# Przetwarzanie Danych w Chmurze Publicznej **Laboratorium**

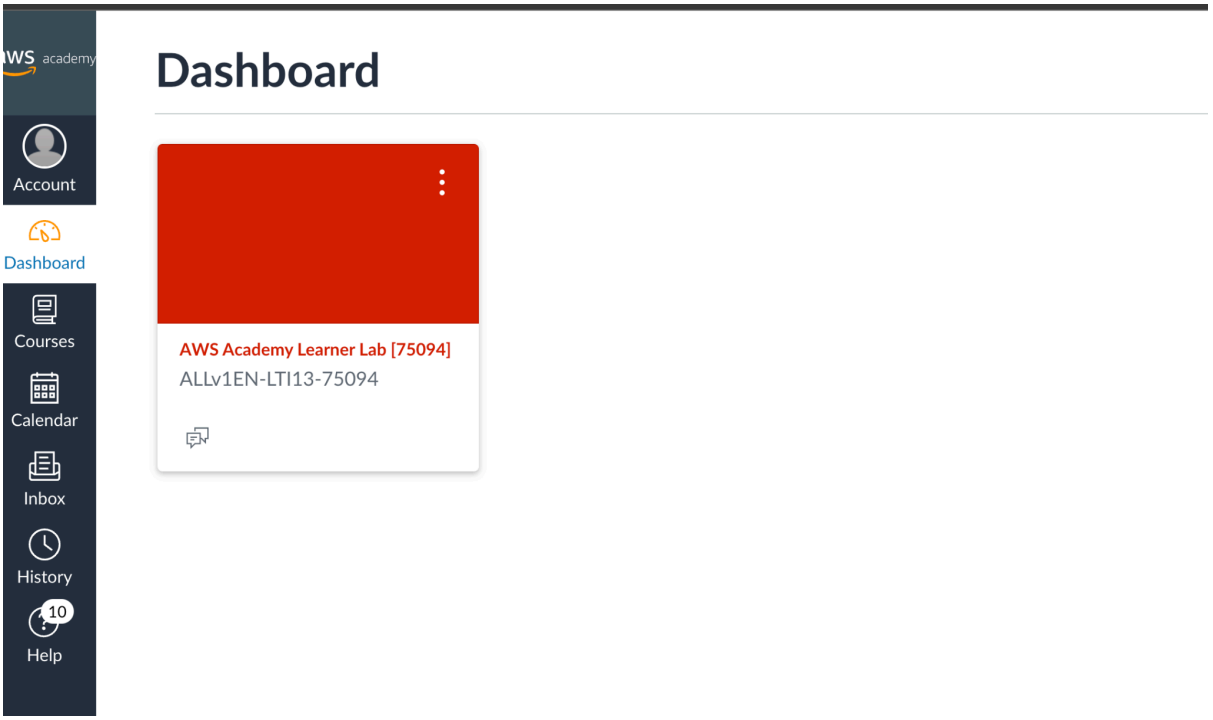
---

## Przygotowanie środowiska

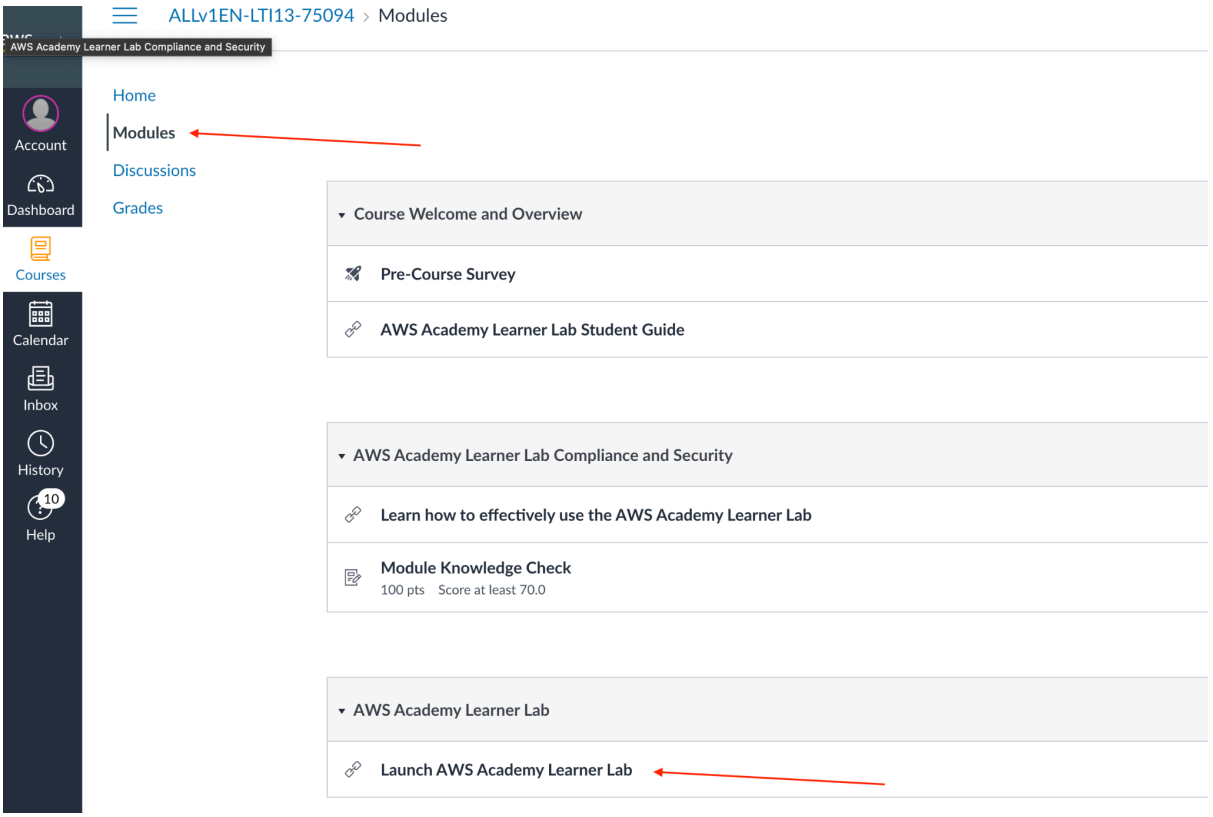
### Dostęp do konta AWS

Całość laboratoriów robimy na koncie AWS Academy Learner Lab.

1. Zaloguj się na https://awsacademy.instructure.com/ i otwórz labs (zaproszenia zostały wysłane na Twoją skrzynkę UAM):

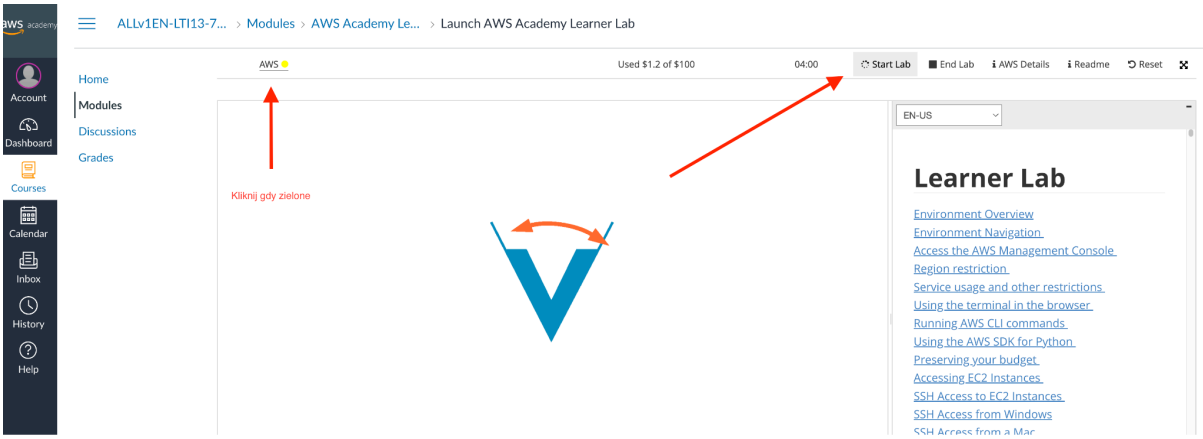


2. Wejdź do modules i uruchom **moduł AWS Academy Learner Lab**




3. Wystaruj laboratorium (**Start Lab**) a następnie kliknij AWS, gdy kółko zmieni się na zielone.
   Otworzy się konsola AWS z której możesz wykonać całość laboratoriów (będziemy pracowali na Cloud9).

   W zakładce “AWS Details” możesz znaleźć tymczasowe AWS Credentials jeśli chcesz pracować z tym kontem z własnego środowiska (local development)

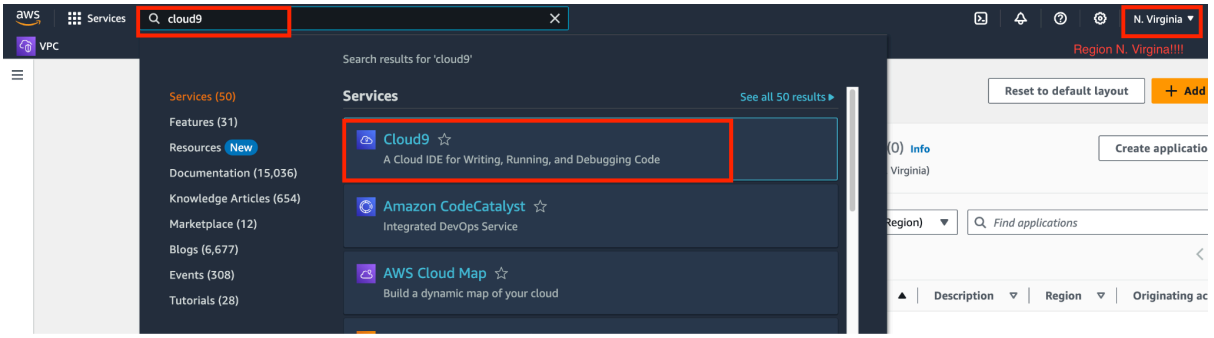


## Setup środowiska (Cloud9)

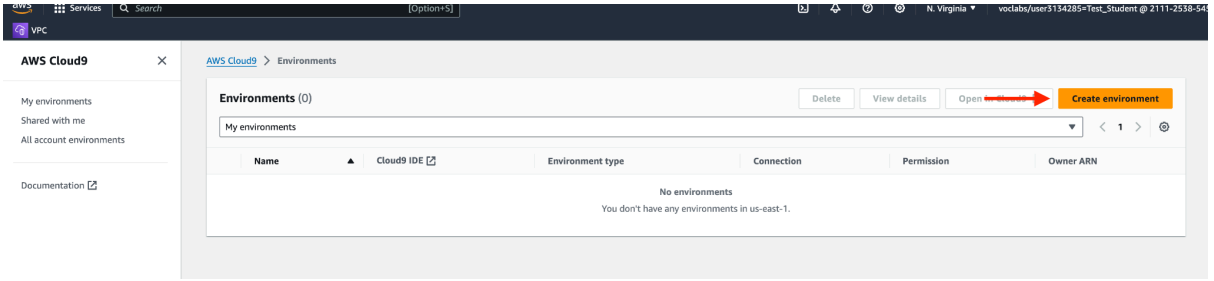
W tym kroku utworzysz nową instancję środowiska Cloud9

1. Otwórz serwis Cloud9 w konsoli AWS (poprzedni krok).

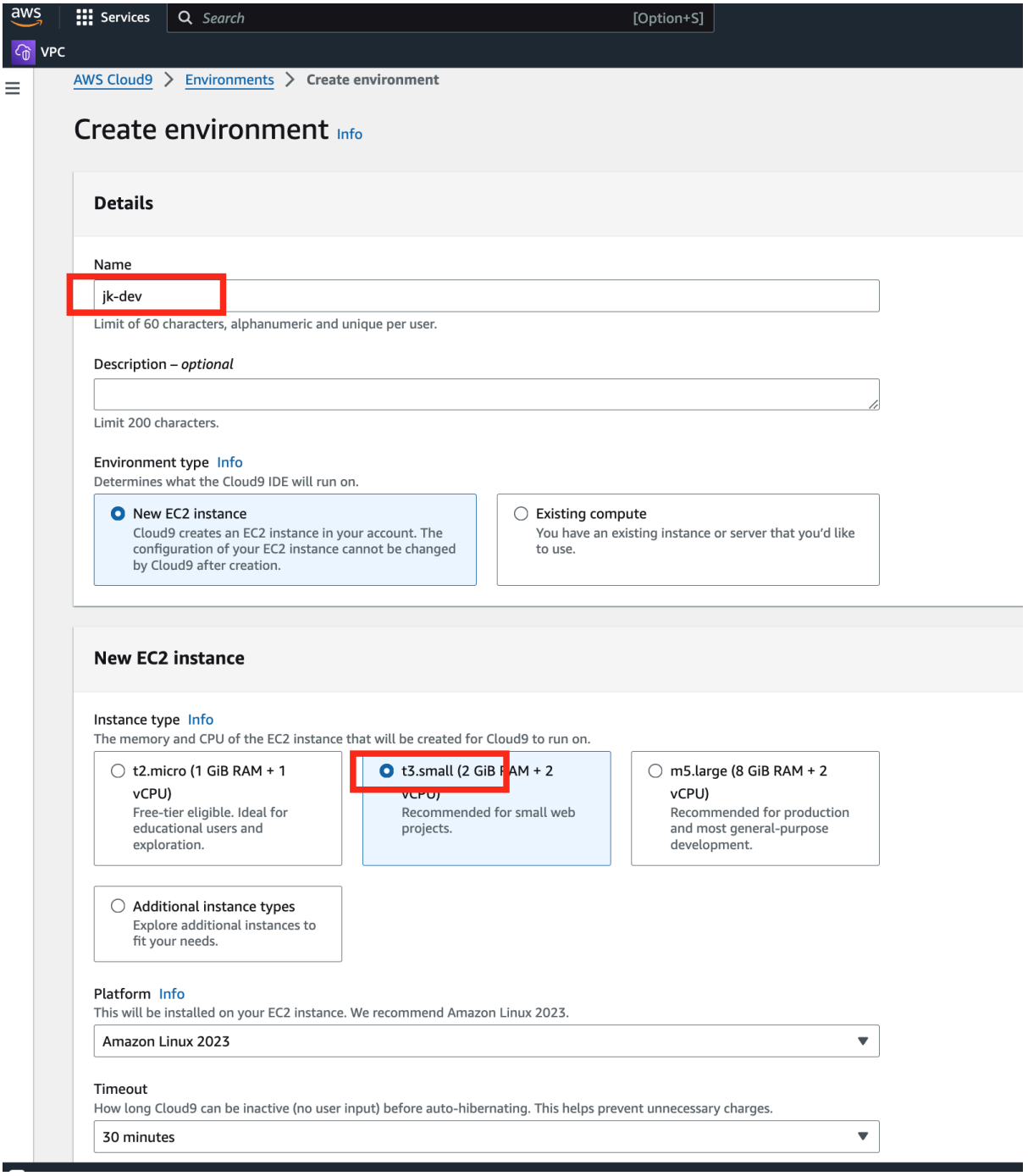
   Zwróć uwagę w którym regionie AWS jestes (pracujemy w **N. Virginia** us-east-1)



