function using a storage trigger.
- 5- Adding code to the function to retrieve metadata about the event that triggered the function.
- 6- Creating a cloud function to count the lines in a csv file.
- 7- Delete the project.

Step 1: Create a Bucket

The `google.storage.object.finalize` trigger type to deploy a function that is invoked whenever the specified Cloud Storage bucket is written to.

1. Log in to your GCP Console account.
2. Create a new project and make sure the Billing Account for Education is enabled for that project.
3. From the left menu, choose Storage → Cloud Storage
4. Create a bucket with the following information:
 - a. Name: `<your-last-name>movieratinglab10`
 - b. Location type: Region
 - c. Location: us-central1 (Iowa)
 - d. Storage class: Standard
 - e. Access control: Uniform
5. Click on CREATE.

Step 2: Enable APIs

1. From the main menu, go to APIs & Services
2. From the top menu, Click on ENABLE APIS AND SERVICES
 - a. Search for Cloud Build API, click on the service, then click ENABLE.
 - b. Go back to APIs& Services main page, search for Cloud Functions API, click on the API name, then click on ENABLE
 - c. Go back to APIs& Services main page, search for Cloud Storage API, click on the API name, then click on ENABLE.

Step 3: Deploy a Cloud Function

1. From the left menu, go to SERVERLESS → Cloud Functions
2. Click on CREATE FUNCTION
 - a. Function name: `<your-last-name>-movierating-function`
 - b. Region: us-central1
 - c. Trigger type: Cloud Storage
 - d. Event type: Finalize/Create
 - e. Bucket: Click on BROWSE and choose the bucket that you created in Step 1. Click on SELECT.
 - f. Click on SAVE.
3. Click on Next
4. From the Runtime drop-down menu, choose Python 3.8
5. From Source code drop-down menu, choose Inline Editor.
6. Entry point: keep the default 'hello_gcs' as this is the function handler that will be invoked when the function is triggered.
7. Examine the default function code (in `main.py`) that does the following:
 - a. Capture the triggering file: `file = event`
 - b. Prints the file name `print(f"Processing file: {file['name']}")`
8. Click on Deploy. Wait until the function is created. This may take few moments.

Lab screenshot #15: take a screenshot of your Cloud Functions dashboard showing your function with a green circle next to it which means that the function is successfully created.

Step 4: Invoke the function

1. Go to Cloud Storage and upload the file `users.csv` to the bucket you created in Step 1. Wait until the file is successfully uploaded.
2. Go to Cloud Function dashboard and click on your function name.
3. Go to the LOGS tab. You should see few lines in the log which means that the function is triggered and run upon the file upload.

Lab screenshot #16: take screenshot of the LOGS page. Make sure function name appears in the screenshot. Make sure to include all lines in the log. There should be a line showing the name of the file that triggered the function execution.

4. Upload the `items.csv` file to the bucket and come back to the Cloud Function dashboard and examine LOGS.

Lab screenshot #17: take screenshot of the LOGS page. Make sure function name appears in the screenshot. Make sure to include all lines in the log. There should be a line showing the names of the two files that triggered the function.

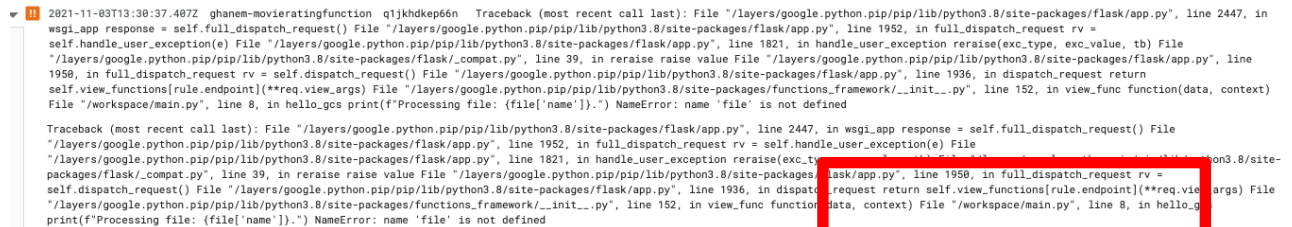
Step 5: Adding code to the function to retrieve event metadata

