Small-Scale Serverless Processing and Integration

Learning objectives

- Understand the need for serverless processing within data pipelines.
- Justify serverless processing for small transformation and integration tasks.
- Implement a solution to ingest data from an API using AWS Lambda.
- Learn about other serverless transformation and integration examples.

Table of contents

1. An era before serverless computing

Servers could be said to be the backbone of all the activities that occur over the internet. They are needed to facilitate the computation, storage, and communication of online artefacts. Therefore, it comes as no surprise that you need a server to have an application running on the web. In the past, it was common procedure to purchase a physical machine that would act as the server or approach a service provider to rent out their servers for a monthly fee. In addition to building and maintaining an application, this meant that developers had to look after the servers to ensure security, OS stability, and other infrastructure dependencies. Often the work required to maintain the servers proved to be taxing and slowed the productivity of the developers. An improved approach was to have additional personnel looking after the servers. Although this freed up some of the developers' time, it also meant that more money was being spent on hiring additional staff.

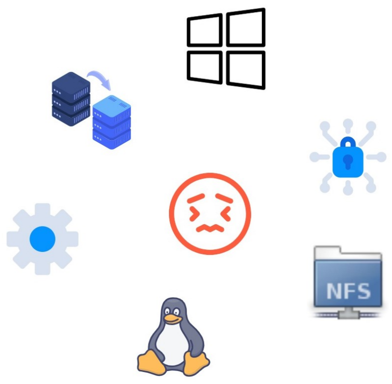


Figure 1: Owning physical servers meant you were also in charge of their maintenance.

Furthermore, having physical servers meant that scalability was limited to the capacity of the commissioned servers. Attempting to increase the scale at which you were operating meant you needed to purchase new physical servers or rent more. This introduced some overheads because you had to revisit the tedious process of configuring the infrastructure to meet the new server specifications or configuring the new infrastructure to be on par with the server specifications in the production environment. To circumvent these issues, virtual machines were introduced. Virtual machines allowed us to abstract the physical hardware so that we only had to deal with the software required to make the server operational. This advancement enabled us to have multiple logical servers (VMs) running on a physical host server â€“ also known as server virtualisation. The virtual machines could be deployed to address different needs, such as the segregation of development and regression testing environments. Developers could finally commission new servers without having to purchase or rent new infrastructure.

At this point, virtual machines seemed like the be-all and end-all of virtualisation. It helped us to abstract the physical infrastructure. However, the reality was that the VM's operating system still needed to be managed by the developers. This can easily turn into a nightmare if you are managing multiple environments (VMs) to accommodate the different needs of the application development cycle. As it turns out, this step could further be eliminated. We then saw the introduction of containers, which are modular units of software that contain both the code and system dependencies required to execute your application. We no longer needed a stand-alone VM to run an application. Instead, we could create small containers which were OS-agnostic to execute on a physical or logical server. With containers, it meant that we could quickly test and deploy applications without having to worry about the underlying operating system dependencies â€“ another win for developers.

Having read this far, it should be apparent that laziness breeds innovation. With all the progress and advancements made to unburden developers, surely they should be more content with the current state of affairs... apparently not. The rise of cloud computing was the excuse they needed to rid themselves of servers entirely. In the following sections, we dive into the realm of serverless computing and how they have made the application creation process a less daunting task.

2. Serverless computing

Serverless computing refers to a cloud service architecture in which the infrastructure required to run an application is managed on behalf of the client (for example, the developer). Although we refer to it as *'serverless'*, it is a bit of a misnomer as the application code is still executed on a server. Serverless in this context refers to how the server is managed and maintained â€“ by the cloud provider and not the client.