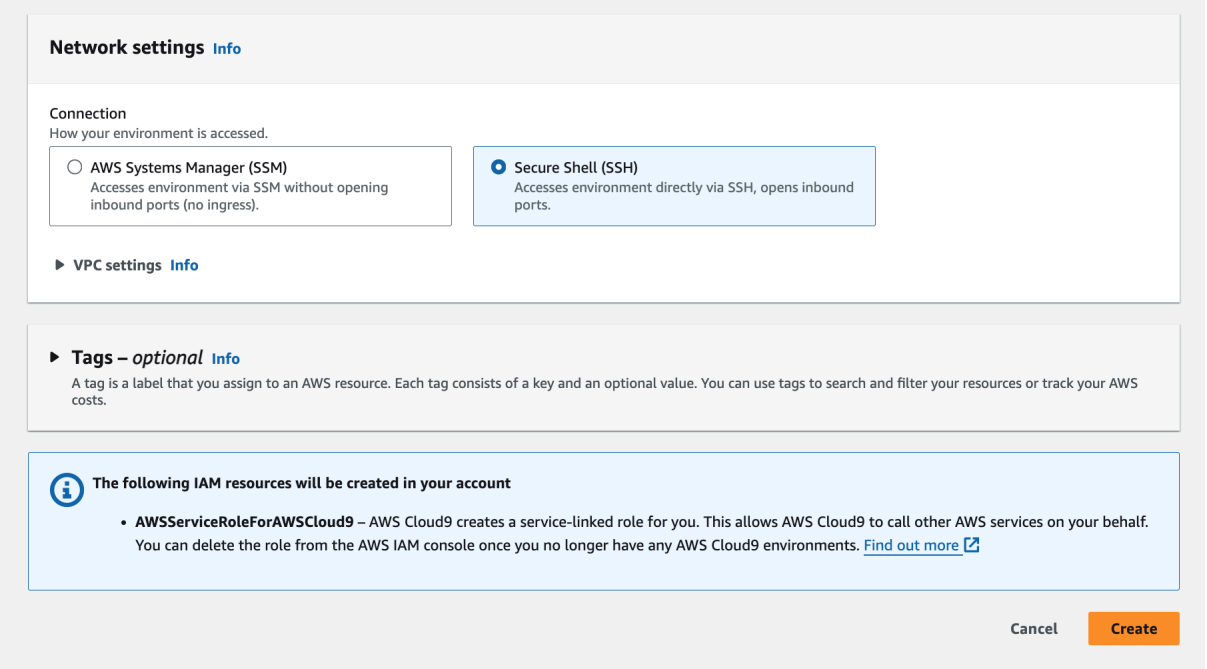

2. Utwórz nową instancję Cloud 9



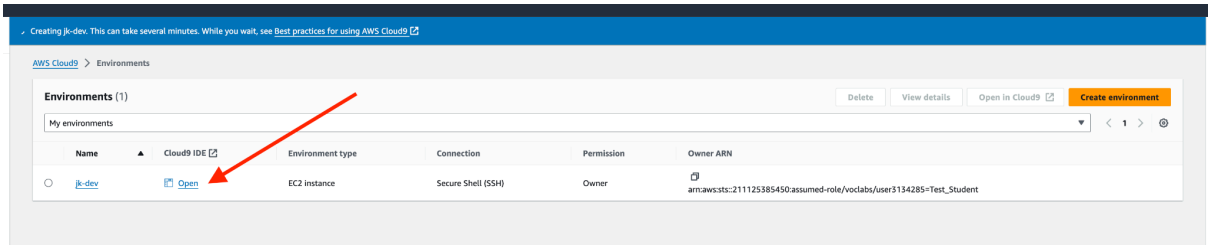
Nazwij ją <inicjały>-dev, wybierz rozmiar maszyny t3-small:



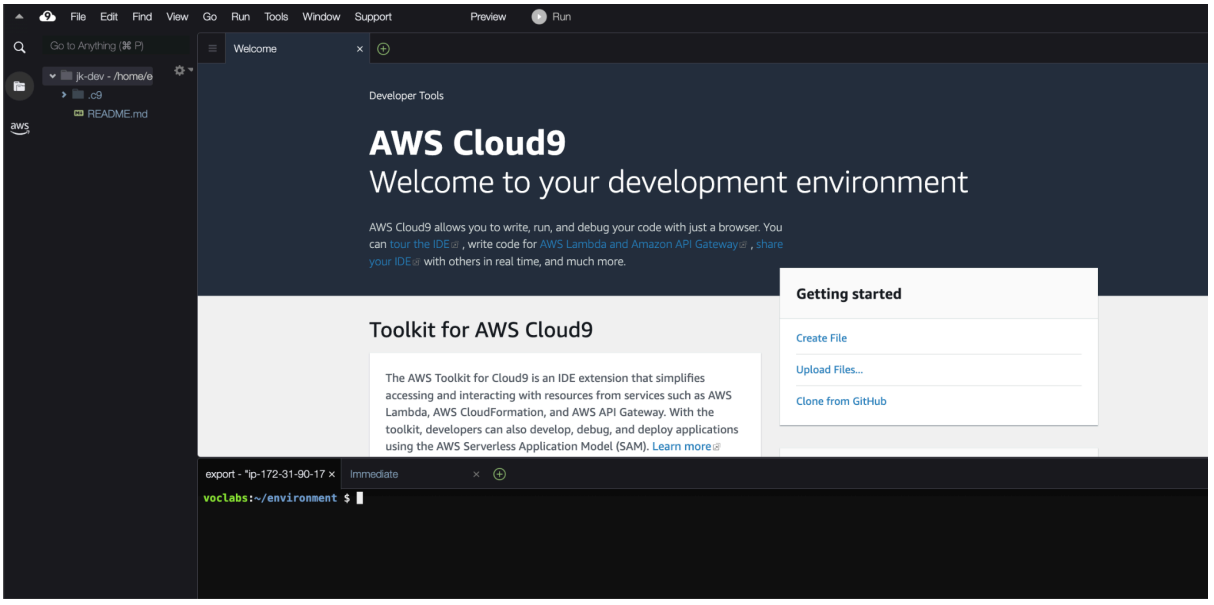
W ustawieniach sieciowych wybierz **Secure Shell** i naciśnij Create




3. Nowa instancja zostanie utworzona - uruchom ją,



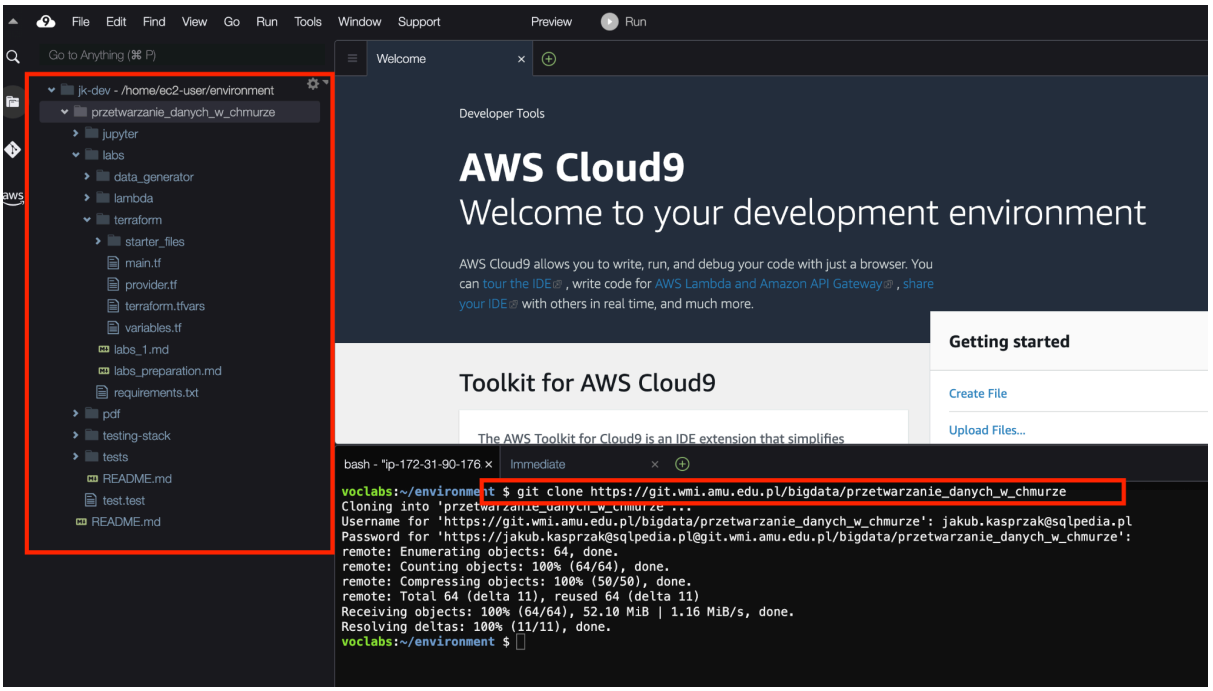

4. Pojawi sie nowe okno z IDE - twoja instancja C9 jest gotowa do pracy.



5. W dolnej części ekranu, masz dostęp do terminala (możesz otworzyć wiele okien terminala w osobnych zakładkach).

   Sklonuj repozytorium używając swoich danych autoryzujących (user / password) https://git.wmi.amu.edu.pl/bigdata/przetwarzanie_danych_w_chmurze

```
$ git clone
https://git.wmi.amu.edu.pl/bigdata/przetwarzanie_danych_w_chmurze.git
```




6. Uruchom skrypt **./labs/setup.sh,** który zainstaluje terraform i inne wymagane pakiety (sprawdź zawartość skryptu i pliku requirements.txt)

```
$ cd przetwarzanie_danych_w_chmurze/labs/
$ source ./setup_c9.sh
```

   Skrypt zainstaluje wszystkie wymagane biblioteki i przygotuje środowisko do pracy.

7. [OPCJONALNY SETUP] - Jeśli chcesz usprawnić swoją prace, możesz użyć tej instancji jako backend dla Twojego IDE (VS Code). Instrukcja jak to zrobić tutaj: Field Notes: Use AWS Cloud9 to Power Your Visual Studio Code IDE | AWS Architecture Blog (amazon.com)

8. Po porawnym uruchomieniu kroku 6 (skrypt) otwórz nowy terminal i sprawdź czy terraform został zainstalowany poprawnie:

```
$ terraform --version
```


Twoja instancja jest gotowa do pracy!

# Terraform 101

Wszystkie obiekty, które będziemy tworzyć w AWS można wykonać za pomocą konsoli lub kodu. Najpopularniejszymi rozwiązaniami **IaaC** (Infrastructure as a Code) jest natywny dla AWS Cloudformation oraz **Terraform**. W przypadku naszych zajęć będziemy korzystali z Terraform.

Jeśli po raz pierwszy masz do czynienia z Terraform - rekomenduję przejście tutorialu aby zrozumieć najważniejsz obiekty i założenia (całość nie powinna zabrać więcej niż ok 30 minut) :

https://learn.hashicorp.com/tutorials/terraform/infrastructure-as-code?in=terraform/aws-get-started

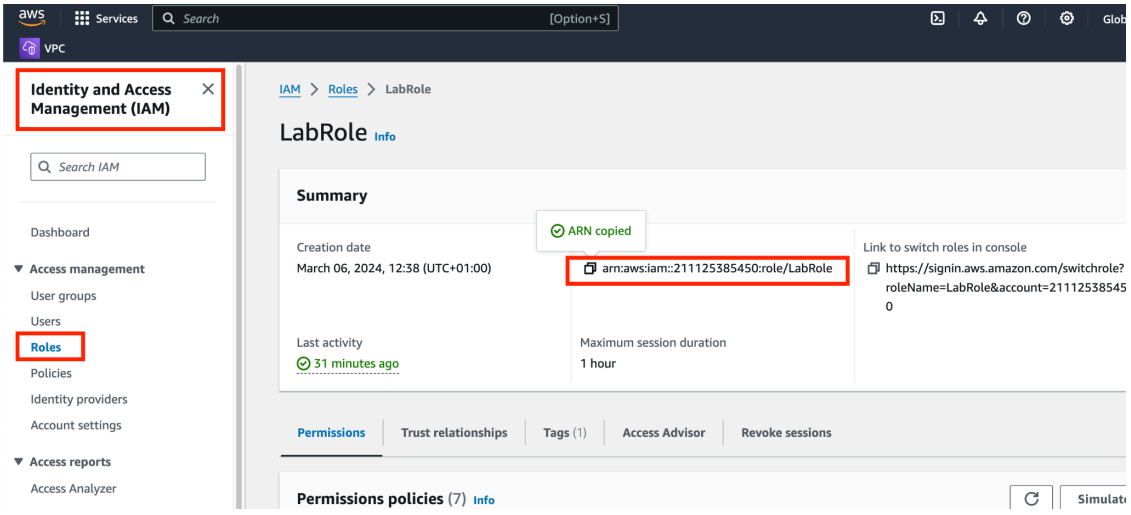
Podstawowe polecenia Terraform które będziemy używali :

1. Terraform **init** - inicjalizacja stanu terraform (tylko raz na początku projektu).
2. Terraform **plan** - po inicjalizacji i utworzeniu pierwszych obiektów, możemy zobaczyć co zostanie stworzone / zmodyfikowane - to jest delta pomiędzy obecnym stanem a docelowym.
3. Terraform **apply** - wdrożenie zmian (nowe obiekty, aktualizacja istniejących lub kasowanie niepotrzebnych) - dokładnie to co pokazał plan.
4. Terraform **destroy** - usunięcie wszystkich obiektów

## Konfiguracja podstawowych obiektów Terraform

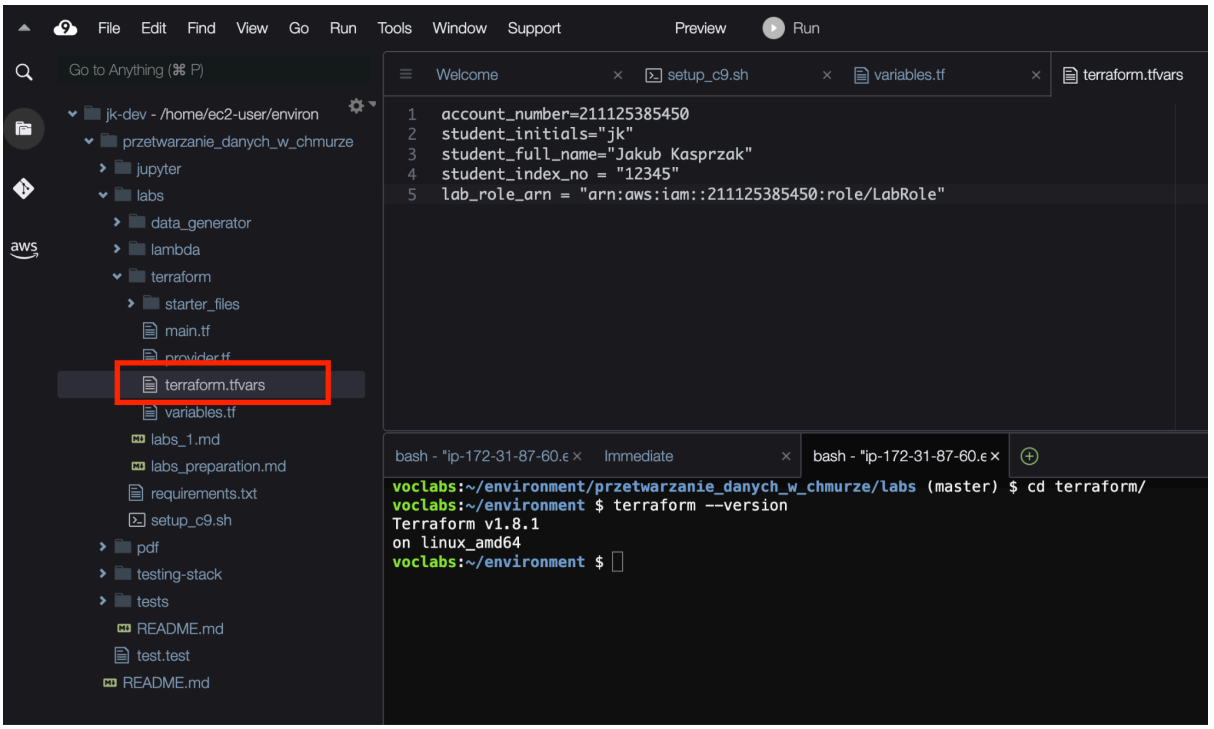
1. Cały Twój kod TF będzie znajdował się w  lokalizacji (sklonowane repo)_ **./przetwarzanie_danych_w_chmurze/labs/terraform/** i wszędzie w zadaniach gdy będziemy odwoływali się do kodu terraform - będzie to właśnie w tym folderze.

