Contents

[Synapse Overview 5](#_Toc155088610)

[Key Terms 6](#_Toc155088611)

[Synapse Linked Services “static” vs “dynamic”: 7](#_Toc155088612)

[Synapse Enterprise Data Flow 11](#_Toc155088613)

[Extraction and Load Step Copy Activity from External to Azure ADLS Parquet. 12](#_Toc155088614)

[Transformation step from parquet to Lakehouse bronze. 13](#_Toc155088615)

[Transformation step from Landing Zone Parquet to Warehouse stage. 13](#_Toc155088616)

[Transformation Step from Delta Bronze to Silver or Silver to Gold. 13](#_Toc155088617)

[Transformation Step from Delta Bronze, Silver, or Gold to Warehouse stage. 14](#_Toc155088618)

[Transformation step from Warehouse Stage to Warehouse 14](#_Toc155088619)

[Transformation Step from Warehouse to Delta Bronze, Silver, or Gold. 14](#_Toc155088620)

[Naming Conventions 15](#_Toc155088621)

[ADLS Data Lake – Landing Zone 15](#_Toc155088622)

[ADLS Data Lake – Bronze (raw) 15](#_Toc155088623)

[ADLS Data Lake – Silver (cleansed) 16](#_Toc155088624)

[ADLS Data Lake – Gold (curated) 16](#_Toc155088625)

[Synapse Linked Services 16](#_Toc155088626)

[KeyVault Secret 17](#_Toc155088627)

[Synapse DevOps 17](#_Toc155088628)

[Old DEV (pbit) to New Dev (enterpriseanalytics) Pipeline 17](#_Toc155088629)

[Power BI Azure Reporting Data Flow 18](#_Toc155088630)

[Autosys 19](#_Toc155088631)

[Catalog Enterprise Data 19](#_Toc155088632)

[Bronze & Gold Delta 19](#_Toc155088633)

[Ingestion Framework 20](#_Toc155088634)

[Transformation Framework 20](#_Toc155088635)

[Versioning SQL repositories 20](#_Toc155088636)

[Current Priorities 20](#_Toc155088637)

[EY Notebook Framework Presentation 21](#_Toc155088638)

[Reading options for ADLS delta lake to get into other relational engines 21](#_Toc155088639)

[Linked Services setup for both direct delta and serverless options POC 22](#_Toc155088640)

[Pipeline copy activity from serverless 23](#_Toc155088641)

[Dataflow from serverless 23](#_Toc155088642)

[Spark notebook Jdbc to serverless 24](#_Toc155088643)

[Spark notebook direct sql 24](#_Toc155088644)

[Data flow from delta adls 25](#_Toc155088645)

[POC delta vs parquet performance testing 25](#_Toc155088646)

[Staging data into landing zone for 1.633M rows of sapshr\_DFKKOP (~12 min) 25](#_Toc155088647)

[25](#_Toc155088648)

[Initialize table from parquet landing into bronze delta (~1 min) 26](#_Toc155088649)

[26](#_Toc155088650)

[Initialize table in silver from delta into silver delta (~2min) 26](#_Toc155088651)

[Method 1: delta serverless copy activity to dedicated using bulk load (~8.5 min) 26](#_Toc155088652)

[Method 2: delta serverless copy activity to dedicated using polybase staging (~3.5 min) 27](#_Toc155088653)

[Method 3: delta notebook to dedicated using sql pool adapter (~1.5 min… not including time to start spark session) 27](#_Toc155088654)

[Old method: adls parquet copy to dedicated using polybase (~2 min) 27](#_Toc155088655)

[27](#_Toc155088656)

[Writeback from dedicated using notebook (~40 sec without spark startup time 29](#_Toc155088657)

[29](#_Toc155088658)

[Writeback to adls parquet using copy activity from dedicated sql pool 29](#_Toc155088659)

[Writeback from dedicated using cetas external table (~20 seconds) 30](#_Toc155088660)

[Domain how to access enterprise data 30](#_Toc155088661)

[Login to portal and find synapse workspace 30](#_Toc155088662)

[Demo notebook and pipelines 31](#_Toc155088663)

[Ad-hoc access via ADLS synapse serverless sql pool 33](#_Toc155088664)

[Access via PowerBI using Domain Synapse Serverless Pool 35](#_Toc155088665)

[Access via PowerBI using SQL Server 38](#_Toc155088666)

[Security For Domains access to Enterprise ADLS 40](#_Toc155088667)

[41](#_Toc155088668)

