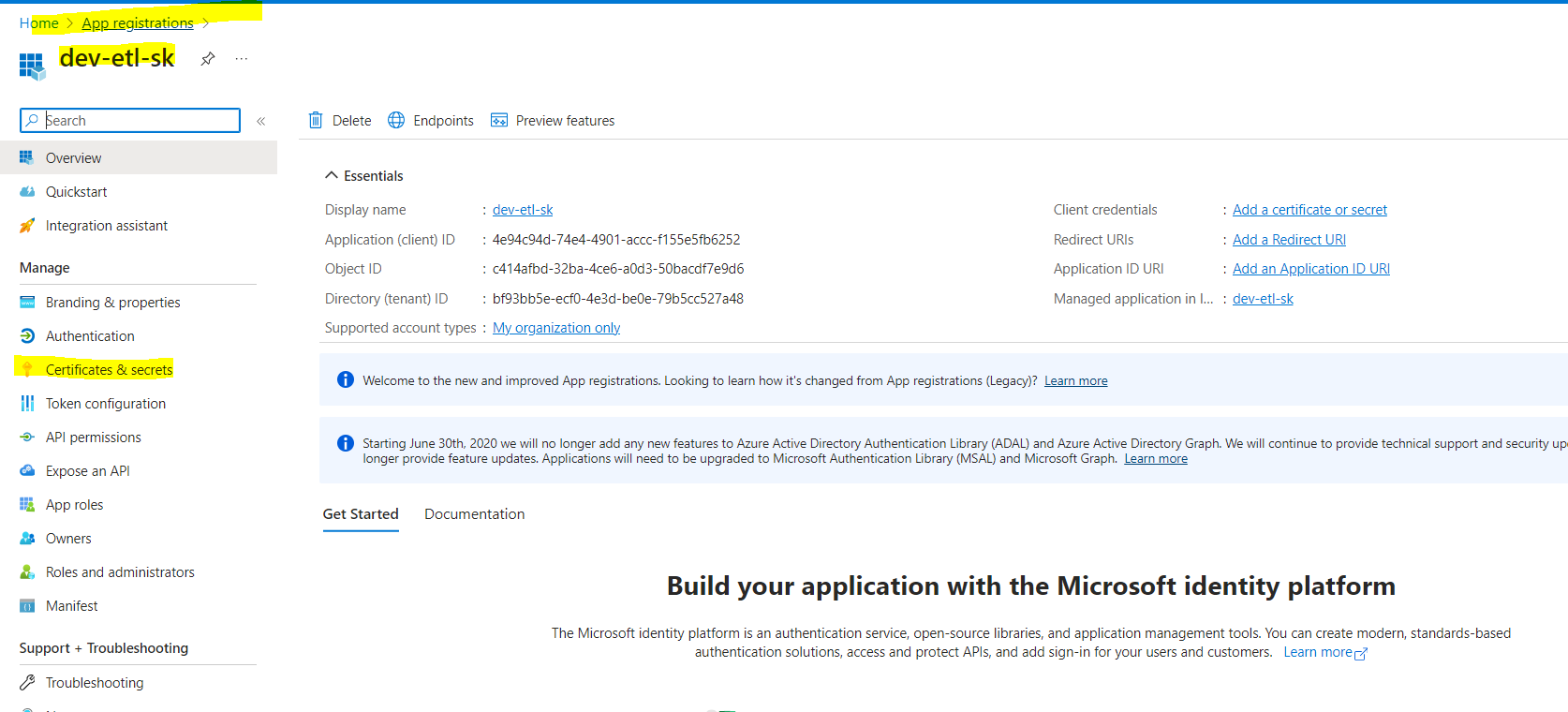
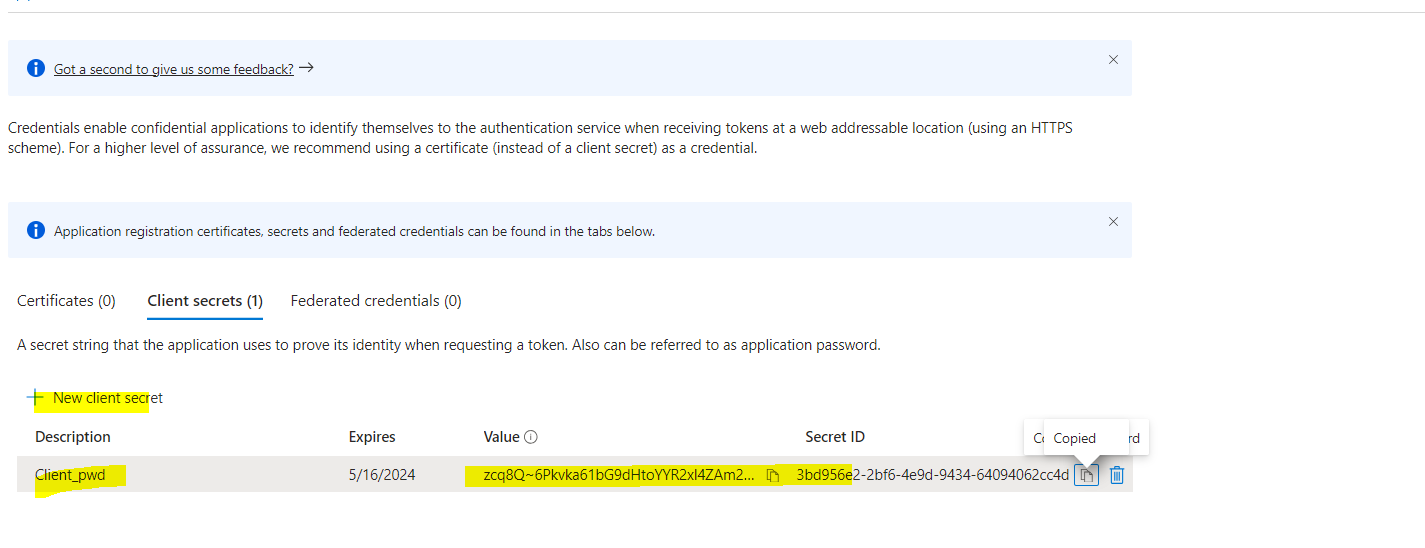
## Metadata Driven ETL Pipeline

Step 1 🡪 Create the resource group say metadatalab and create the keyvault say dev-akv-182023

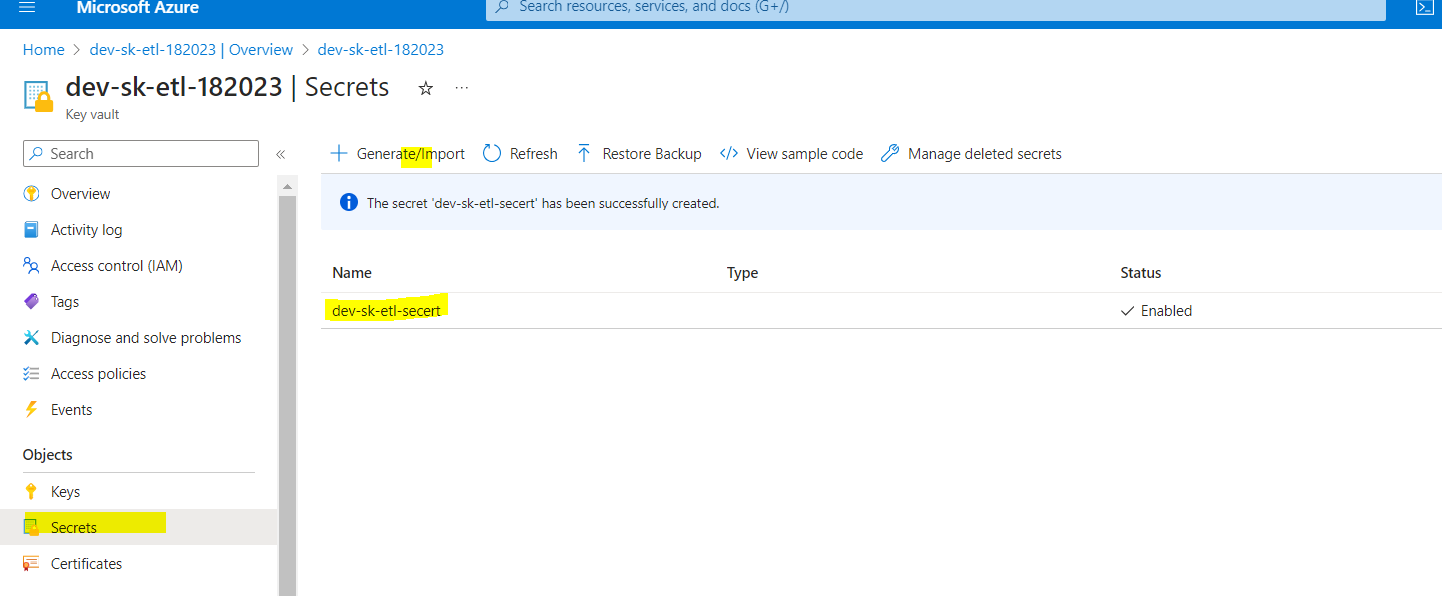
Step2 🡪 Create a service principal from app registration

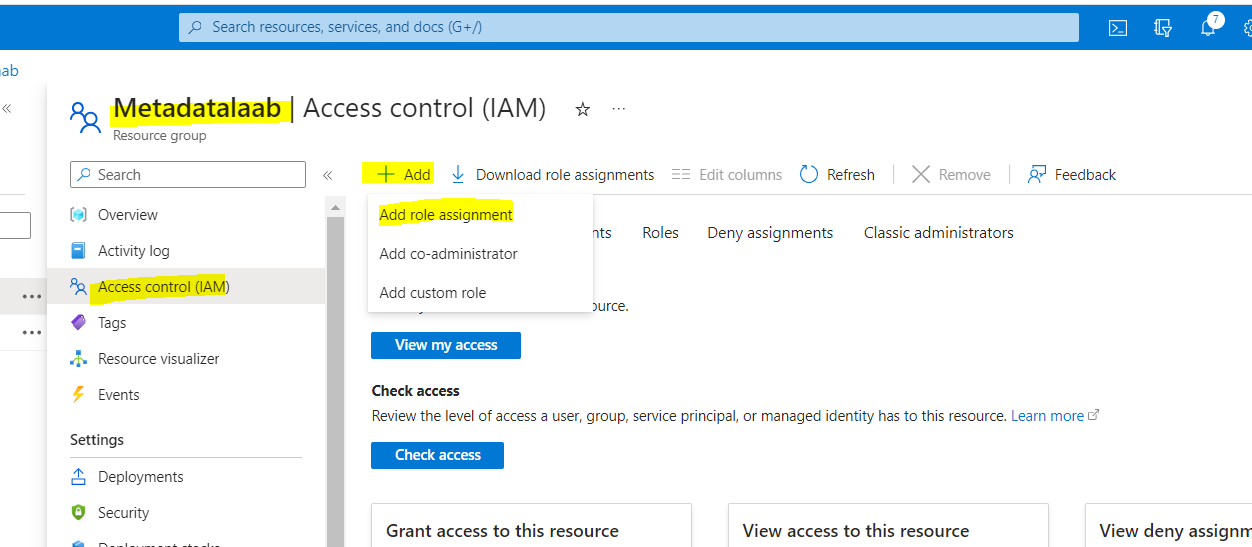


Create a new client secret

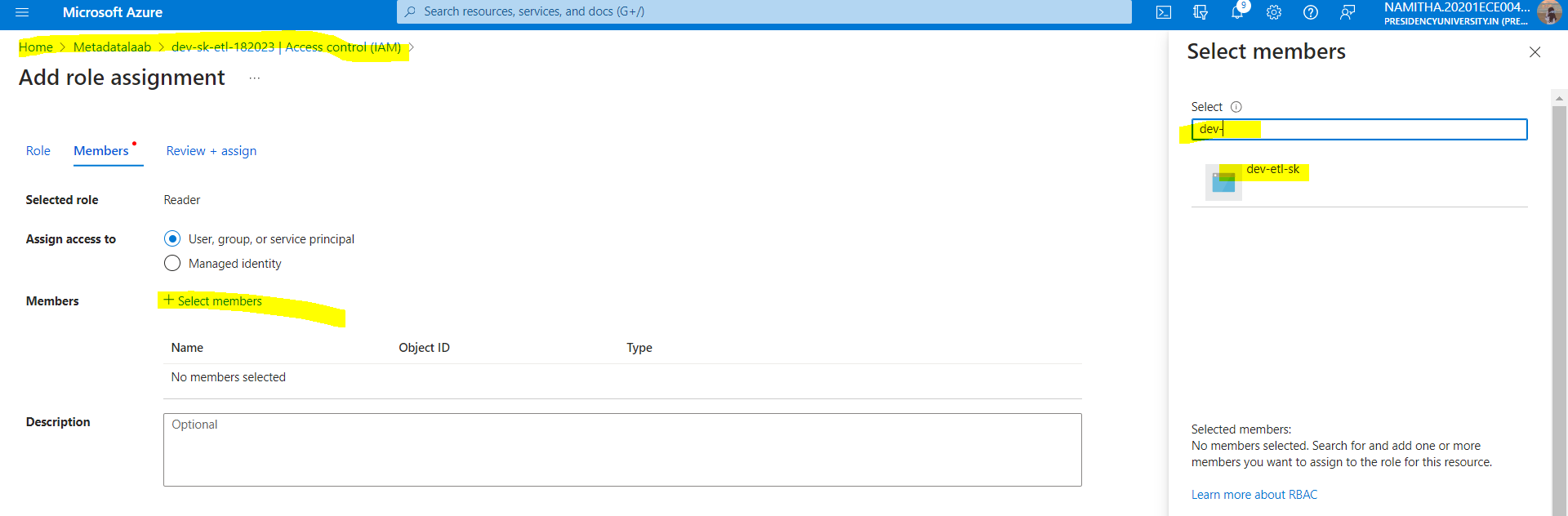


Place the secret in key vault

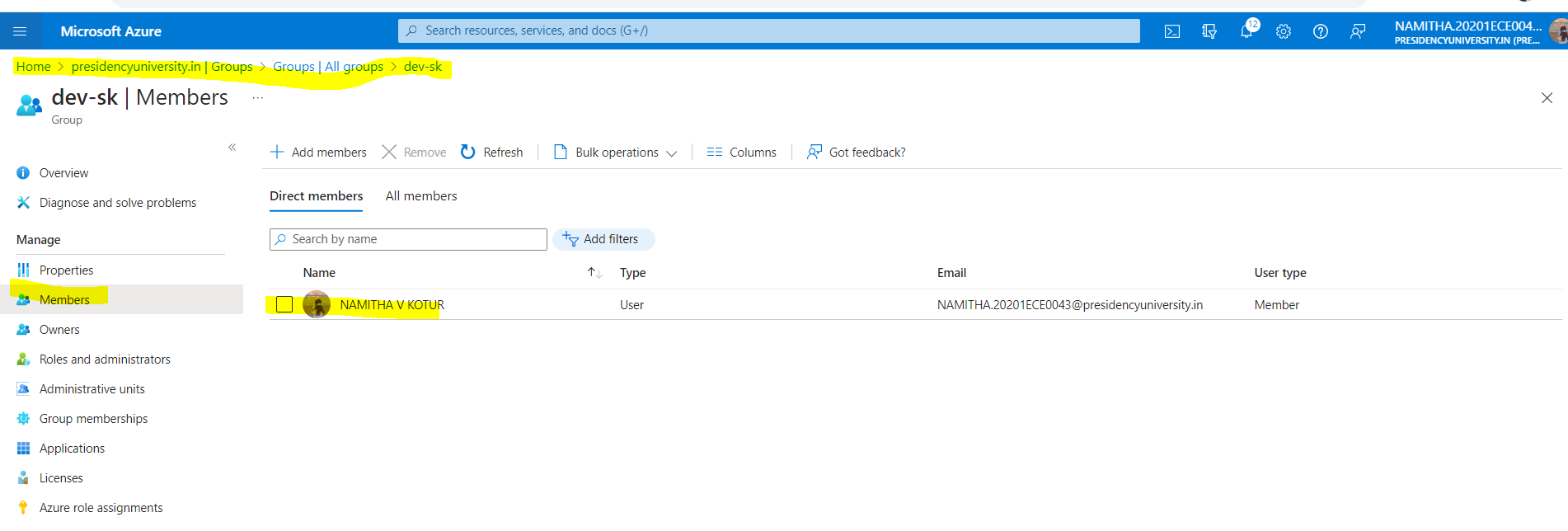
  
give reader access to your resource group



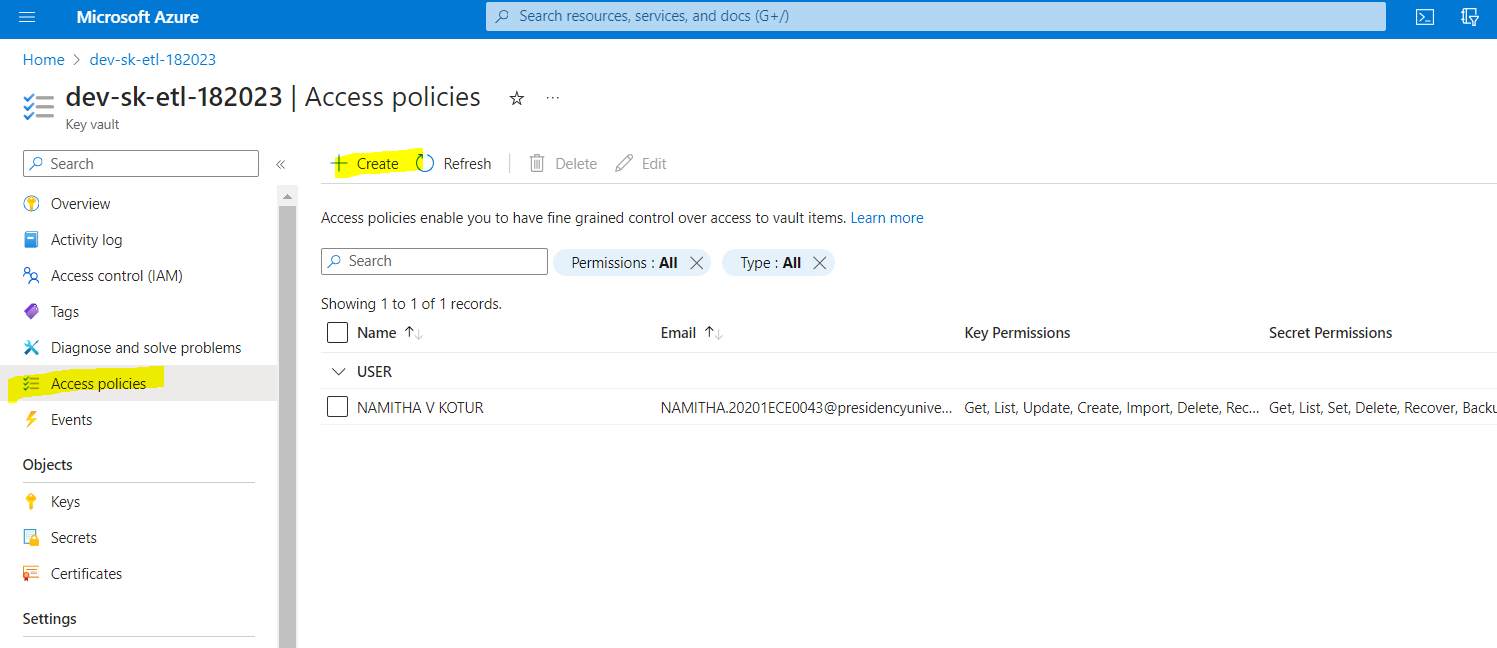
Give reader role to service principal also through key vault



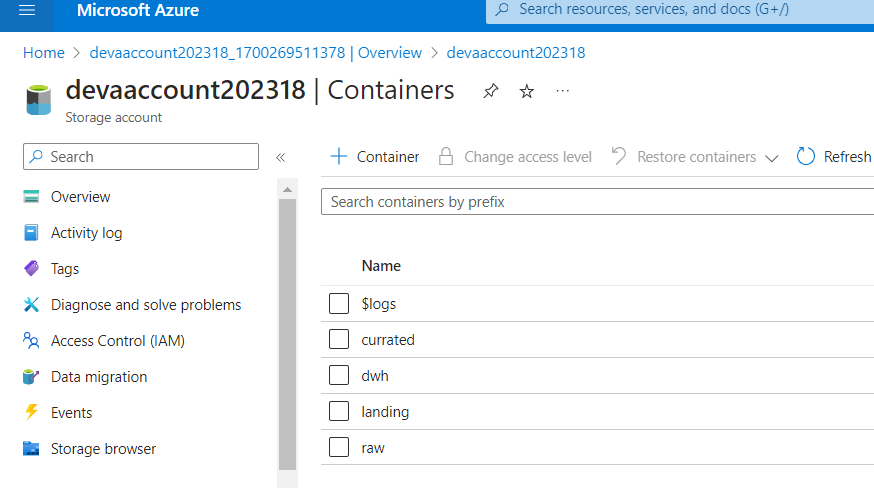
First create the user group dev-sk in active directory and add your member