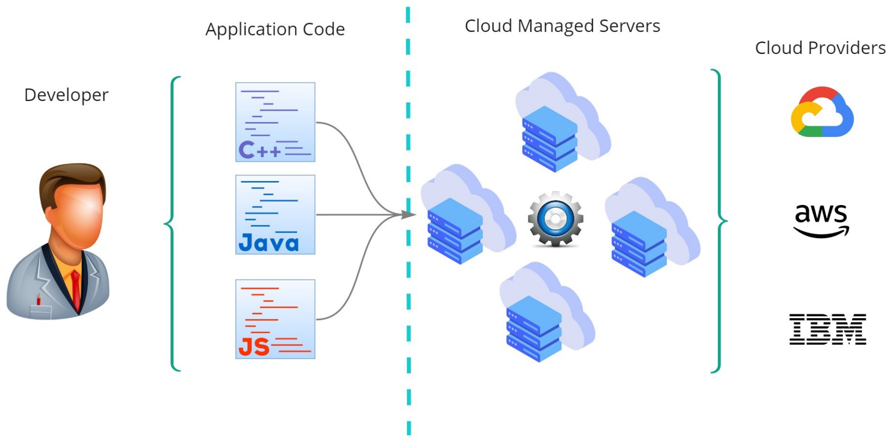


Figure 2: Serverless computing.

Having the servers managed by a cloud provider means that from a developer's perspective, they no longer have to deal with issues relating to routine maintenance and administration tasks such as provisioning resources, operating system upgrades, security patches, and so forth. More than that, the execution cost is fixed for each execution of the serverless function, and the client does not have to pay for the time a cluster is active but not in use. It is for this reason that it is often called a Function-as-a-Service (FaaS).

Freeing a developer from the pains of configuring and maintaining a robust server means they can focus purely on the application code. As a data engineer, this can speed up the creation of data pipelines required to ingest, manipulate, and store data in relevant target locations. All of this can be done without having to worry about the intricacies involved with computing resource allocations, cost, and other aforementioned considerations.

Furthermore, not all ETL tasks require the constant up-time of conventional server-based applications. To a large extent, data pipelines can be event-driven. Data can be ingested if changes are made to the existing data source or if we require updated versions of the data every hour. Both of these actions are event or schedule-based, which means they naturally lend themselves to serverless processing. Provisioning a fully-fledged server to form part of the pipeline can result in you paying for idle resources. If the server only executes every hour for one minute, the server was only ever required for 24 minutes per day. With serverless computing, you would only pay for the 24 minutes of computation used, instead of paying for the up-time of a server.

Serverless computing serves us well when dealing with applications that do not need low-latency performance. Low-latent applications require services that are running idle in the background, ready to execute when needed. With a serverless architecture, your application is started every time it is evoked (via an event or schedule) and then stopped when it has completed. So over and above the computation time of the application, the restart will drastically affect the latency, resulting in poor performance. Therefore, you should exercise caution when selecting your architecture. Serverless has great benefits, but does it suit your business needs?

3. Serverless transformation and integration

When designing a data pipeline there are often transformation tasks and integration points that you need to address.

One of the most basic tasks would be to connect to an API and send requests to retrieve data. In this instance, you are using serverless processing as an integration layer between the API and your data pipelines. Here you would construct a serverless function that can authenticate against the API, perform the correct API calls, and ingest the data into a location from where you can access the data again using your data pipelines (for example, blob storage).

After receiving data from an API, you would then prepare it so that it can be loaded into the appropriate target destination. We sometimes prefer to ingest data in its raw format and dump it in a data lake or warehouse. However, sometimes itâ€™s not as simple as dumping the data into a file location. More often you have to transform the data so that it's ready for the target source. An example of this may be using a function to transform incoming data in JSON format into a more structured format, such as a table structure for a relational database. It is important to keep in mind that the idea is not to change the table structure from the serverless function, but rather, to prepare it so that it conforms to the existing table structure.

One should avoid the temptation to perform data aggregation at this step. Only deal with the small data transformations that are required. Analytical tasks such as complex computations and aggregations required to obtain insights should be reserved for use on a target source, which could be a data lake or warehouse. Data aggregation and transformation segregation will serve you well when writing your code or debugging your application. If the work is separated, it means you only have to look at different modules when applying software fixes and upgrades.

Serverless processing greatly complements the streaming paradigm. Serverless functions can becombined with streaming services such as AWS Kinesis to transform data as part of a streaming pipeline, before being loaded into a database, data warehouse, or data lake.