2. Otwórz serwis IAM w kosnoli AWS i odszukaj role LabRole - skopiuj jej ARN - bedzie potrzebny do następnego kroku.

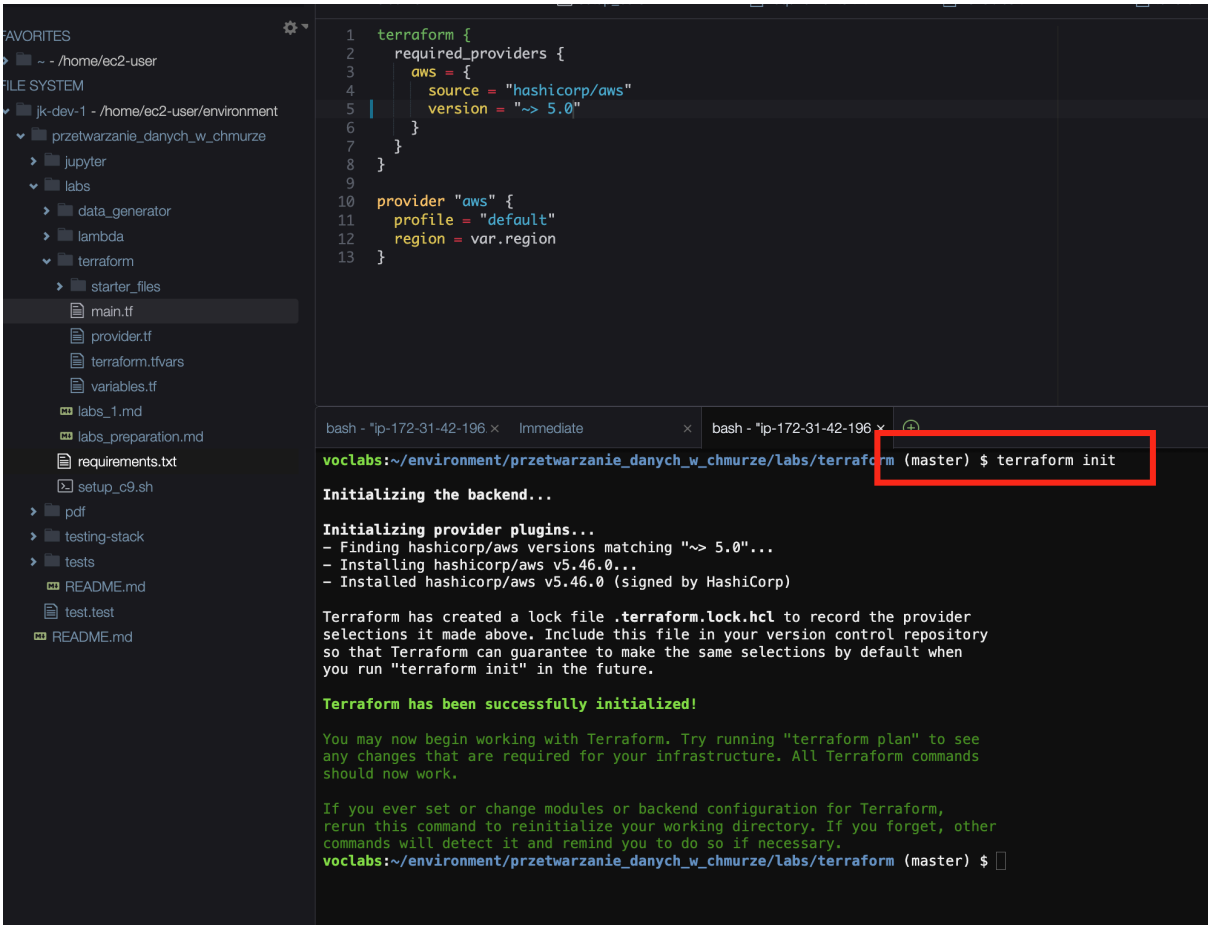


3. Zaktualizuj **i ZAPISZ** plik **./labs/terraform/terraform.tfvars** ze swoimi danymi (numer konta AWS, numer indeksu i inicjały oraz ARN roli LabRole) np:

```
account_number=211125385450
student_initials="jk"
student_full_name="Jakub Kasprzak"
student_index_no = "12345"
lab_role_arn = "arn:aws:iam::211125385450:role/LabRole"
```




4. Zainicjuj plik stanu Terraform poleceniem ***terraform init*** (od tego momentu wszystkie zmiany będą dokonywane na Twoim koncie AWS). Nigdy nie modyfikuj pliku stanu ręcznie!
   Uruchom w katalogu **./labs/terraform/**




Otwórz repozytorium - powinieneś mieć strukturę jak poniżej

```
<your repo>
├────jupyter
├────labs
│    ├────data_generator
│    ├────lambda
│    └────terraform
│         │    main.tf          <
│         │    provider.tf      <
│         │    terraform.tfvars <
│         │    variables.tf     <
│         └────starter_files
│
├────ppt
├────testing-stack
└────tests
```

5. Masz już skonfigurowane wszystkie narzędzia i zainicjowany terraform.

# Przetwarzanie Danych w chmurze publicznej **Laboratorium**

---

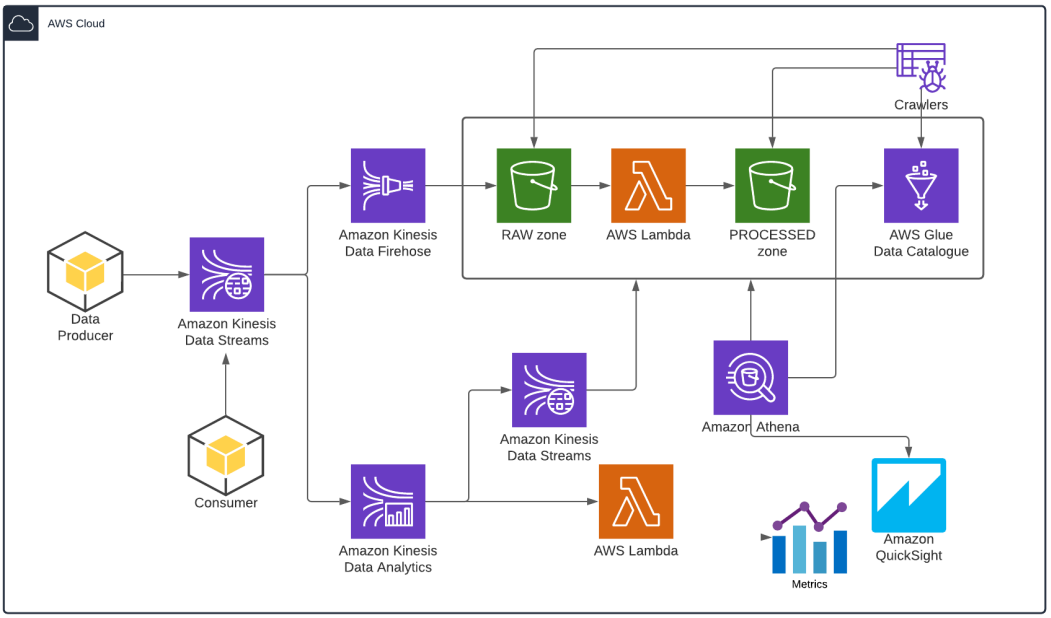
## Scenariusz Laboratorium

W scenariuszu będziemy symulować ładowanie danych *near real-time* z giełdy kryptowalut Bitstamp (zlecenia sell/buy) do *Data Lake* z wykorzystaniem usług Kinesis. Dane będziemy przetwarzać pomiędzy **Raw** a **Processed** zone z wykorzystaniem funkcji Lambda.

Zarejestrujemy tabele w Glue Data Catalogue aby móc wykonywać zapytania ad-hoc w Athena (SQL).

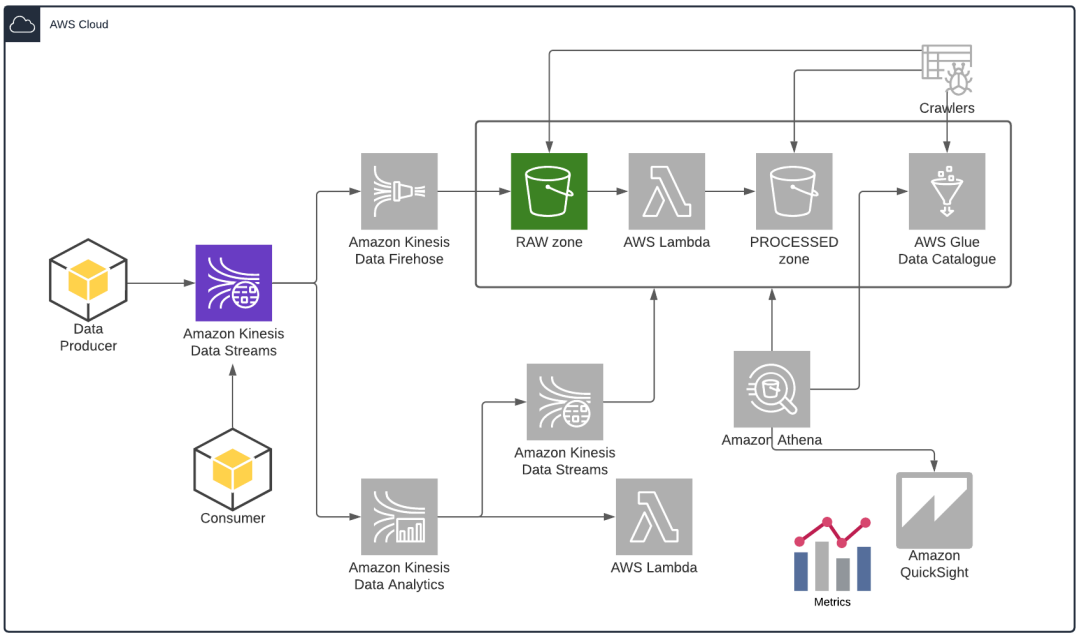
Do zbudowania systemu użyjemy następujących usług serverless :

- broker wiadomości Kinesis Data Stream
- zasilanie data lake Kinesis Delivery Stream
- Glue Data Catalogue jako Hive Metastore do przechowywanie metadanych (tabele, partycje)
- silnik przetwarzania danych Presto udostępniający interfejs zgodny z ANSI SQL - usługa Athena (analityka ad-hoc)
- Ciągłe przetwarzanie danych, funkcja lambda odpalana w momencie wystąpienia zdarzenia zapisu pliku do S3
- Przykładowe dane do symulacji pochodzą z https://www.cryptodatadownload.com/data/bitstamp/



# Ćwiczenie 1

W ćwiczeniu tym, stworzymy pierwsze, podstawowe elementy naszego Data Lake i infrastruktury do ładowania danych - **S3 buckety (RAW i PROCESSED) oraz Kinesis Data Stream**. Poznasz podstawowe zasady tworzenia obiektów w Terraform.




1. Utwórz nowy plik TF **./labs/terraform/s3.tf** zawierający definicję S3 bucketów RAW i PROCESSED  w którym będziemy przechowywali dane.
    a. Opis struktury S3 w TF : https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/s3_bucket
    b. Stwórz buckety opisany następującymi parametrami :
        i. Nazwa bucketu powinna składać się z : *datalake-<zone>-${var.account_number}-${var.student_initials}-${var.student_index_no}*

            *Np:* ***datalake-raw-100603781557-jk-12345***
        ii. Wykorzystaj parametr ***force_destroy*** (sprawdź do czego służy)
        iii. Jako dobrą praktykę wykorzystaj tagowanie obiektów - dodaj tag **Environment** = DEV

c. Przykładowa definicja jednego z bucketów **./labs/terraform/s3.tf**

```
resource "aws_s3_bucket" "raw_bucket" {
 bucket =
"datalake-raw-${var.account_number}-${var.student_initials}-${var.stude
nt_index_no}"
 force_destroy = true

 tags = {
   Purpose = "UAM Cloud Data Processing"
   Environment = "DEV"
 }
}
```

d. Uruchom ***terraform plan***, aby sprawdzić jakie obiekty zostaną stworzone zgodnie z tą definicją (w folderze **./labs/terraform/**)

```
$ terraform plan

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_s3_bucket.raw_bucket will be created
  + resource "aws_s3_bucket" "dl_raw_bucket" {
      + acceleration_status         = (known after apply)
      + acl                         = "private"
      + arn                         = (known after apply)
      + bucket                      =
"datalake-raw-100603781557-jk-12345"
      + bucket_domain_name          = (known after apply)
      + bucket_regional_domain_name = (known after apply)
      + force_destroy               = true
      + hosted_zone_id              = (known after apply)
      + id                          = (known after apply)
      + region                      = (known after apply)
      + request_payer               = (known after apply)
```

```
      + tags                         = {
          + "Environment" = "DEV"
          + "Purpose"     = "UAM Cloud Data Processing"
        }
      + website_domain               = (known after apply)
      + website_endpoint             = (known after apply)

      + versioning {
          + enabled    = (known after apply)
          + mfa_delete = (known after apply)
        }
    }

Plan: 1 to add, 0 to change, 0 to destroy.

------------------------------------------------------------------------
-

Note: You didn't specify an "-out" parameter to save this plan, so
Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.
```

e. Wykonaj kod poleceniem **terraform apply** i postępuj zgodnie z instrukcją**.** Sprawdź w konsoli AWS czy bucket został utworzony.

```
$ terraform apply

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:
```

```
  # aws_s3_bucket.raw_bucket will be created
  + resource "aws_s3_bucket" "raw_bucket" {
      + acceleration_status         = (known after apply)
      + acl                         = "private"
      + arn                         = (known after apply)
      + bucket                      =
"datalake-raw-100603781557-jk-12345"
      ...

  Enter a value: yes

aws_s3_bucket.raw_bucket: Creating...
aws_s3_bucket.raw_bucket: Creation complete after 9s
[id=datalake-raw-100603781557-jk-12345]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Otwórz serwis S3 w konsoli AWS i sprawdź czy bucket został stworzony.

2. Analogicznie  stwórz bucket dla zony **Processed** (w tym samym pliku s3.tf)**.** Przykład nazwy:

   ***datalake-processed-100603781557-jk-12345***

3. Utwórz nowy plik **./labs/terraform/kinesis_ds.tf** zawierający definicię **Kinesis Data Stream**
   a. Opis obiektu TF znajdziesz w : https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/kinesis_stream
   b. Stwórz nowy stream z wykorzystaniem następujących parametrów:
      i. Liczba shardów = 1
      ii. Nazwa powinna pasować do tego wzorca : *cryptostock-${var.account_number}-${var.student_initials}-${var.student_index_no}*

*Np:* ***cryptostock-100603781557-jk-12345***

iii. **Retention period** = default (24h)
iv. Tag **Environment** = DEV
v. **enforce_consumer_deletion** = True (sprawdź w dokumentacji jakie ma znaczenie)
vi. Włącz metryki na poziomie shardu dla : ["IncomingBytes", "OutgoingBytes", "IncomingRecords", "OutgoingRecords"]

c. Przykładowa definicja