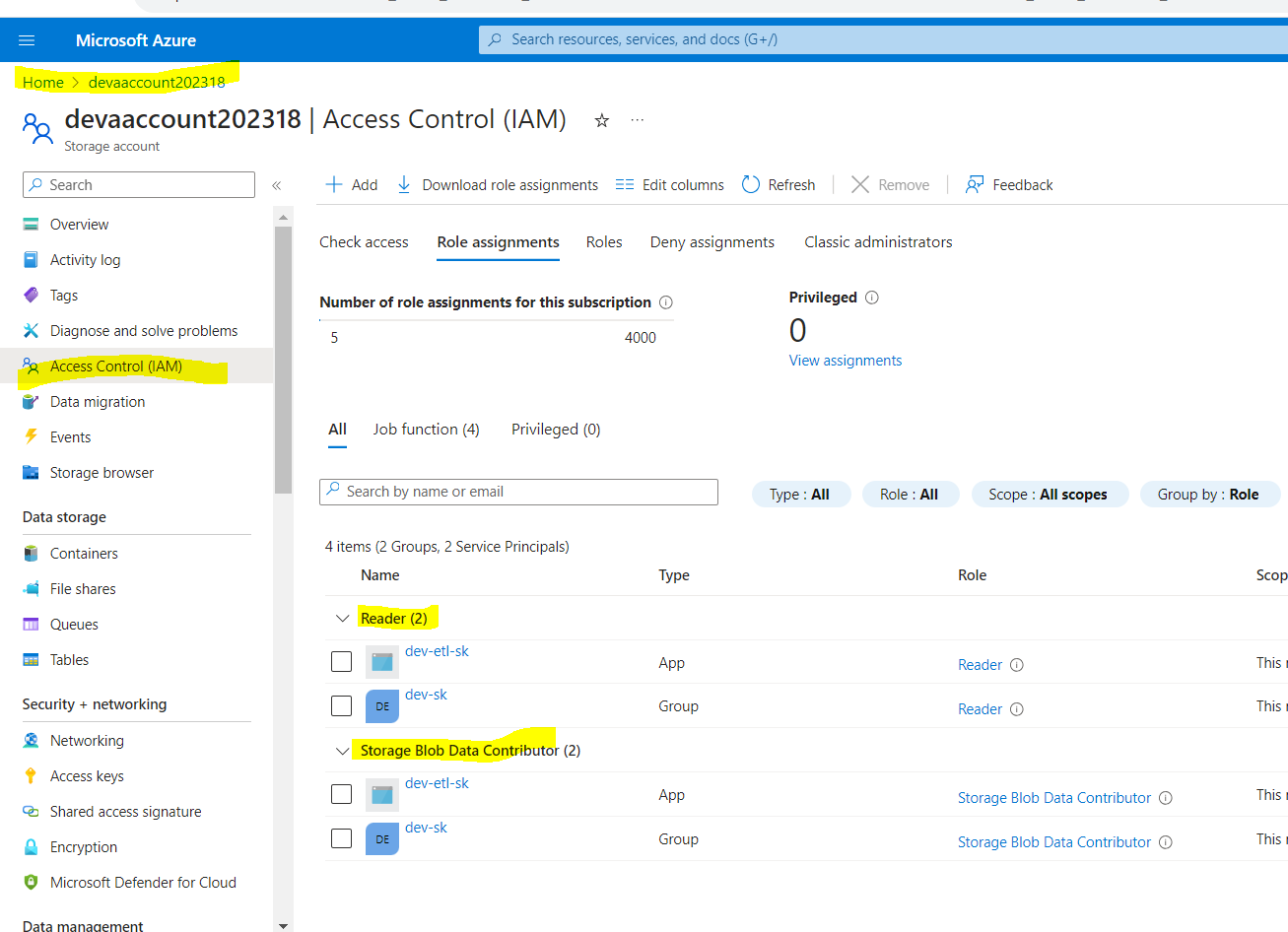
Give get aand list permission in key vault to service principaal



Crete the ADLS storage 🡪 under it create 3 container landing raw and curated and one for datawarehouse



Give access of storage account



Create the sql data base

Excute the following scripts for metadata tables

--create schema config;

--go

--create schema control;

--go

--create schema log;

--config has below tables

--jobs, tasks, task\_parameter, job\_tasks, job\_task\_parameter, enviorment

--control schema have below tables

--job\_task\_control, task\_paramter\_control, job\_task\_watermark

--log schema have below tables

--job\_task\_log

drop table if exists config.jobs

go

create table config.jobs

(

job varchar(4000) not null,

active char(1) not null,

insert\_dt datetime not null default getdate(),

update\_dt datetime

)

on [primary]

go

alter table config.jobs

add constraint uk\_jobs\_job unique(job)

go

drop table if exists config.tasks

go

create table config.tasks

(

task varchar(4000) not null,

active char(1) not null,

insert\_dt datetime not null default getdate(),

update\_dt datetime

)

on [primary]

go

alter table config.tasks

add constraint uk\_tasks\_task unique(task)

go

drop table if exists config.task\_parameters

go

create table config.task\_parameters

(

task varchar(4000) not null,

parameter varchar(4000) not null,

parameter\_type varchar(4000) not null,

active char(1) not null,

insert\_dt datetime not null default getdate(),

update\_dt datetime

)

on [primary]

go

alter table config.task\_parameters

add constraint uk\_task\_parameters\_task\_parameter unique(task,parameter)

go

alter table config.task\_parameters

add constraint fk\_task\_parameters\_task

foreign key(task) references config.tasks(task)

go

drop table if exists config.job\_tasks

go

create table config.job\_tasks

(

job varchar(4000) not null,

task varchar(4000) not null,

task\_sequence int not null,

active char(1) not null,

insert\_dt datetime not null default getdate(),

update\_dt datetime

)

on [primary]

go

alter table config.job\_tasks

add constraint uk\_job\_tasks\_job\_task unique(job,task)

go

alter table config.job\_tasks

add constraint fk\_job\_tasks\_job

foreign key(job) references config.jobs(job)

go

alter table config.job\_tasks

add constraint fk\_job\_tasks\_task

foreign key(task) references config.tasks(task)

go

drop table if exists config.job\_task\_parameters

go

create table config.job\_task\_parameters

(

job varchar(4000) not null,

task varchar(4000) not null,

parameter varchar(4000) not null,

value varchar(4000) not null,

active char(1) not null,

task\_sequence int not null,

insert\_dt datetime not null default getdate(),

update\_dt datetime

)

on [primary]

go

alter table config.job\_task\_parameters

add constraint uk\_job\_task\_parameters\_job\_task\_parameter unique(job,task,parameter)

go

alter table config.job\_task\_parameters

add constraint fk\_job\_task\_parameters\_job\_task

foreign key(job,task) references config.job\_tasks(job,task)

go

alter table config.job\_task\_parameters

add constraint fk\_job\_task\_parameters\_task\_parameter

foreign key(task,parameter) references config.task\_parameters(task,parameter)

go

drop table if exists config.environment

go

create table config.environment

(

parameter varchar(4000) not null,

value varchar(4000) not null,

insert\_dt datetime not null default getdate(),

update\_dt datetime

)

on [primary]

go

alter table config.environment

add constraint uk\_environment\_parameter unique(parameter)

go

drop table if exists control.job\_task\_control

go

create table control.job\_task\_control

(

job\_id varchar(4000) not null,

task\_id varchar(4000) not null,

job varchar(4000) not null,

job\_dt date not null,

task varchar(4000) not null,

task\_sequence int not null,

task\_status varchar(4000) not null,

insert\_dt datetime not null default getdate(),

update\_dt datetime

)

on [primary]

go

drop table if exists control.task\_parameter\_control

go

create table control.task\_parameter\_control

(

job\_id varchar(4000) not null,

task\_id varchar(4000) not null,

job varchar(4000) not null,

task varchar(4000) not null,

task\_sequence int not null,

parameter varchar(4000) not null,

value varchar(4000) not null,

insert\_dt datetime not null default getdate(),

update\_dt datetime

)

on [primary]

go

drop table if exists control.job\_task\_watermark

go

create table control.job\_task\_watermark

(

job varchar(4000) not null,

task varchar(4000) not null,

last\_run\_date datetime not null,

insert\_dt datetime not null default getdate(),

update\_dt datetime

)

on [primary]

go

drop table if exists log.job\_task\_log

go

create table log.job\_task\_log

(

job\_id varchar(4000) not null,

task\_id varchar(4000) not null,

job varchar(4000) not null,

task varchar(4000),

task\_sequence int,

job\_status varchar(4000),

task\_status varchar(4000),

status\_description varchar(4000) not null,

insert\_dt datetime not null default getdate()

)

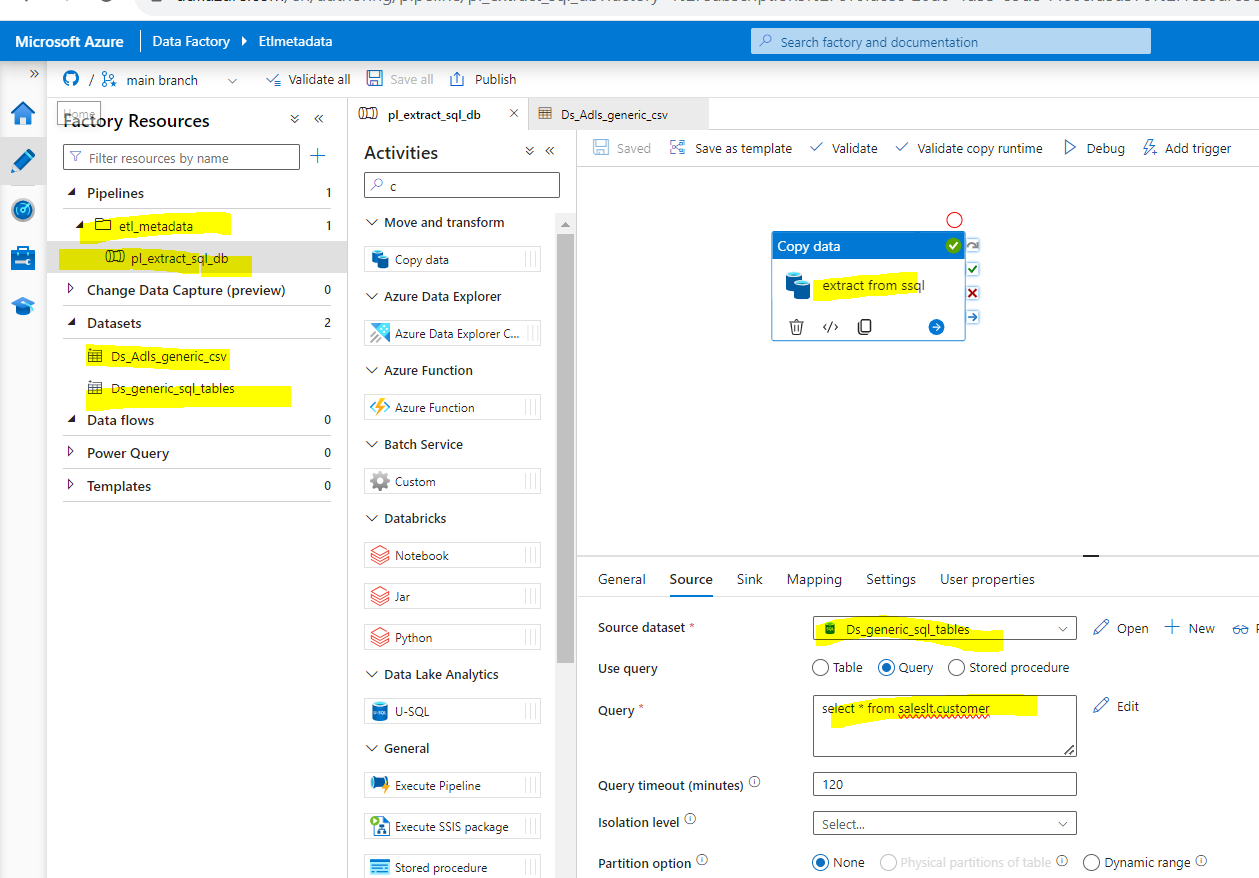
on [primary]

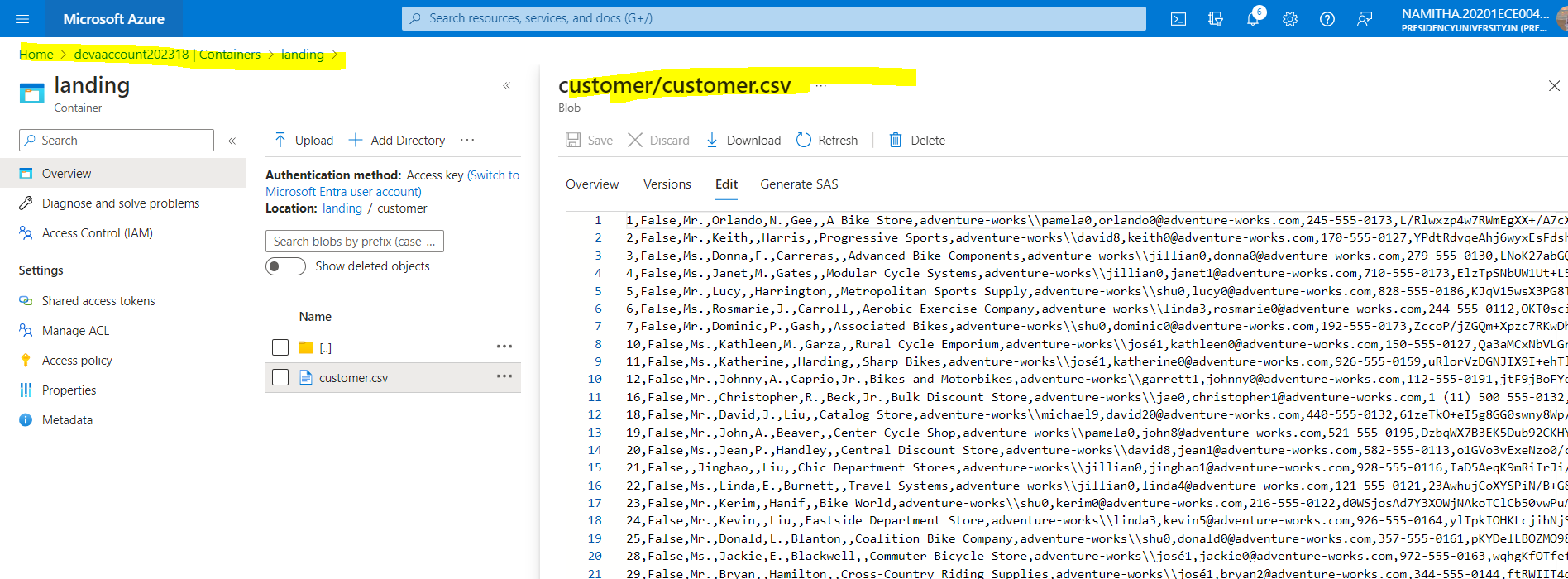
go

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

## Create a adf and start building pipeline

First 🡪 Create a copy activity 🡪 in it create the dataset for sql server and adls gen 2 generic , check for customer data from saleslt.customer table, move data to landing folder with customer subfolder in csv





Just for checking we have parameter the directory and filename of adls ds, and hardcoded, we need to change it to dynamic

So We have these tables where we will pass all the parameter in the tables information about the query, adls parameter for folder and filename, job info, pipeline name as below, and then write the storeprocdure to get the required info and pass that to copy data activity   
  
  
  
Lab Script - Metadata Table Configurations

1. Config.Jobs

==========

insert into config.jobs (job,active)

values ('load\_customer','Y')

2. Config.Tasks

============

insert into config.tasks(task,active)

values ('pl\_extract\_from\_sqldb','Y')

3. Config.Job\_Tasks

===============

insert into config.job\_tasks(job,task,task\_sequence,active)

values ('load\_customer','pl\_extract\_from\_sqldb',1,'Y')

4. Config.Task\_Parameters

====================

insert into config.task\_parameters(task,parameter,parameter\_type,active)

values ('pl\_extract\_from\_sqldb','src\_sql','static','Y')

insert into config.task\_parameters(task,parameter,parameter\_type,active)

values ('pl\_extract\_from\_sqldb','tgt\_folder','static','Y')

insert into config.task\_parameters(task,parameter,parameter\_type,active)

values ('pl\_extract\_from\_sqldb','tgt\_filename','static','Y')

5. Config.Job\_Task\_Parameters

=======================

insert into config.job\_task\_parameters(job,task,parameter,value,active,task\_sequence)

values ('load\_customer','pl\_extract\_from\_sqldb','src\_sql','select \* from saleslt.customer','Y',1)

insert into config.job\_task\_parameters(job,task,parameter,value,active,task\_sequence)

values ('load\_customer','pl\_extract\_from\_sqldb','tgt\_folder','customer','Y',1)

insert into config.job\_task\_parameters(job,task,parameter,value,active,task\_sequence)

values ('load\_customer','pl\_extract\_from\_sqldb','tgt\_filename','customer.csv','Y',1)

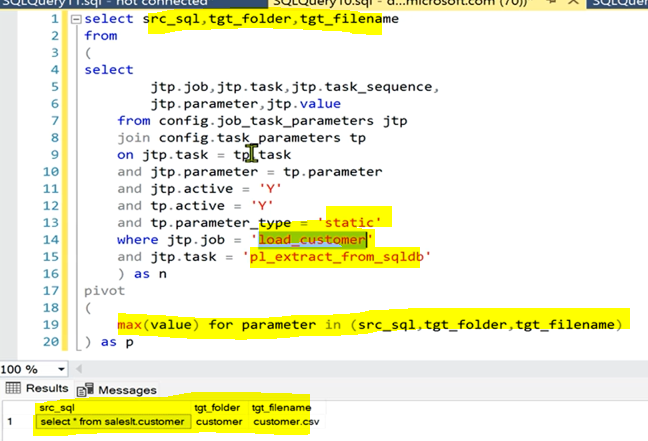
We have to create the store proc for dynamic implementation  
  
the functionality of the **control.sp\_task\_parameters** stored procedure step by step:

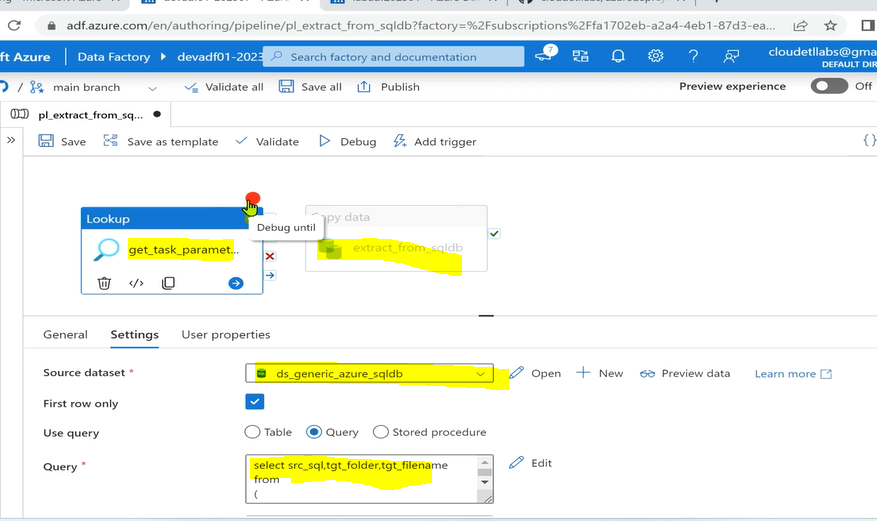
1. **Drop Procedure (IF EXISTS):**
   * Checks if the procedure **control.sp\_task\_parameters** already exists.
   * If it exists, it's dropped to ensure a fresh creation.
2. **Create Procedure:**
   * Defines a new stored procedure named **control.sp\_task\_parameters**.
   * Accepts input parameters:
     + **@job\_id**, **@task\_id**: IDs related to the job and task.
     + **@job**, **@task**: Names of the job and task.
     + **@task\_sequence**: Indicates the sequence of the task.
3. **Insert Statement:**
   * Populates the **control.task\_parameter\_control** table.
   * Selects and inserts parameter-value pairs from **config.job\_task\_parameters** and **config.task\_parameters**.
   * Conditions for insertion:
     + Matching parameters between job and task configurations.
     + Active status (**'Y'** indicates active).
     + Specific parameter type (**'static'**).
4. **Dynamic SQL Generation:**
   * Constructs dynamic SQL statements for selecting and pivoting data.
   * Constructs two variables:
     + **@sql**: Stores the dynamically generated SQL query.
     + **@columns**: Holds column names for the pivot operation.
5. **Column Selection for Pivot:**
   * Selects columns (parameters) for pivoting based on specific conditions from **control.task\_parameter\_control**.
   * Appends these columns to the **@columns** variable.
6. **Dynamic SQL Execution (Pivot Operation):**
   * Builds a SQL query using the **@columns** variable for pivoting the data.
   * Executes the dynamic SQL query using **sp\_executesql**.
   * The pivot operation transforms rows into columns based on the selected parameters.
7. **End of Procedure:**
   * Ends the stored procedure's definition.