As a data engineer, you may also be required to expose data to other teams within your company. In most instances, you would be able to expose data through a database or data lake, however, sometimes you deal with very large enterprises where security may be of concern, isolation of who gets access to what, and self-service is more important, in which case you cannot give access to entire databases. Serverless functions also shine here. They allow you to create a very lightweight API from which users can then request data for their own use.

4. Practical exercise

Now that you understand what serverless computing entails, let's move on to build a simple data pipeline. Before you proceed, we'll set the scene with a case study.

A small financial services company, FX Inc., which mainly deals with foreign exchange, wants to get ahead of the game in terms of its product offering. After researching current trends, it was decided that a data-driven approach was the way to go. However, there was one small caveat. Previously, their investments were not technology-focused, and as such, they do not have the in-house expertise to set up the necessary infrastructure to get them started with their journey.

As a data engineer, you were hired to assist FX Inc. in its new journey. Your manager explains that the company is looking to track the G10 currencies alongside gold in the commodities market to understand the impact that one has on the other and hopefully spot a correlation. The project requirements are as follows:

- Use available financial market APIs to source data.
- Extract the G10 currencies.
- Append the gold market data to the currency data.
- Store the data into a database.
- The process needs to execute daily (at the close of business) to capture closing prices in the commodities market.

After considering the requirements and taking into account the limited manpower within the company, you realise that serverless is your best option if you want to reduce upfront costs. You proceed to draft the below architecture to present to the stakeholders.

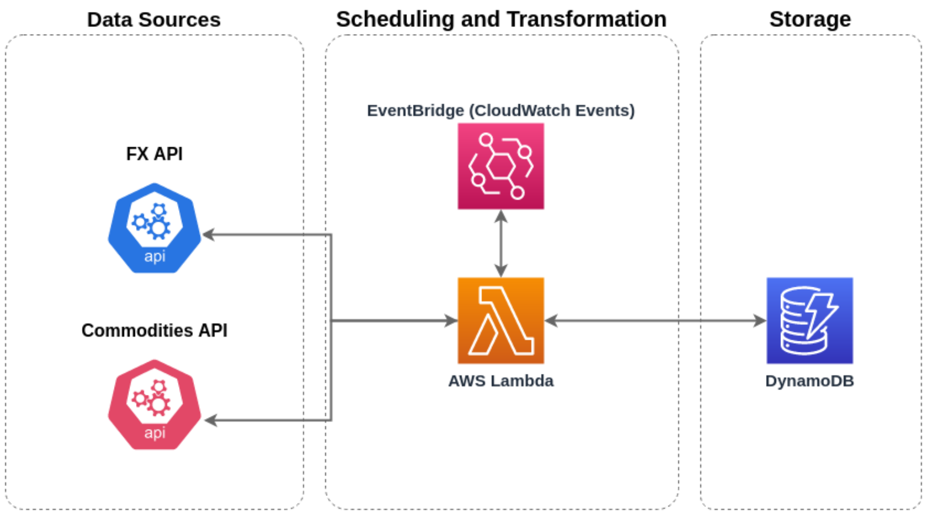


Figure 3: Event-driven data pipeline.

The stakeholders are happy with your approach and give you the go-ahead to implement the suggested architecture.

4.1. APIs

To source the data, you have identified the following APIs for your implementation:

1. Exchangerate.host to source the currency data.
2. Goldapi.io, which requires you to have an API key to access the data. Click on the **Get FREE API Key Now or Sign In** button at the bottom of the page to retrieve your API key.

   Note: You are encouraged to make an effort to read the documentation that accompanies the API. This will give you an understanding of how the endpoints work and how to make use of parameters to filter out data that is required.