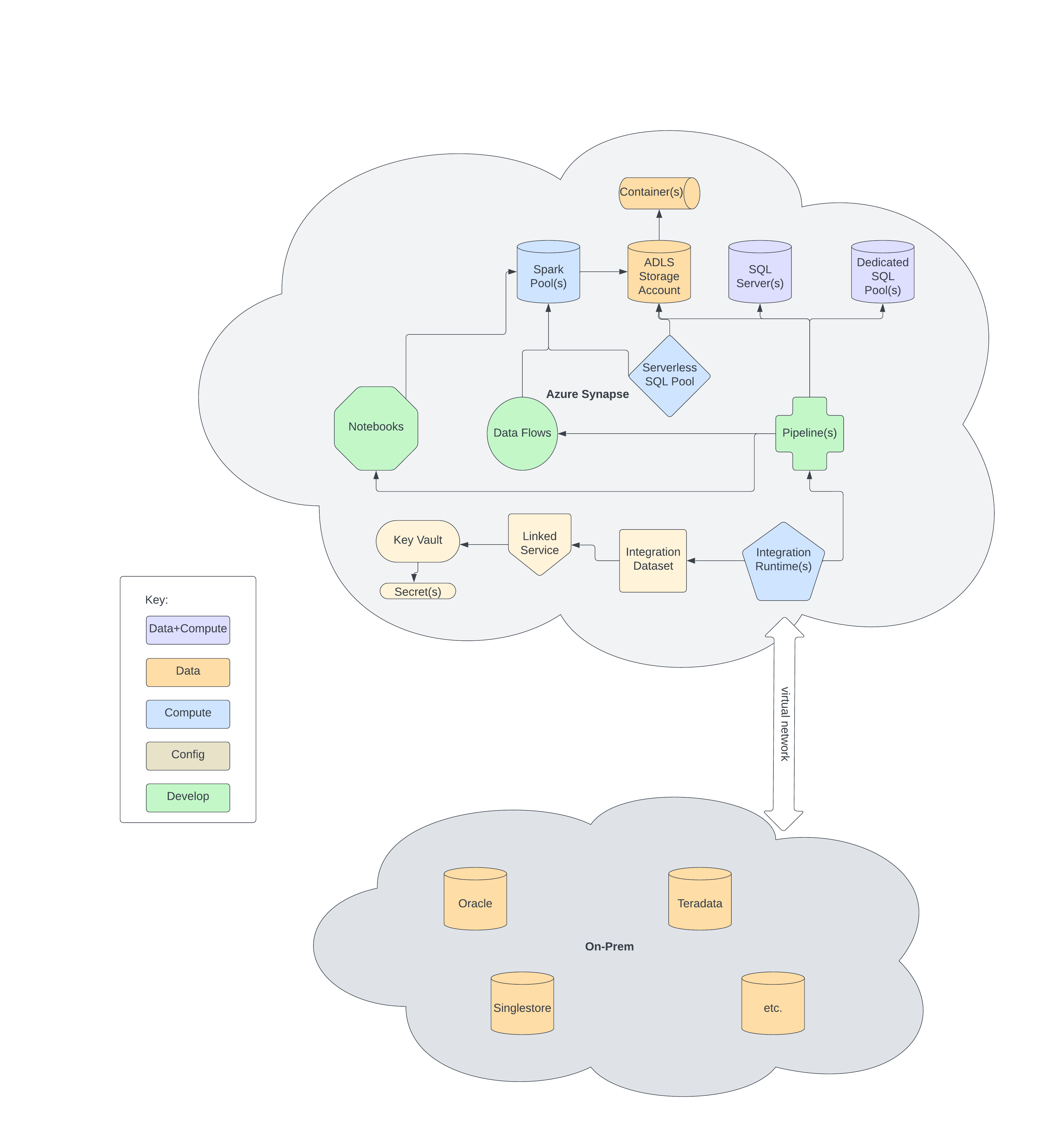
[**Qlik to PBI Project Handoff from Earnst & Young** 42](#_Toc155088669)

[Synapse Frameworks 43](#_Toc155088670)

[Ingestion 43](#_Toc155088671)

[Pipelines 43](#_Toc155088672)

[Tables 46](#_Toc155088673)

Synapse Overview

## Key Terms

*Spark Pool:* A pool of compute to run Spark jobs. These jobs can be developed in Notebooks, Data Flows, or Custom Spark Job definitions. It is largely used to transform data that already exists within the Azure data space. Synapse spark pools also will catalog a table that will then make it available within “Lake Database” and thus can be queried via Serverless SQL pool.

*ADLS2 Storage Account (like DFS):* A collection of containers that uses azures ADLS2 format.

*Container (like Logical Disk):* Disk for flat files or blobs that look like any typical filesystem tree to store your data.

*SQL Server (SQL Server):* few flavors available within Azure. But this is the long-standing SQL Server db engine. Two common types are Managed Instance (collection of databases) and SQL Database (one database).

*Dedicated SQL Pool (like SingleStore):* A pool of compute to run Polybase/SQL Data Warehouse (previous terms for this service). This is a MPP database much like Teradata and SingleStore. This compute costs whatever you provision as far as CPU and Memory. It can be scaled but it’s “dedicated”.

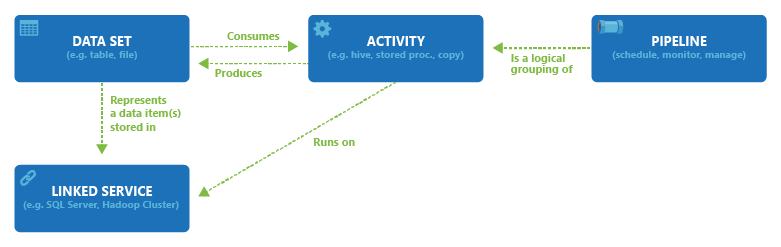
*Serverless SQL Pool:* An “elastic” pool of compute that queries ADLS2 containers to make flat files consumable. This compute is shared and scaled automatically in a “black box” type compute and costs per query (how many GB are processed per query).

*Pipelines (like PowerCenter Workflows):* a code-less development environment to create “activities”. From an integration perspective you can create complex workflows together to pull/push data to/from on prem using a linked service and dataset (See definitions below). This linked service communication is accomplished through a virtual machine in Azure that is called an integration runtime. So, when you run an activity (e.g. Copy Activity) it submits the job to the integration runtime server which then acts as a gateway to our on prem environment. From a workflow point of view, there are many types of activities that can be logically grouped together. A couple examples of those activities are Notebook Activity and Data Flow activity. Say you wanted to pull data from an external source and land it in ADLS. You’d start with a copy activity which sources data from dataset <- linked service <- runtime. Sink-ed it to an ADLS container via dataset <- linked service <- runtime. Then if you want to transform data that was staged in ADLS from a Copy Activity. You’d create a Data Flow activity or Notebook Activity that referenced a specific notebook or data flow. See below on differences between those two development environments.