In essence, this procedure:

* Inserts relevant parameter-value pairs into **control.task\_parameter\_control** based on specific conditions.
* Dynamically constructs and executes a SQL query to pivot and retrieve parameter values based on job, task, and sequence provided as input parameters.

It's designed to manage and retrieve task parameters associated with specific jobs and tasks, allowing for dynamic data retrieval based on different job and task configurations.





This is just for checking purpose whether pivot works or not, we have to create the below store proc and call the store proc and have to get the similar results

DROP PROCEDURE IF EXISTS control.sp\_task\_parameters;

GO

CREATE PROCEDURE control.sp\_task\_parameters

(

@job\_id VARCHAR(4000),

@task\_id VARCHAR(4000),

@job VARCHAR(4000),

@task VARCHAR(4000),

@task\_sequence INT

)

AS

BEGIN

INSERT INTO control.task\_parameter\_control

(

job\_id,

task\_id,

job,

task,

task\_sequence,

parameter,

value

)

SELECT

@job\_id,

@task\_id,

@job,

@task,

@task\_sequence,

jtp.parameter,

jtp.value

FROM

config.job\_task\_parameters jtp

JOIN

config.task\_parameters tp ON jtp.task = tp.task

AND jtp.parameter = tp.parameter

AND jtp.active = 'Y'

AND tp.active = 'Y'

AND tp.parameter\_type = 'static'

WHERE

jtp.job = @job

AND jtp.task = @task

AND jtp.task\_sequence = @task\_sequence;

DECLARE @sql NVARCHAR(MAX);

DECLARE @columns NVARCHAR(MAX);

SET @sql = N'';

SET @columns = N'';

SELECT @columns += N', ' + QUOTENAME(parameter)

FROM

(

SELECT parameter

FROM control.task\_parameter\_control tpc

WHERE tpc.job = @job

AND tpc.task = @task

AND tpc.task\_sequence = @task\_sequence

AND tpc.job\_id = @job\_id

AND tpc.task\_id = @task\_id

) AS N;

SET @sql = N'

SELECT ' + STUFF(@columns, 1, 1, '') + '

FROM

(

SELECT parameter, value

FROM control.task\_parameter\_control tpc

WHERE tpc.job = @j\_name

AND tpc.task = @t\_name

AND tpc.job\_id = @j\_id

AND tpc.task\_id = @t\_id

AND tpc.task\_sequence = @t\_sequence

) AS N

PIVOT

(

MAX(value) FOR parameter IN (' + STUFF(@columns, 1, 1, '') + ')

) AS P;';

EXEC sp\_executesql @sql,

N'@j\_name NVARCHAR(4000),

@t\_name NVARCHAR(4000),

@j\_id NVARCHAR(4000),

@t\_id NVARCHAR(4000),

@t\_sequence NVARCHAR(4000)',

@job,

@task,

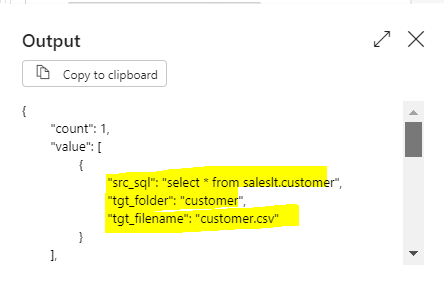
@job\_id,

@task\_id,

@task\_sequence;

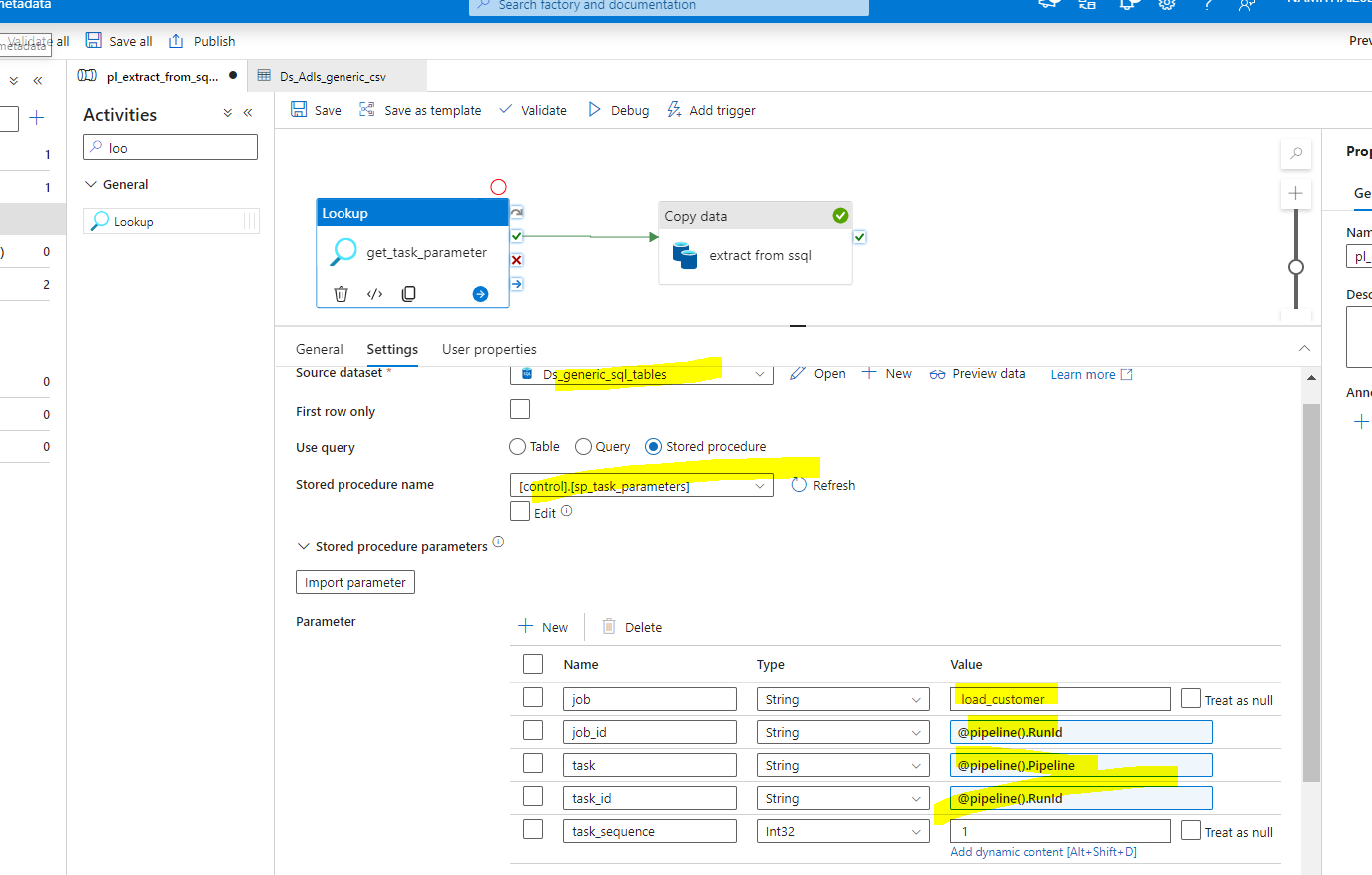
END

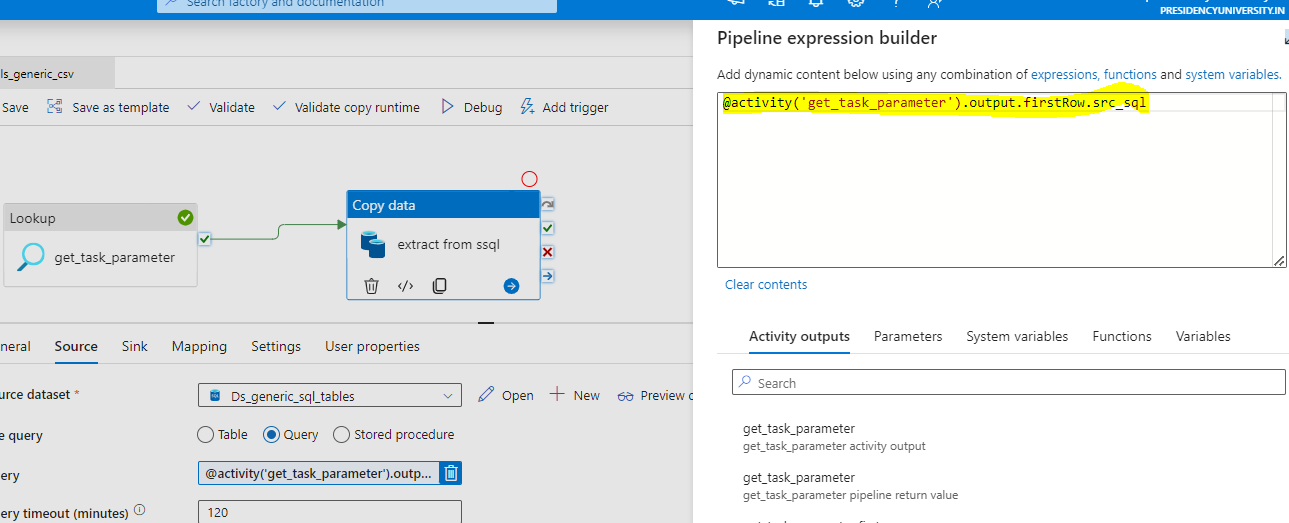
Note🡪 for now we will hardcode the job name and take taskId and job\_id same, while orschtration we will change the job\_id



Output of store proc lookup we which was directly passed before

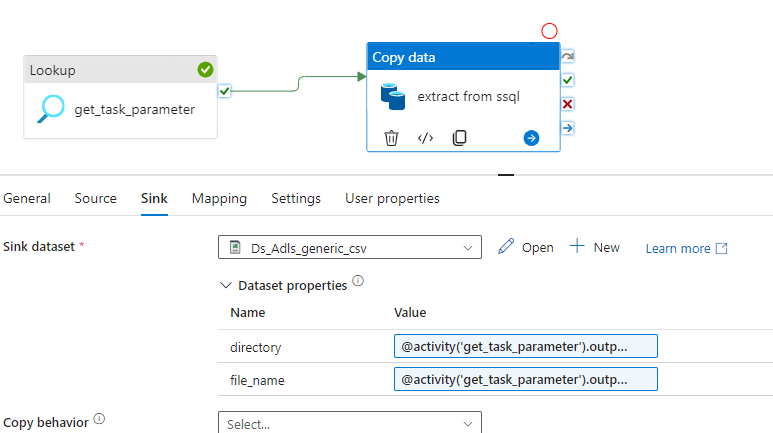
(take firstrow in lookup missed it)

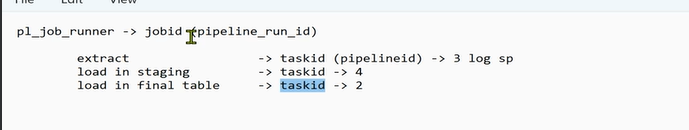




@activity('get\_task\_parameter').output.firstRow.tgt\_folder

@activity('get\_task\_parameter').output.firstRow.tgt\_filename





One jobId and different tasked based on task, we extract from sp

🡪now we will add the sp for logging purpose   
-- Check if the procedure exists; if so, drop it

DROP PROCEDURE IF EXISTS control.sp\_job\_task\_log;

GO

-- Create the procedure 'sp\_job\_task\_log' in the 'control' schema

CREATE PROCEDURE control.sp\_job\_task\_log

(

@job\_id VARCHAR(4000),

@task\_id VARCHAR(4000),

@job VARCHAR(4000),

@task VARCHAR(4000),

@task\_sequence INT,

@job\_status VARCHAR(4000),

@task\_status VARCHAR(4000),

@status\_description VARCHAR(4000)

)

AS

BEGIN

-- Insert data into the 'log.job\_task\_log' table

INSERT INTO log.job\_task\_log

(

job\_id,

task\_id,

job,

task,

task\_sequence,

job\_status,

task\_status,

status\_description

)

VALUES

(

@job\_id,

@task\_id,

@job,

@task,

@task\_sequence,

@job\_status,

@task\_status,

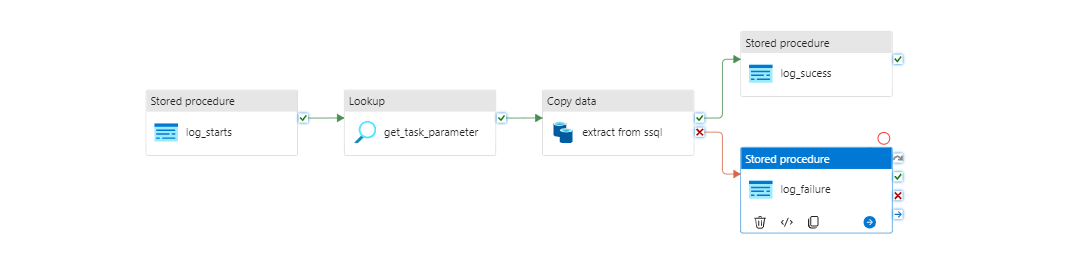
@status\_description

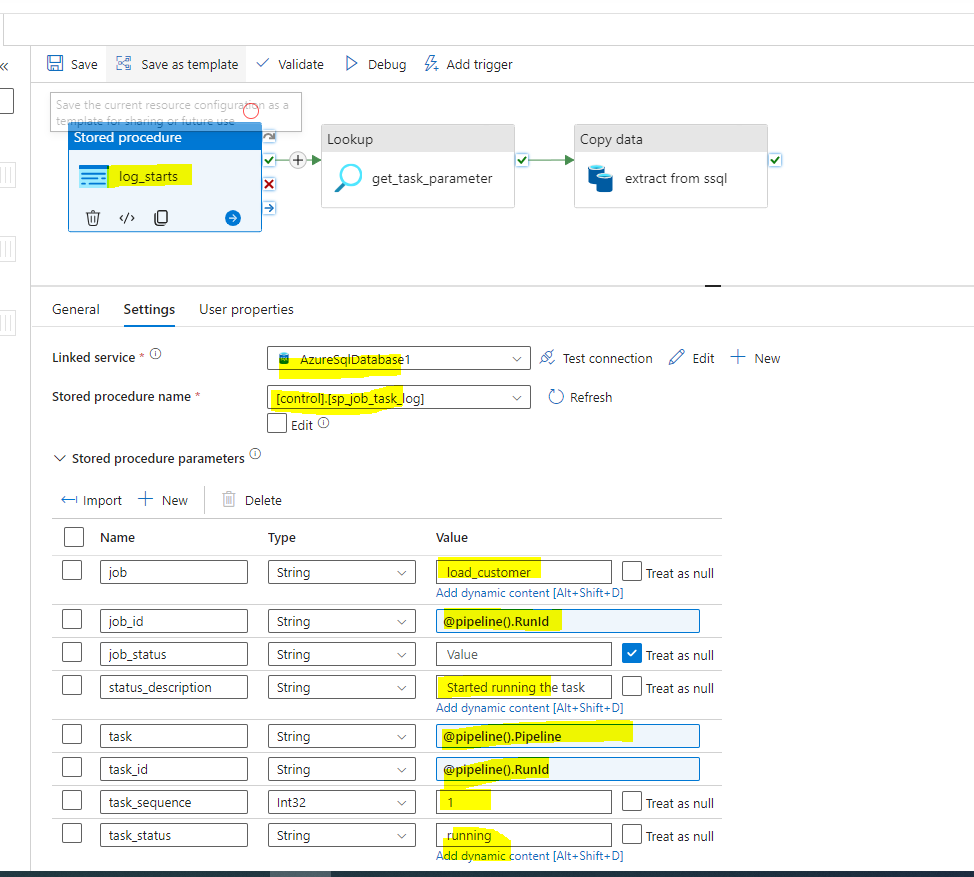
);

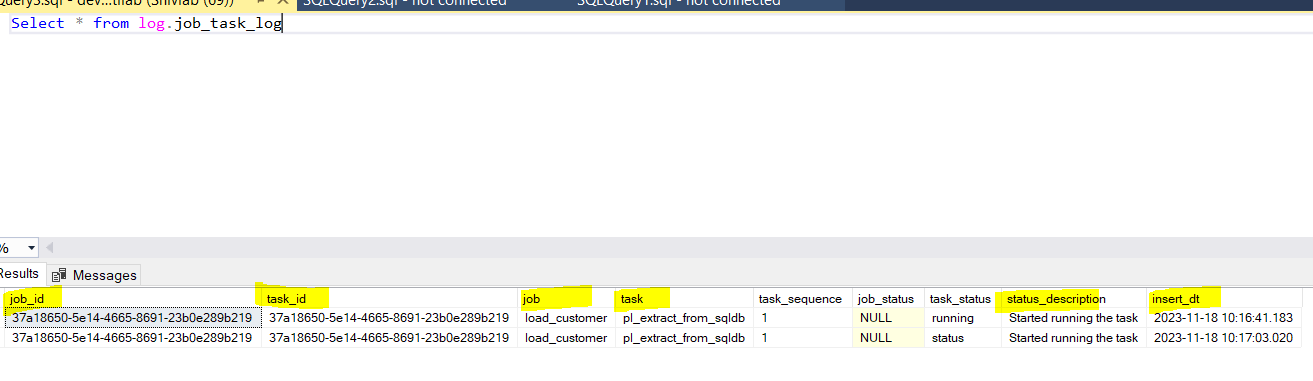
END

GO

🡪add the start log and end log as store proc activity in pipeline, for now few values are hardcode in further these are beginning changed🡪 just clone store proc and add them add end for success and failure, change the status respective in parameter







🡪next we re going to excute the job control store proc

-- Check if the procedure exists; if so, drop it

DROP PROCEDURE IF EXISTS control.sp\_update\_job\_control;

GO

-- Create the procedure 'sp\_update\_job\_control' in the 'control' schema

CREATE PROCEDURE control.sp\_update\_job\_control

(

@job\_id VARCHAR(4000),

@job VARCHAR(4000),

@task\_id VARCHAR(4000),

@task VARCHAR(4000),

@task\_status VARCHAR(4000)

)

AS

BEGIN

-- Declare variables

DECLARE @job\_dt DATE, @error VARCHAR(4000)

-- Set @job\_dt variable to the current date and time

SET @job\_dt = GETDATE()

-- Update the 'control.job\_task\_control' table

UPDATE control.job\_task\_control

SET

task\_id = @task\_id,

task\_status = @task\_status,

update\_dt = GETDATE()

WHERE

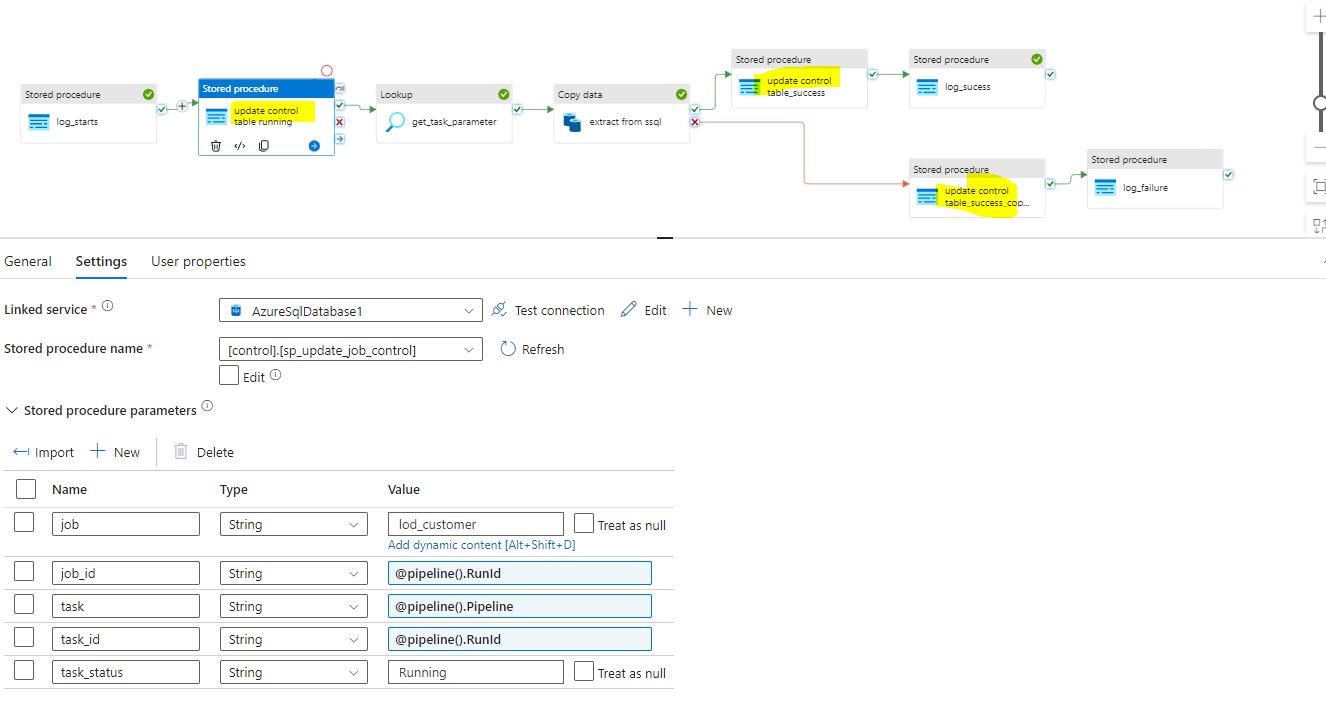
job = @job

AND job\_id = @job\_id

AND task = @task

END

GO



### 1. ****Extract Task Implementation****

* This procedure focuses on extracting data from the SQL database.
* Initially, it might have involved hardcoding parameters, but it was later made metadata-driven.
* It incorporates tasks such as querying the database based on dynamic parameters, extracting necessary data, and potentially transforming or preparing it for further processing.