1. Go to the bucket and delete the two files. **Don't delete the bucket.**
2. Go to your cloud function's page and click on Edit.
3. In the configuration page, click on NEXT, to go to the Code page.
4. Add code to `main.py` to retrieve and print metadata about the event that triggers the function.

```
1 def hello_gcs(event, context):
2     """Triggered by a change to a Cloud Storage bucket.
3     Args:
4         event (dict): Event payload.
5         context (google.cloud.functions.Context): Metadata for the event.
6     """
7     print('File: {}'.format(event['name']))
8     print('Event type: {}'.format(context.event_type))
9     print('Bucket: {}'.format(event['bucket']))
10    print('Created: {}'.format(event['timeCreated']))
11    print('Updated: {}'.format(event['updated']))
12
13
```

5. Click on Deploy. Wait until the function deployment is complete.

- Go to the bucket and upload `users.csv`. Wait until the file upload is complete.
- Check your cloud function's LOGS and examine the output of the function that should include more details about the triggering event. Note that you can check the times in the log to now which log lines correspond to your most recent function invocation.
- If you have any syntax errors in your function, then the errors will be displayed in the log. You may expand any line in the log to read more details. For example, the following figure shows an error in `main.py` line 8.



```

2021-11-03T13:30:37.407Z ghanem-movieratingfunction q1jkhdkp66n Traceback (most recent call last): File "/layers/google.python.pip/pip/lib/python3.8/site-packages/flask/app.py", line 2447, in wsgi_app response = self.full_dispatch_request() File "/layers/google.python.pip/pip/lib/python3.8/site-packages/flask/app.py", line 1952, in full_dispatch_request rv = self.handle_user_exception(e) File "/layers/google.python.pip/pip/lib/python3.8/site-packages/flask/app.py", line 1821, in handle_user_exception reraise(exc_type, exc_value, tb) File "/layers/google.python.pip/pip/lib/python3.8/site-packages/flask/_compat.py", line 39, in reraise raise value File "/layers/google.python.pip/pip/lib/python3.8/site-packages/flask/app.py", line 1950, in full_dispatch_request rv = self.dispatch_request() File "/layers/google.python.pip/pip/lib/python3.8/site-packages/flask/app.py", line 1936, in dispatch_request return self.view_functions[rule.endpoint](**req.view_args) File "/layers/google.python.pip/pip/lib/python3.8/site-packages/functions_framework/_init_.py", line 152, in view_func function(data, context) File "/workspace/main.py", line 8, in hello_gcs print(f'Processing file: {file['name']}') NameError: name 'file' is not defined

Traceback (most recent call last): File "/layers/google.python.pip/pip/lib/python3.8/site-packages/flask/app.py", line 2447, in wsgi_app response = self.full_dispatch_request() File "/layers/google.python.pip/pip/lib/python3.8/site-packages/flask/app.py", line 1952, in full_dispatch_request rv = self.handle_user_exception(e) File "/layers/google.python.pip/pip/lib/python3.8/site-packages/flask/app.py", line 1821, in handle_user_exception reraise(exc_type, exc_value, tb) File "/layers/google.python.pip/pip/lib/python3.8/site-packages/flask/_compat.py", line 39, in reraise raise value File "/layers/google.python.pip/pip/lib/python3.8/site-packages/flask/app.py", line 1950, in full_dispatch_request rv = self.dispatch_request() File "/layers/google.python.pip/pip/lib/python3.8/site-packages/flask/app.py", line 1936, in dispatch_request return self.view_functions[rule.endpoint](**req.view_args) File "/layers/google.python.pip/pip/lib/python3.8/site-packages/functions_framework/_init_.py", line 152, in view_func function(data, context) File "/workspace/main.py", line 8, in hello_gcs print(f'Processing file: {file['name']}') NameError: name 'file' is not defined

```

- If you get any error, go back to the function code and fix error. Make sure to click **Deploy** and wait until the deployment is completed before you test the function again. To test the function again, delete the file from the bucket and upload it again because the function is triggered only upon file upload.