```
resource "aws_kinesis_stream" "cryptostock_stream" {
 name             =
"cryptostock-${var.account_number}-${var.student_initials}-${var.studen
t_index_no}"
 shard_count      = 1

 enforce_consumer_deletion = true

 shard_level_metrics = [
   "IncomingBytes",
   "OutgoingBytes",
   "IncomingRecords",
   "OutgoingRecords"
 ]

 tags = {
   Purpose = "UAM Cloud Data Processing"
   Environment = "DEV"
   Owner = var.student_full_name
 }
}
```

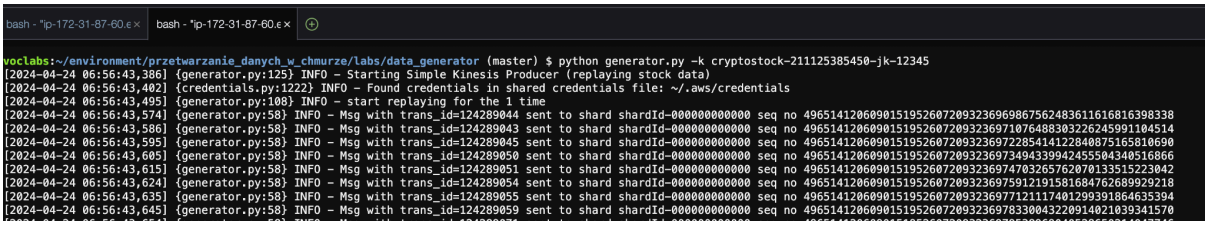
d. Utwórz Kinesis DS (uwaga ! koszty stały $0.015 / h) - **terraform apply**. Więcej na temat cen tej usługi https://aws.amazon.com/kinesis/data-streams/pricing/

e. Sprawdź w konsoli AWS czy data stream został utworzony (wejdź do usługi Kinesis -> Data Streams)

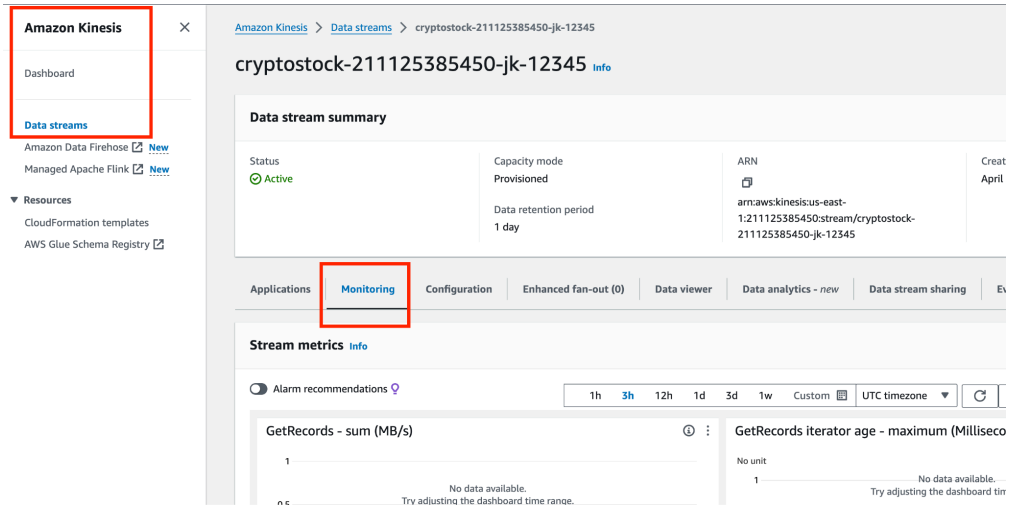
4. Wysyłanie danych do Kinesis DS
    a. Przejrzyj kod prostego generatora danych **./labs/data_generator/generator.py**. Zwróć uwagę na sposób w jaki dane są wysyłane do Kinesis - wykorzystujemy tu SDK (API), które charakteryzuje się niską wydajnością. Iterujemy po danych z pliku i wywołujemy metodę put_record.

    **(*)** W ramach dodatkowego ćwiczenia (dla ambitnych :)) możesz udoskonalić generator, przepisując go na put_records lub generować rekordy wielowątkowo / ew asynchronicznie aby zwiększyć wydajność.
    b. Uruchom generator w nowym oknie linii poleceń (będzie nam potrzebny aby cały czas wysyłał dane).
    Użyj parametru -k aby przekazać nazwę kinesis data stream'a np. :

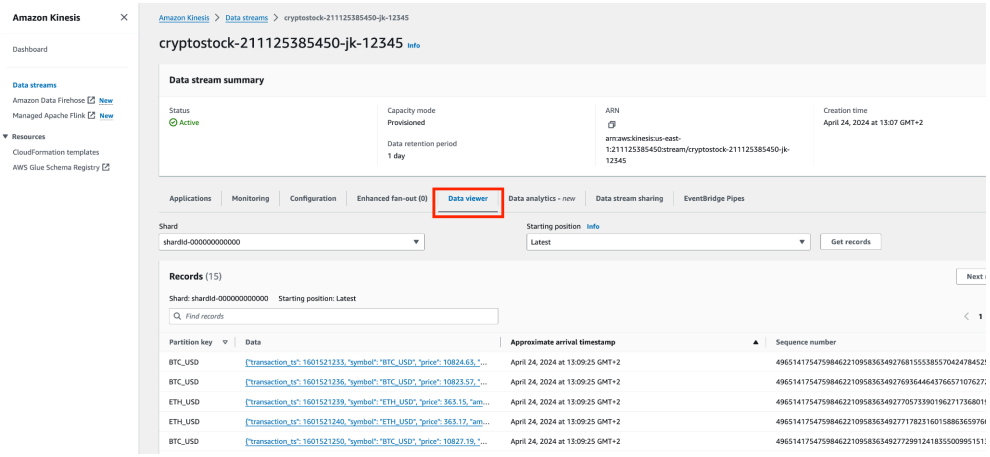
    **$ python generator.py -k cryptostock-211125385450-jk-12345**




    c. Przejdź do Konsoli AWS i zobacz jak wyglądają metryki KDS, czy dane faktycznie docierają do sharda (może minąć pare minut zanim metryki będą zaktualizowane) - zakładka monitoring



d. Otwórz zakładkę Data Viewer i sprawdź czy eventy są wysyłane do streama:




e. **(*) Dla chętnych** - napisz skrypt (np. w jupyter notebook) który będzie czytał wiadomości z Kinesis Data Stream.

5. Zatrzymaj generator danych i usuń cały stack poleceniem ***terraform destroy*** (uruchom w katalogu w którym masz utworzone obiekty terraform)

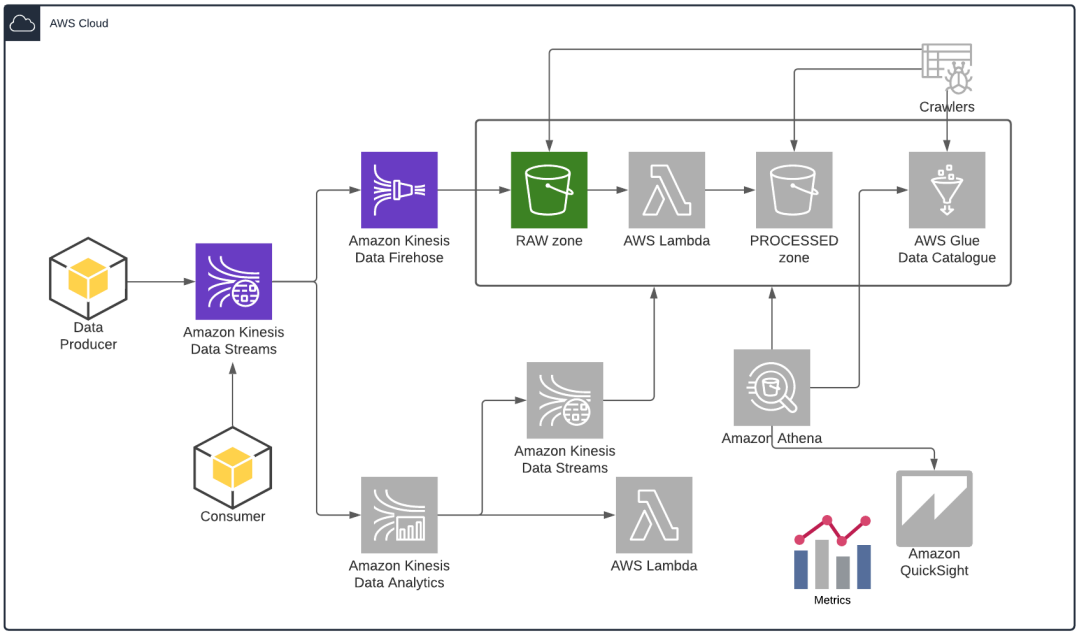
```
$ terraform destroy

...

Plan: 0 to add, 0 to change, 3 to destroy.

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown
above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes
```

6. Sprawdź w konsoli AWS czy faktycznie wszystko zostało posprzątane. Jeśli S3 bucket nie został usunięty - prawdopodobnie zapomniałeś o parametrze ***force_destroy***. Usuń więc najpierw pliki (z konsoli AWS) a potem bucket.

## Ćwiczenie 2

Do naszego systemu dołożymy kolejny element, odpowiedzialny za ładowanie danych do Data Lake.

W tym celu wykorzystamy usługę Kinesis Firehose



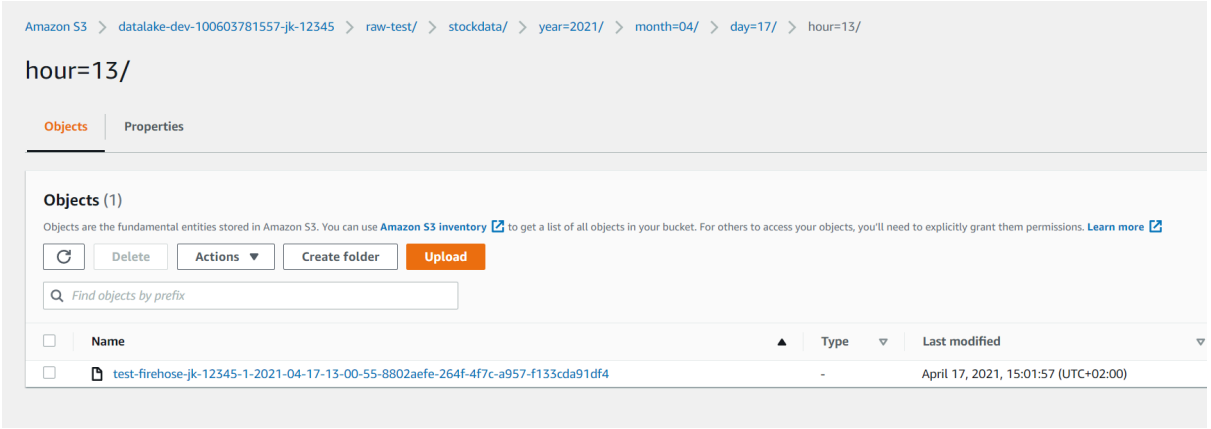

1. Utwórz obiekty zdefiniowane w ćwiczeniu 1 (*terraform **apply***) S3 buckety + Kinesis Data Stream - zostały one usunięte w ostatnim punkcie.
2. Utwórz nowy Kinesis **Delivery Stream z poziomu konsoli AWS**, przejrzyj możliwe opcje.

   Wybierz następujące parametry :
   a. Nazwa = *test-firehose-{inicjały}-{nr_indeksu}*
   b. Źródło danych = Kinesis Data Stream (wybierz utworzony w p. 1 stream)
   c. Brak transformacji
   d. Miejsce docelowe S3 - Twój bucket utworzony w p.1
      Wzorce prefiksów:
      i. Dane :
      *raw-test/stockdata/year=!{timestamp:yyyy}/month=!{timestamp:MM}/day=!{timestamp:dd}/hour=!{timestamp:HH}/*
      ii. Błędy:
      *raw-test/stockdata_errors/!{firehose:error-output-type}/year=!{timestamp:yyyy}/month=!{timestamp:MM}/day=!{timestamp:dd}/hour=!{timestamp:HH}/*

e. Rozmiar bufora = 1MB
f. Częstotliwość zrzucania danych do S3 = 60s
g. Rola IAM - utwórz nową (domyślna opcja)

3. Uruchom generator danych testowych (patrz ćwiczenie 1 p. 3). Upewnij się że dane są poprawnie generowane do KDS.

4. Poczekaj pare minut (5-6) i sprawdź czy dane są zapisywane w S3 z odpowiednim prefixem. Początkowy data dump może się odbyć z pewnym opóźnieniem (inicjalizacja serwisu), kolejne porcje danych powinny napływać zgodnie z przyjętą strategią.




5. Utwórz obiekt Kinesis Firehose Delivery Stream.

   aws_kinesis_firehose_delivery_stream | Resources | hashicorp/aws | Terraform | Terraform Registry

   Utwórz nowy plik **./labs/terraform/kinesis_fh.tf** zawierający definicję **Kinesis Delivery Stream (Firehose)**

   a. Nazwa (wzorzec) : *"firehose-${var.account_number}-${var.student_initials}-${var.student_index_no}"*
   b. Źródło danych : Kinesis Data Stream (wybierz utworzony w p. 1 stream) - referencja terraform.
   c. Brak transformacji
   d. Miejsce docelowe S3 - Twój bucket utworzony w p.1 (referencja terraform)
      Wzorce prefiksów:
      i. Dane : *raw-zone/stockdata/year=!{timestamp:yyyy}/month=!{timestamp:MM}/day=!{timestamp:dd}/hour=!{timestamp:HH}/*

ii. Błędy:
*raw-zone/stockdata_errors/!{firehose:error-output-type}/year=!{timestamp:yyyy}/month=!{timestamp:MM}/day=!{timestamp:dd}/hour=!{timestamp:HH}/*

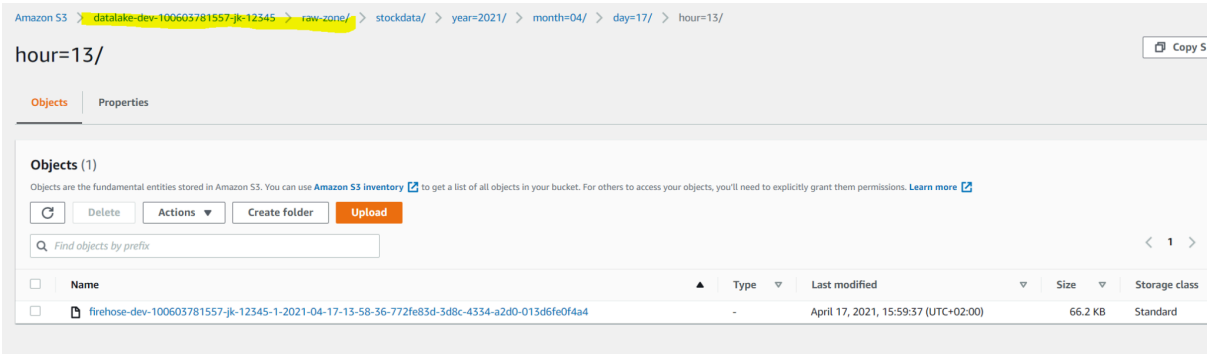
e. Rozmiar bufora = 1MB
f. Częstotliwość zrzucania danych do S3 = 60s
g. Rola IAM - LabRole (użyj zmiennej)

Przykładowa definicja

```
resource "aws_kinesis_firehose_delivery_stream" "stock_delivery_stream"
{
 name =
"firehose-${var.account_number}-${var.student_initials}-${var.student_in
dex_no}"
 destination = "extended_s3"

 kinesis_source_configuration {
   kinesis_stream_arn = aws_kinesis_stream.cryptostock_stream.arn
   role_arn = var.lab_role_arn
 }

 extended_s3_configuration {
   role_arn = var.lab_role_arn
   bucket_arn = aws_s3_bucket.raw_bucket.arn
   buffering_size = 1
   buffering_interval = 60
   prefix =
"raw-zone/stockdata/year=!{timestamp:yyyy}/month=!{timestamp:MM}/day=!{t
imestamp:dd}/hour=!{timestamp:HH}/"
   error_output_prefix =
"${"raw-zone/stockdata_errors/!{firehose:error-output-type}/year=!{times
tamp:yyyy}"}${"/month=!{timestamp:MM}/day=!{timestamp:dd}/hour=!{timesta
mp:HH}"}/"
 }