### 2. ****Logging Procedure****

* The logging procedure is responsible for capturing and recording activities within the pipeline.
* It's designed to log specific details such as:
  + Job ID: Identification of the main job being executed.
  + Task ID: Identification of individual tasks within the job.
  + Job/Task Name: Descriptive name of the job or task being executed.
  + Sequence: Indicates the sequence or order of the task within the job.
  + Status: Captures the status of the job or task (e.g., running, success, failure).
  + Status Description: Additional information or description related to the status.
  + Insert Date: Automatically records the timestamp of the log entry.

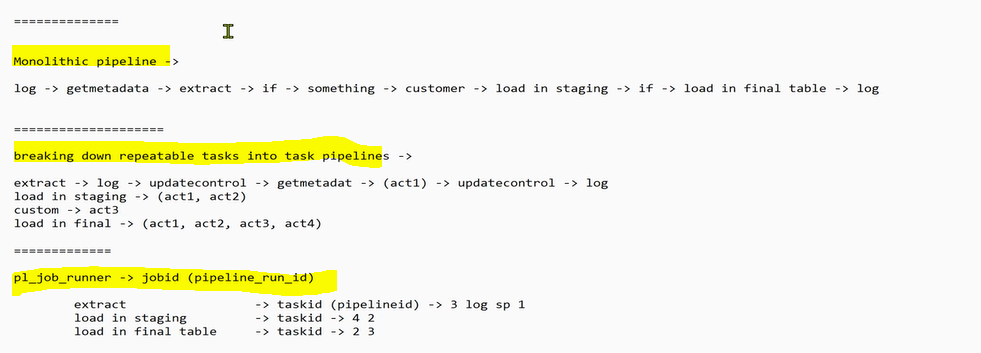
### 3. ****Control Table Procedures****

* **Init Job Control** Procedure:
  + This procedure is intended to initialize control table entries for the main job and its associated tasks.
  + It might involve inserting initial entries with status as 'Scheduled' or 'Running' for each task and the main job.
  + The initialization process ensures a starting point to track the status and progress of each job and task.
* **Update Control Table** Procedures:
  + These procedures are responsible for updating the control table entries during the pipeline execution lifecycle.
  + They update the control table with the current status of the job or task (e.g., 'Running', 'Success', 'Failure').
  + The updates are triggered at key stages, such as the start and completion of each task or the entire job.

### 4. ****Future Job Runner Procedures (Planned)****

* These procedures will likely be developed as part of the Job Runner pipeline.
* They might handle the initialization of control table entries for jobs and tasks within the pipeline execution flow.
* These procedures could dynamically update control table entries as the pipeline progresses through different stages.

Each procedure serves a specific purpose in managing the execution, tracking, and logging of activities within the SQL DB Pipeline, ensuring better control, monitoring, and traceability of the pipeline's execution flow.

  
here is the idea of design,

**Metadata-Driven Framework and Design Challenges:**

* **Initial Development Approach:** The process commenced by constructing an extract pipeline without a comprehensive framework. Initial steps involved hardcoding values, but later realized the necessity of a metadata-driven approach for scalability and reusability.
* **Monolithic Pipeline Concerns:** There was an emphasis on the drawbacks of creating monolithic pipelines where all activities are stacked sequentially. This method quickly becomes complex, especially when tasks vary across different data loads.
* **Addressing Design Challenges:** Stress was laid on the importance of addressing design challenges upfront to foresee potential complexities during development and subsequent maintenance phases.

**Task Pipelines and Control Framework Significance:**

* **Task Pipeline Strategy:** The proposed solution involved breaking down tasks into smaller, reusable task pipelines. Each task pipeline represents a work unit that can be composed, reused, or repeated across multiple jobs.
* **Control Framework:** Introduced the significance of a control framework to efficiently manage ETL processes. This framework assists in sequencing tasks, avoiding duplicates, and enabling status reporting for jobs.

**Development Progress and Critical Steps:**

* **Individual Task Pipelines:** Demonstrated the process of developing individual task pipelines (such as extract, load, custom tasks) and highlighted the need for a job runner to link these pipelines together.
* **Parameter Passing:** Emphasized the importance of passing common parameters from the job runner to individual task pipelines for consistency and manageability.
* **Incremental Loading:** Mentioned the pending implementation of the job task watermark, crucial for incremental loading. This implementation prevents re-extraction of previously loaded data.
* **Upcoming Steps:** Future steps involve finalizing the loading staging pipeline, completing the job task watermark implementation, and ultimately building the job runner to orchestrate the task pipelines.

**Focus and Conclusion:**

The overarching focus remains on designing a scalable, reusable, and maintainable framework for managing ETL processes effectively. The discussion highlighted the need for a structured approach to accommodate varying tasks and enhance overall manageability and efficiency.

Next step is incremental load

### Overall Strategy

1. **Full Load vs. Incremental Load:**
   * Currently performing a full load from SQL DB, fetching all records each time.
   * Want to switch to an incremental load to retrieve only new or modified records since the last run.
2. **Timestamp Column:**
   * Tables must contain a timestamp column (**modified date** in this case) for tracking changes.
3. **Job Watermark Table:**
   * Maintaining a table (**watermark table**) with the last run date/time for each job/task.
   * Using this table to identify records modified/inserted after the last run.

### Proposed Implementation Steps

1. **Stored Procedure for Fetching Last Run Date:**
   * A stored procedure retrieves the last run date/time based on the job and task.
   * If no record found (first run), sets the last run date to an old value (**1900-01-01**).
2. **Stored Procedure for Updating Last Run Date:**
   * Another stored procedure updates the last run date in the watermark table.
   * If no record exists, it inserts a new record with the current last run date.
3. **Pipeline Modifications:**
   * Before fetching tasks, add a stored procedure activity to retrieve the last run date for the current job/task.
   * Use the fetched last run date to modify the SQL query in the extract activity to fetch only newer records.
4. **Update Watermark Table:**
   * After completing the extraction, update the last run date in the watermark table using the **updateLastRunDate** stored procedure.

### Considerations

* **Dynamic Queries:**
  + The SQL query for extraction needs to be dynamically modified to include the condition based on the last run date.
* **Generic Approach:**
  + For different timestamp columns in different tables, additional logic might be needed.
* **Handling First Run:**
  + For the very first run, as there's no record in the watermark table, it defaults to a full load.

### Suggestions

* **Error Handling:** Implement error handling in case the procedures fail or encounter unexpected issues.
* **Logging:** Consider logging for better traceability and debugging purposes.

here's a step-by-step breakdown of the ETL (Extract, Transform, Load) pipeline implementation discussed:

### Initial Setup:

1. **Database Preparation:**
   * Ensure source and target databases are appropriately configured.
   * Confirm tables have necessary columns (like timestamp for tracking modifications).

### Incremental Load Implementation:

1. **Lookup Activity:**
   * Create a lookup activity to fetch the last run timestamp using a stored procedure.
   * Configure the stored procedure to return the last run date based on job and task parameters.
2. **SQL Query Generation:**
   * Construct SQL queries dynamically using the fetched last run timestamp.
   * Update the query to retrieve only new or modified records since the last run.
3. **Update Last Run Timestamp:**
   * Develop a stored procedure to update the last run date in a watermark table.
   * Trigger this procedure only upon successful completion of the pipeline run.

### Full Load Implementation:

1. **Extract Activity Setup:**
   * Configure the extract activity to fetch all records from the source table without conditions.

### Error Handling and Logging:

1. **Error Handling Mechanisms:**
   * Implement error handling within the pipeline to manage and handle failures gracefully.
2. **Logging and Monitoring:**
   * Set up logging mechanisms to track pipeline execution details, errors, and warnings.
   * Implement monitoring tools to track the pipeline's performance and status.

### Testing and Validation:

1. **Validation Testing:**
   * Execute the pipeline to validate the extraction process and verify data consistency.
   * Test error scenarios to ensure error handling mechanisms work effectively.

### Additional Considerations:

1. **Optimization:**
   * Optimize the pipeline by merging multiple activities where possible to enhance performance.
   * Tune activities to minimize queuing and maximize efficiency.
2. **Timezone Handling:**
   * Ensure consistency in timestamp handling if dealing with databases and pipelines in different time zones.
3. **Staging and Loading:**
   * Plan for staging and loading processes to move the extracted data into staging and final tables using appropriate operations (e.g., truncate and load, merge).

Implementing these steps systematically ensures the effective execution of both incremental and full load strategies within the ETL pipeline, enabling efficient data extraction and management.

Here is the store proc used

-- Drop the procedure if it already exists to avoid conflicts

DROP PROCEDURE IF EXISTS control.sp\_get\_last\_run\_date;

GO

-- Create a new stored procedure to retrieve the last run date based on job and task

CREATE PROCEDURE control.sp\_get\_last\_run\_date

(

@job VARCHAR(4000), -- Parameter: Job name

@task VARCHAR(4000) -- Parameter: Task name

)

AS

BEGIN

-- Declare a variable to hold the last run date

DECLARE @last\_run\_date DATETIME;

-- Retrieve the last run date from the job\_task\_watermark table based on job and task

SELECT @last\_run\_date = last\_run\_date

FROM control.job\_task\_watermark

WHERE job = @job

AND task = @task;

-- If last\_run\_date is empty or null, set it to a default date ('1900-01-01')

IF @last\_run\_date = '' OR @last\_run\_date IS NULL

SET @last\_run\_date = '1900-01-01';

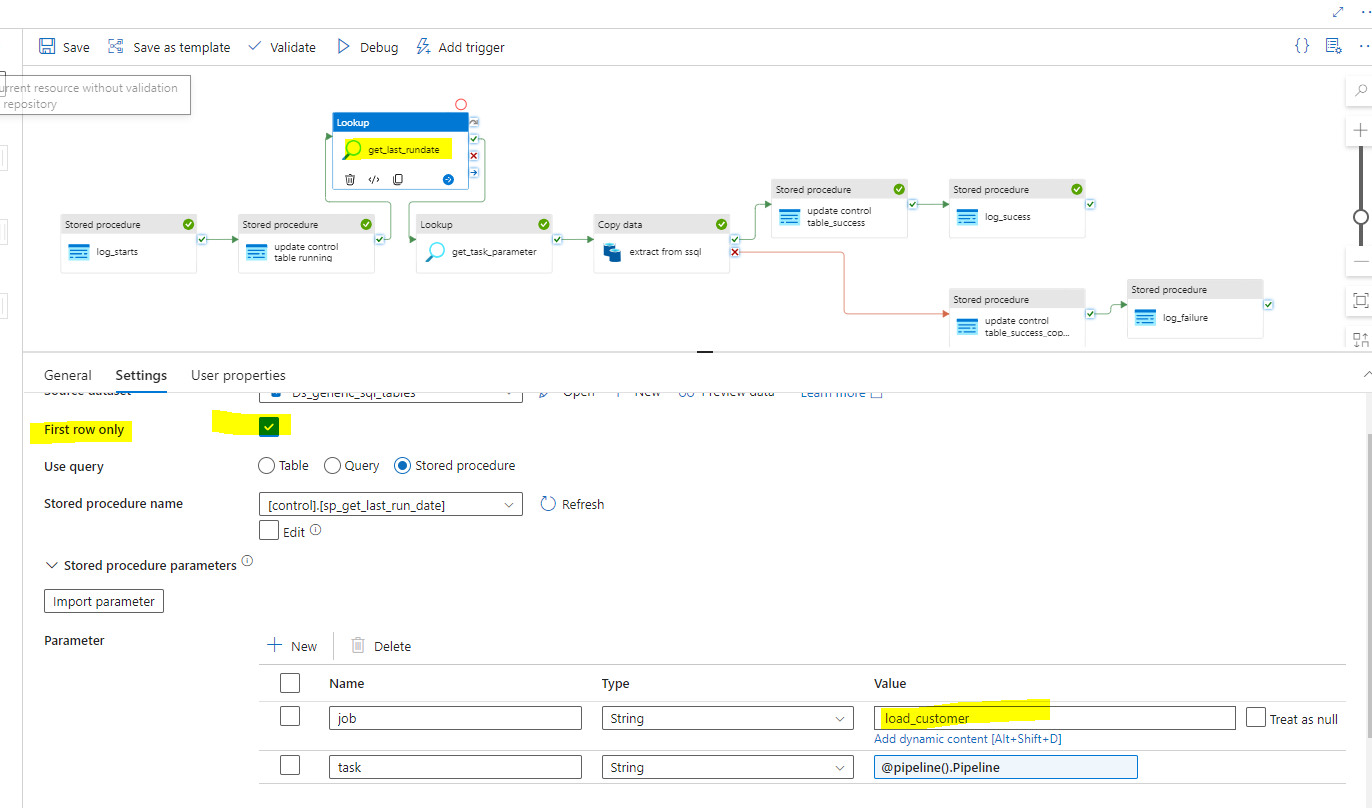
-- Return the last\_run\_date in a specific format (yyyy-mm-dd hh:mi:ss.mmm)

SELECT CONVERT(VARCHAR, @last\_run\_date, 121) AS last\_run\_date;

END

GO

This script creates a stored procedure **sp\_get\_last\_run\_date** that takes **@job** and **@task** parameters to fetch the last run date from the **job\_task\_watermark** table. It ensures that if no date is found, it defaults to '1900-01-01'. The **CONVERT** function formats the retrieved date to a specific style (yyyy-mm-dd hh:mi:ss.mmm) before returning it.



Go to copy data activity and change the dynamic activity

@concat(activity('get\_task\_parameter').output.firstRow.src\_sql, ' where ModifiedDate > ''', activity('get\_last\_rundate').output.firstRow.last\_run\_date, '''')

Now add one more store proc for update last run date

-- Drop the procedure if it already exists to avoid conflicts

DROP PROCEDURE IF EXISTS control.sp\_update\_last\_run\_date;

GO

-- Create a new stored procedure to update the last run date based on job, task, and provided date

CREATE PROCEDURE control.sp\_update\_last\_run\_date

(

@job VARCHAR(4000), -- Parameter: Job name

@task VARCHAR(4000), -- Parameter: Task name

@last\_run\_date DATETIME -- Parameter: Last run date to be updated

)

AS

BEGIN

-- Update the job\_task\_watermark table with the provided last run date and update date

UPDATE control.job\_task\_watermark

SET last\_run\_date = @last\_run\_date,

update\_dt = GETDATE() -- Record the update timestamp

WHERE job = @job

AND task = @task;

-- If no rows were affected by the update, insert a new record with job, task, and last run date

IF @@ROWCOUNT = 0

BEGIN

INSERT INTO control.job\_task\_watermark (job, task, last\_run\_date)

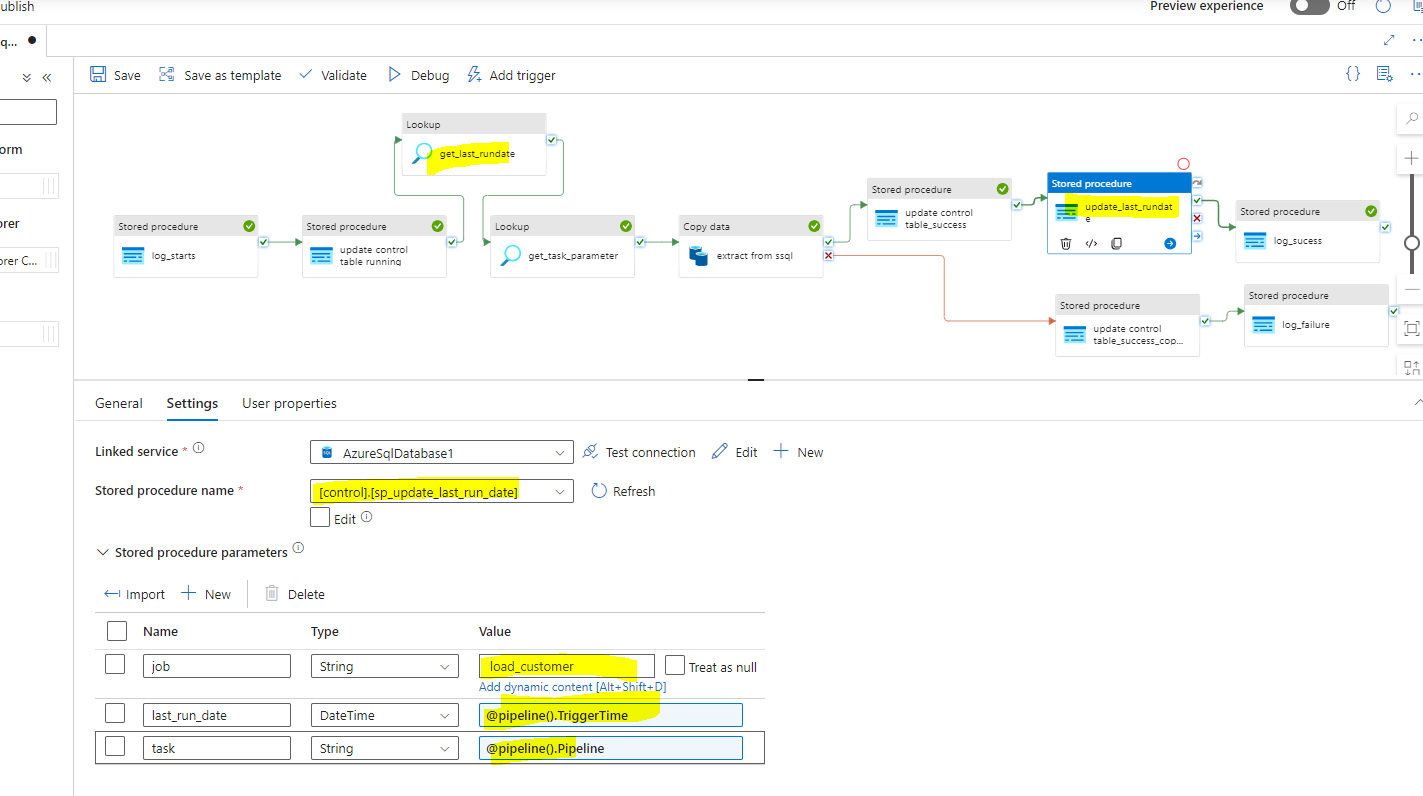
VALUES (@job, @task, @last\_run\_date);

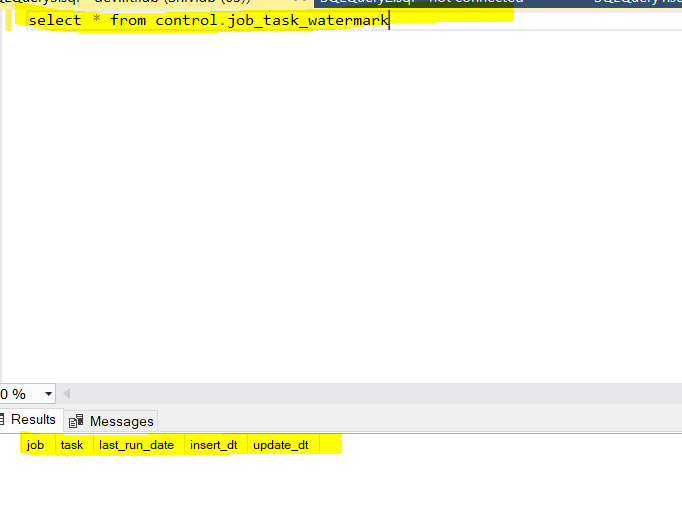
END

END

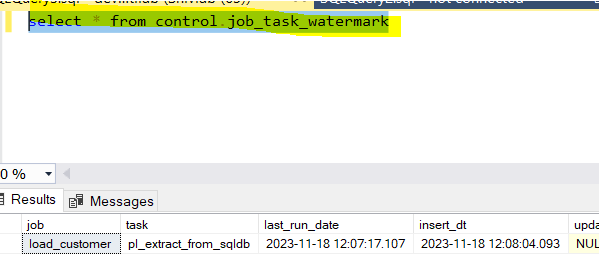
GO

This script creates a stored procedure **sp\_update\_last\_run\_date** that takes **@job**, **@task**, and **@last\_run\_date** parameters. It updates the **job\_task\_watermark** table with the provided **last\_run\_date** for the corresponding **job** and **task**. If no rows are affected by the update (indicating that there is no existing record for the job and task), it inserts a new record with these details.