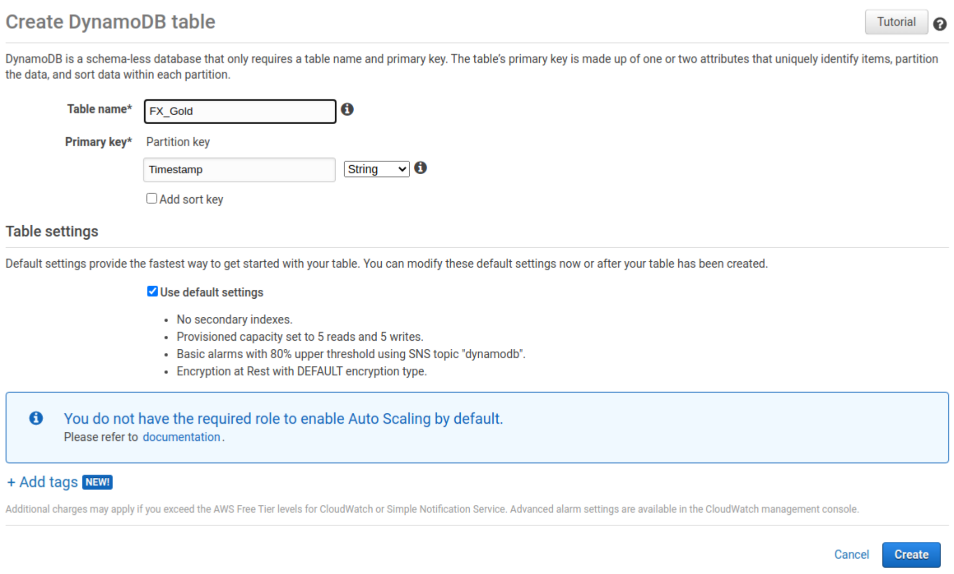
4.2. Serverless database: DynamoDB

Now that the data sources have been identified, the next logical step is to create a target source to store the data retrieved from the API. To keep the architecture serverless, Amazon's DynamoDB will be employed for storage purposes. By using DynamoDB, charges are only incurred on read/write operations and storage used, meaning that charges will only accumulate in proportion to usage. As you're using a serverless database you do not have to worry about the maintenance tasks required to keep the database afloat.

To create the database, open the AWS console and perform the following steps:

1. In the **Services** palette, navigate to **DynamoDB**, which can be found under the **Database** section.
2. Click on **Create table** to configure the table properties.
3. Enter the following details for the table:

   Table name: FX_Gold \ **Primary key**: Timestamp


Create DynamoDB table

DynamoDB is a schema-less database that only requires a table name and primary key. The table's primary key is made up of one or two attributes that uniquely identify items, partition the data, and sort data within each partition.

Table name* FX_Gold

Primary key* Partition key

Timestamp String

☐ Add sort key

Table settings

Default settings provide the fastest way to get started with your table. You can modify these default settings now or after your table has been created.

☑ Use default settings

- No secondary indexes.
- Provisioned capacity set to 5 reads and 5 writes.
- Basic alarms with 80% upper threshold using SNS topic "dynamodb".
- Encryption at Rest with DEFAULT encryption type.

You do not have the required role to enable Auto Scaling by default.
Please refer to documentation.

Additional charges may apply if you exceed the AWS Free Tier levels for CloudWatch or Simple Notification Service. Advanced alarm settings are available in the CloudWatch management console.

Figure 4: Create a NoSQL database with DynamoDB.

4.3. Create Lambda function

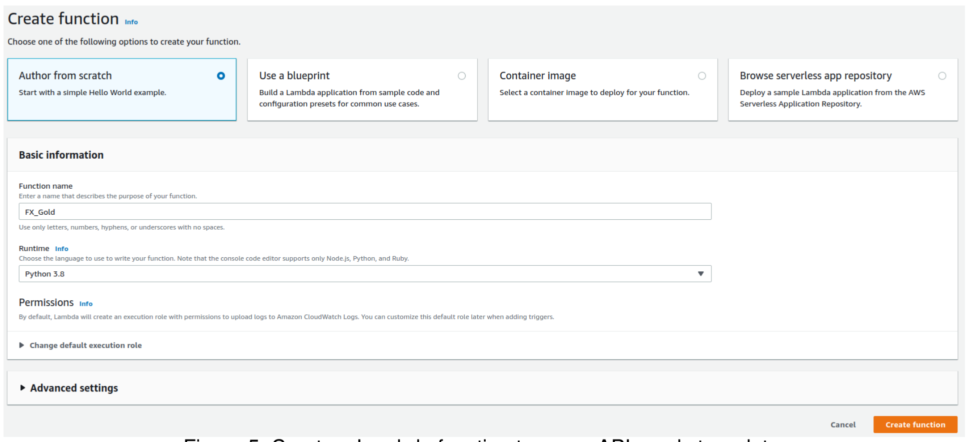
The APIs for the implementation have been identified and the serverless database created. The next step is to set up the serverless computing resources â€“ AWS Lambda will be used for this purpose. By using AWS Lambda, you'll only be charged for the computation resources used to execute your code, as opposed to paying for the up-time of a conventional server. Moreover, it comes with the added benefit that the server is entirely managed by the cloud provider, so no human or monetary resources have to be allocated to server maintenance and administration. We'll create a Lambda function to *request* data from the external APIs.

