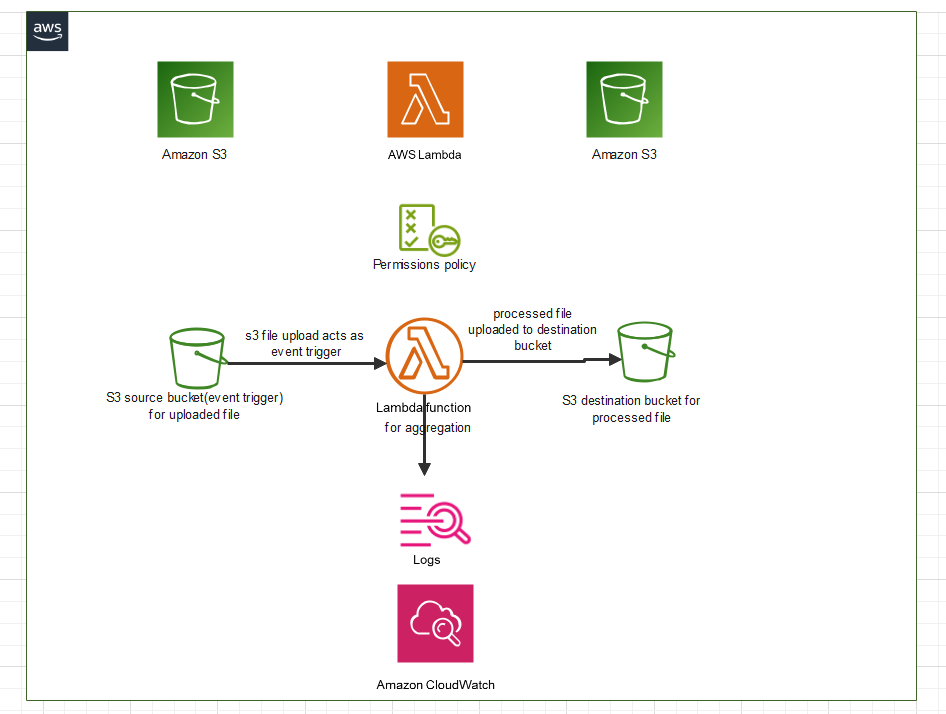
**Lambda function for aggregation**

The main purpose of this lambda function is to get invoked on S3 file upload. On S3 file upload event ,the Lambda reads the data ,aggregates on certain columns and then writes this aggregate data to a file in memory ,which is then uploaded to the destination bucket.



The detailed end to end process is underlined below:

AWS resources used:

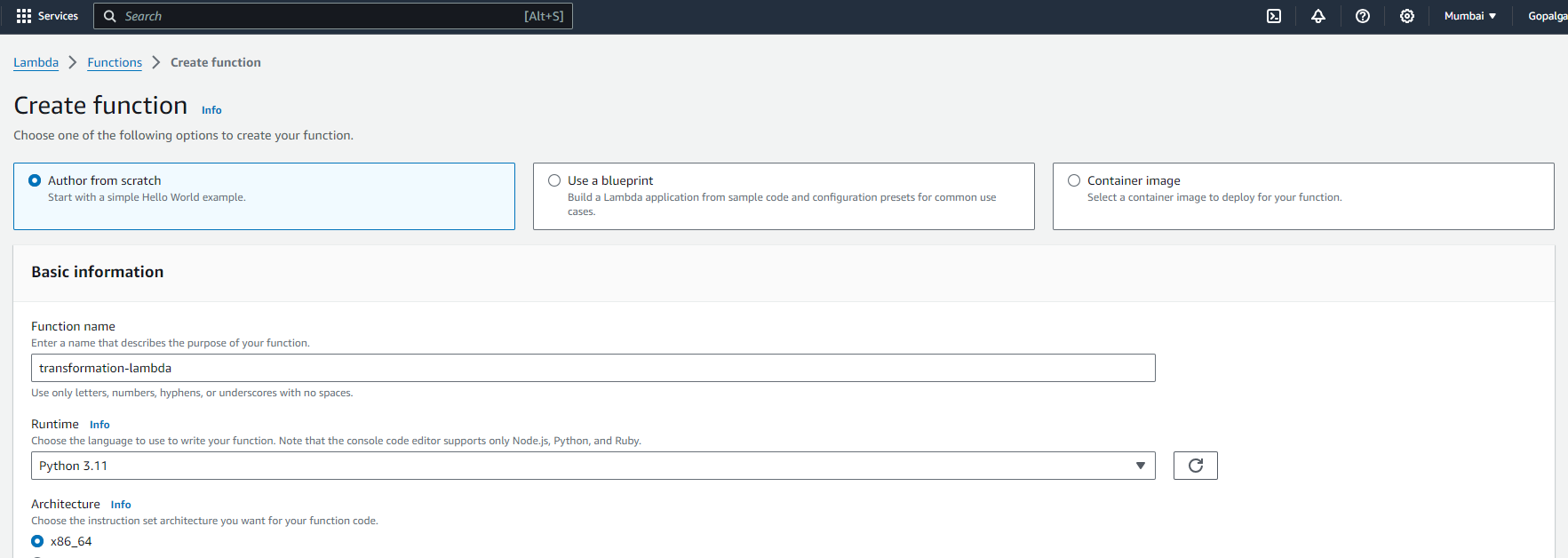
**1.S3**

**2.Lambda**

**3.Cloudwatch**

1.Click on Create function and select author from scratch , since we will be specifying the configuration

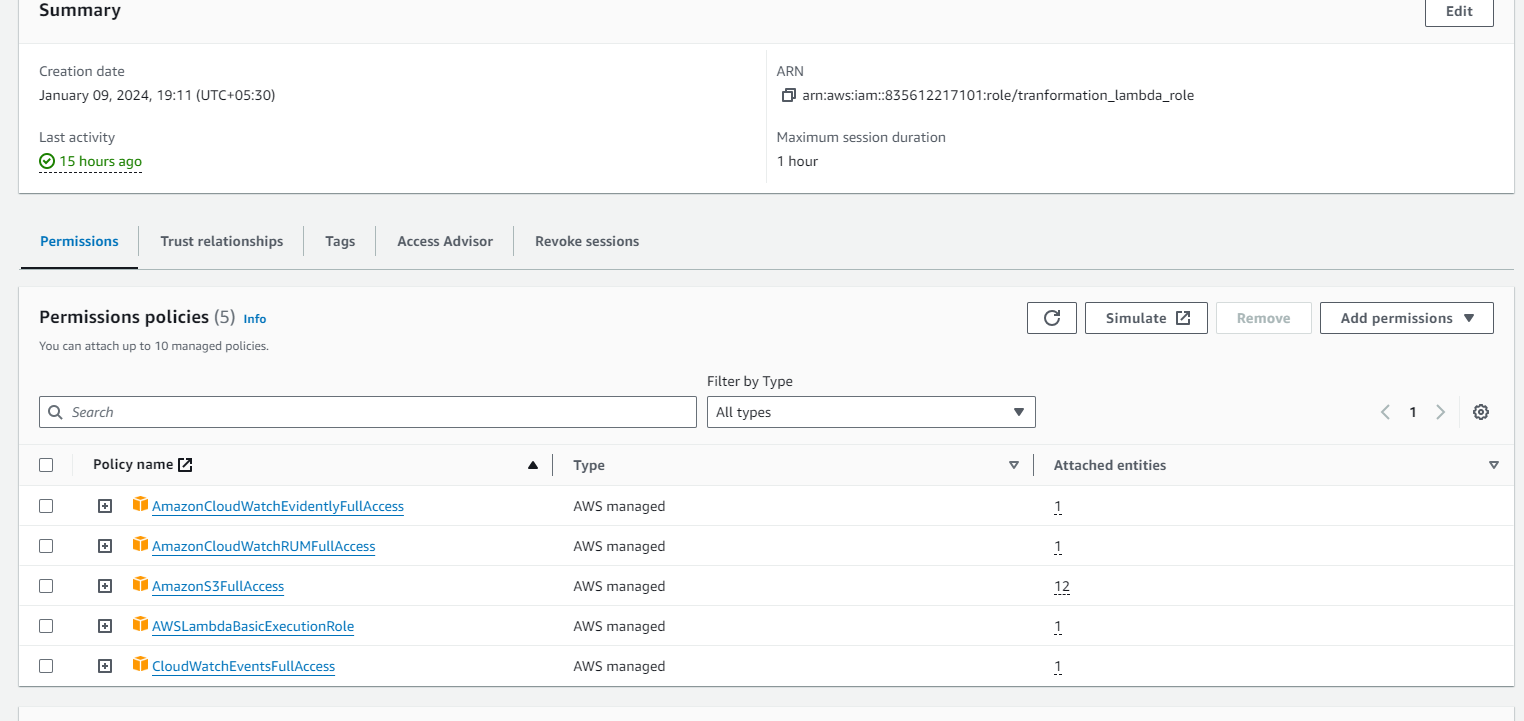
2.Select the runtime as Python 3.11 and architecture as x86\_64