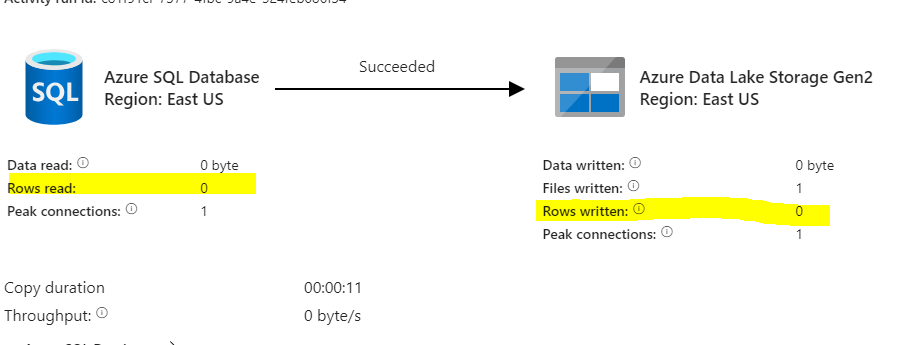




Before excuting the pipeline is above, after is below

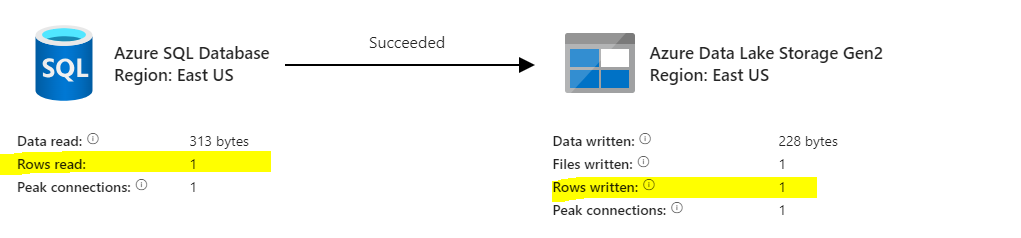


Now rerun pipeline and check that there should no records should be moved, and in watermark table lastrundate gets updated



Add one new record/update in table and check its incremental

update saleslt.customer set modifieddate = getdate() where customerid=1



Now for testing purpose of full load in other enviormnent say testing, all u have to do is delte the watermark table and excute pipeline again it will do full load

Point to be noted, while using watermark table is that take care of time format.

🡪next pipeline is to load data from landing to stage, so excute the following scripts for the same

Lab Script - Metadata Table Configurations

1. Config.Jobs

===========

insert into config.jobs (job,active)

values ('load\_customer','Y')

select \* from config.jobs

2. Config.Tasks

===========

insert into config.tasks(task,active)

values ('pl\_load\_into\_stg','Y')

select \* from config.tasks

3. Config.Job\_Tasks

===============

insert into config.job\_tasks(job,task,task\_sequence,active)

values ('load\_customer','pl\_load\_into\_stg',2,'Y')

select \* from config.job\_tasks

4. Config.Task\_Parameters

====================

insert into config.task\_parameters(task,parameter,parameter\_type,active)

values ('pl\_load\_into\_stg','src\_path','static','Y')

insert into config.task\_parameters(task,parameter,parameter\_type,active)

values ('pl\_load\_into\_stg','tgt\_stg\_table','static','Y')

select \* from config.task\_parameters

5. Config.Job\_Task\_Parameters

========================

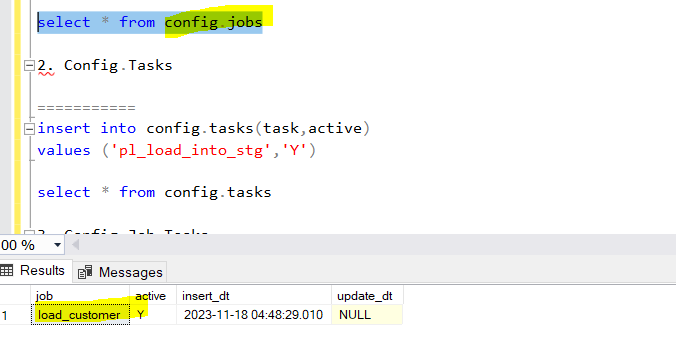
insert into config.job\_task\_parameters(job,task,parameter,value,active,task\_sequence)

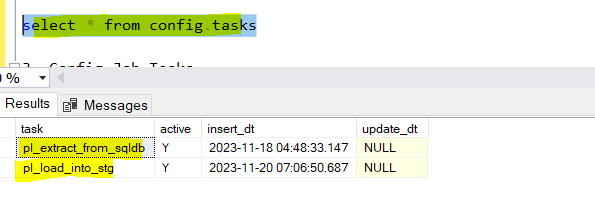
values ('load\_customer','pl\_load\_into\_stg','src\_path','landing/customer/customer.csv','Y',2)

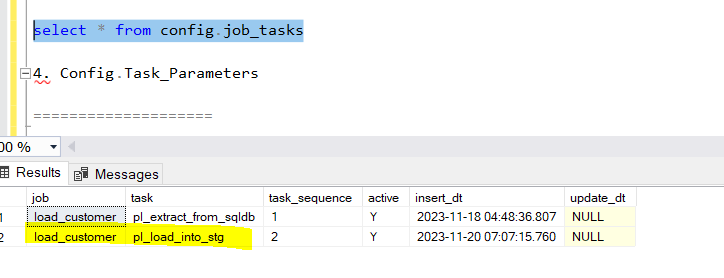
insert into config.job\_task\_parameters(job,task,parameter,value,active,task\_sequence)

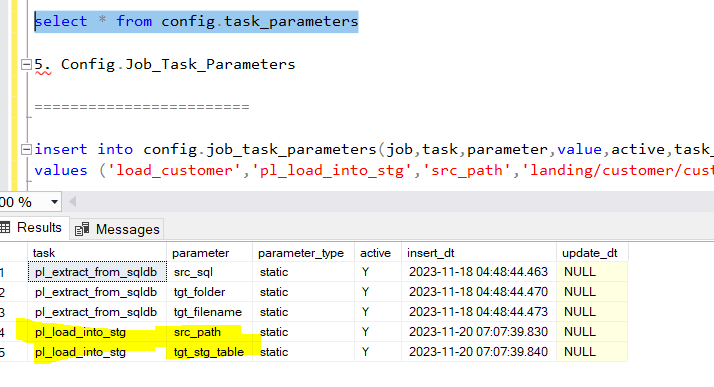
values ('load\_customer','pl\_load\_into\_stg','tgt\_stg\_table','saleslt.customer\_target\_Stg','Y',2)

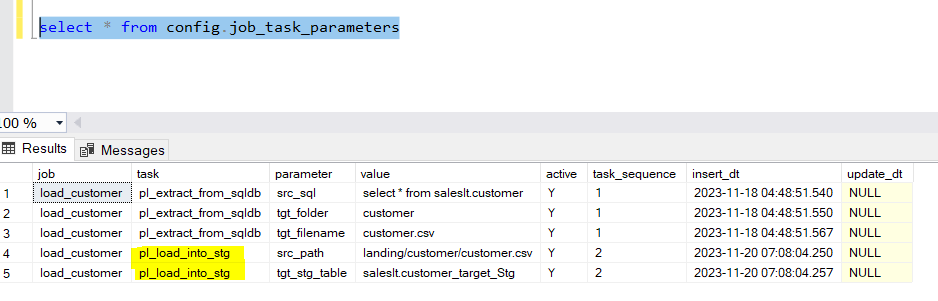
select \* from config.job\_task\_parameters











Staging table

-- Drop the table if it already exists

IF OBJECT\_ID('saleslt.Customer\_Target\_Stg', 'U') IS NOT NULL

DROP TABLE saleslt.Customer\_Target\_Stg;

GO

-- Create the Customer\_Target\_Stg table

CREATE TABLE saleslt.Customer\_Target\_Stg (

CustomerID varchar(4000),

NameStyle varchar(4000),

Title varchar(4000),

FirstName varchar(4000),

MiddleName varchar(4000),

LastName varchar(4000),

Suffix varchar(4000),

CompanyName varchar(4000),

SalesPerson varchar(4000),

EmailAddress varchar(4000),

Phone varchar(4000),

PasswordHash varchar(4000),

PasswordSalt varchar(4000),

rowguid varchar(4000),

ModifiedDate varchar(4000)

);

GO

This script is a series of SQL commands aimed at configuring a database for bulk loads, likely involving Azure Blob Storage as an external data source.

-- Drop the existing master key if it exists

DROP MASTER KEY;

-- Create a new master key encrypted by a strong password

CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'Svk@7019368730';

-- Create a scoped credential for Azure Blob Storage access

CREATE DATABASE SCOPED CREDENTIAL DevAdlsContainer

WITH IDENTITY = 'SHARED ACCESS SIGNATURE',

SECRET = '?sv=2022-11-02&ss=bfqt&srt=sco&sp=rwdlacupyx&se=2023-11-30T15:29:20Z&st=2023-11-20T07:29:20Z&spr=https&sig=bOAyqI1VAn81YsqZ67hoWBC5d9AnY1uGSDzS35n7ptg%3D';

This creates a scoped credential named **DevAdlsContainer**, likely used to authenticate against an Azure Blob Storage account using a Shared Access Signature (SAS) token.

-- Drop the existing external data source if it exists

DROP EXTERNAL DATA SOURCE DevStorage;

-- Create an external data source for Azure Blob Storage

CREATE EXTERNAL DATA SOURCE DevStorage

WITH (

TYPE = BLOB\_STORAGE,

LOCATION = 'https://devaaccount202318.blob.core.windows.net',

CREDENTIAL = DevAdlsContainer

);

This command creates an external data source named **DevStorage**, specifying it as a blob storage type and linking it to the **DevAdlsContainer** credential. It points to an Azure Blob Storage URL.

-- Grant necessary privileges to roles and users

ALTER ROLE db\_datawriter ADD MEMBER [dev-etl-sk];

ALTER ROLE db\_executor ADD MEMBER [dev-etl-sk];

GRANT SELECT, UPDATE, DELETE, INSERT, ALTER, CONTROL ON SCHEMA::SalesLt TO [dev-etl-sp];

GRANT ADMINISTER DATABASE BULK OPERATIONS TO [dev-etl-sp];

These commands grant permissions to the database roles and the specific user/group **[dev-etl-sp]**. It allows them to perform read, write, and schema modification operations (**SELECT**, **UPDATE**, **DELETE**, **INSERT**, **ALTER**, **CONTROL**) in the **SalesLt** schema and administers database bulk operations.

This script essentially sets up the necessary keys, credentials, and permissions to facilitate bulk loading operations from Azure Blob Storage into the database schema named **SalesLt**

To find user and give permission

USE database;

SELECT name, type\_desc, create\_date

FROM sys.database\_principals

WHERE type IN ('S', 'U', 'G') -- Filter for SQL users, Windows users, and Windows groups

ORDER BY type\_desc, name;

-- Drop the stored procedure if it exists

IF OBJECT\_ID('saleslt.sp\_load\_stg', 'P') IS NOT NULL

DROP PROCEDURE saleslt.sp\_load\_stg;

GO

-- Create the stored procedure sp\_load\_stg

CREATE PROCEDURE saleslt.sp\_load\_stg

(

@stgtablename varchar(4000),

@filepath varchar(4000)

)

AS

BEGIN

DECLARE @sql varchar(4000);

-- Truncate the target table

SET @sql = 'TRUNCATE TABLE ' + @stgtablename;

EXEC (@sql);

-- Perform bulk insert into the target table

SET @sql = 'BULK INSERT ' + @stgtablename +

' FROM ' + '''' + @filepath + '''' +

' WITH (' +

' DATA\_SOURCE = ''DevStorage'',' +

' DATAFILETYPE = ''char'',' +

' FIELDTERMINATOR = '','''',' +

' ROWTERMINATOR = ''\n''' +

')';

EXEC (@sql);

END

GO

-- You can try testing the SP manually using below script

delete from saleslt.customer\_target\_Stg

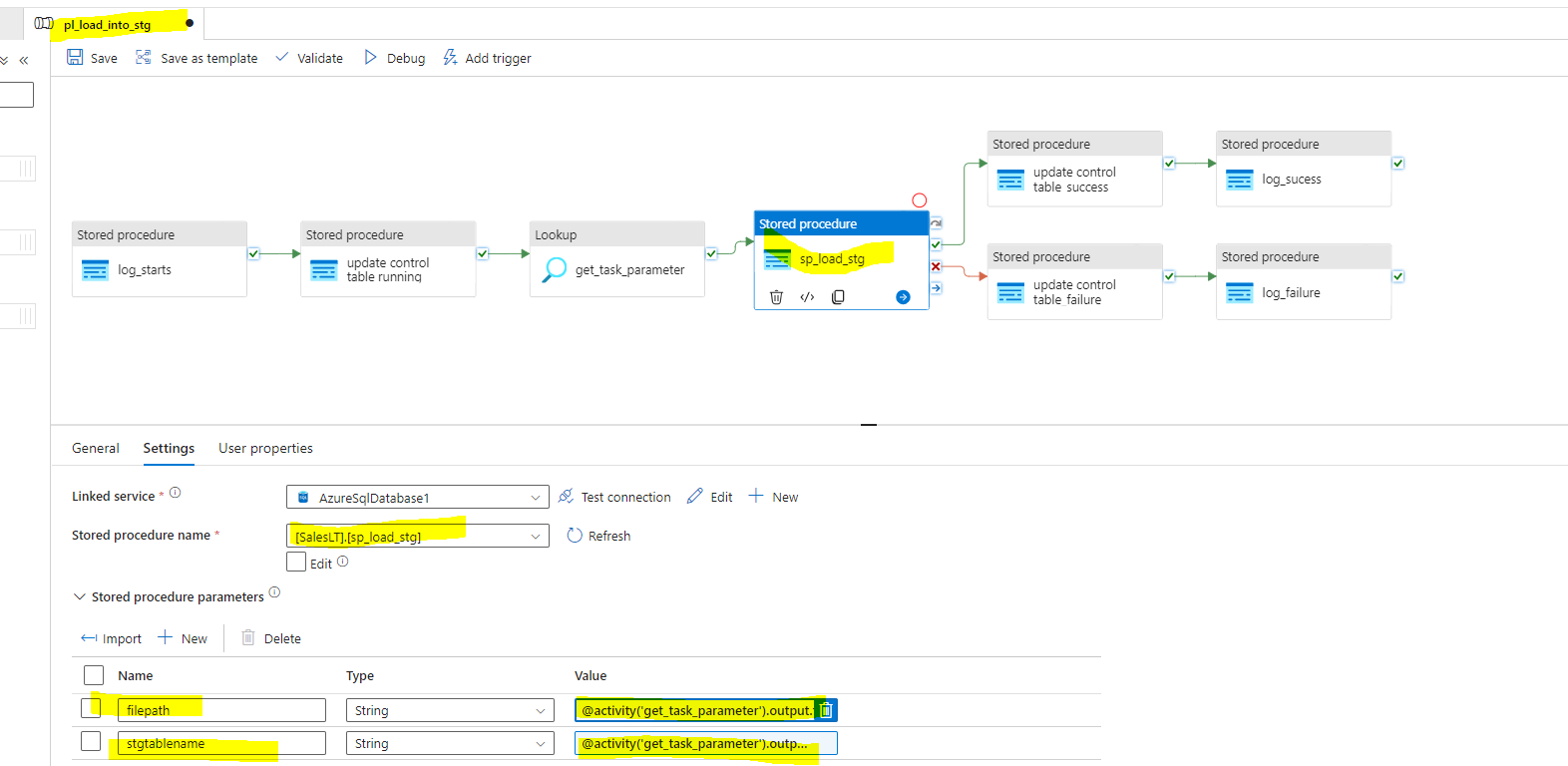
exec saleslt.sp\_load\_stg 'saleslt.customer\_target\_Stg', 'landing/customer/customer.csv'

select count(\*) from saleslt.customer\_target\_Stg

This script creates a stored procedure named **sp\_load\_stg** in the **saleslt** schema. This procedure takes in two parameters: **@stgtablename** (the target table name) and **@filepath** (the file path for bulk loading).

Inside the procedure, it truncates the target table specified by **@stgtablename** and then performs a bulk insert operation into that table from the file specified by **@filepath**, using the **BULK INSERT** SQL command. Adjust the parameters and options in the **BULK INSERT** statement as needed for your specific use case.