}
```

6. Utwórz nowe obiekty - **terraform plan / apply**
7. Sprawdź czy nowy firehose stream jest aktywny (konsola AWS)

8. Upewnij się że generator generuje dane i po około 5-7 minutach  zweryfikuj czy dane pojawiają się w bucket'cie (prefix - **raw-zone**)

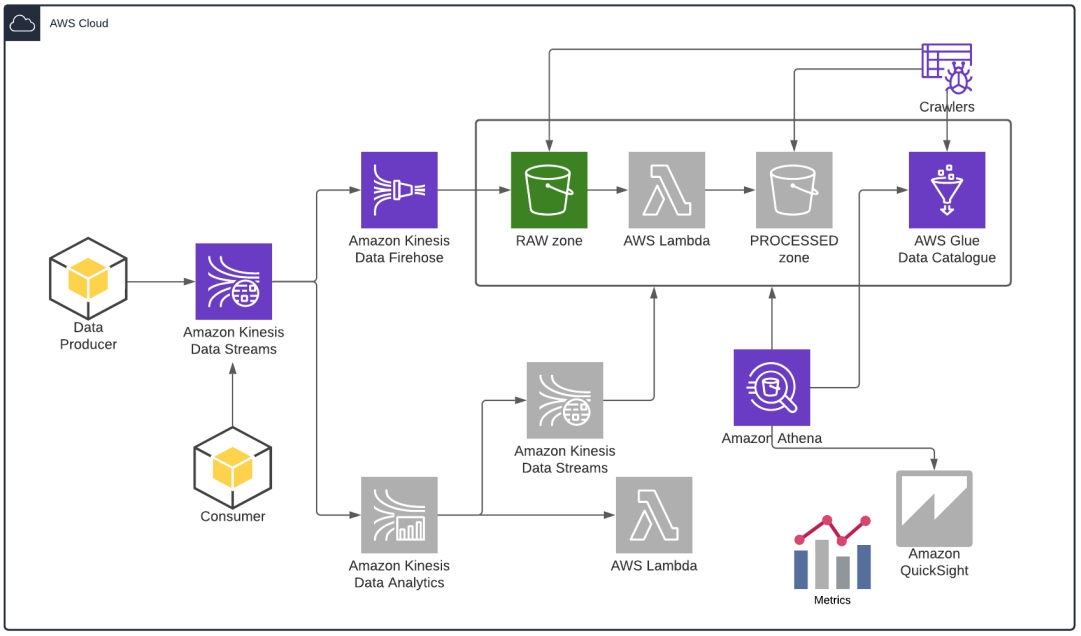



9. Pozostaw generator włączony i przejdź do ćwiczenia 3

## Ćwiczenie 3

Rejestrowanie tabel w Glue Data Catalogue i aktualizowanie partycji.
W tym ćwiczeniu wykorzystamy:

- Glue Crawlers do automatycznego rejestrowania tabeli w katalogu Glue.
- Athena do wykonania zapytanie SQL zwracające największe oferty kupna / sprzedaży w przedziałach godzinowych




1. Utwórz nową bazę danych Glue w terraform. Stwórz plik **./labs/terraform/glue_dc.tf**
    a. Opis obiektu bazy glue TF : https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/glue_catalog_database
    b. Nazwa bazy danych (wzorzec): *"datalake_raw_${var.account_number}_${var.student_initials}_${var.student_index_no}"*

    Zwróć uwagę, że nazwa nie zawiera '-' tylko '_'. Powinna być zgodna z https://docs.aws.amazon.com/athena/latest/ug/glue-best-practices.html

    Przykładowa definicja

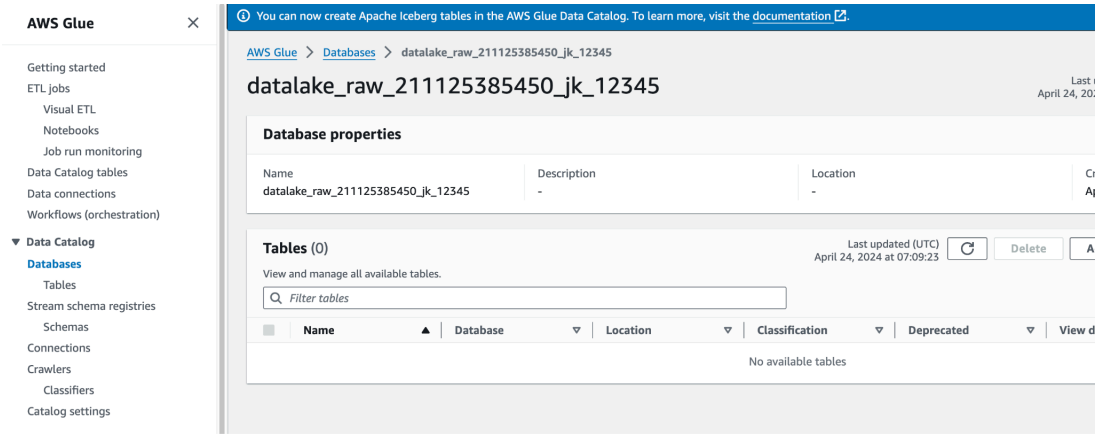
```
resource "aws_glue_catalog_database" "datalake_db_raw_zone" {
 name =
```

```
"datalake_raw_${var.account_number}_${var.student_initials}_${var.stude
nt_index_no}"
}
```

c. Uruchom terraform apply aby stworzyć bazę

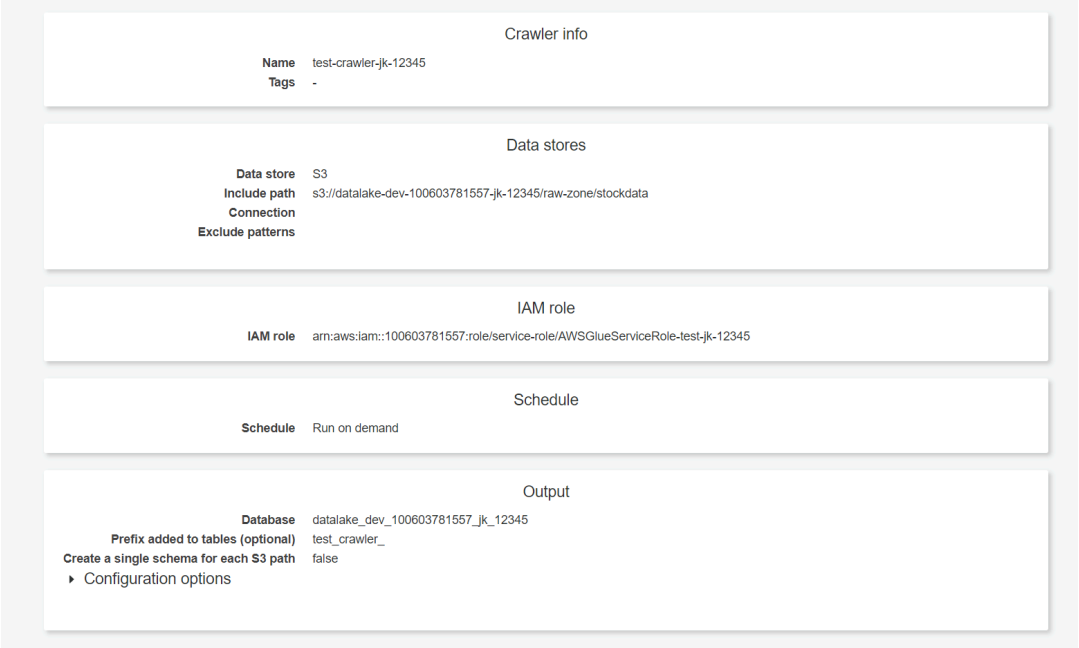
```
$ terraform apply
...

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_glue_catalog_database.datalake_db_raw_zone will be created
  + resource "aws_glue_catalog_database" "datalake_db_raw_zone" {
      + arn        = (known after apply)
      + catalog_id = (known after apply)
      + id         = (known after apply)
      + name       = "datalake_raw_100603781557_jk_12345"
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_glue_catalog_database.datalake_db_raw_zone: Creating...
aws_glue_catalog_database.datalake_db_raw_zone: Creation complete
after 1s [id=100603781557:datalake_raw_100603781557_jk_12345]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

d. Zweryfikuj poprawność jej utworzenia w konsoli AWS

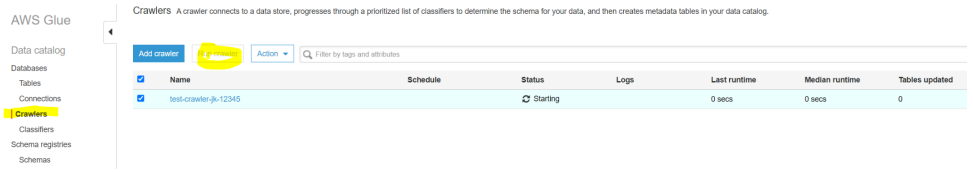



2. Uruchom ręcznie **Glue Crawler** (z konsoli AWS), który przeskanuje dane w prefixie do którego trafiają nasze dane i utworzy nową tabelę w twojej bazie danych w Glue, Wybierz następujące opcje :
   a. Nazwa crawlera - *test-crawler-{inicjaly}-{nr_indeksu}*
   b. Data Store - ścieżka S3 do danych zrzucanych przez firehose np. *s3://datalake-raw-100603781557-jk-12345/raw-zone/stockdata*
   c. Uprawnienia - wybierz istniejącą role LabRole
   d. Częstotliwość uruchamiania - on demand
   e. Wyniki powinny być zapisywane w Twojej bazie danych glue (utworzonej w p.1)
   f. Wybierz prefiks dla nowo tworzonych tabel - ***test_crawler_***

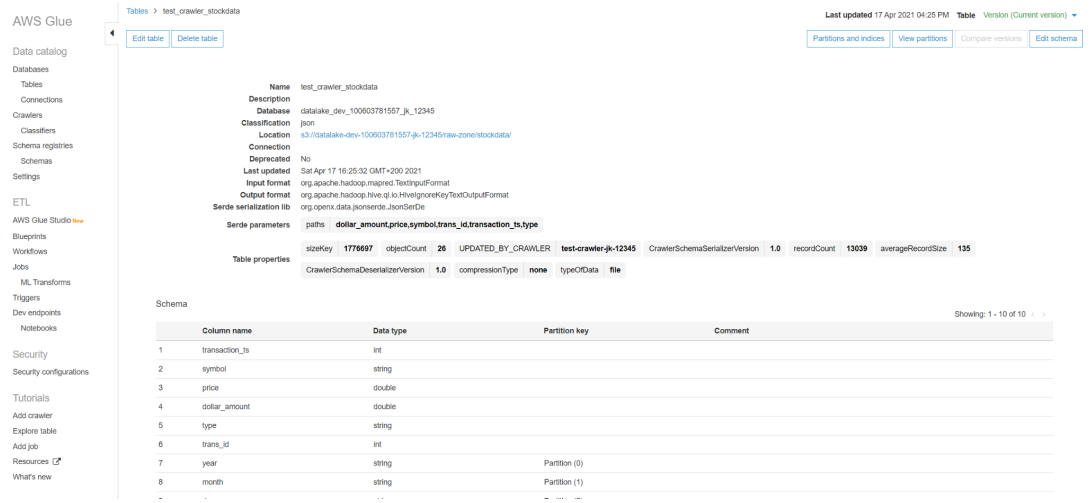
   Przykład konfiguracji:



g. Uruchom Crawler’a




h. Po paru minutach powinieneś zobaczyć rezultaty - nowa tabela w Twojej bazie danych




i. Sprawdź jej schemat i partycje.

j. Stwórz nowego Crawlera za pomocą kodu TF.
Dodaj właściwą definicję Glue Crawlera do pliku **./labs/glue.tf** :

```
resource "aws_glue_catalog_database" "datalake_db_raw_zone" {
 name =
"datalake_raw_${var.account_number}_${var.student_initials}_${var.stude
nt_index_no}"
}



resource "aws_glue_catalog_database" "datalake_db_processed_zone" {
 name =
"datalake_processed_${var.account_number}_${var.student_initials}_${var
.student_index_no}"
}




resource "aws_glue_crawler" "glue_crawler_raw_zone" {
 database_name = aws_glue_catalog_database.datalake_db_raw_zone.name
 name =
"gc-raw-${var.account_number}-${var.student_initials}-${var.student_ind
```

```
ex_no}"
 role = var.lab_role_arn
 table_prefix = "crawler_"

 s3_target {
   path = "s3://${aws_s3_bucket.raw_bucket.bucket}/raw-zone/stockdata/"
 }

 tags = merge(local.common_tags, )
}
```

Uruchom ***terraform plan / apply*** aby wdrożyć zmiany. Uruchom nowo utworzonego Crawlera i sprawdź czy nowa tabela z prefiksem crawler_ została utworzona.

3. Stwórz dedykowaną infrastrukture dla serwisu **Athena**

   Obiekty do stworzenia:
   - nowy bucket dla wyników zapytań wraz z lifecycle_policy (tymczasowe wyniki - powinny być usunietę po jednym dniu automatycznie)
   - dedykowan Athena Workgroup https://registry.terraform.io/providers/hashicorp/aws/3.64.2/docs/resources/athena_workgroup z domyślną lokalizacja athena-results bucket, prefix output.

   a. Utwórz nowy plik **athena.tf** dla obiektów dedukowanych dla Athena.
   b. Dodaj nowy S3 bucket (podobnie jak w LAB1) do tego pliku.

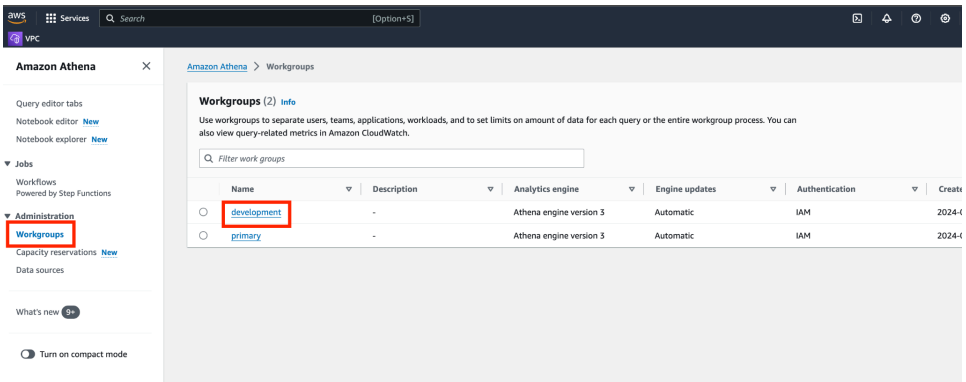
      Bucket powinin mieć również zdefiniowany lifecycle_policy i jego nazwa zaczynać się do athena-results (reszta podobnie jak w poprzednich bucketach)

      Przykład:

```
resource "aws_s3_bucket" "athena_results" {
 bucket =
"athena-results-${var.account_number}-${var.student_initials}-${var.stu
dent_index_no}"
 force_destroy = true

 tags = merge(local.common_tags, )
```

```
}

resource "aws_s3_bucket_lifecycle_configuration"
"athena_results_lifecycle" {
 bucket = aws_s3_bucket.athena_results.id

 rule {
   id = "standard-expiration"
   status = "Enabled"
   expiration {
       days=1
   }
 }
}

resource "aws_athena_workgroup" "athena_workgroup" {
 name = "development"

 configuration {
   enforce_workgroup_configuration    = true

   result_configuration {
     output_location =
"s3://${aws_s3_bucket.athena_results.bucket}/output/"
   }
 }

 force_destroy = true
}
```

Zweryfikuj stworzone obiekty (S3 bucket i Athena workgroup)

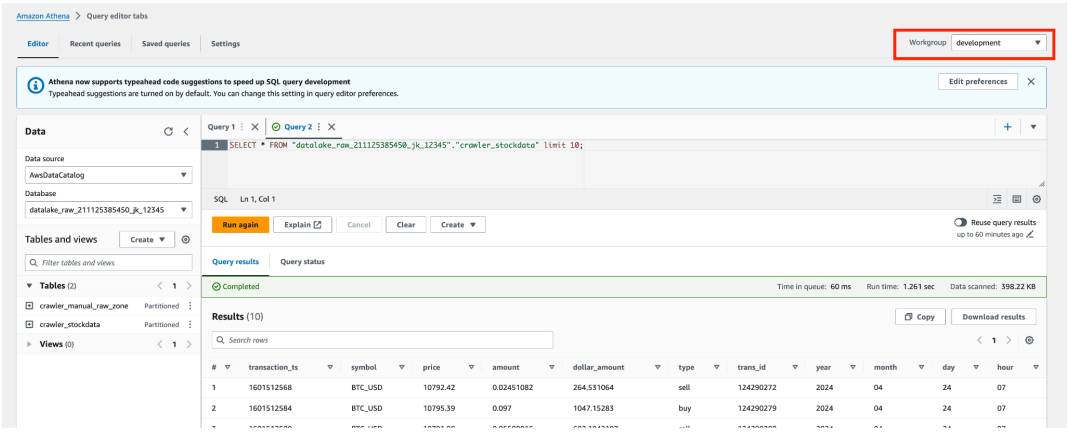


4. Otwórz konsole AWS i serwis Athena. Jeśli pierwszy raz otwierasz tą usługę, konieczne będzie skonfigurowanie **Workgroup Primary lub wybrania nowo utworzonej workgroup - Development** aby wskazać miejsce składowania wyników.