*Linked Services (like ODBC):* a linked service is just a configuration to some endpoint that can be used within a pipeline. Think of it as ODBC for synapse. You create a linked service for your type of endpoint/system (Oracle, SQL Server, MySQL, SAP, etc.). That’s the “what” which tells it which driver or api is used for the call to that system. That driver or api needs to be installed or enabled on the Integration Runtime environment. Then depending on the type of linked service, it has configuration settings. Which essentially tells it the “where”. The where configuration can be stored in Key Vault as a secret (see def below) to protect sensitive information.

*Integration Runtime (like PowerCenter Integration Runtime):* in our case we almost always use a self-hosted collection of virtual machines within our Azure resource group. Their main capabilities are being the integration piece between on-premise and cloud environments. For instance, when data is retrieved via a copy activity from an on prem resource these servers facilitate that request. It does so via a linked service (def above) that contains config information and the necessary installed binaries that know how to talk to said system. This runtime also is used for communication to azure resources in that same way. By default, when creating a synapse workspace an integration runtime is created. But it cannot be changed or configured to communicate with on prem or external resources. This is why we don’t use that runtime. And is why we use the self-hosted runtime.

*Integration Dataset (like PowerCenter source and targets)*: can be thought of as a layer overtop of a linked service that contains the specifics of the source or sink your after. Meaning, after creating a linked service to some source datastore (e.g. Oracle system). You’d create a dataset that contains the settings to use it as either an input (source) or output (sink).



*Data Flows (like Powercenter mappings w/o external Azure recources)*: a code-less development environment that creates transformation tasks that compile down to spark jobs that execute on the spark pool. This environment is used to transform source data that exists WITHIN Azure (SQL, Flat, etc.) into Azure destinations (SQL, Flat, etc.). These data flows are run/orchestrated via a pipeline or interactively in debug mode.

*Notebooks (like Powercenter Java):* a code-first development environment that creates spark jobs that execute on the spark pool. This environment because it is code-first is more flexible in it’s use cases than data flows. You can interactively profile data, create visuals, read/write data to/from azure, include markdown to tell a story, etc. It’s very similar to something like Jupyter Notebooks. But by default it has hooks into ADLS and a Spark Pool to then run api commands against anything in the synapse workspace. Languages used are any of the following spark api’s; SQL, .Net, Python, R, and Scala. These notebooks are run/orchestrated via a pipeline or interactively in debug mode. So, if needing a quick flexible way to control logic of transformations this is a good route to go.

*Parameters:* a lot of what is configurable within Synapse can be “parameterized”. These parameters typically allow for whatever has been developed to be reusable. What this means is values can be passed in, evaluated, and then executed within Synapse. Any pipeline, data flow, notebook, etc. provides the use of parameters throughout. Some config services allow for parameters too (integration dataset and linked services).

*Key Vault (SecVault):* is a storage environment for storing key value pairs in encrypted format. It’s main purpose it to securely store sensitive information for retrieval with data pipelines, ci/cd pipelines, etc. This keys value can be retrieved as long as the user or resource has access to the key from an api or many built in features of Azure. For instance, when creating a linked service within Azure portal the GUI provides a built-in mechanism to define which Key you would like to put in place of the configuration setting. So it’s great place to store passwords for instance. As then the value is never passed around or saved in clear text. At runtime for the particular configuration setting it is decrypted and then used.

*Secrets (SecVault account) :* these are just key value pairs where the value is encrypted. They are sored in a Key Vault and can be consumed in many ways. Key value pairs are “Key”:”Value”. So there could be a key value pair defined as “users\_password”:”pizza1”. But pizza1 is stored at rest in an unreadable format. As well as in flight it is sent in unreadable and the decrypted at its destination.

# Synapse Linked Services “static” vs “dynamic”:

*Background*: Linked services are a bit like ODBC configurations on your machine. With ODBC you’ve got your config file (odbc.ini); in it an ODBC name, driver and driver config settings exist. In Synapse a linked service name, connection type and connection config settings are defined. A static linked service means all 3 items (name, type, settings) are hard coded into that service. A dynamic linked service means name and type are static. But the individual settings are parameterized and can be changed at runtime. Screenshots below depict two copy activities. Static example is using a static linked service. The dynamic example is using a dynamic linked service.

*Static example*:

*Linked service*:

A screenshot of a computer

Description automatically generated

*Copy activity:*

A screenshot of a computer

Description automatically generated

*Dynamic example*:

*Linked service*:

A screenshot of a computer

Description automatically generated

*Copy activity:*

A screenshot of a computer

Description automatically generated

*Why dynamic*: Within Synapse development environments, almost all of it can be parameterized. For example, pipelines within Synapse. If that pipeline is to be reused for multiple linked services; you’d parameterize as the screenshot show above. One portion you cannot change is the linked service being used itself. Meaning you can’t parameterize a linked service “name”. You can however parameterize settings within a linked service. So, with the above copy activity at runtime a different ls\_secret\_name can be passed in and the query executed on whatever Oracle instance that secret provides.