Lab screenshot #18: take a screenshot of the part of LOGS that shows the output of the function. You may need to refresh the logs few times until the new invocation appears.

Step 6: Creating a Cloud function to count the lines in a csv file.

- Follow the same steps to create another function called `<your-last-name>-csv-data-processing`. Make sure to configure the function exactly as you configured the first function in Step 3.
- The following figure shows the code to be written in `main.py`. Examine the code and make sure you understand how the code works. Pay attention to the syntax when you write the code. Note that the code imports `storage` library from Python `google.cloud` which is the same as used in Lab 8.

Runtime
Python 3.8

Entry point *
hello_gcs

Source code
Inline Editor

+

main.py ...

requirements.txt ...

```

1  from google.cloud import storage
2
3  def hello_gcs(event, context):
4      file_name = event['name']
5      bucket = event['bucket']
6      client = storage.Client()
7      bucket = client.get_bucket(bucket)
8      file_blob = bucket.get_blob(file_name)
9      file_str = file_blob.download_as_string()
10     print(file_name)
11     count = 0
12     lines = file_str.splitlines()
13     for line in lines:
14         count = count + 1
15     print("number of lines in %s is %s"%(file_name, count))
16
17

```

Lab screenshot #19: take a screenshot of your `main.py`.

- Because this function is importing some libraries, you need to add the following to the `requirements.txt` file to configure the Python runtime environment. You can access `requirements.txt` from the function's code page.

```

google-cloud-storage==1.30.0
gcsfs==0.6.2

```

Configuration — 2 Code

Runtime
Python 3.8

Entry point *
hello_gcs

Source code
Inline Editor

+

main.py ...

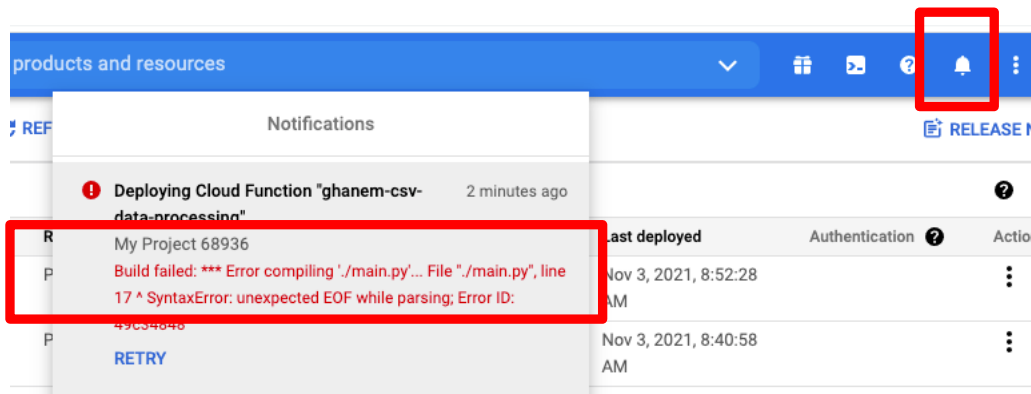
requirements.txt ...

```

1  google-cloud-storage==1.30.0
2  gcsfs==0.6.2
3
4

```

- Click on Deploy. If your function gives error for not being able to deploy successfully, you can check the notifications from the GCP console's top menu (bell symbol) to get more information about the error. For example, the following notifications tells that there is an error in line 17 of `main.py`.



Lab screenshot #20: take a screenshot your Cloud Functions dashboard showing that your function is successfully deployed (i.e., has a green circle with a check mark)

5. To test your function, go the bucket and upload the `ratings.csv` file. Wait until the file is successfully uploaded.
6. Go to the LOGS of your function.

Lab screenshot #21: take a screenshot LOGS showing the output of the function that includes the number of lines in `ratings.csv`.

Step 7: Delete the project.

After you complete the lab, delete the project to save your credits.