   Dla **primary** - wybierz odpowiedni bucket, dodaj prefix

   Dla **development** workgroup - wszystko jest już skonfigurowane w TF!

   a. Sprawdź czy możesz odpytać tabelę stworzoną przez Crawlera




   b. Napisz zapytanie SQL, które z nowo utworzonej tabeli pobierze najwyższe wartości zleceń (BUY / SELL) w przedziałach godzinowych **per symbol** . Posortuj wyniki od najstarszych przedziałów do najnowszych.

      Czas pobierz z atrybutu **transaction_ts.** Jest to informacja o faktycznym czasie złożenia zlecenia buy/sell. Format czasu unix epoch, który możesz przekonwertować na czytelną dla ludzi postać za pomocą funkcji **from_unixtime**() oraz **date_format**().

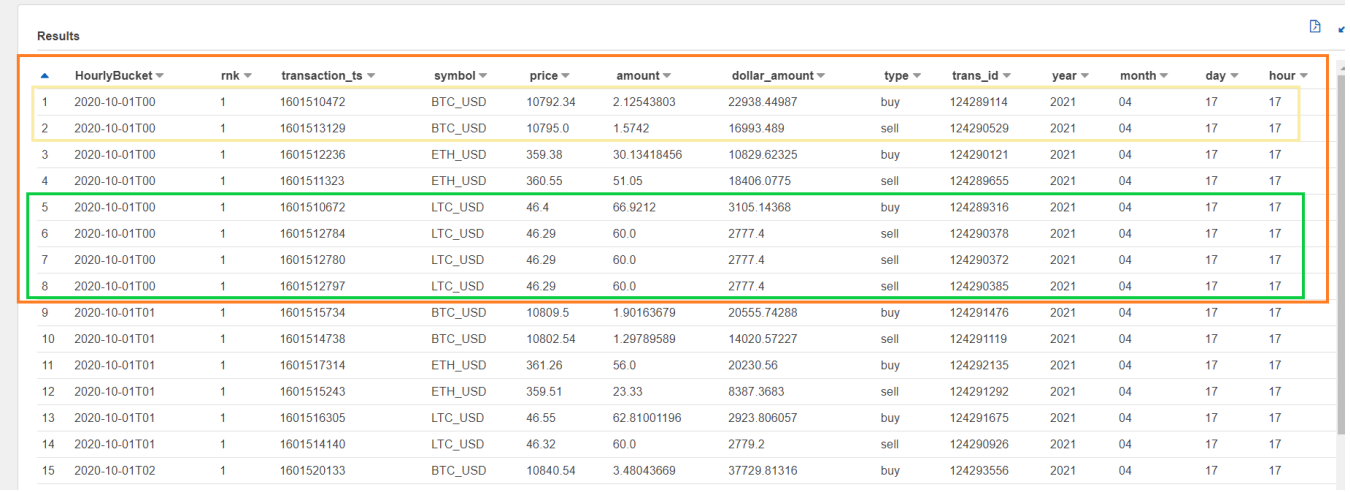
      Przykład wykorzystania funkcji from_unixtime() i date_format() :

```
SELECT date_format(from_unixtime(1618669914),'%Y-%m-%dT%H:%i:%sZ')

-----
2021-04-17T14:31:54Z
```

   c. Porównaj Twoje “buckety godzinowe” (nazwij tą kolumnę HourlyBucket, format `%Y-%m-%dT%H`)otrzymane z konwersji **transaction_ts** z bucketami godzinowymi wynikającymi z partycjonowania danych (uwaga ! generator mógł wygenerować te same event kilka razy - rolling strategy).

Widać tutaj podstawowy problem związany z czasem zasilania data lake a rzeczywistym czasem ich generowaniem.
Informacje o czasie zupełnie nie pasują do naszych partycji.

Partycjonowanie następuje zgodnie z czasem nadejścia danych do systemu (Kinesis Data Stream używa **ApproximateArrivalTimestamp)**
Fragment wyniku kwerendy - porównaj twój rezultat dla pierwszego okresu **2020-10-01 godzina 00** (jeśli twój wynik się różni - prawdopodobnie nie wszystkie dane zostały wygenerowane lub usługa firehose została utworzona zbyt późno - po prostu zrestartuj generator i upewnij się że przynajmniej 600 próbek zostało wygenerowanych i zrzuconych do S3 (pierwsza godzina danych)) :



Results

| | HourlyBucket | rnk | transaction_ts | symbol | price | amount | dollar_amount | type | trans_id | year | month | day | hour |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2020-10-01T00 | 1 | 1601510472 | BTC_USD | 10792.34 | 2.12543803 | 22938.44987 | buy | 124289114 | 2021 | 04 | 17 | 17 |
| 2 | 2020-10-01T00 | 1 | 1601513129 | BTC_USD | 10795.0 | 1.5742 | 16993.489 | sell | 124290529 | 2021 | 04 | 17 | 17 |
| 3 | 2020-10-01T00 | 1 | 1601512236 | ETH_USD | 359.38 | 30.13418456 | 10829.62325 | buy | 124290121 | 2021 | 04 | 17 | 17 |
| 4 | 2020-10-01T00 | 1 | 1601511323 | ETH_USD | 360.55 | 51.05 | 18406.0775 | sell | 124289655 | 2021 | 04 | 17 | 17 |
| 5 | 2020-10-01T00 | 1 | 1601510672 | LTC_USD | 46.4 | 66.9212 | 3105.14368 | buy | 124289316 | 2021 | 04 | 17 | 17 |
| 6 | 2020-10-01T00 | 1 | 1601512784 | LTC_USD | 46.29 | 60.0 | 2777.4 | sell | 124290378 | 2021 | 04 | 17 | 17 |
| 7 | 2020-10-01T00 | 1 | 1601512780 | LTC_USD | 46.29 | 60.0 | 2777.4 | sell | 124290372 | 2021 | 04 | 17 | 17 |
| 8 | 2020-10-01T00 | 1 | 1601512797 | LTC_USD | 46.29 | 60.0 | 2777.4 | sell | 124290385 | 2021 | 04 | 17 | 17 |
| 9 | 2020-10-01T01 | 1 | 1601515734 | BTC_USD | 10809.5 | 1.90163679 | 20555.74288 | buy | 124291476 | 2021 | 04 | 17 | 17 |
| 10 | 2020-10-01T01 | 1 | 1601514738 | BTC_USD | 10802.54 | 1.29789589 | 14020.57227 | sell | 124291119 | 2021 | 04 | 17 | 17 |
| 11 | 2020-10-01T01 | 1 | 1601517314 | ETH_USD | 361.26 | 56.0 | 20230.56 | buy | 124292135 | 2021 | 04 | 17 | 17 |
| 12 | 2020-10-01T01 | 1 | 1601515243 | ETH_USD | 359.51 | 23.33 | 8387.3683 | sell | 124291292 | 2021 | 04 | 17 | 17 |
| 13 | 2020-10-01T01 | 1 | 1601516305 | LTC_USD | 46.55 | 62.81001196 | 2923.806057 | buy | 124291675 | 2021 | 04 | 17 | 17 |
| 14 | 2020-10-01T01 | 1 | 1601514140 | LTC_USD | 46.32 | 60.0 | 2779.2 | sell | 124290926 | 2021 | 04 | 17 | 17 |
| 15 | 2020-10-01T02 | 1 | 1601520133 | BTC_USD | 10840.54 | 3.48043669 | 37729.81316 | buy | 124293556 | 2021 | 04 | 17 | 17 |

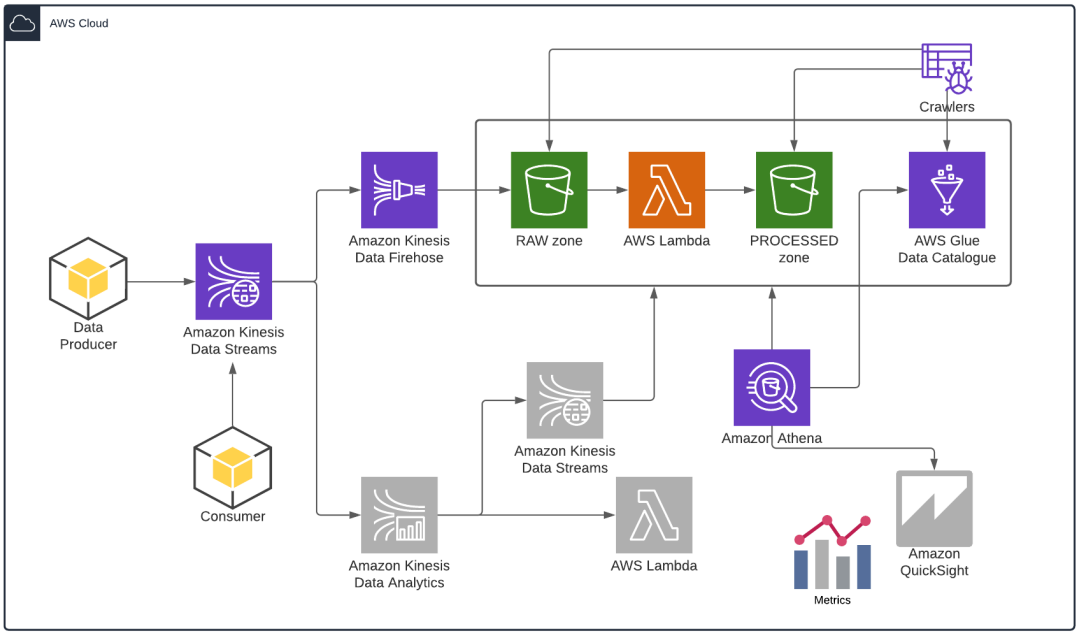
W następnym ćwiczeniu będziemy konwertowali te dane do formatu **parquet** i zmieniali partycjonowania ze względu na datę i czas wystąpienia zdarzenia (producer timestamp).

Zapisz swoją kwerendę SQL w pliku **./labs/sql_queries.sql**

5. Zatrzymaj generator danych i usuń cały stack poleceniem ***terraform destroy*** (uruchom w katalogu w którym masz utworzone obiekty terraform)

## Ćwiczenie 4

Automatyczne procesowanie danych z RAW zone do Processed zone za pomocą wyzwalaczy S3 i funkcji Lambda. Zmiana formatu danych na **Parquet** oraz zmiana partycjonowania - tym razem ze względu na czas wystąpienia zdarzenia (transakcji) - ***transaction_ts***.




Utworzymy trigger (S3 notification), który będzie wyzwalał funkcję Lambda. Ta przetworzy dane z pliku w momencie jego zapisu na S3. Zmieni format i sposób partycjonowania (***parquet***, kompresja snappy).

1. Utwórz funkcję Lambda która będzie czytała dane z pliku S3 (tuż po zapisie) i przenosiła je do nowej lokalizacji - processed-zone, zapisując w formacie **parquet**. Możesz to wykonać najpierw testowo z konsoli AWS, ale finalnie zrób poniższe kroki aby była ona zdefiniowana za pomocą Terraform,
    a. Do tego celu wykorzystamy kilka dołączonych plików. Przejrzyj zawartość **./labs/lambda/lambda_definition.py** - to kod naszej funkcji.
    Zwróć uwagę w jaki sposób będziemy zapisywali dane - zmian sposobu partycjonowania, zmieniona lista kolumn (np. artybut symbol będzie częścią partycjonowania a nie pliku z danymi)
    b. Funkcja lambda będzie używała spakowanej (**zip**) wersji tego skryptu (znajdziesz go w tym samym folderze)
    c. Do odczytu i zapisu danych będziemy wykorzystywali pakiet Python **awswrangler** : [://pypi.org/project/awswrangler/](://pypi.org/project/awswrangler/)
    Trzeba  go będzie dołączyć do funkcji lambda (również **zip**) jako dodatkowa “warstwa” rozszerzająca możliwości obrazu na którym nasza funkcja lambda

będzie wykonywana. Funkcja lambda to nic innego jak kawałek kodu wykonywanego na docker image.

d. Najpierw stworzymy w TF dodatkową warstwę (aws_lambda_layer_version) z pakietem awswrangler. Sprawdź w dokumentacji TF i AWS jakie możliwości i ograniczenia mają dodatkowe warstwy (rozmiar, ilość). Zwróć uwagę, że mamy możliwość również tworzenia własnych obrazów na których możemy uruchamiać swoją funkcję (nowość ogłoszona na ReInvent 2020)

   Utwórz plik ./**labs/terraform/lambda.tf** i zdefiniuj obiekt jak poniżej :

```
resource "aws_lambda_layer_version" "aws_wrangler" {
filename            = "../lambda/awswrangler-layer-2.7.0-py3.8.zip"
layer_name          = "aws_wrangler_${var.account_number}_${var.student_initials}_${var.student_index_no}"
source_code_hash    = "${filebase64sha256("../lambda/awswrangler-layer-2.7.0-py3.8.zip")}"
compatible_runtimes = ["python3.8"]
}
```

   Sprawdź w dokumentacji do czego służą poszczególne atrybuty i funkcja *filebase64sha256.*

e. Funkcja lambda będzie musiała być uruchamiana z odpowiednimi uprawnieniami - tutaj znów użyjemy roli **LabRole**.

f. Kolejnym krokiem będzie zdefiniowanie właściwej funkcji lambda. Dopisz do pliku ./**labs/terraform/lambda.tf** kolejny obiekt. Przykładowa definicja :

```
resource "aws_lambda_function" "etl_post_processing" {

 function_name = "etl-post-processing-${var.account_number}-${var.student_initials}-${var.student_index_no}"
 filename      = "../lambda/lambda_definition.zip"
 handler = "lambda_definition.etl_function"
 runtime = "python3.8"
 role = var.lab_role_arn
 timeout = 300
 memory_size = 512
```

```
 source_code_hash= filebase64sha256("../lambda/lambda_definition.zip")
 layers           = ["${aws_lambda_layer_version.aws_wrangler.arn}"]

}
```

g. Ostatnim elementem potrzebnym do wyzwolenie funkcji, są notyfikacje S3 oraz odpowiednie uprawnienia bucketu do wyzwalania funkcji lambda. Utwórz w pliku ./**labs/terraform/lambda.tf** te obiekty. Sprawdź szczegółowo w dokumentacji do czego służą.