|  |  |  |
| --- | --- | --- |
| Scenario | Static | Dynamic |
| Pipeline has bug specific to linked service or dataset (SQL Server connection option quoted\_identifier). How would we fix? | In specific pipeline where bug exists. Check for specific linked service and alter linked service to include quoted\_identifier option. Any other pipelines using linked service would be affected | During execution of pipeline pass in parameter for linked server to alter driver options for quoted\_identifier. If not wanting in other runs of dynamic pipeline, make it default to what it was before and only change in needed pipeline parameters. |
| Linked service driver was updated and is causing all pipelines that use it to fail. How would we fix? | You could backout change to driver on integration runtime. Or you could create another linked service to use old driver. Then change affected pipelines to use it. If many pipelines were changed would need to change each one. | You could backout change to driver on integration runtime and flip linked service to using. Or you could create another linked service to use old driver. Then change affected pipeline to use it. |
| Pipeline has bug specific to an activity’s linked service type (e.g., Oracle dates 1-1-0001 needed scrubbed as lower than SQL Server minimum of 1-1-1900. How would we fix? | Whatever the means of transforming column to then get into SQL Server (pushdown sql, in flight derived column, change to string, etc). Change it in specific pipeline that references linked service. If multiple pipelines have similar dates, need to change there too. | Whatever the means of transforming column to then get into SQL Server (pushdown sql, in flight derived column, change to string, etc). Change it in specific pipeline that references linked service. |
| Find pipeline logs for a specific table or extract when troubleshooting? | Log into portal and go to monitoring. If name of pipeline was intelligent and contained something searchable you could do so using that name. If not, intelligent you could use annotations of pipeline. | Log into portal and go to monitoring. On pipeline execution pass in a parameter that is referenced in its annotation’s definition. That annotation or tag is searchable. |
| Rerun failed pipeline without manually entering dynamic parameters? | Log into portal monitoring tab and find failed pipeline run. Click rerun. If for some reason a parameter needs change (some timestamp or something not related to linked service). You’d trigger a new run and copy any other previous parameters. | Log into portal monitoring tab and find failed pipeline run. Click rerun (parameters are saved as part of run). If for some reason a parameter needs change (some timestamp or something not related to linked service). You’d trigger a new run and copy any other previous parameters. |
| Quickly replicate a development task that has been solved for before? | Can copy a pipeline, change things manually to where things should point. But could lead to outdated copies if something universal needs changed. | Can change pipeline parameters during execution to replicate needed result. |
| Change a username/password? | Change within KeyVault and changes will propagate through. | Change within KeyVault and changes will propagate through. |
| If we find issue with either static or dynamic approach, can we switch? | Could switch to dynamic by editing any static pipelines to include parameters for a dynamic linked service. | Could switch to static by copying each dynamic pipeline and changing each linked service to appropriate static one. |
| Can we secure a secret from being used in a dynamic linked service? | You can granularly down to user or group level apply security using workspace rbac access control. So you could give operator access which would allow user to see pipeline runs, run them, etc. But not allow them to create linked services or use them. | You can granularly down to user or group level apply security using workspace rbac access control. So you could give operator access which would allow user to see pipeline runs, run them, etc. But not allow them to create linked services or use them. |

**Other useful links:**

*Datasets in Azure Data Factory and Synapse Analytics*

<https://learn.microsoft.com/en-us/azure/data-factory/concepts-datasets-linked-services?tabs=data-factory>

*Parameterize linked services in Azure Data Factory and Azure Synapse Analytics*

<https://learn.microsoft.com/en-us/azure/data-factory/parameterize-linked-services?tabs=data-factory>

# Synapse Enterprise Data Flow

A diagram of a cloud

Description automatically generated

NOTE: we can combine Landing and Bronze in Lakehouse above. But clear delineation of what is landing vs bronze needs to be a part of the folder structure, security, and policies.

## Extraction and Load Step Copy Activity from External to Azure ADLS Parquet.

A green cross with black text

Description automatically generated

Pipeline copy activity can query or move over data from linked services and integration service.

## Transformation step from parquet to Lakehouse bronze.

A screenshot of a computer

Description automatically generated

Two options, Notebook or Data Flow to read landing zone parquet. Either of them compiles down to same thing, Spark. Notebooks are more flexible in that it’s code. Data Flows are stricter being code-less. Both allow for parameters. Cannot use pipeline as there are no activities that allow you to write to delta at this time.

## Transformation step from Landing Zone Parquet to Warehouse stage.

A screen shot of a computer

Description automatically generated

External tables in dedicated SQL pool can read parquet in ADLS using External Tables. NOTE: dedicated SQL pool cannot read delta at this time.

## Transformation Step from Delta Bronze to Silver or Silver to Gold.

A screenshot of a computer

Description automatically generated

Two options; Notebook or Data Flow to read landing zone parquet. Either of them compiles down to same thing, Spark. Notebooks are more flexible in that it’s code. Data Flows are stricter being code-less. Both allow for parameters. Cannot use pipeline as there are no activities that allow you to write to delta at this time.

## Transformation Step from Delta Bronze, Silver, or Gold to Warehouse stage.

A screenshot of a computer

Description automatically generated

Three options; Notebook, Data Flow or Pipeline Copy Activity (using serverless pool as source). Notebook and Data Flow either of them compiles down to Spark. Both allow to query a version or from a timestamp of the delta table. Both allow for parameters. A pipeline copy activity can read delta from the serverless sql pool. As the serverless sql pool allows for reading of delta. Serverless though does not allow you to query a specific version or from a timestamp. All can write to dedicated sql pool table.

## Transformation step from Warehouse Stage to Warehouse

A screenshot of a computer

Description automatically generated

Depending on path prior, either parquet->external table or delta->table. A sql script or a stored procedure can be used for transforming into production table. Store procedures allow for parameters and are compiled into a sql plan for reuse. SQL scripts cannot do either of those.

## Transformation Step from Warehouse to Delta Bronze, Silver, or Gold.

A screenshot of a computer

Description automatically generated

Two options; Notebook, Data Flow. Notebook and Data Flow either of them compiles down to Spark. Both allow to write delta file format from data warehouse. Both allow for parameters.