Now make a clone of first pipeline, delete the lastrun date sp activities and make changes in other activities for sequence 2



🡪Now third pipeline pl\_load\_into\_tgt

Lab Script - Metadata Table Configurations

1. Config.Jobs

==========

insert into config.jobs (job,active)

values ('load\_customer','Y')

2. Config.Tasks

===========

insert into config.tasks(task,active)

values ('pl\_load\_into\_tgt','Y')

3. Config.Job\_Tasks

===============

insert into config.job\_tasks(job,task,task\_sequence,active)

values ('load\_customer','pl\_load\_into\_tgt',3,'Y')

4. Config.Task\_Parameters

====================

insert into config.task\_parameters(task,parameter,parameter\_type,active)

values ('pl\_load\_into\_tgt','sp\_name','static','Y')

5. Config.Job\_Task\_Parameters

========================

insert into config.job\_task\_parameters(job,task,parameter,value,active,task\_sequence)

values ('load\_customer','pl\_load\_into\_tgt','sp\_name','saleslt.sp\_load\_customer','Y',3)

-- Drop the table if it already exists

IF OBJECT\_ID('saleslt.Customer\_Target', 'U') IS NOT NULL

DROP TABLE saleslt.Customer\_Target;

GO

-- Create the Customer\_Target table

CREATE TABLE saleslt.Customer\_Target (

[CustomerID] [varchar](4000),

[NameStyle] [varchar](4000),

[Title] [varchar](4000),

[FirstName] [varchar](4000),

[MiddleName] [varchar](4000),

[LastName] [varchar](4000),

[Suffix] [varchar](4000),

[CompanyName] [varchar](4000),

[SalesPerson] [varchar](4000),

[EmailAddress] [varchar](4000),

[Phone] [varchar](4000),

[PasswordHash] [varchar](4000),

[PasswordSalt] [varchar](4000),

[rowguid] [varchar](4000),

[ModifiedDate] [varchar](4000)

);

To implement SCD below sp is used

-- Drop the procedure if it already exists

IF OBJECT\_ID('saleslt.sp\_load\_customer', 'P') IS NOT NULL

DROP PROCEDURE saleslt.sp\_load\_customer;

GO

-- Create the sp\_load\_customer stored procedure

CREATE PROCEDURE saleslt.sp\_load\_customer

AS

BEGIN

MERGE [SalesLT].[Customer\_Target] AS target

USING [SalesLT].[Customer\_Target\_Stg] AS source

ON source.CustomerID = target.CustomerID

WHEN NOT MATCHED BY TARGET THEN

INSERT (

[CustomerID], [NameStyle], [Title], [FirstName], [MiddleName], [LastName],

[Suffix], [CompanyName], [SalesPerson], [EmailAddress], [Phone], [PasswordHash],

[PasswordSalt], [rowguid], [ModifiedDate]

)

VALUES (

Source.CustomerID, Source.NameStyle, Source.Title, Source.FirstName,

Source.MiddleName, Source.LastName, Source.Suffix, Source.CompanyName,

Source.SalesPerson, Source.EmailAddress, Source.Phone, Source.PasswordHash,

Source.PasswordSalt, Source.rowguid, Source.ModifiedDate

)

WHEN MATCHED THEN

UPDATE SET

Target.NameStyle = Source.NameStyle,

Target.Title = Source.Title,

Target.FirstName = Source.FirstName,

Target.MiddleName = Source.MiddleName,

Target.LastName = Source.LastName,

Target.Suffix = Source.Suffix,

Target.CompanyName = Source.CompanyName,

Target.SalesPerson = Source.SalesPerson,

Target.EmailAddress = Source.EmailAddress,

Target.Phone = Source.Phone,

Target.PasswordHash = Source.PasswordHash,

Target.PasswordSalt = Source.PasswordSalt,

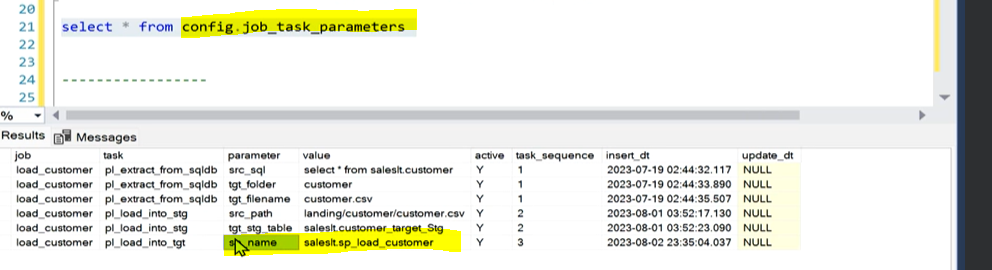
Target.rowguid = Source.rowguid,

Target.ModifiedDate = Source.ModifiedDate;

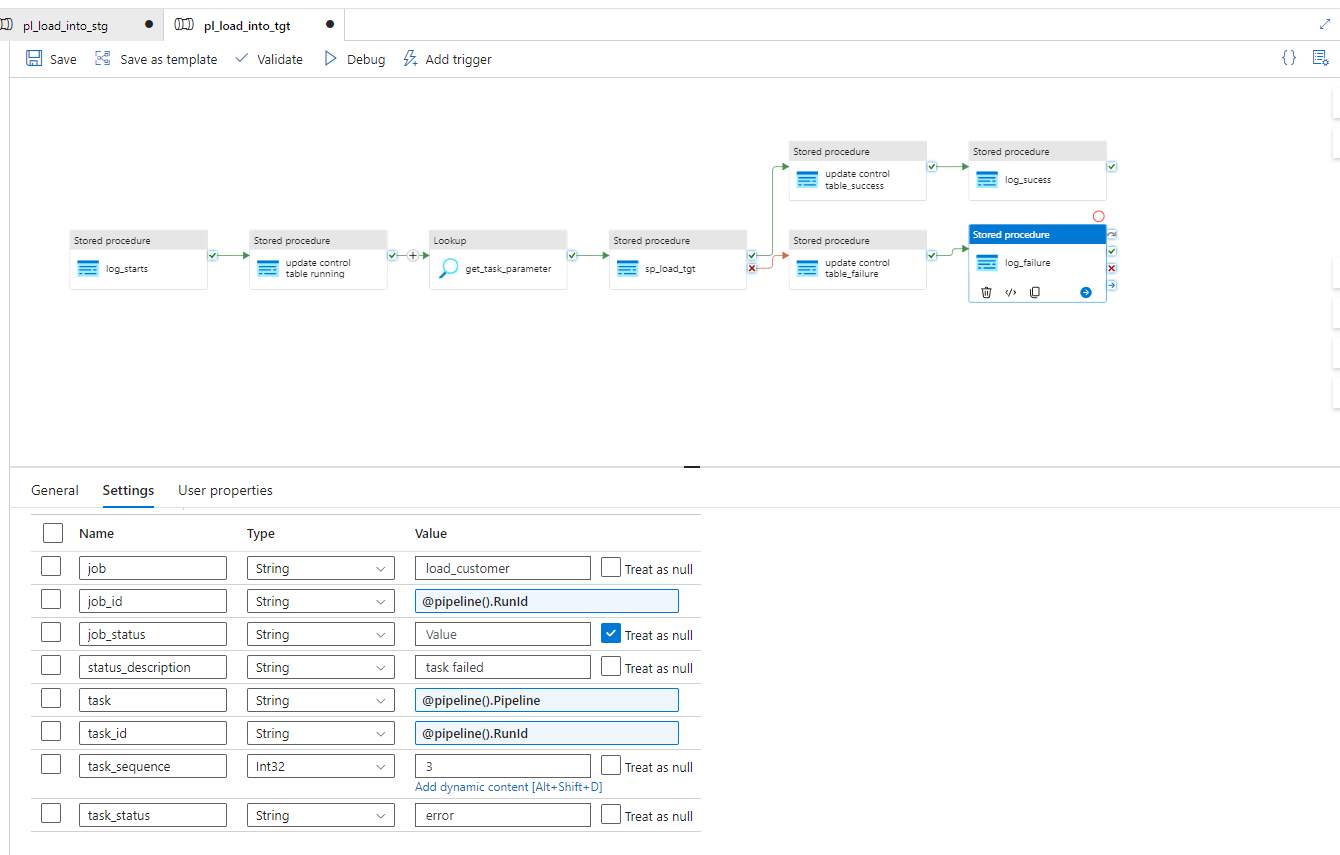
END

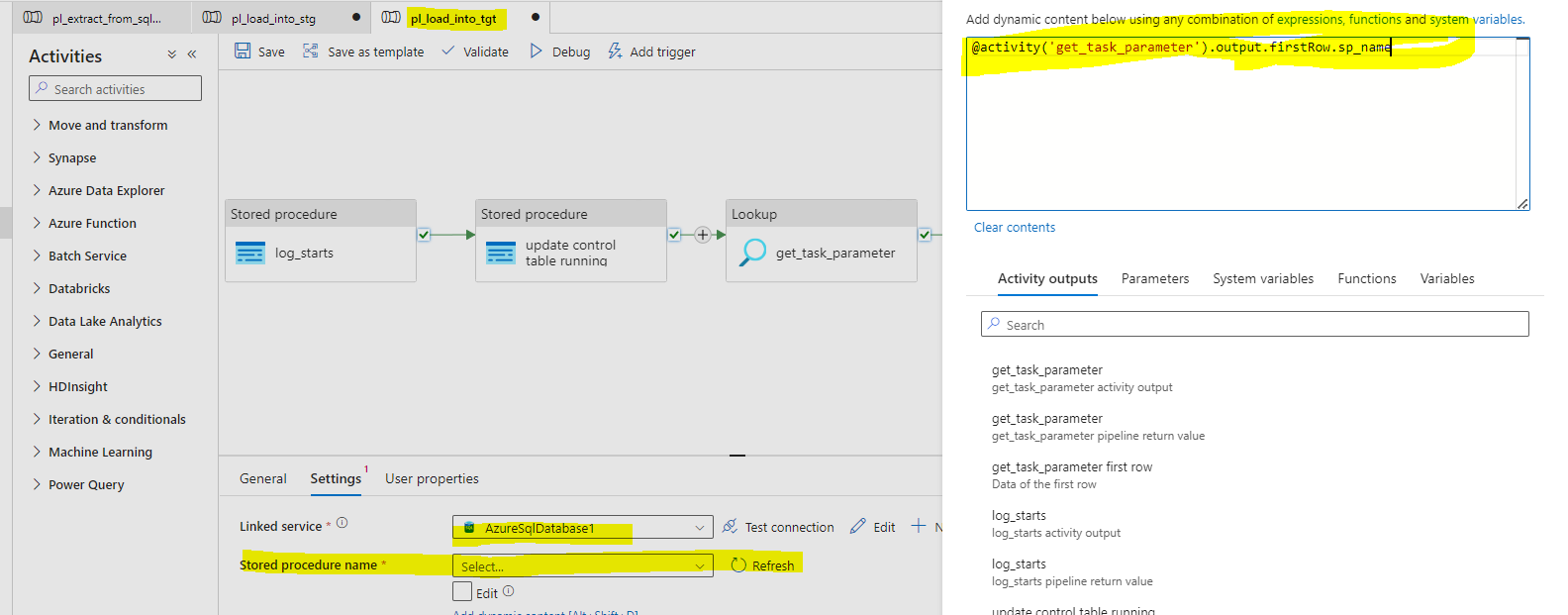
GO

Now this sp id for customer tble only , now in ADF we have to pass the store proc itself as parameter



This store proc don’t have any parameter





Next is to bulid orchestration

### Steps to Create the Job Orchestrator Pipeline:

1. **Input Parameter Handling**:
   * The orchestrator pipeline should accept an input parameter specifying the job name (e.g., "load customer").
2. **Retrieve Associated Tasks**:
   * Query the job task table to retrieve all tasks associated with the specified job.
   * Extract the task details such as pipeline names, sequence, and other relevant information for execution.
3. **Setup Control Values**:
   * For each task retrieved:
     + Determine the sequence of tasks to execute in the specified order.
     + Set up control values in the control table for each task before execution:
       - Insert records into the control table for each task with relevant job ID, task name, sequence, initial status as "Scheduled," and other necessary details.
4. **Iterative Task Execution**:
   * Implement a loop to iterate through the retrieved tasks based on their sequence.
   * For each task in the sequence:
     + Update the control table to mark the task as "Running" before executing the corresponding task pipeline.
     + Trigger the execution of the task pipeline (extract, load, etc.) associated with the task.
     + Capture the execution outcome (success or failure) of the task pipeline.
     + Update the status in the control table based on the outcome (e.g., "Success" or "Error").
     + Proceed to the next task in the sequence.
5. **Log Execution Details**:
   * Record relevant execution details in log tables or logs within your system, including start and end times, execution statuses, errors, and any necessary notes or metadata.
6. **Completion and Cleanup**:
   * After executing all tasks for the specified job:
     + Update the overall job status in your control or metadata tables.
     + Perform any necessary cleanup or post-execution tasks.

### Key Considerations:

* **Error Handling**: Implement robust error handling mechanisms to capture and manage failures during task execution.
* **Monitoring and Logging**: Ensure comprehensive logging to track the execution progress, success, or failure of each task pipeline.
* **Parameter Passing**: Pass necessary parameters or configurations from the orchestrator pipeline to the task pipelines for proper execution.
* **Validation**: Validate inputs, sequences, and control values to ensure correctness and consistency throughout the orchestration process.

Building an orchestrator pipeline involves intricate logic and control mechanisms to manage task execution. Make sure to test and validate the pipeline thoroughly before deploying it for production use.

Job\_runner\_pipeline

insert into config.environment(parameter, value)

values ('subscription', 'c16facec-26d0-4d53-89de-f460cfd5a370');

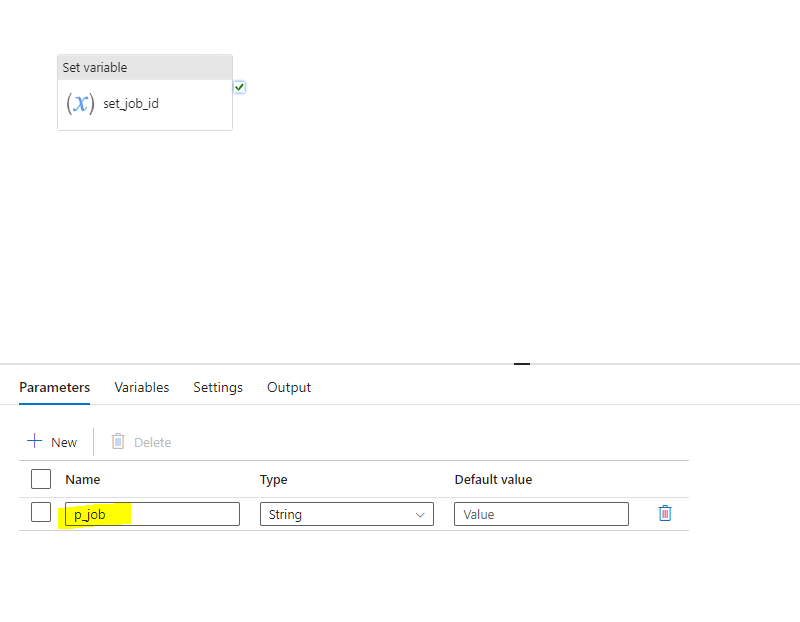
insert into config.environment(parameter, value)

values ('rg','Metadatalaab');

insert into config.environment(parameter, value)

values ('adf','Etlmetadata');

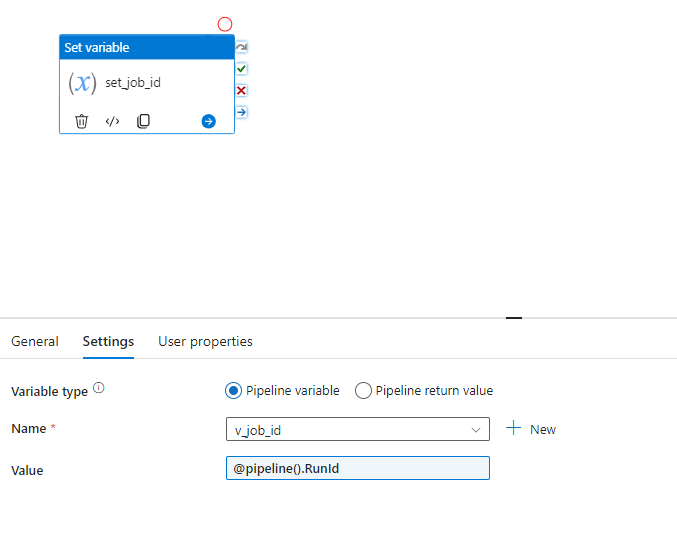
this is done so that whenever the environment is changed easily we can make changes.



Now whenever u trigger the pipeline, it will ask for job say we did in pipeline we say load\_customer

So this same pipeline I can use to trigger multiple jobs, it can be load, customer load, product

or any other table that I want to extract from our SQL.

Define variable and take that variable in job id   
IF OBJECT\_ID('control.sp\_get\_tasks', 'P') IS NOT NULL

DROP PROCEDURE control.sp\_get\_tasks;

GO

CREATE PROCEDURE control.sp\_get\_tasks

(

@job\_id varchar(4000),

@job varchar(4000)

)

AS

BEGIN

DECLARE @job\_date date, @error varchar(4000), @tasks int;

SET @job\_date = GETDATE();

/\* Consider adding logging within sp\_job\_task\_log \*/

IF NOT EXISTS (

SELECT job

FROM config.job\_tasks

WHERE job = @job

AND active = 'Y'

)

BEGIN

SET @error = 'No active tasks found for the job';

EXEC control.sp\_job\_task\_log @job\_id, '', @job, '', '', 'ERROR', '', @error;

THROW 50001, @error, 1;

END

SELECT job, task, task\_sequence

FROM config.job\_tasks

WHERE job = @job

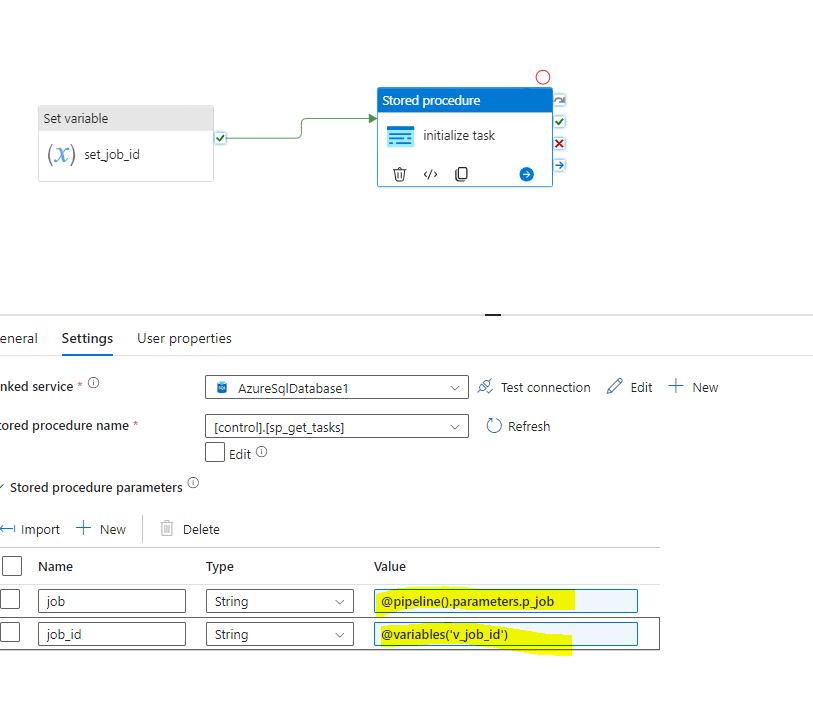
AND active = 'Y';

END

GO

This script creates a stored procedure **control.sp\_get\_tasks** that takes **@job\_id** and **@job** parameters. It retrieves active tasks associated with the specified job from the **config.job\_tasks** table. If no active tasks are found for the provided job, it logs an error using **sp\_job\_task\_log** and throws an exception.

Adjust the logic and error handling as needed for your specific requirements and ensure that the **sp\_job\_task\_log** procedure exists and handles logging appropriately



Thefollowing script creates a stored procedure **control.sp\_init\_job\_control** that initializes job task control for a specified job. It performs checks for job existence and status in the **config.jobs** and **control.job\_task\_control** tables, respectively. Then, it inserts task control entries based on active tasks from the **config.job\_tasks** table, setting their initial status as 'SCHEDULED'.

Modify the error handling and logic as needed to suit your specific requirements and ensure the stored procedure aligns with your overall job control and task scheduling process.