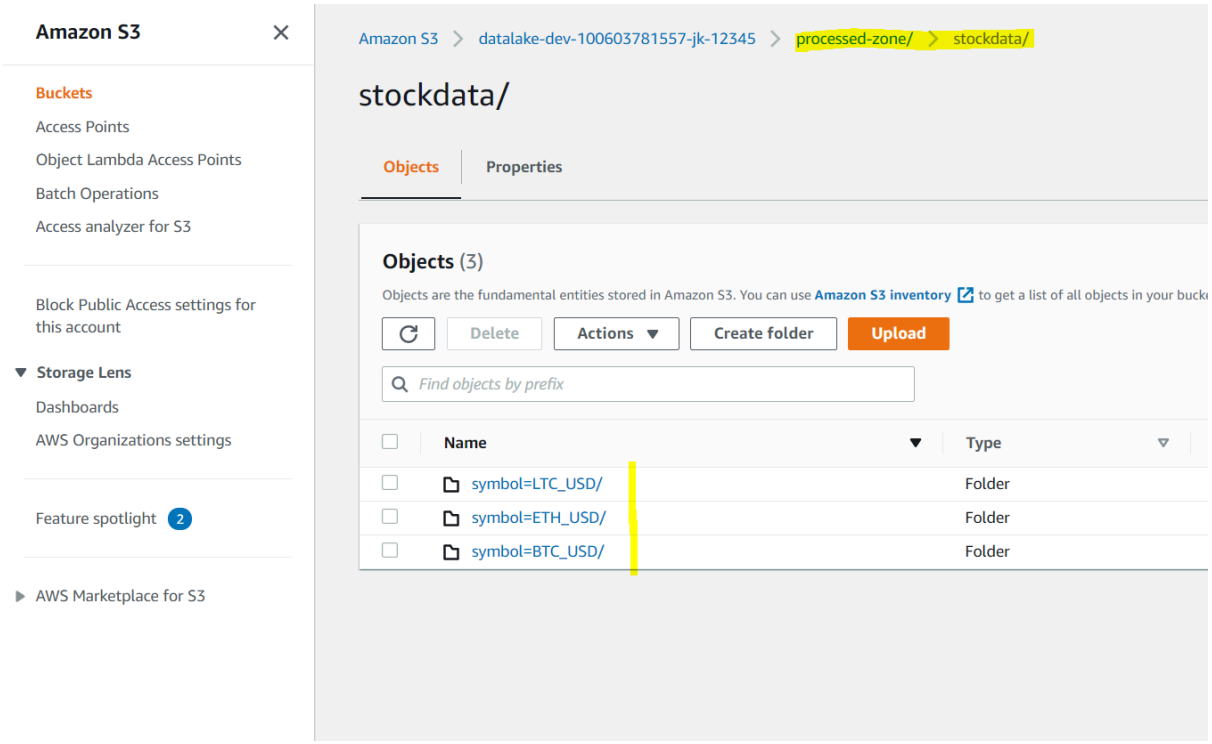
Przykładowe definicje:

```
resource "aws_lambda_permission" "allow_bucket" {
 statement_id  = "AllowExecutionFromS3Bucket"
 action        = "lambda:InvokeFunction"
 function_name = aws_lambda_function.etl_post_processing.arn
 principal     = "s3.amazonaws.com"
 source_arn    = aws_s3_bucket.raw_bucket.arn
}

resource "aws_s3_bucket_notification" "trigger_etl_lambda" {
 bucket = aws_s3_bucket.raw_bucket.id

 lambda_function {
   lambda_function_arn = aws_lambda_function.etl_post_processing.arn
   events              = ["s3:ObjectCreated:*"]
   filter_prefix       = "raw-zone/"
 }

 depends_on = [aws_lambda_permission.allow_bucket]
}
```

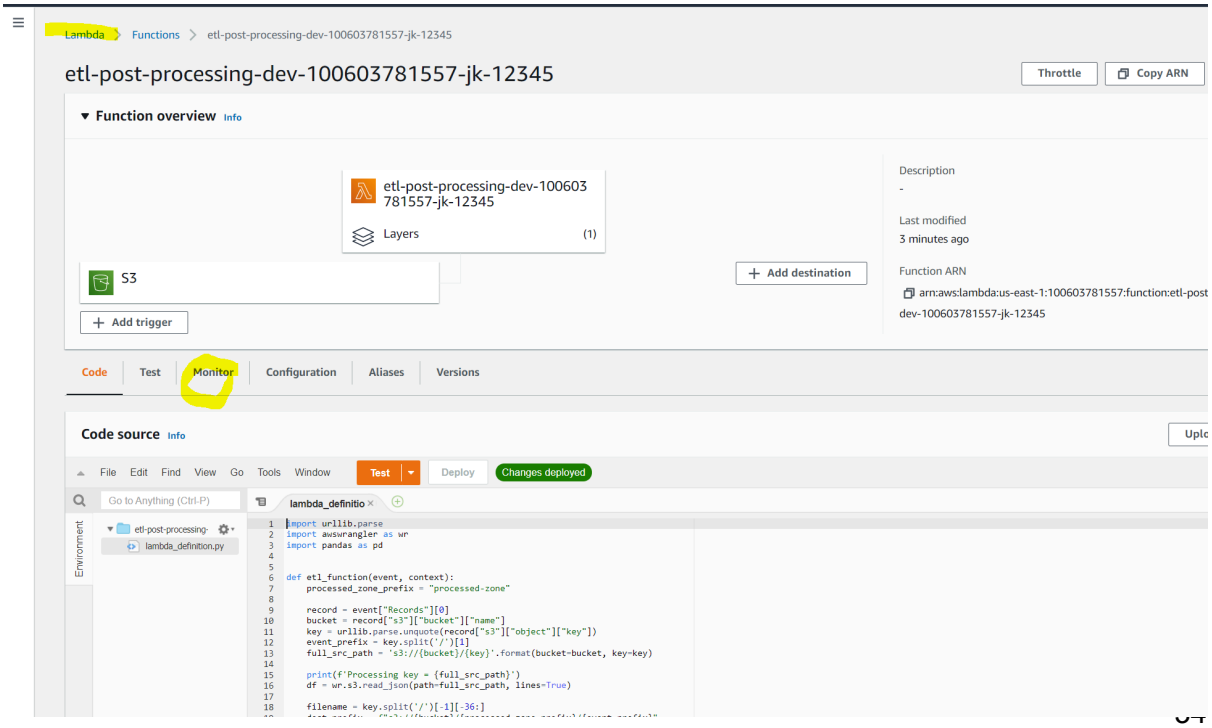
2. W tym momencie możemy wdrożyć zmiany. Uruchom **terraform apply.**

3. Po utworzeniu wszystkich obiektów **wystartuj ponownie generator danych** - odczekaj minutę.
    a. Sprawdź czy dane są automatycznie przenoszone do prefixu processed-zone (dotyczyć to będzie wszsystkich nowych plików zrzucanych do S3 przez Firehose).
    Zwróć uwagę, że funkcja Lambda, zmienia także schemat partycjonowania.

Nadrzędną partycją jest informacja o typie symbolu (notowania). Dzięki temu nasze zapytania będą znacznie bardziej wydajne w kwerendach dotyczących konkretnych symboli (eliminacja partycji). Sprawdź w jakich datach (y/m/d/h) transakcje zostały wygenerowane.




b. Sprawdź w konsoli AWS jak wygląda funkcja lambda i czy faktycznie jest wywoływana (metryki i logi pojawiają się z pewnym opóźnieniem).



4. Utwórz dodatkową tabelę w Glue. Możesz stworzyć / zaktualizować istniejące Crawlera lub zarejestrować tabelę z poziomu Athena
    a. Przykład jak tworzyć tabele z poziomy Athena (SQL DDL) - dostosuj lokalizację S3 i nazwę bazy dancych :

```
CREATE EXTERNAL TABLE <twoj raw db name>.processed_stockdata(
  transaction_date timestamp,
  price double,
  amount double,
  dollar_amount double,
  type string,
  trans_id bigint)
PARTITIONED BY (
  symbol string,
  year integer,
  month integer,
  day integer,
  hour integer
  )
ROW FORMAT SERDE
  'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT
  'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
LOCATION
  's3://<twoj raw zone bucket>/processed-zone/stockdata/';
```

Zostanie utworzona nowa tabela w Glue DC. Na Razie sama definicja bez partycji (sprawdź w Glue DC).

Aby odświeżyć partycje wykonaj poniższe polecenie

```
MSCK REPAIR TABLE processed_stockdata;
```

Zostanie przeskanowana lokalizacja główna S3 podana w parametrach tabeli. Automatycznie zostaną dodane wszystkie brakujące partycje. Spróbuj odpytać tabelę.

Zwróć uwagę, że wykorzystując atrybuty partycjonowania, ograniczasz ilość skanowanych danych. Dodatkowo wybierając pojedyncze atrybuty również ograniczasz koszty (ilość skanowanych danych).

W tym momencie RAW zone i Processed-zone nie są w pełni zsynchronizowane. Funkcja Lambda jest wyzwalana tylko dla nowo

pojawiających się plików.

Spróbuj zaproponować sposób synchronizacji danych (weź pod uwagę idempotentność).

5. Wykonaj porównanie strefy RAW z PROCESSED (reconciliation).
    a. Wyczyść całe środowisko **(terraform destroy & apply)** i rozpocznij generowanie danych od początku.
    Uruchom generator z opcją single run **-r (single run)**

    **$ python generator.py -k cryptostock-211125385450-jk-12345 -r**

    Poczekaj aż przynajmniej dwie pierwsze godziny z transakcjami zostaną wysłane i zrzucone do RAW zone.
    Uruchom stworzonego crawlera i utwórz raz jeszcze tabele w strefie processed (odśwież partycje).
    b. Porównaj pierwszy bucket godzinowy w obu warstwach lake'a

```
SELECT count(*)
FROM "datalake_raw_100603781557_jk_12345"."crawler_stockdata"
WHERE symbol = 'BTC_USD' and
date_format(from_unixtime(transaction_ts),'%Y-%m-%dT%H') =
'2020-10-01T00';

SELECT count(*)
FROM "datalake_raw_100603781557_jk_12345"."processed_stockdata"
WHERE HOUR=0 and DAY=1 and MONTH=10 and YEAR=2020
      and symbol = 'BTC_USD';
```

    c. Powinieneś otrzymać te same ilości rekordów w obu warstwach (337 sztuk dla pary BTC_USD)

6. Modyfikacja funkcji lambda i jej parametryzacja z wykorzystanie Parameter Store

    Obecnie dane są przerzucane pomiędzy prefixami tego samego bucketu. Nasze główne założenie jest takie, aby dane przetworzone były składowane w osobnym buckecie.

    Zaproponuj rozwiązanie w którym funkcja lambda będzie sparametryzowana i wrzucała dane do bucketu **Processed**.

Rozważ gdzie przechowywać parametry ? Czy w funkcji Lambda?
Jednym z rozwiązań może byc serwis parameter store.
aws_ssm_parameter | Resources | hashicorp/aws | Terraform | Terraform Registry

Stwórz osobną bazę danych (datalake_processed….. ) dla nowej tabeli.
Stwórz osobny proces do katalogowania danych w tej bazie (bucket processed).

**Parameter store 101**

a. Otwórz Cloud9
b. Wykorzystaj pakiet boto3 (jest już zainstalowany) i utwórz nowy parametr używając boto3 (możesz napisać skrypt .py)

   Nazwa paramteru - **foo**
   Wartość parametry - **bar**
   Typ parametru - **String**

   Kroki do wykonania

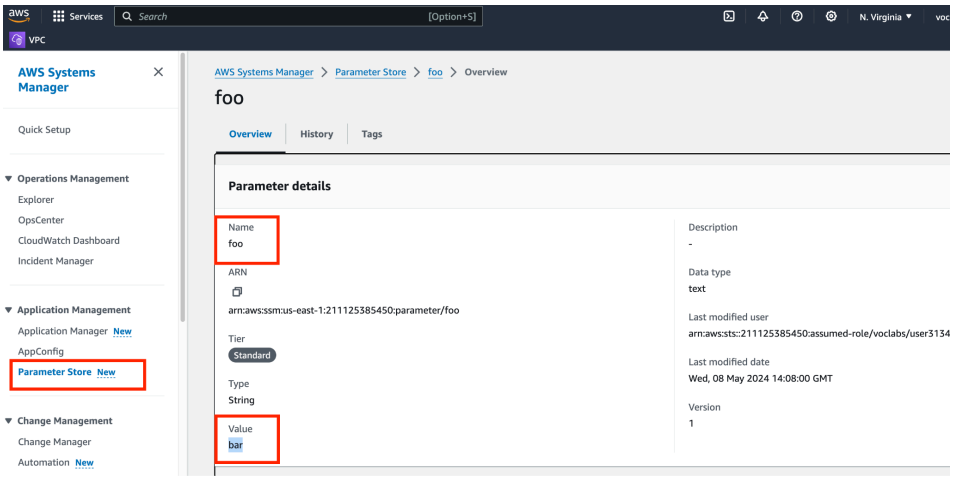
   i. Utwórz w skrypcie lub w repl python klienta ssm
      SSM - Boto3 1.34.99 documentation (amazonaws.com)

   ii. Wykorzystaj metodę describe_parameters clienta ssm aby pobrać informację o parametrach w SSM

      describe_parameters - Boto3 1.34.100 documentation (amazonaws.com)

   iii. Utwórz nowy parametr metodą **put_parameter**
      put_parameter - Boto3 1.34.100 documentation (amazonaws.com)

iv. Przejdź w konsoli AWS do usługi parameter store i upewnij się że nowy parametr został poprawnie utworzony




c. Utwórz nowy parametr z kodu terraform.

d. Wykorzystaj ten parameter w funkcji lambda (zmodufikuj kod funkcji lambda aby pobierała zmienne z parameter store (bucket name - odzwierciedlający Twoją warstwę Data Lake - processed))

## Ćwiczenie 5

Wykorzystanie frameworka GLUE ETL do tworzenia i uruchamiania Spark jobów.

W tym ćwiczeniu wykorzystasz podejście “low code” do utworzenia joba Spark oraz zaznajomisz się z frameworkiem Glue ETL.

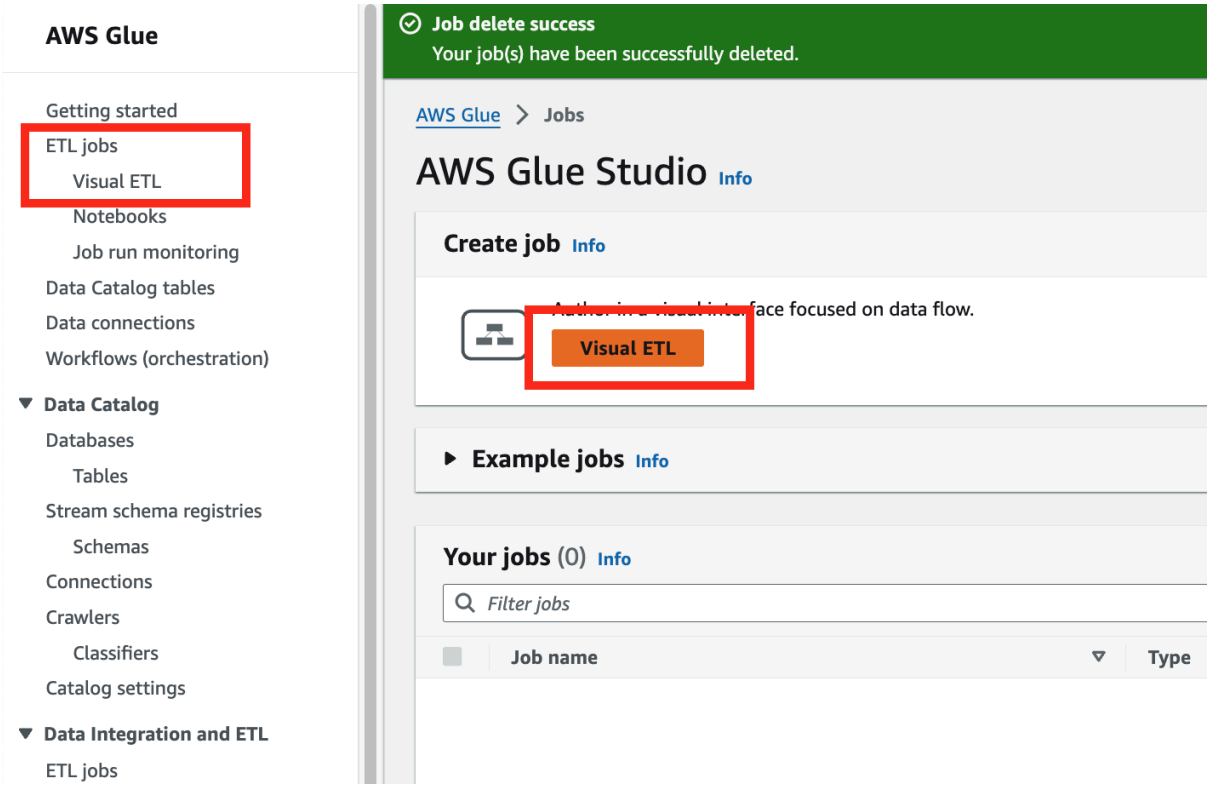
Scenariusz zakłada przerzucenie danych do bucketu **Processed** z wykorzystaniem joba **Spark** (proces curation / transformation) jak również katalogowanie danych znanymi już metodami.