# Naming Conventions

## ADLS Data Lake – Landing Zone

Optional to use landing zone, but if data needs landed before bringing into medallion it should be separate from bronze, silver or gold layers to identify it is only a ephemeral data asset.

|  |  |
| --- | --- |
| **Description** | **Example** |
| Landing zone folder in bronze container for crews raw data of table CREWSSSP.WORK\_REQUEST | bronze\lzone\crews\crewsssp\work\_request\20230815\_105301\ |
| Landing zone container for crews raw data of table CREWSSSP.WORK\_REQUEST | lzone\crews\crewsssp\work\_request\20230815\_105301\ |
| Landing zone container for sapshr raw data of table SAPR3.SET\_HEADER | lzone\sapshr \sapr3\set\_header\20230815\_105301\ |

## ADLS Data Lake – Bronze (raw)

Bronze layer should consist of the raw data needed to construct an accurate representation of the source. The folder structure should include application\schema\table. After that the folder names should be self-documenting on what is included in the files underneath. It is a best practice in data lakes to use “Hive partitioning” (key=value) naming convention to then be able to exclude files when reading and writing data.

|  |  |
| --- | --- |
| **Description** | **Example** |
| Bronze container for crews raw data of table CREWSSSP.WORK\_REQUEST | bronze\crews\crewsssp\work\_request\ |
| Bronze container for crews raw data of table CREWSSSP.WORK\_REQUEST partitioned on year(entry\_date) | bronze\crews\crewsssp\work\_request\entry\_date\_year=2023\ |
| Bronze container for sapeast raw data of table SAPR3.SET\_HEADER that is snapshot daily | bronze\sapshr\sapr3\set\_header\snapshot\_dt=20230815\ |
|  | bronze\singlestore\finance\_dw\table\_name? |

## ADLS Data Lake – Silver (enriched)

Silver layer should consist of cleansed data needed to construct an accurate representation of the source with friendly names and values. The folder structure should include application\schema\table. After that the folder names should be self-documenting on what is included in the files underneath. It is a best practice in data lakes to use “Hive partitioning” (key=value) naming convention to then be able to exclude files when reading and writing data.

|  |  |
| --- | --- |
| **Description** | **Example** |
| Silver container for crews curated data of table CREWSSSP.WORK\_REQUEST | silver\crews\crewsssp\work\_request\ |
| Silver container for crews curated data of table CREWSSSP.WORK\_REQUEST partitioned on year(entry\_date) | silver\crews\crewsssp\work\_request\entry\_date\_year=2023\ |
| Silver container for sapshr curated data of table SAPR3.EVER that is snapshot daily | silver\sapshr\sapr3\set\_header\snapshot\_dt=20230815\ |

## ADLS Data Lake – Gold (curated)

Gold layer should consist of curated data needed to construct an accurate representation of what the solution or analytical needs are. The folder structure should include domain\table type\table. After that the folder names should be self-documenting on what is included in the files underneath. It is a best practice in data lakes to use “Hive partitioning” (key=value) naming convention to then be able to exclude files when reading and writing data. Table types are fact and dim.

|  |  |
| --- | --- |
| **Description** | **Example** |
| Gold container for work managements crews curated data of tables CREWSSSP.WORK\_REQUEST and CREWSSSP.A\_WORK\_REQUEST | gold\wm\fact\work\_request\ |
| Gold container for work managements crews curated data of table CREWSSSP.WORK\_REQUEST and CREWSSSP.A\_WORK\_REQUEST partitioned on year(entry\_date) | gold\wm\fact\work\_request\entry\_date\_year=2023\ |
| Gold container for sapshr Finance’s curated data of tables SAPR3.SET\_HEADER, SAPR3.SET\_HEADERT, SAP3.SET\_NODES to represent their hierarchies | gold\fin\dim\cost\_element\_hierarchy\ |

Dedicated SQL Pool Tables

Dedicated SQL Pool External Tables

Dedicated SQL Pool Views

Dedicated SQL Pool Stored Procedures

## Synapse Linked Services

Linked services should be re-usable. Names should have synapse environment and a unique identifier that self-documenting to where it is linked. In general, it should follow fe-<synapse env>-ls-<identifier>.

|  |  |
| --- | --- |
| **Description** | **Example** |
| Dev synapse workspace that is linked to an oracle database via a dynamic secret | fe\_d\_ls\_oracle\_secret |
| Statically linked to sapeast production in dev synapse workspace. Note, production is implied in absence of no dev, qa, sbx, etc. in name. | fe\_d\_ls\_oracle\_sapeast |
| Statically linked to sapeast development oracle environment in dev synapse workspace | fe\_d\_ls\_oracle\_sapeastdev |

## KeyVault Secret

|  |  |
| --- | --- |
| **Description** | **Example** |
| Dev enterprise anaylytics synapse workspace key vault secret. Which is a connection string to the dedicated sql pool for dev. | fe-d-secret-enteranalytics-dwdev-connstring |
| Dev enterprise anaylytics synapse workspace key vault secret. Which is a connection string to the dedicated sql pool in prod (missing dev, qa, etc. implies prod). | fe-d-secret-enteranalytics-dw-connstring |
| Prod enterprise anaylytics synapse workspace key vault secret. Which is a connection string to the singlestore prod finance database (missing dev, qa, etc. implies prod). | fe-p-secret-enteranalytics-ss-fin-cnnstr |

Synapse Pipelines

Synapse Dataset

Synapse DataFlow

Synapse Notebook

# Synapse DevOps

## Old DEV (pbit) to New Dev (enterpriseanalytics) Pipeline

NOTE: WASN’T PLANNING ON USING… WAS GOING TO MOVE OVER THINGS INTO FRAMEWORK 1 BY 1. THIS DEVOPS PIPELINE WAS DONE PRIOR TO EY DEPARTURE AS AN OLIVE BRANCH.