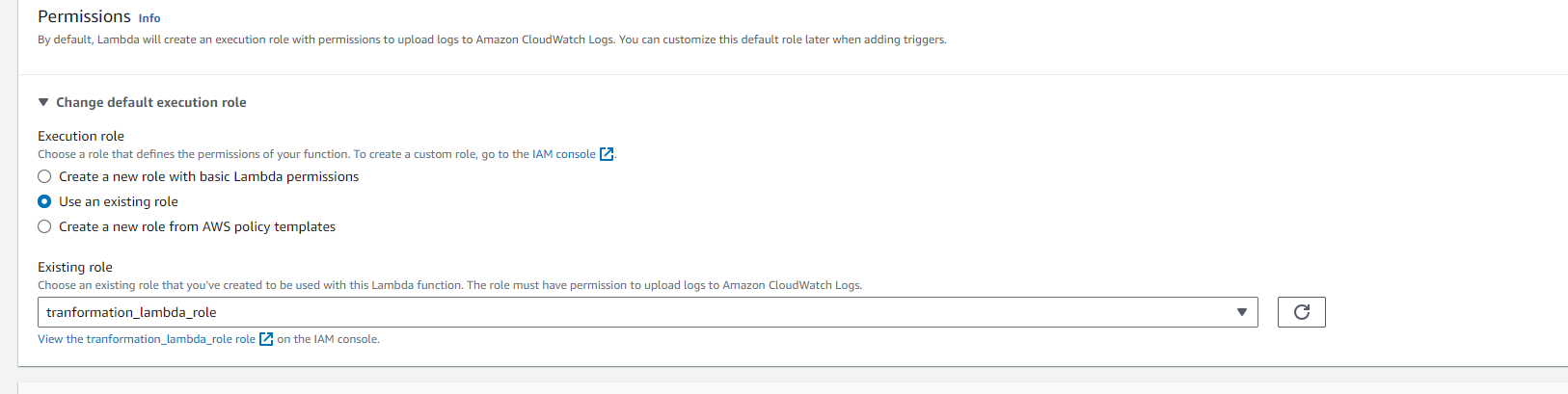
3.Then give suitable permissions to the lambda . For this create an IAM role that will allow the Lambda to interact with S3,i.e. read from S3 and write back to S3.Create a IAM role transformation\_lambda\_role with the following permissions:

a .Cloudwatch full access : to monitor Cloudwatch logs for lambda

b. S3fullaccess:to read FROM and put object back to S3 bucket

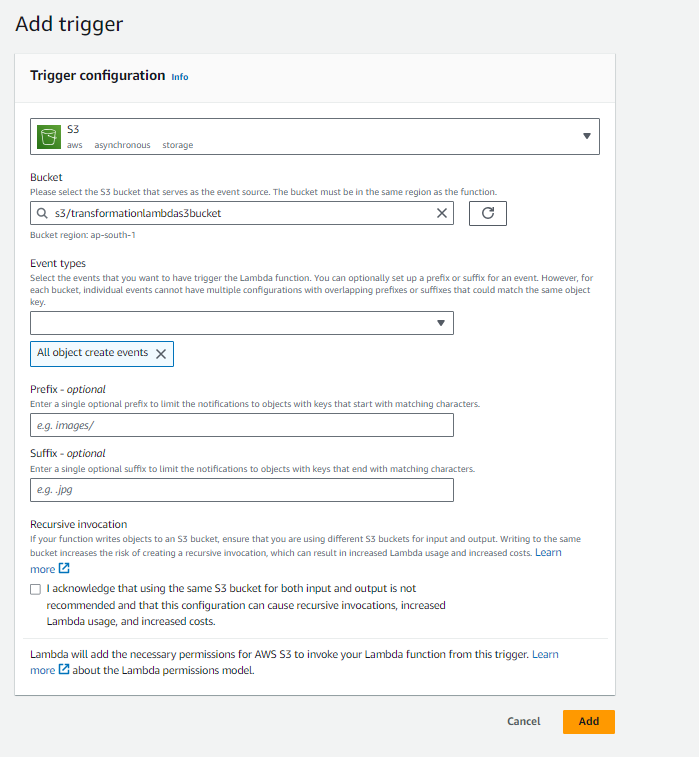


4.Attach this IAM role to lambda

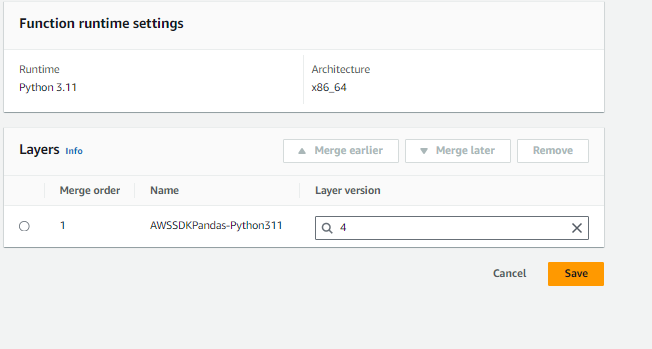


4.Click on create function .