### Proces Curation

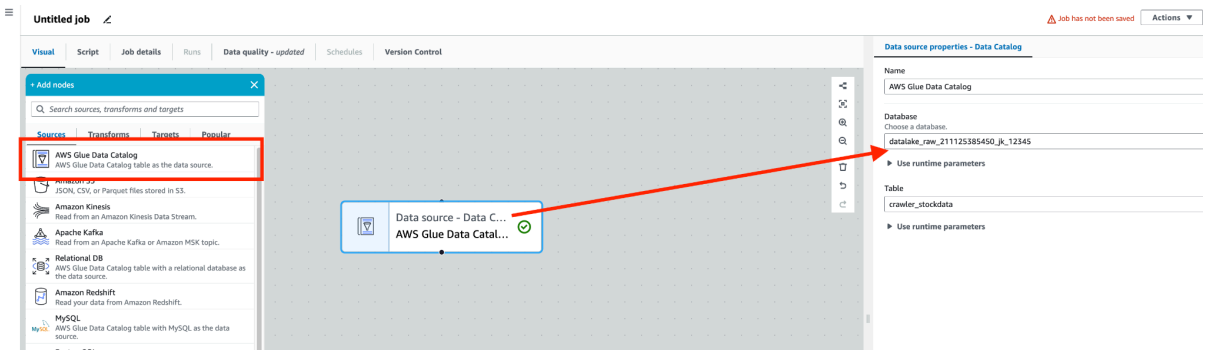
1. Wykorzystasz dane które zostały załadowane do prefixu **raw** bucketu **RAW** celem przeprocesowania ich do warstwy **PROCESSED.**
2. Stwórz **nową** tabele Glue w Twoje bazie danych **Processed**, bucket **Processed** o następującej strukturze (zwróć uwagę na dodatkowy parametr na końcu - `'useGlueParquetWriter'='true'`):

```
CREATE EXTERNAL TABLE <twoja processed db name>.processed_stockdata(
  transaction_date timestamp,
  price double,
  amount double,
  dollar_amount double,
  type string,
  trans_id bigint)
PARTITIONED BY (
  symbol string,
  year integer,
  month integer,
  day integer,
  hour integer
  )
ROW FORMAT SERDE
  'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT
  'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
LOCATION
  's3://<twoj processed zone bucket>/stockdata/'
TBLPROPERTIES (
   'useGlueParquetWriter'='true');
```

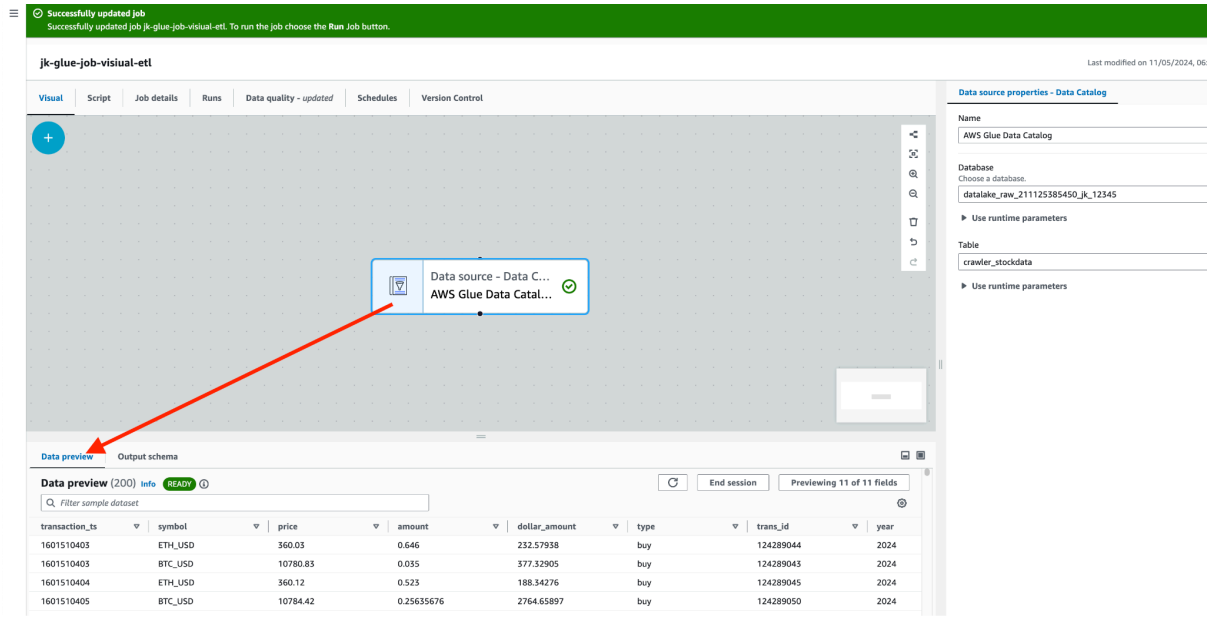
3. Otwórz w konsoli AWS serwis GLUE i wybierz ETL



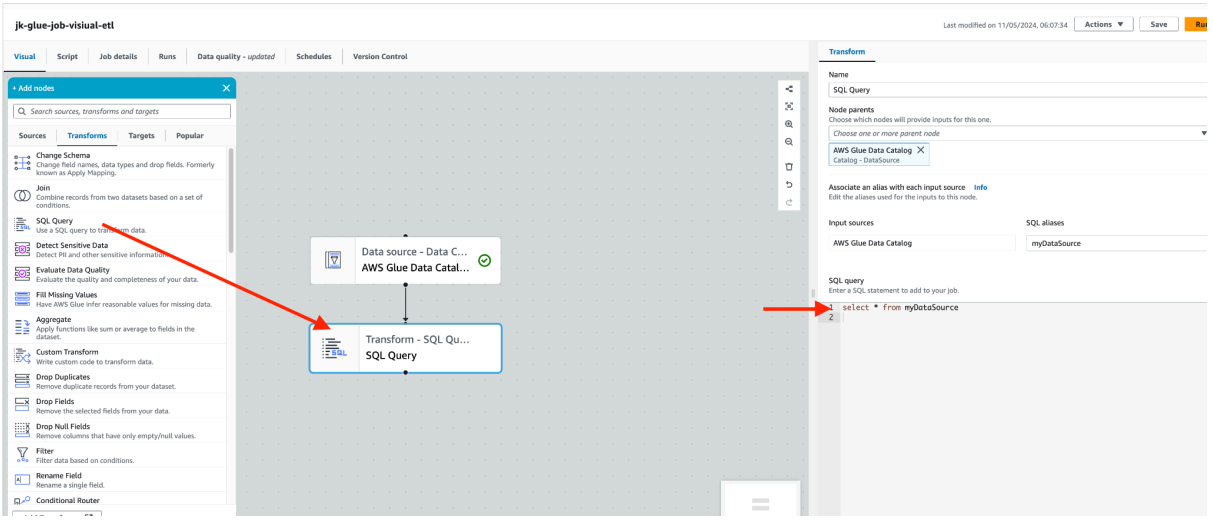

4. Jako źródło danych wybierz tabelę z bazy RAW (surowe dane) - crawler-stockdata



5. Możesz podejrzeć dane (wybierz jako rolę do wykonywania joba LabRole)




6. Teraz czas na transformacje - wybierz SQL

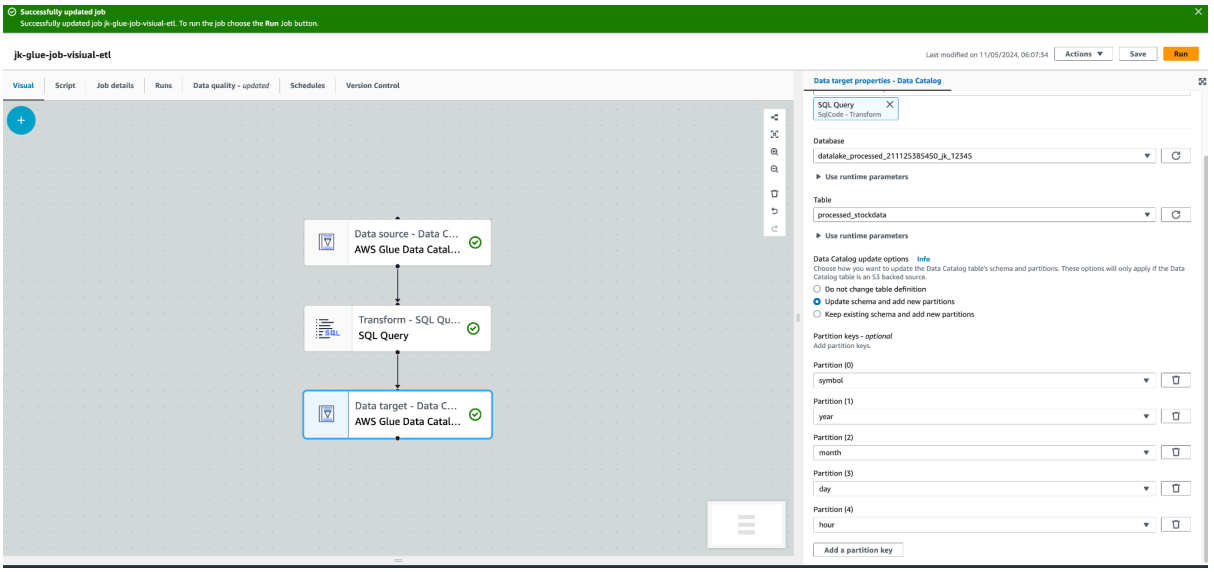



7. Napisz samodzielnie zapytanie - transformacje, które realizuje obecnie funkcja Lamba przerzucająca dane pomiędzy prefiksami Raw zone. Musisz zmienić sposób partycjonowania - wykorzystać informacje o dacie i czasie z kolumny timestamp_ts. Pamiętaj że w SQL trzeba najpierw dokonać konwersji typów danych na timestamp:

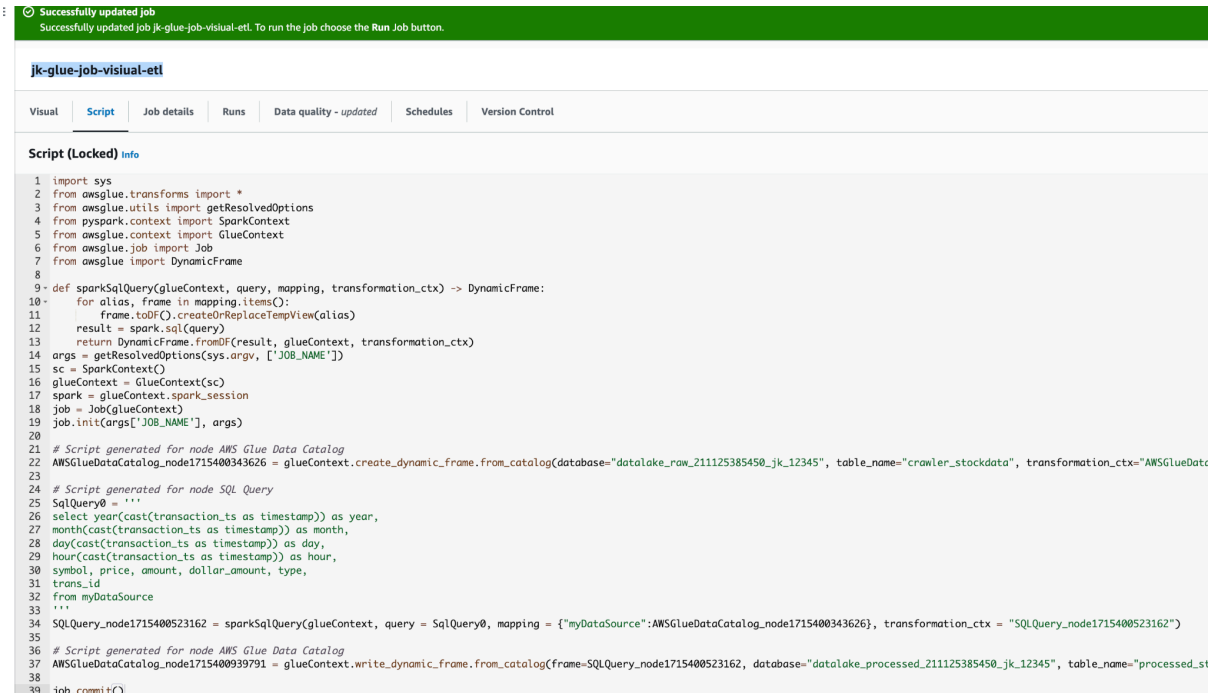
   **CAST**(transaction_ts as timestamp)

   A następnie wykorzystaj skalarne funkcje wbudowane w SQL - **YEAR, MONTH, DAY, HOUR** celem estrakcji odpowiednich części daty.

8. Dodaj jako destination nowo utworzoną tabelę glue (processed_stockdata z bazy processed).Skonfiguruj schemat partycjonowania i ustaw aby JOB aktualizował schemat i partycje..




9. Zapisz i nazwji Joba np. : <inicjały>-visual-etl-raw-to-processed.
10. Kliknij w zakładkę Script i przejrzyj kod Sparkowy




```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from awsglue import DynamicFrame

def sparkSqlQuery(glueContext, query, mapping, transformation_ctx) -> DynamicFrame:
    for alias, frame in mapping.items():
        frame.toDF().createOrReplaceTempView(alias)
    result = spark.sql(query)
    return DynamicFrame.fromDF(result, glueContext, transformation_ctx)
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)

# Script generated for node AWS Glue Data Catalog
AWSGlueDataCatalog_node1715400343626 = glueContext.create_dynamic_frame.from_catalog(database="datalake_raw_211125385450_jk_12345", table_name="crawler_stockdata", transformation_ctx="AWSGlueData

# Script generated for node SQL Query
SqlQuery0 = '''
select year(cast(transaction_ts as timestamp)) as year,
month(cast(transaction_ts as timestamp)) as month,
day(cast(transaction_ts as timestamp)) as day,
hour(cast(transaction_ts as timestamp)) as hour,
symbol, price, amount, dollar_amount, type,
trans_id
from myDataSource
'''
SQLQuery_node1715400523162 = sparkSqlQuery(glueContext, query = SqlQuery0, mapping = {"myDataSource":AWSGlueDataCatalog_node1715400343626}, transformation_ctx = "SQLQuery_node1715400523162")

# Script generated for node AWS Glue Data Catalog
AWSGlueDataCatalog_node1715400939791 = glueContext.write_dynamic_frame.from_catalog(frame=SQLQuery_node1715400523162, database="datalake_processed_211125385450_jk_12345", table_name="processed_st

job.commit()
```

11. Przejrzyj zakładkę Job Details i dostosuj liczbę workerów (wybierz 2 - domyślnie jest to aż 10).
12. Uruchom Joba (RUN) i zweryfikuj poprawność działania.

### Proces Transformation

1. Wykorzystaj dane zmaterializowane w poprzednim ćwiczeniu - tabela stockdata z bazy processed

2. Utwórz tabelę z agreagacjami jak poniżej

```
CREATE EXTERNAL TABLE `agg_stockdata`(
  `total_volume` double,
  `total_dollars` double,
  `total_cnt_of_transactions` int,
  'type' string
)
PARTITIONED BY (
  `symbol` string,
  `year` int,
  `month` int,
  `day` int)
ROW FORMAT SERDE
  'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'
STORED AS INPUTFORMAT
  'org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat'
LOCATION
  's3://<Twoj bucket processed>/agg_stockdata'
```

3. Utwórz Spark joba który
    a. Pobierze dane ze źródła (baza processed, tabela stockdata)
    b. wykona agregację (grupowanie)

        Partycjonowanie po :
            **Symbol**
            **Rok**
            **Miesiąc**
            **Dzień**
        Agregacje:
            **Total_volume** - jako suma wolumenu wszystkich transakcji w ramach danej partycji - wuyrażona liczbą handlowanych pozycji - uwzględnij w grupowaniu typ transacakcji (kolumna type) ale nie partycjonuj po niej)

**Total_dollars** - jako suma wolumenu wszystkich transakcji w ramach danej partycji - wuyrażona w dolaraach - uwzględnij w grupowaniu typ transacakcji (kolumna type) ale nie partycjonuj po niej)

**Total_cnt_of_transactions** - jako liczba transakcji w ramach partycji (count) z uwzględnieniem typu (buy/sell)

c. Zapisze dane do nowej tabeli

4. Dostosuj parametry uruchomieniowe joba (liczba workerów)
5. Uruchom Spark joba i sprawdź poprawność jego działania - zweryfikuj dane w tabeli docelowej.
6. Uruchom Spark joba ponownie i sprawdź dane jeszcze raz. Znajdź rozwiązanie aby job był zbudowany zgodnie z zasadą idempotentności (dowolna liczba jego uruchomień powinno dać odpowiedni rezultat).