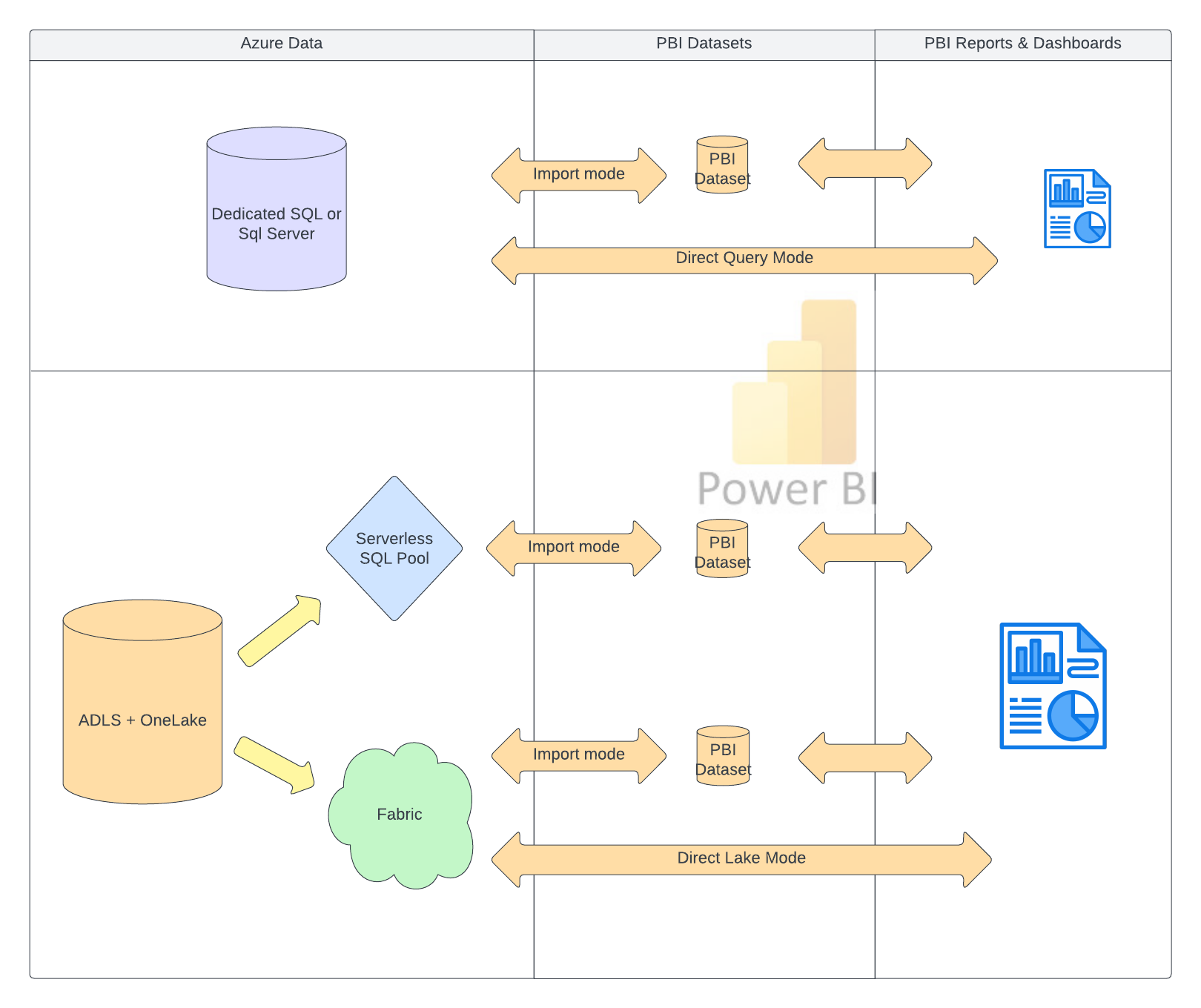
Link to DevOps Repo for pbit: <https://fe-general.visualstudio.com/Analytics/_git/Enterprise_Analytics>

Two working branches and three pipelines. Branch to deploy from pbit to enterpriseanalytics for **synapse objects** is **dev\_synapse\_objects**. Branch to deploy **dedicated sql pool objects** from pbit to enterpriseanalytics is **dev\_dw\_objects**.  Pipeline Enterprise\_Analytics\_ENT\_DEV\_DW\_Validate (builds and validates dacpac can be created from dev\_dw\_objects), Enterprise\_Analytics\_ENT\_DEV\_Synapse\_Validate (validates synapse objects can be deployed by compiling and checking for dependencies from dev\_synapse\_objects), and Enterprise\_Analytics\_ENT-DEV\_Deploy (deploys database and synapse objects).

**dev\_synapse\_objects:** controlled by csv in **Deploy\Scripts\Synapse\listOfFilesToDeploy.csv**.  **Truncation of everything in enterpriseanalytics will still occur when this branches deployment is ran.** You create a branch in Enterprise\_Analytics repo through synapse studio.  You create your stuff or make changes in synapse studio within pbit and then commit to that new branch. You then create pull request and merge into dev\_synapse\_objects branch (we should code review at this step). Last, you change the csv in the dev\_synapse\_object branch to include the objects needed. From there you run the devops deployment pipeline (Enterprise\_Analytics\_ENT-DEV\_Deploy).

**dev\_dw\_objects:** a database project is used for this (Azure Data Studio). You create a new database project locally on your machine within Azure Data Studio.  You do a schema compare of enterpriseanalytics database and your new empty project. You copy / select the objects you need to include in the changes (if any).  Once all pulled down you initialize a new branch in the devops project with what you copied from enterpriseanalytics.  Make your changes (add table, alter column, change proc, etc); if done outside of Azure Data Studio (e.g, done with Synapse or Management Studio), you run schema compare again with enterpriseanalytics and then commit to your new branch. Last you then in azure devops create a pull request and your branch merge back into dev\_dw\_objects (should have code review step here).  Which will update the working repo.  Validation pipeline is then ran. From there the deployment pipeline Enterprise\_Analytics\_ENT-DEV\_Deploy can be run which builds a dacpac and pushes changes to the database.