IF OBJECT\_ID('control.sp\_init\_job\_control', 'P') IS NOT NULL

DROP PROCEDURE control.sp\_init\_job\_control;

GO

CREATE PROCEDURE control.sp\_init\_job\_control

(

@job\_id varchar(4000),

@job varchar(4000)

)

AS

BEGIN

DECLARE @job\_dt date, @error varchar(4000);

SET @job\_dt = GETDATE();

IF NOT EXISTS (

SELECT 1

FROM config.jobs

WHERE job = @job

AND active = 'Y'

)

BEGIN

SET @error = 'Job ' + @job + ' is inactive in control.jobs table';

THROW 50001, @error, 1;

END

IF EXISTS (

SELECT 1

FROM control.job\_task\_control

WHERE job = @job

AND task\_status IN ('RUNNING', 'SCHEDULED')

)

BEGIN

SET @error = 'Previous job run for ' + @job + ' is not yet complete';

THROW 50002, @error, 1;

END

INSERT INTO control.job\_task\_control

(

job\_id,

task\_id,

job,

job\_dt,

task,

task\_sequence,

task\_status

)

SELECT

@job\_id,

'0',

@job,

@job\_dt,

task,

task\_sequence,

'SCHEDULED'

FROM config.job\_tasks

WHERE job = @job

AND active = 'Y'

ORDER BY task\_sequence;

END

GO

🡪this query in get\_enviorment\_variable lookup

Select \* from

(

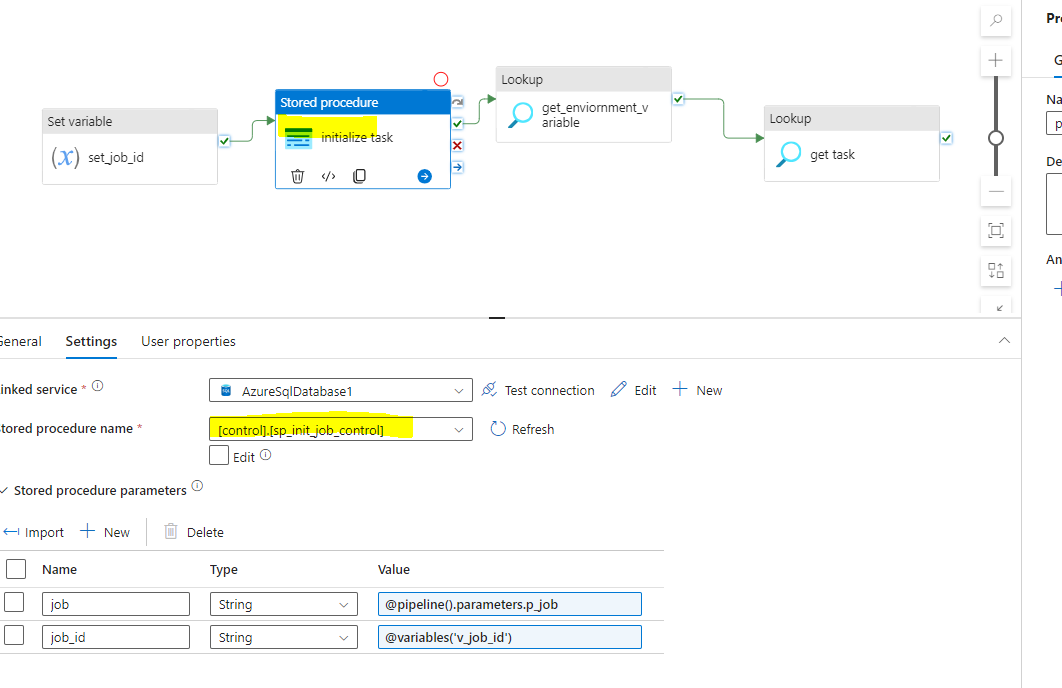
select parameter, value

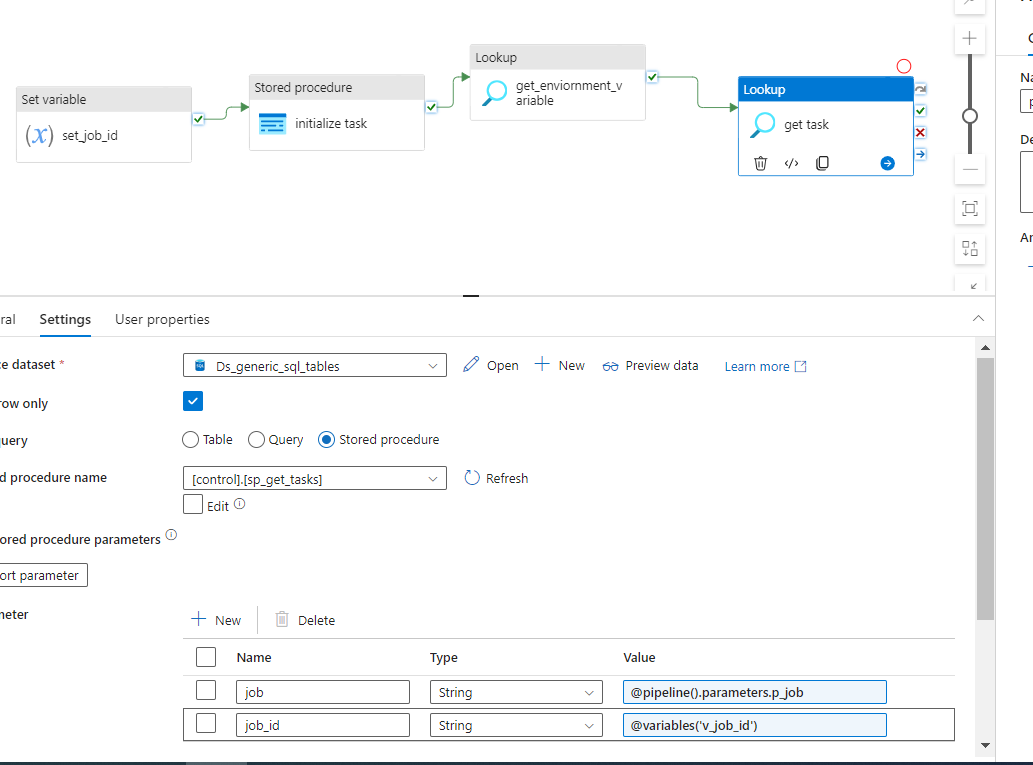
from config.enviornment) as i

pivot

(

max(value) for parameter in (subscription,rg,adf)) as o



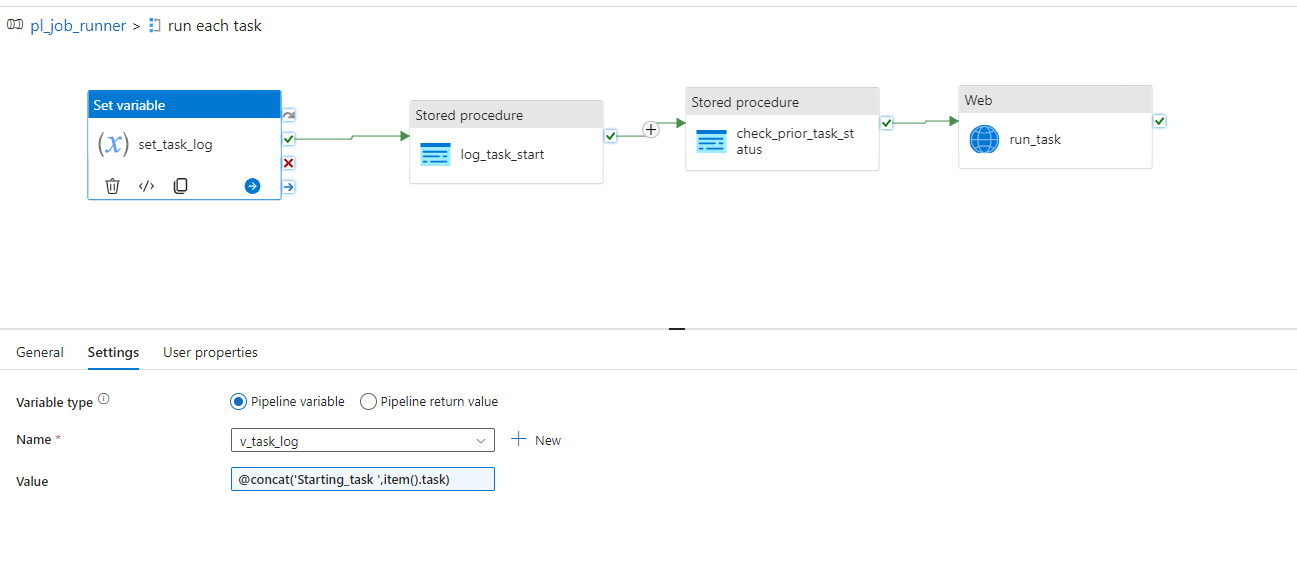


Absolutely! Let's break down the steps systematically:

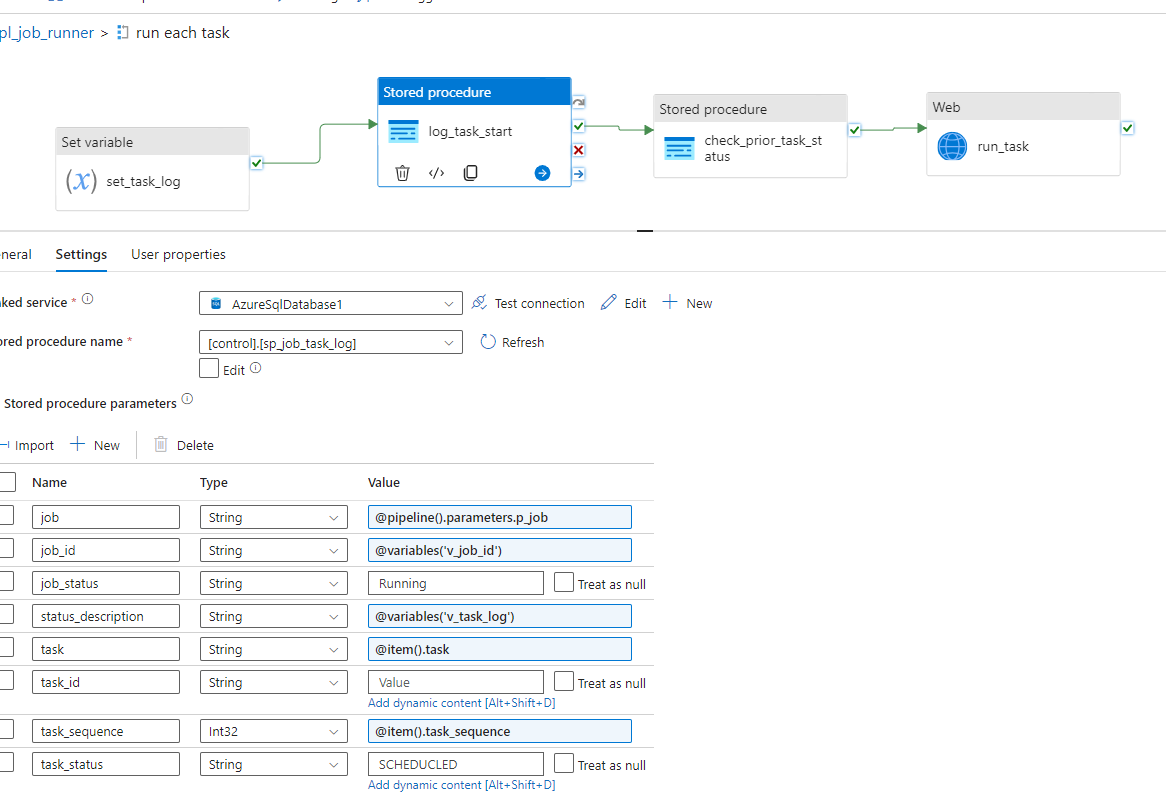
1. **Trigger Naming Convention:**
   * Assign a name to the trigger, like "load\_customer," signifying its association with loading the customer table.
2. **Parameter Configuration:**
   * Create a parameter, say "P," within the trigger to accept a value.
   * Set this parameter to take the value "load\_customer."
3. **Job ID Assignment:**
   * Declare a variable, "job\_id," and set its value to the pipeline run ID from the system variable.
   * The objective is to unify and track the three task pipelines under the same job ID for easier monitoring and correlation.
4. **Initializing Task Control:**
   * Use a stored procedure, like "sp\_init\_job\_control," to initialize the task control.
   * This procedure accepts parameters such as job\_id and job, initializing the job date and an error variable.
   * Conduct validations:
     + Check if the job exists and is active in the **config.jobs** table.
     + Verify no other instances of the same job are currently running or scheduled in the **control.job\_task\_control** table.
   * Insert task control entries for each task pipeline associated with the specified job:
     + Populate fields like job\_id, task\_id, job, job\_dt, task, task\_sequence, setting the task\_status as 'SCHEDULED.'
5. **Parameter Lookup & Assignment:**
   * Utilize a pivot statement to fetch environment parameter values.
   * Collect these values into a single record, mapping subscription ID, environment, and ADF name to respective columns for reference.
6. **Fetching Task Pipelines:**
   * Use a stored procedure, such as "SP\_get\_task," passing the job and job ID parameters to retrieve the task parameters (pipelines).
7. **Summary of Executed Steps:**
   * Ensure each step is documented comprehensively, describing its purpose and execution flow.

This structured approach ensures the trigger's initiation, sets job-related parameters, initializes task control, fetches environment values, and retrieves the task pipelines, all contributing to the orchestration and monitoring of the job execution.

Now take foreach loop and following activiries



**@concat('Starting\_task ',item().task) 🡪for set\_task\_log**



IF OBJECT\_ID('control.sp\_task\_status', 'P') IS NOT NULL

DROP PROCEDURE control.sp\_task\_status;

GO

CREATE PROCEDURE control.sp\_task\_status

(

@task\_status varchar(4000)

)

AS

BEGIN

IF @task\_status = 'ERROR'

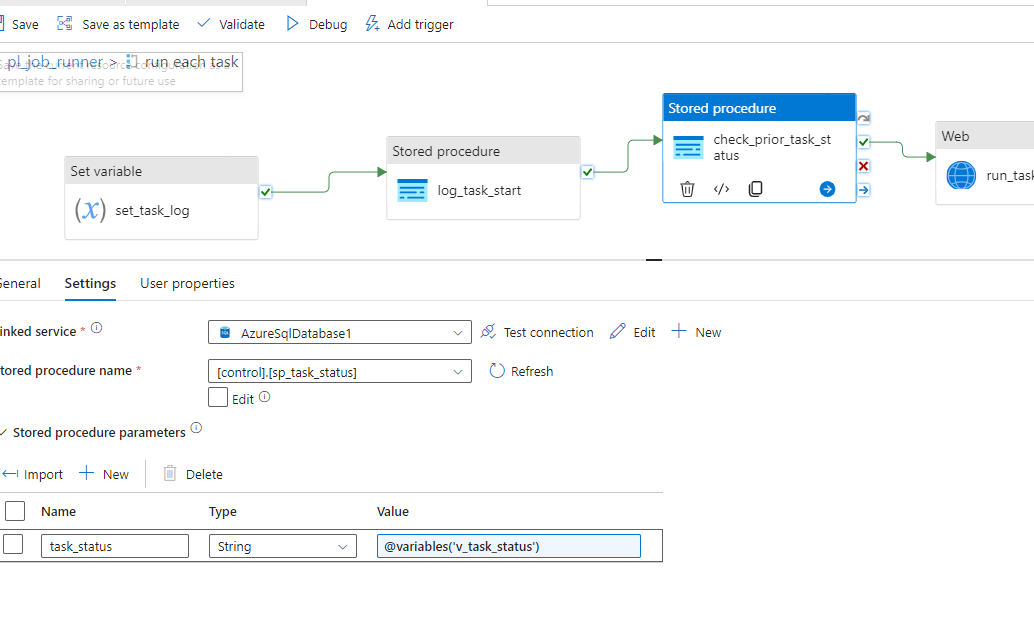
BEGIN

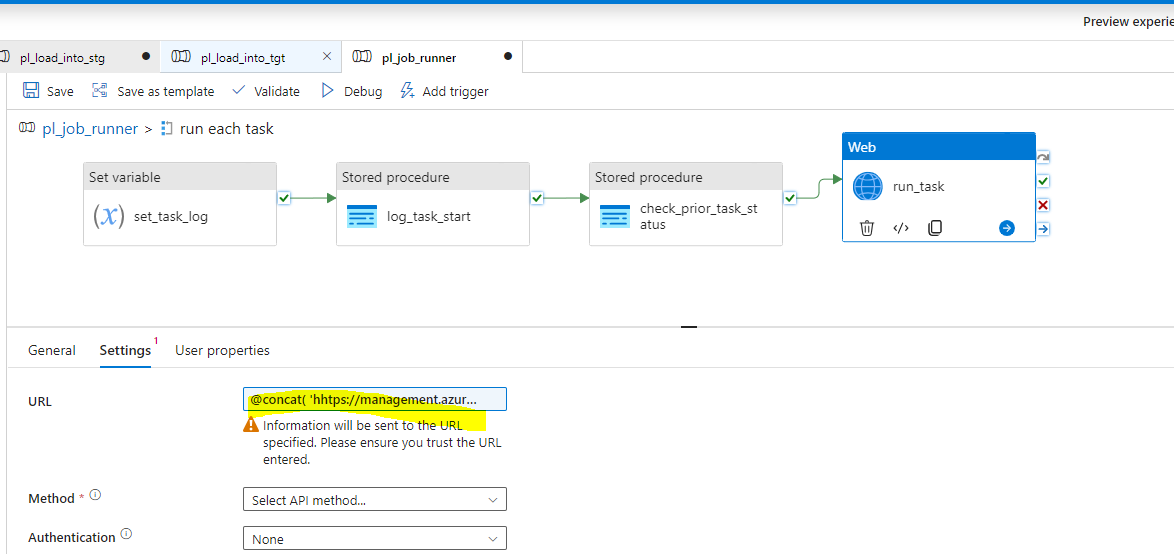
THROW 50001, 'Prior task has failed execution, unable to start next task run', 1;

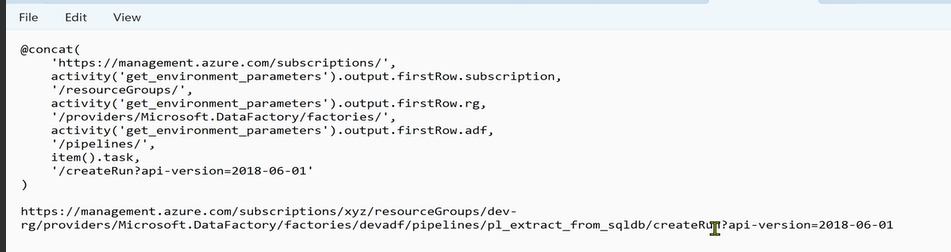
END

END

GO







In web activity url pass below

@concat(

'hhtps://management.azure.com/subscriptions/',

activity('get\_enviornment\_variable').output.firstRow.subscription,

'/resourceGroups/',

activity('get\_enviornment\_variable').output.firstRow.rg,

'/providers/Microsoft.DataFactory/factories/',

activity('get\_enviornment\_variable').output.firstRow.adf,

'/pipelines/',

item().tasks,

'/createRun?api-version=2018-06-01'

)

Here through web activities we are doing the dynamic to get sub, rg,adf and activity

Body

@concat (

'{ ',

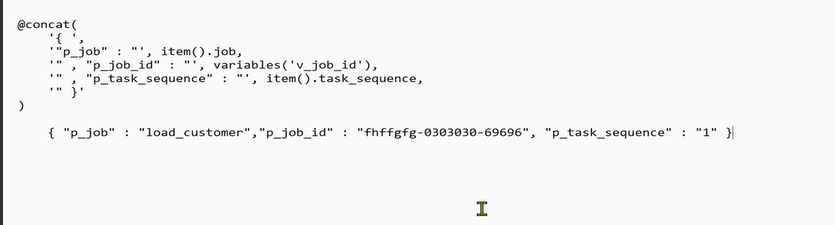
'"p\_job" : "', item().job,

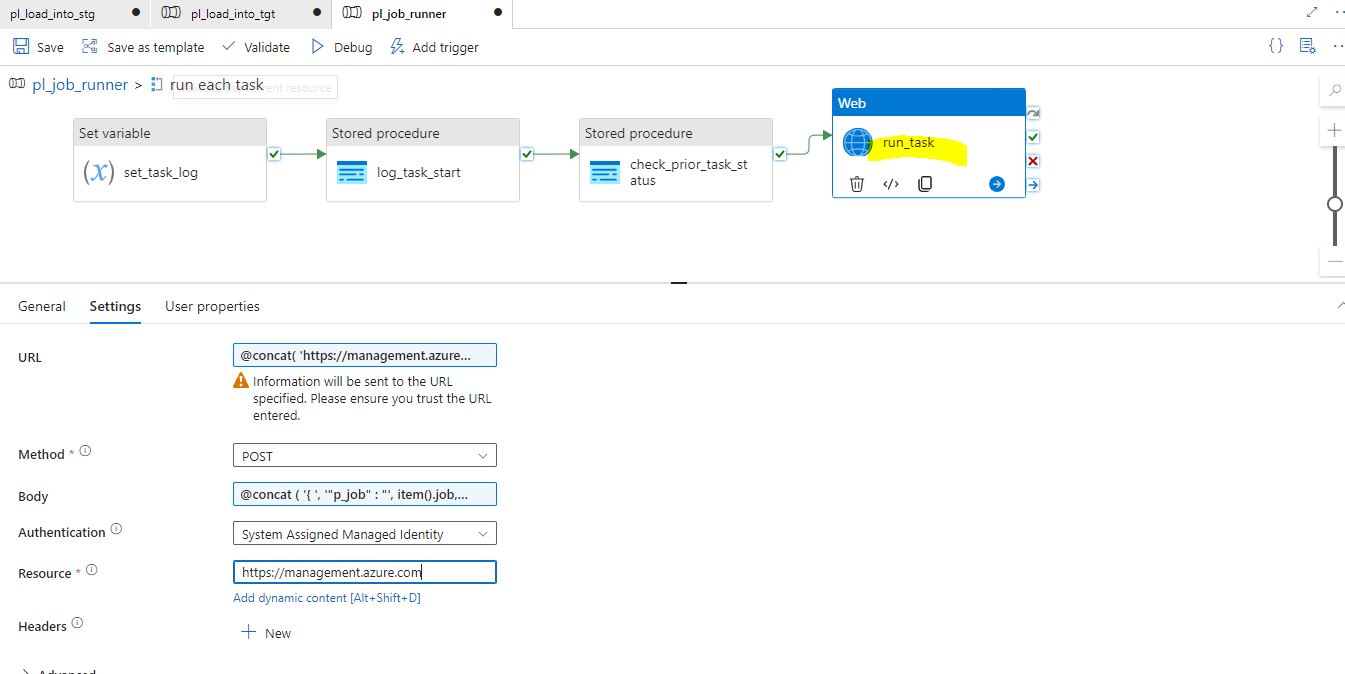
'" , "p\_job\_id" : "', variables('v\_job\_id'),