Through the AWS Management Console, create a Lambda function to query the API, transform the data, and store the result into the database:

1. In the **Services** palette, navigate to **Lambda**, which can be found under the **Compute** section.
2. Click on **Create function**.

3. Make sure you have **Author from scratch** selected, and enter the following details under **Basic Information**:

    Function name: FX_Gold\ **Runtime**: Python 3.8



Create function Info

Choose one of the following options to create your function.

Author from scratch [✓]
Start with a simple Hello World example.

Use a blueprint
Build a Lambda application from sample code and configuration presets for common use cases.

Container image
Select a container image to deploy for your function.

Browse serverless app repository
Deploy a sample Lambda application from the AWS Serverless Application Repository.

Basic information

Function name
Enter a name that describes the purpose of your function.
FX_Gold
Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime Info
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.
Python 3.8

Permissions Info
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

Figure 5: Create a Lambda function to query APIs and store data.

4. Since you'll need to insert records into DynamoDB, you need to attach a policy that permits you to do so. Select your newly created Lambda and navigate to **Configuration**>**Permissions**. Under **Execution role**, click on the role name, and then attach the **AmazonDynamoDBFullAccess** inline policy.

4.4. Python script

Now that the Lambda function has been set up, you need to write the application code to achieve the desired result. For this step, a Python script named `lambda_function.py` is located in the directory. You will need to edit this file to get the Lambda function working.

Step 1: Connect to the APIs that were earmarked as the data source.

- Following the project requirements, you'll only need to make use of the G10 currencies. Make sure you set the base currency to 'USD'.
- For the commodities API, make sure that you provide the API key you obtained from Goldapi.io.

```
parameters = {"base": <base_currency>,"symbols":<list_of_G10_currencies>}
currency_url = "https://api.exchangerate.host/latest"

# Commodities API endpoint.
commodities_url = "https://www.goldapi.io/api/XAU/USD"

# Include the API key in the headers as part of your API call.
headers = {
'x-access-token': '<Insert your API key here>', ## <-- Insert your API key here.
'Content-Type': 'application/json'
   }

# Send a get request call to the API to obtain commodity information.
commodities_resp = requests.get(url=commodities_url, headers=headers).json()
currency_resp = requests.get(url=currency_url, params=parameters).json()
```

Step 2: Prepare the data to be inserted into DynamoDB by introducing transformations.

- Change the timestamp from the commodities API into a DateTime object.
- Create a dictionary to store the incoming data from the API. For the commodities API you are interested in the following attributes:
    - The previous day's gold closing price.

    - The opening gold price.
    - The current gold price.

```
    # Convert the timestamp to a human-readable DateTime object.
    timestamp = commodities_resp['timestamp']
    timestamp= None #<-- TODO: Write code here to convert timestamp accordingly

    # Initialise a dictionary that will be used to store the data from the two API calls.
    record = {}

    # Insert data into the dictionary using data from the commodities and foreign exchange API.
    record['Timestamp'] = timestamp.strftime("%Y-%m-%d, %H:%M:%S")

    # Use the appropriate API response to insert the data into record dictionary.
    record['GoldPrice'] = None          #<-- TODO: Write code here
    record['GoldClosingPrice'] = None   #<-- TODO: Write code here
    record['GoldOpeningPrice'] = None   #<-- TODO: Write code here
    record['BaseCurrency'] = None       #<-- TODO: Write code here
    record['Rates'] = None              #<-- TODO: Write code here
```