# Power BI Azure Reporting Data Flow



# Autosys

Scheduling in Azure will still be handled by Autosys. Two VM’s (dev and prod) are created in azure with autosys agent installed. Autosys provides a plugin to call synapse pipelines and that is installed too. Security of those calls is using an app in Azure which provides a service principal (essentially an AD account). This service principal has a client id and a client secret that are used for logging in to synapse or other Azure resources. Those are stored in our azure key vault.

# Catalog Enterprise Data

When creating hive cataloged table in lake database via Synapse pyspark notebook. If column is varchar(max) from source it converts to string in spark. Then when replicated to lake database it creates an external table in sql serverless pool. But the varchar(max) or string column is then defaulted to varchar(8000). Which leads to truncation of data and a warning/error when querying the external table via serverless SQL pool. To bypass this issue we have a couple options:

1. create views in serverless using with clause to strongly type data
   1. solves problems with varchar(max) or geometry types
   2. increases performance as you can type your columns smaller if need be
   3. can be called via a linked service and serverless endpoint (synapse pipelines copy activity)
   4. messy as this would involve creation from outside of spark notebook or at the very least a separate step when initializing table
   5. you can’t use delta time travel or any delta spark api calls… thus you wouldn’t be able to leverage change data feed to get only data that has changed
2. create spark package that will assist domains and enterprise in pulling data via synapse notebooks
   1. Solves problems with varchar(max) or geometry types
   2. Can use time travel or other spark delta api’s
   3. Involves notebooks and spark only so provisioning takes a few minutes to spin up
3. Read full or incremental parquet or flat files in landing zone (temp) and transform into data assets from there

Option 2 is our recommended approach as the con of time delay of each notebook running can be mitigated with having control loop / orchestration be inside of spark notebook.

# Bronze & Gold Delta

Pros

* Change data feed and leveraging it for all downstream medallion layers.
* Superior format provides acid compliant transactions, audit trailing, performance, etc <https://medium.com/untienots/delta-lake-when-reliability-meets-performance-2db967303d71>
* Fabric / Databricks / Lakehouse use it

Cons

* DW Dedicated SQL pool and Synapse pipelines can’t read it directly.

Silver and gold are fine as delta (EY opinion). And we can share with business those formats.  Well, each domain will have same issue.... how do you get the delta data into sql? Answer would be shared python package that can be distributed and used within synapse notebooks (see “Sharing Enterprise Data”).

<https://piethein.medium.com/medallion-architecture-best-practices-for-managing-bronze-silver-and-gold-486de7c90055>

# Versioning SQL repositories

Flyway

MS Database Projects

# EY Notebook Framework Presentation

* New notebook for every transformation… not metadata driven
  + It’s a template… standards can be changed.. just clone template and then change it all up.. leads to problems
  + Only full reload for framework… incremental is separate pipeline and notebooks
* No views or serverless catalogue for quick consumption
* Looping is in pipelines… performance will be bad
* Doesn’t tie to sql at all…. So not sure what is the problem with delta in bronze
* Parquet… bronze concurrency and read/write to file. Ideally won’t happen.. if does then it fails

# Reading options for ADLS delta lake to get into other relational engines

**Serverless**: For serverless options you’ll need a linked server to a serverless pool.. either their own domain synapse serverless or enterprise.. in this case I setup to enterprise serverless directly for the screenshots. But could be their own (domain) serverless.

* Pipeline copy activity from serverless enterprise or domain
* Dataflow from serverless enterprise or domain
* Notebook from serverless enterprise using jdbc

**Direct Delta:** A linked server to the adls storage acct. is needed. In screenshots below I setup ACLS to folder for vegman only folder locations in silver container of enterprise storage acct. This allows user to browse via synapse studio as well as read the data.

* Spark notebook direct sql or pyspark
* Dataflow using adls linked service from delta

### Linked Services setup for both direct delta and serverless options POC

A screenshot of a computer

Description automatically generated

**Security:** anyone with synapse domain workspace (vegman workspace) access would have access to read delta data for their folders / containers. This would be same for parquet or any file format. If wanting to exclude files or folders they should be in locations up the tree from where acls grants were given for users and synapse managed identity.

Note, when setting up ACLs… execute permissions are needed at each level in the tree. But read is only needed at the level for where the files/folders exist. Example:

Silver/ … the fe-d-syn-vegman has execute at root level of silver container.

Silver/esriveg01d and Silver/ssplcycle01d … fe-d-syn-vegman has read and execute. Make sure to set default to read and execute for those folders as well so any new files/folders underneath inherit those permissions. Last step if there are existing files/folders already… propagate acls to objects underneath (easy option in storage explorer).

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

### Pipeline copy activity from serverless

A screenshot of a computer

Description automatically generated

## Dataflow from serverless

A screenshot of a computer

Description automatically generated

### Spark notebook Jdbc to serverless

A screenshot of a computer

Description automatically generated

### Spark notebook direct sql

A screenshot of a computer

Description automatically generated

### Data flow from delta adls

A screenshot of a computer

Description automatically generated

# POC delta vs parquet performance testing

For this test we ingested data from sapshr\_DFKKOP into bronze and then silver using the following methodologies. This is roughly a 3 GB 1.6 million row table.