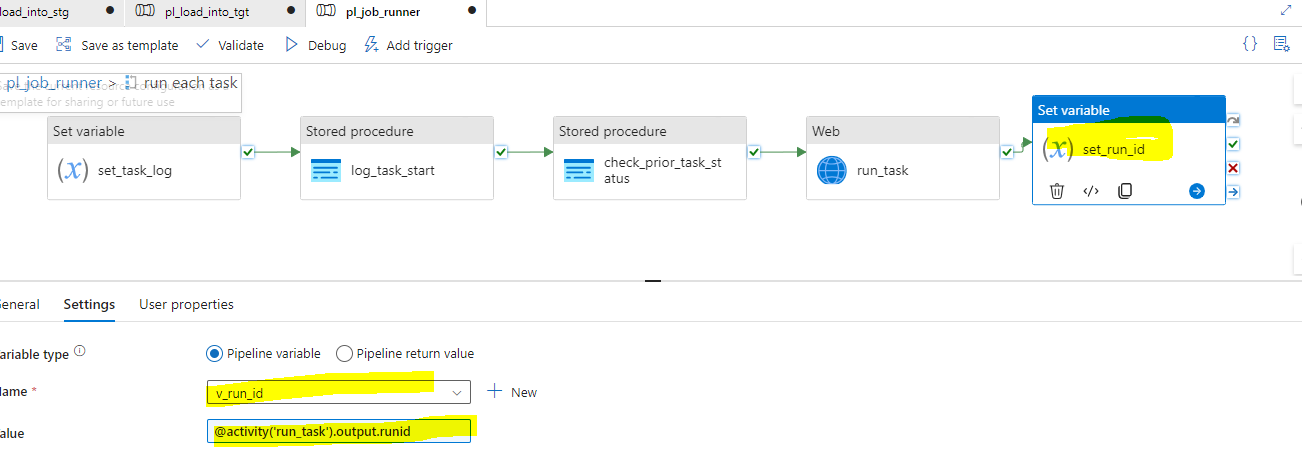
'" , "p\_task\_sequence" : "', item().task\_sequence,

'" }'

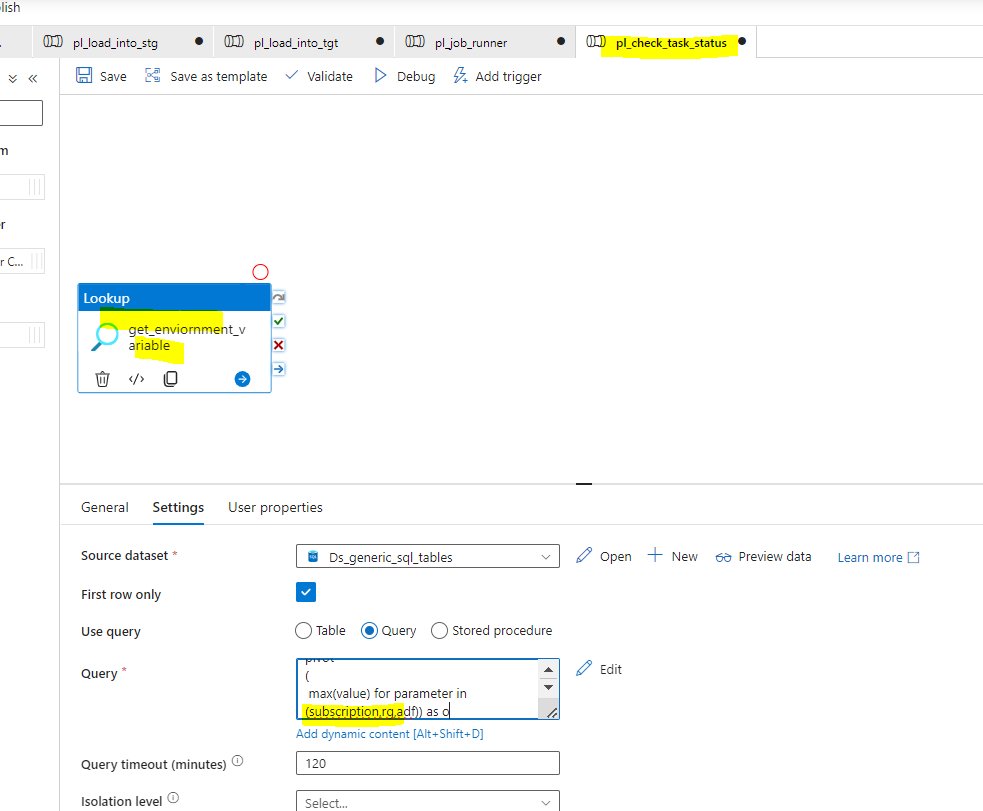
)



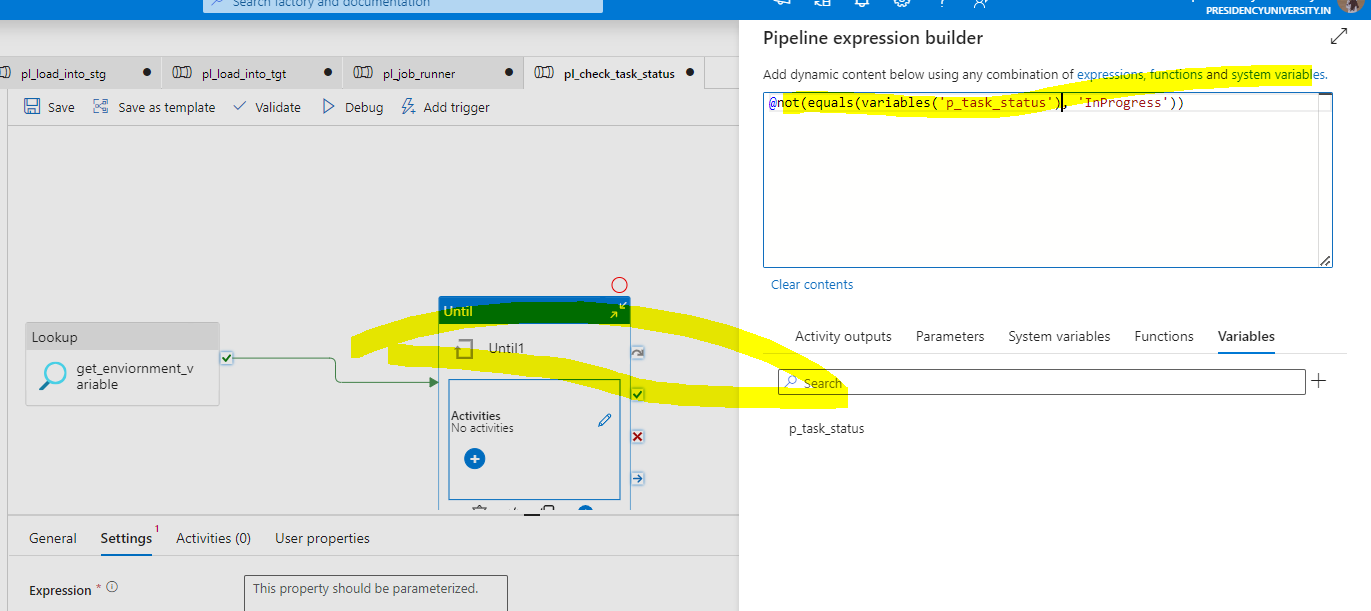


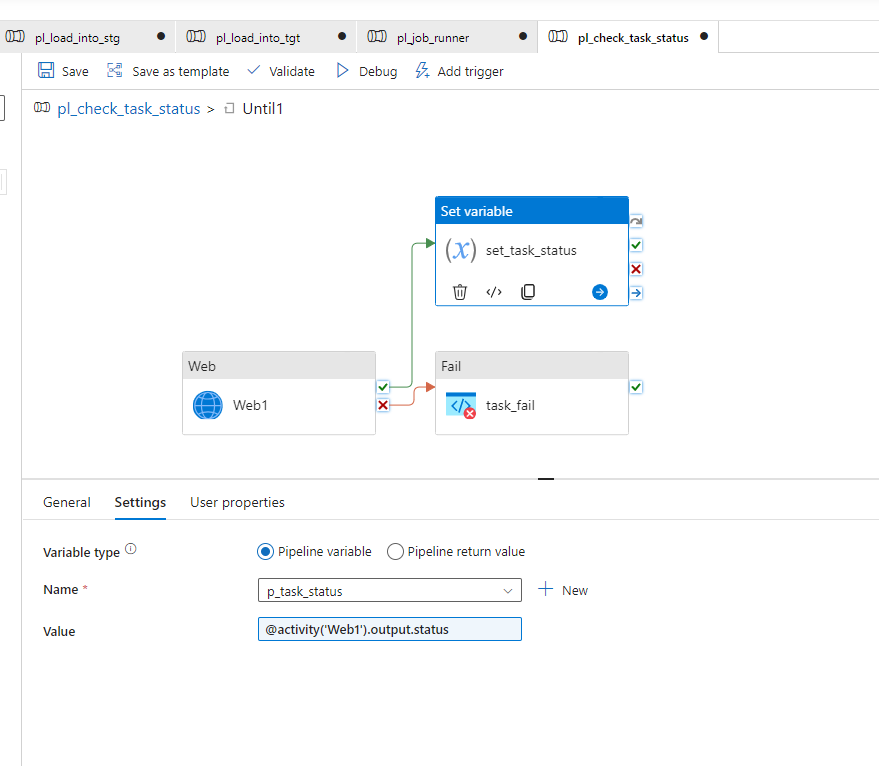


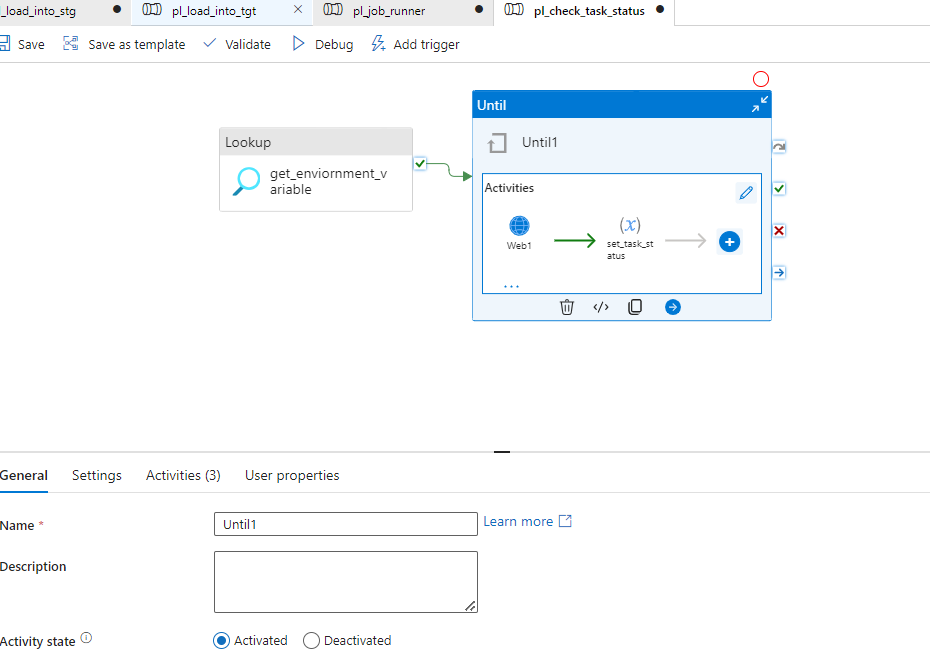
Next is do another pipeline which is called next here

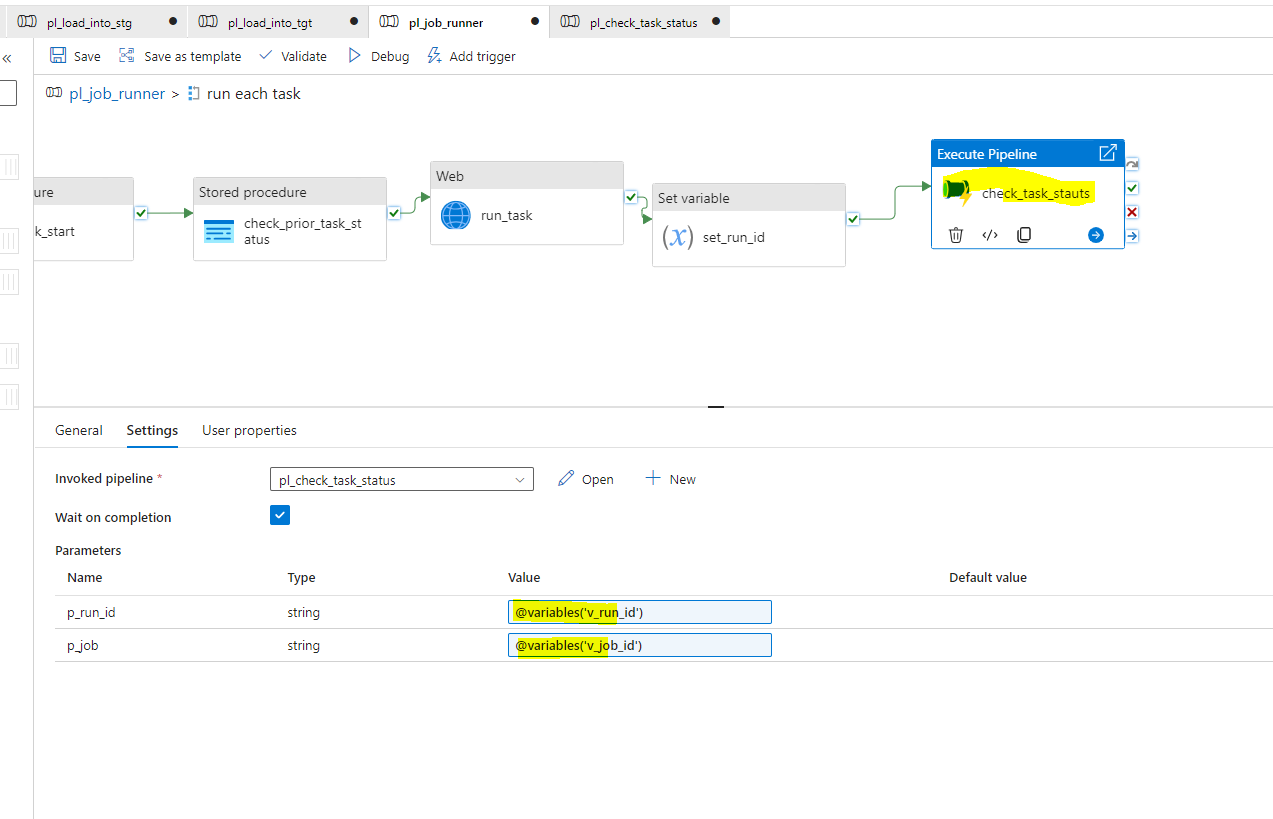


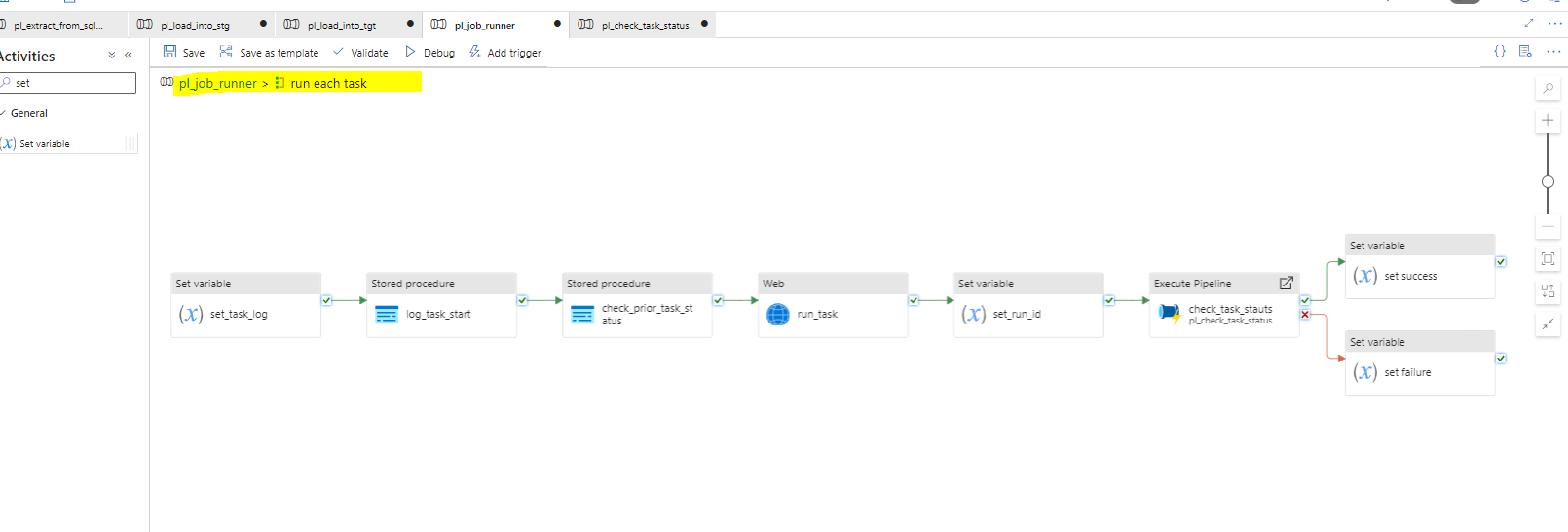
Pivot query to fetch enviorment

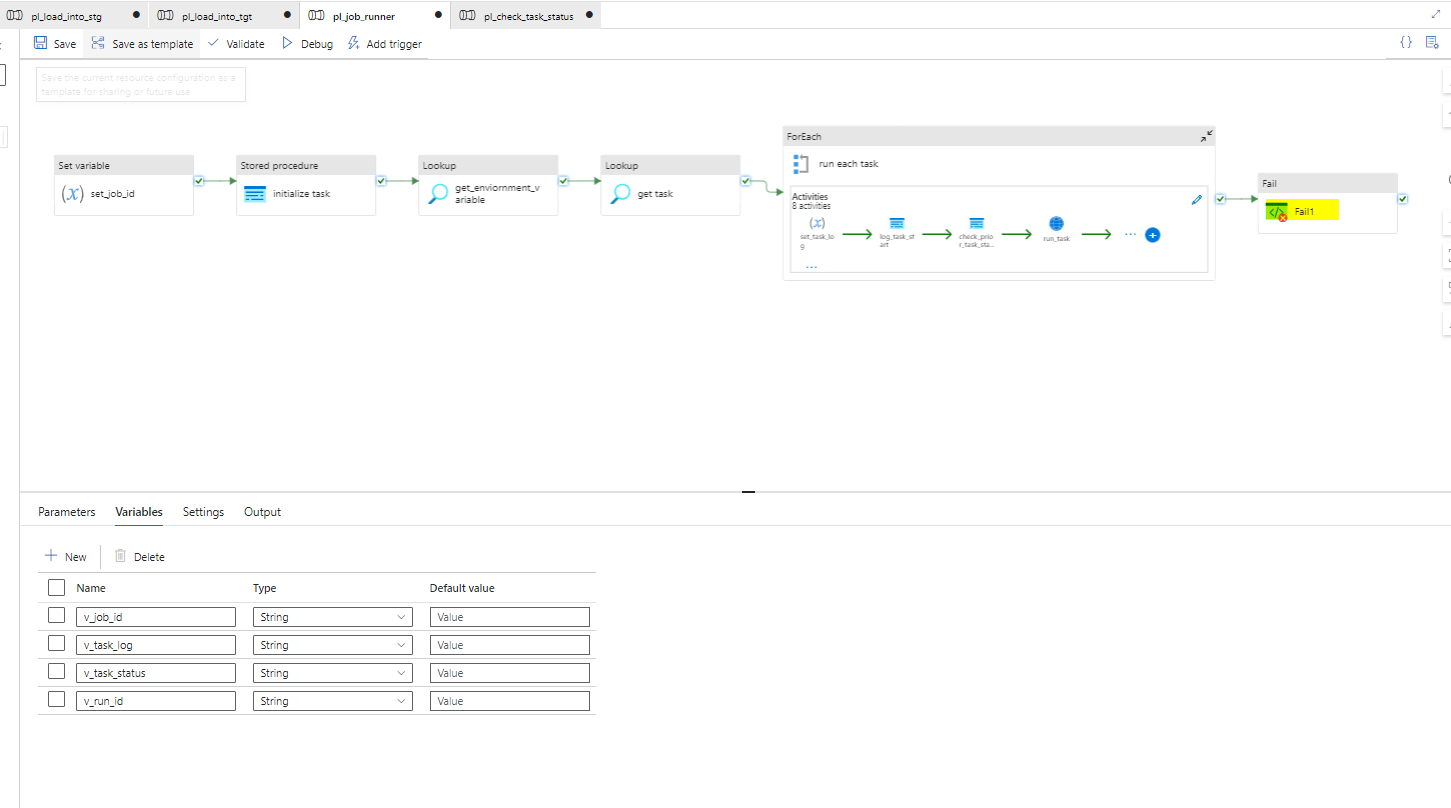












Now when u run the generic pipeline all the pipeline are taken and runs