Step 3: Connect to DynamoDB and insert the newly created record into the table. You will need to use the AWS SDK to achieve this.

```
# Use AWS SDK to instantiate a DynamoDB object.
dynamodb = None  #<-- TODO: Write code here
table = None     #<-- TODO: Write code here

# Float types are not supported by DynamoDB so you parse the float data type as Decimal.
record = json.loads(json.dumps(record), parse_float=Decimal)

# Insert a new record into DynamoDB.
#TODO: Write code to put an item into the DB
```

4.5. Layers

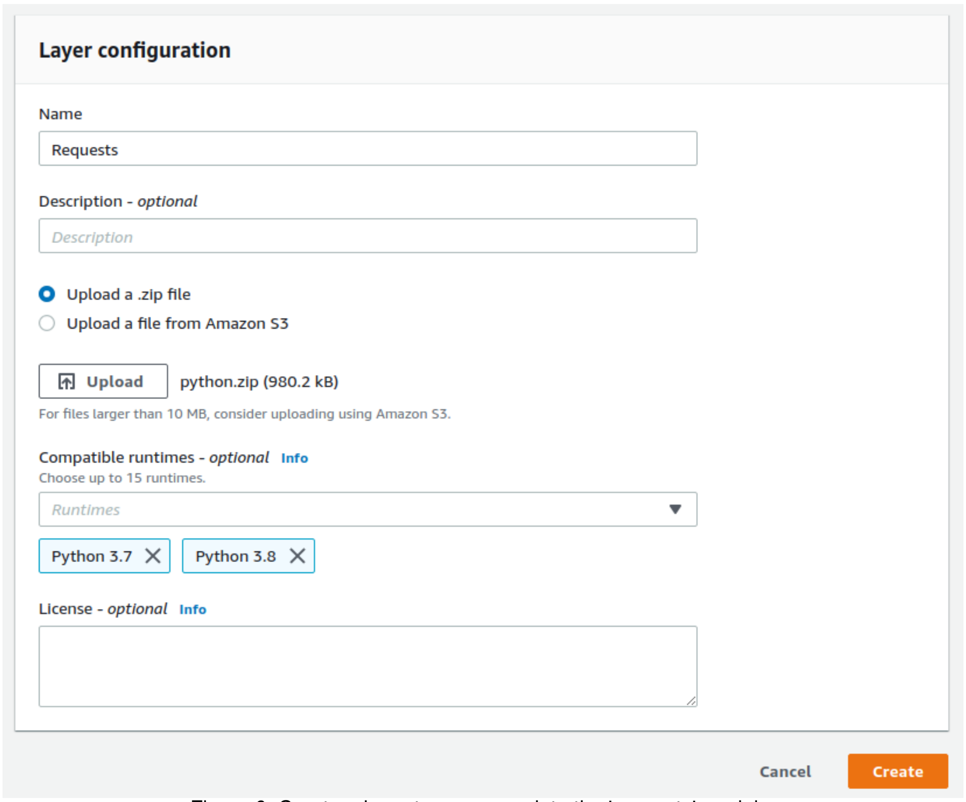
If you try to run the completed version of your code, you will run into issues on the*requests* module. This simply means that the module is not available for the current runtime and you need to create a layer to ensure it's supported. The layers in the Lambda function allow you to upload any dependencies that might be required to run the application code.

You will need to create a new layer to accommodate the*requests* module used in the Lambda function. To do this, enter the following commands into your terminal:

```
mkdir python
cd python
pip install requests -t ./
```

This will create a new folder called`python`, and install *requests* binary files into this location. Once that is done, zip the folder in preparation for an upload.

1. In the left-hand navigation pane, go to **Layers** and then click on **Create Layer**.
2. In the **Layer configuration**, add the following details:
    - **Name**: Requests
    - Select the **Upload .zip file** radio button and upload the python folder that you've just zipped.
3. Click **Create** to complete the set-up.


Layer configuration

Name

Requests

Description - *optional*

Description

● Upload a .zip file

○ Upload a file from Amazon S3

Upload python.zip (980.2 kB)

For files larger than 10 MB, consider uploading using Amazon S3.