5.Create a trigger based on which the Lambda function will be invoked. Click on S3 ,select the bucket that will act as the event source and select event types as all object create events.



6.We need to add a pandas layer to our lambda .for this we use a pandas layer that’s already present .this can be obtained using the ARN of the source, also the architecture and runtime should be specified to avoid compatibility issues



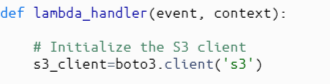
7.Under general configuration we set the timeout and storage.

8.Now we come to the main code .

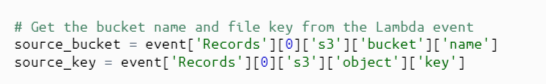
9.Import all the necessary modules.

10 .Define the lambda .

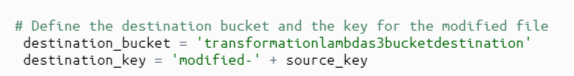
11.Create a connection using boto3 that is used to interacting with S3.



12.Extract details of the source bucket and key from the trigger event.

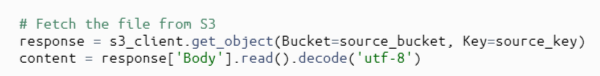


13.Provide details of S3 bucket and key where the output file will be uploaded to



**read()** method reads its contents into memory.

**response['Body']**: This accesses the **Body** attribute of the response. The **Body** is a streaming object provided by AWS, which contains the binary content of the S3 object. Since the file content is in binary format, it needs to be decoded into a string for it to be usable in Python. The **.decode('utf-8')** method decodes the binary content into a UTF-8 encoded string. UTF-8 is a standard text encoding format that can represent a wide range of characters

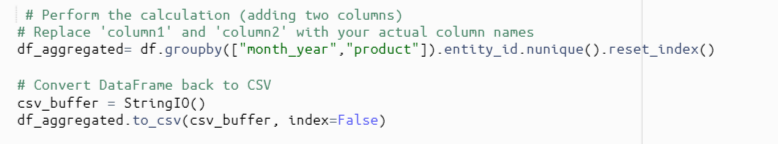


15.  **String IO(content)**: **StringIO** is a class from Python's **io** module. It allows to treat a string as a file-like object. This is useful because **pd.read\_csv** typically expects a file path or a file-like object as its input. Since we have the CSV data as a string (**content**), not an actual file, **StringIO** is used to simulate a file-like object from that string. This allows **pd.read\_csv** to read the CSV data from the string as if it were reading from an actual file.

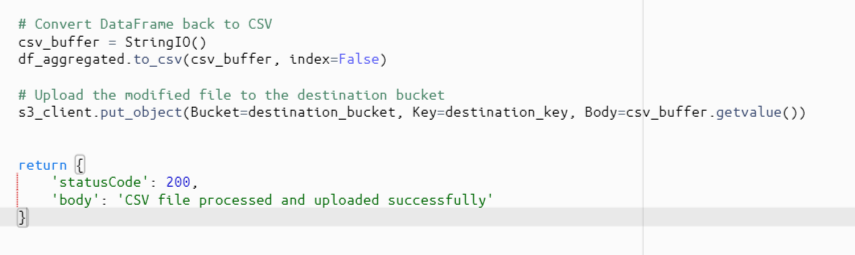
**content**: This is the string variable containing the CSV data that was previously read from the S3 object

15.Do aggregation and assign it to a new dataframe.

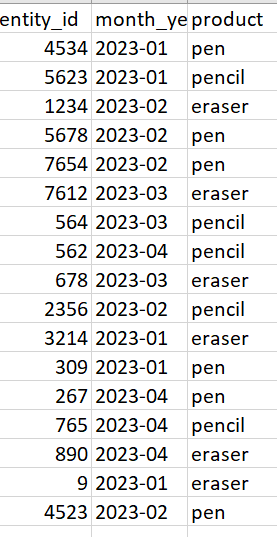
16. **StringIO()**: This is a class from Python's **io** module. It creates an in-memory file-like object. This object can be used as a temporary storage space (or buffer) for data. The data stored in this buffer can be accessed and manipulated just like file data, but it resides in memory and does not require a physical file on the disk.



16.This dataframe is written back to in memory file object which is then uploaded to destination bucket.

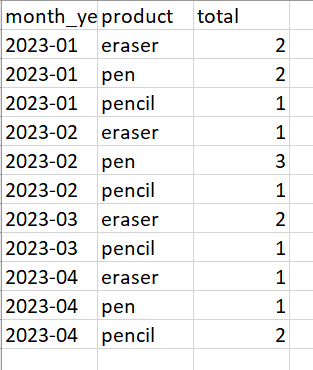


**Input file**



For seeing the logs of the Lambda function ,use cloudwatch

**Output file**



We also define a test condition to check if the lambda function is working properly or not.