Compatible runtimes - *optional* Info

Choose up to 15 runtimes.

Runtimes

Python 3.7 Python 3.8

License - *optional* Info

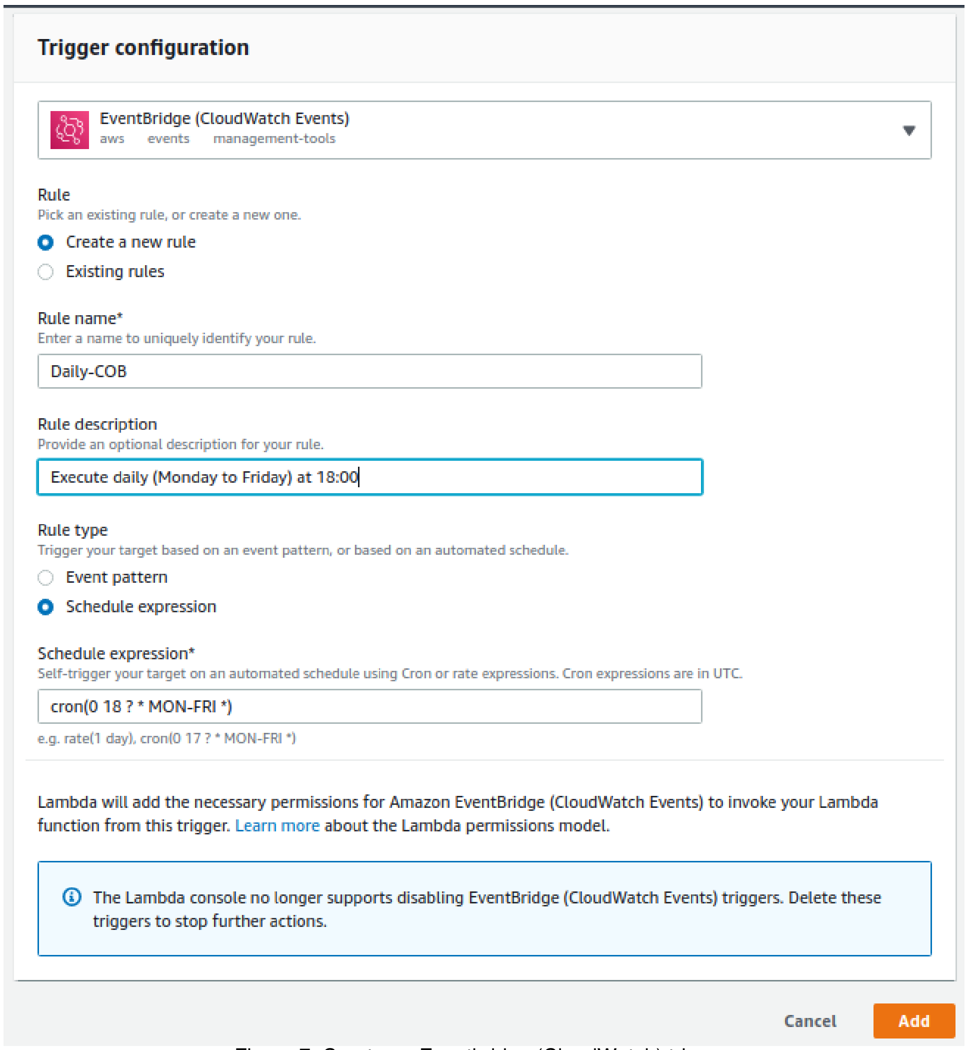
Cancel Create

Figure 6: Create a layer to accommodate the 'requests' module.

4.6. Triggers

The architecture that you've designed is finally coming together. The only thing missing is running the data pipeline on a schedule. The requirement is to have it run daily at 18:00 to try and capture the closing prices in the commodities market. To do this, you'll make use of **CloudWatch Events**.

1. Navigate back to your Lambda function and click **Add trigger**
2. Under **Trigger configuration**, select **EventBridge (CloudWatch Events)**.
3. Click on the **Create a new rule** radio button.
4. Give your rule a name: **Daily-COB**, and give it a description (optional).
5. For the **Rule type**, make sure that you select **Schedule expression**.
6. For the **Schedule expression**, enter the following: `cron(0 18 ? * MON-FRI *)`.


Trigger configuration

EventBridge (CloudWatch Events)
aws events management-tools

Rule
Pick an existing rule, or create a new one.
[✓ Create a new rule]
Existing rules

Rule name*
Enter a name to uniquely identify your rule.
Daily-COB

Rule description
Provide an optional description for your rule.
Execute daily (Monday to Friday) at 18:00

Rule type
Trigger your target based on an event pattern, or based on an automated schedule.
Event pattern
[✓ Schedule expression]

Schedule expression*
Self-trigger your target on an automated schedule using Cron or rate expressions. Cron expressions are in UTC.
cron(0 18 ? * MON-FRI *)
e.g. rate(1 day), cron(0 17 ? * MON-FRI *)

Lambda will add the necessary permissions for Amazon EventBridge (CloudWatch Events) to invoke your Lambda function from this trigger. Learn more about the Lambda permissions model.

The Lambda console no longer supports disabling EventBridge (CloudWatch Events) triggers. Delete these triggers to stop further actions.

Figure 7: Create an Eventbridge (CloudWatch) trigger.

5. Integration on Microsoft Azure

There is more than one way to skin a cat, and the same goes for serverless processing in the cloud. In this section, we introduce you to serverless processing on Microsoft Azure and some of the advantages that you are afforded by utilising the Azure cloud. Microsoft Azure has two services for serverless processing: an AWS Lambda equivalent â€“ Azure Functions, and a user interface-based version â€“ Azure Logic Apps.

5.1. Azure Functions

Similar to Lambdas, Azure Functions allow you to write blocks of code and organise them into functions to be executed in response to events. Similarly, Azure Functions can scale indefinitely in response to changes in demand. Some of the use cases include:

- building a web API;
- processing incoming files;
- building a serverless workflow;
- responding to database changes;
- running scheduled tasks; and
- processing data in real-time, and processing data from streams or message queues.

All but one of the above use cases are stateless â€“ building a workflow. What does this mean? For all the above use cases, individual functions do not need to be aware of other runs and do not have to transfer data to any of the other functions. In contrast, with a workflow, you build a pipeline with multiple functions strung together, between which you have to transfer the state and keep track of the files that have been processed and those that have not. This is where durable functions come in. Durable functions are somewhat unique in the serverless processing world in that they allow you to keep track of variables each time the function is executed. As a result, you can write orchestrator functions to orchestrate pipelines and entity functions to allow you to keep track of stateful entities.

5.2. Azure Logic Apps

Sometimes you need to build a pipeline in collaboration with a non-technical person, or you are building a data pipeline that will be maintained by a team of business analysts who are not well versed in programming. Fortunately, this is where services such as Azure Logic Apps and AWS AppSync come in.

Logic Apps provides you with a way to create automated workflows to integrate multiple apps, data, services, and systems. This could be as simple as transferring a file from Azure Storage to a database when the file arrives in storage, or by querying an API on a regular schedule to create a file in storage. Logic Apps have integrations with a very large number of connectors, which allows you to integrate with other services inside and outside of the cloud and across cloud providers. The visual interface allows you to easily explain the app to non-technical users and lets you hand it over to them for maintenance. Supporting this is a JSON configuration of the Logic App, which ensures that you can edit the Logic App in code, and you can use standard DevOps practices for version control and code deployment.

6. Conclusion

You have explored the details surrounding serverless computing in the context of data pipelines, and by doing so, you were able to apply your knowledge to build a small pipeline to ingest data from two different API sources. This train captures the advantages that are associated with opting for a serverless architecture which include:

- saving on human and capital resources as there is no need to manage servers in-house;
- upfront costs are reduced as you only pay for the usage of computing and storage resources and not the up-time of a running server; and
- inherent scalability is provided as you do not have to worry about provisioning resources as the data pipeline matures.

7. Resources

- Serverless Computing
- Serverless Databases
- Event-Driven Architecture