### Staging data into landing zone for 1.633M rows of sapshr\_DFKKOP (~12 min)

# A white screen with black text Description automatically generated

### Initialize table from parquet landing into bronze delta (~1 min)

# A screenshot of a computer Description automatically generated

### Initialize table in silver from delta into silver delta (~2min)

A screenshot of a computer

Description automatically generated

### Method 1: delta serverless copy activity to dedicated using bulk load (~8.5 min)

A screenshot of a computer

Description automatically generated

### Method 2: delta serverless copy activity to dedicated using polybase staging (~3.5 min)

A white background with black and white text

Description automatically generated

### Method 3: delta notebook to dedicated using sql pool adapter (~1.5 min… not including time to start spark session)

A screenshot of a computer

Description automatically generated

### Old method: adls parquet copy to dedicated using polybase (~2 min)

### A white background with black text Description automatically generated

|  |  |  |
| --- | --- | --- |
| **Method** | **Time taken / ratio** | **Comments** |
| Current Pipeline parquet polybase | 142 sec | Current method of getting data into dedicated sql pool from parquet |
| Serverless delta pipeline (bulk load) | 502 sec / 353.5 % | Bulk load is the slowest of all copies into dedicated. But started there. |
| Serverless delta pipeline (polybase load) | 214 sec / 150.7 % | This approach would be minimal to changing how things are currently working. |
| Spark notebook delta | 86 sec / 60.5 % | This does not include spark session startup. But if doing batch once session is started all tables will be processed in same session. This though would involve changing from looping in pipelines to looping in python. Also, this approach allows us to use delta as it was designed (cdf, versioning, etc.) |

### Writeback from dedicated using notebook (~40 sec without spark startup time

### A screenshot of a computer program Description automatically generated

### Writeback to adls parquet using copy activity from dedicated sql pool

A screenshot of a computer

Description automatically generated

### Writeback from dedicated using cetas external table (~20 seconds)

A screenshot of a computer

Description automatically generated

|  |  |  |
| --- | --- | --- |
| **Method** | **Time taken / ratio** | **Comments** |
| cetas (external table) to parquet | 21 sec | Internal data into adls parquet from dedicated sql pool |
| Copy activity to adls | 499 sec | Internal data into adls parquet from copy activity |
| Spark notebook read/write | 37 sec / 176.2 % | Read using spark sql and write using pyspark api into delta |

# Domain how to access enterprise data

## Login to portal and find synapse workspace

Navigate to browser and go to <https://portal.azure.com/> . You’ll be prompted to login. User your [sapid@fenetwork.com](mailto:sapid@fenetwork.com) username and password. In search bar type synapse and select “Azure Synapse Analytics”. From there you should see any synapse workspaces you have access to.

A screenshot of a computer

Description automatically generated

Select the appropriate workspace. And Open Synapse Studio link shown below.

A screenshot of a computer

Description automatically generated

## Demo notebook and pipelines

Find the demo notebook and pipelines created by default within your workspace. There should be 2 pipelines and 2 notebooks.

Pipelines:

A screenshot of a computer

Description automatically generated

These two pipelines are just calling the two notebooks and passing parameters in.

Notebooks:

A screenshot of a computer

Description automatically generated

These two notebooks show how within pyspark you would move data from adls delta into sql server instance. Or move sql server data into adls delta.

## Ad-hoc access via ADLS synapse serverless sql pool

Navigate to desired container in the synapse workspace. It should be present within the domain workspace as we’ve added a linked service using the synapse managed identity. You should see something like:

A screenshot of a computer

Description automatically generated

Silver is where your permissions for raw tables that have been ingested into Azure are located. Above shows a subfolder devsqlfarm and subfolders under that esriveg01d and ssplcycle01d. The path of the raw tables in silver is representing where the data came from. For instance, devsqlfarm is the sql server where database esriveg01d exists. Within the database exists tables/views. Under the esriveg01d folder in ADLS (Azure Data Lake Storage) exists another subfolder called “sde”. This is the schema in the database where those tables and views exist. Under sde are the tables.

A screenshot of a computer

Description automatically generated

If you right click on any table folder you’ll see a menu with the option for new sql script. Select top 100 rows will open a window with a selection of what type of file format is it.

A screenshot of a computer

Description automatically generated

Select “delta format“ for the file format.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

The code above is auto-generated. You can see it’s selecting top 100 from the path of the folder where you selected in ADLS. What this is doing is it is querying a table in ADLS using serverless sql pool ( <https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/on-demand-workspace-overview> ).

## Access via PowerBI using Domain Synapse Serverless Pool

First you’ll want to create views within your serverless sql pool over that ADLS delta formatted tables. To do this you can grab the script from doing the select top 100 option in ADLS as described in “Ad-hoc access via ADLS” above… Something like:

create view [db].[sde\_VEGTASKTREEOUTAGE\_EVW] as

SELECT \*

FROM OPENROWSET(

BULK 'https://fedsaentanalytics.dfs.core.windows.net/silver/devsqlfarm/ESRIVEG01D/sde/VEGTASKTREEOUTAGE\_EVW/',

FORMAT = 'DELTA'

) AS v

Some tables require stricter control around the data types that are returned from the delta tables. For instance, if a table contains a data type that is not a general data type (varchar,int,datetime,etc) like geometry or xml. Or maybe you want to adhere standards for how the data is formatted. Those strict types and formatting can be put into the view you define within serverless as well. Something like:

create view [db].[sde\_CORRIDOR\_EVW] as

SELECT [Shape], [GDB\_GEOMATTR\_DATA], convert(uniqueidentifier,[GlobalID]) GlobalID, [CORRIDORNAME], [VMSOPCO], [HISTCORRNAME], [CORRIDORDESCRIPTION], [CORRIDORMILEAGE], [SPANQUANTITY], [VMSID], [CORRIDORSEQID], [CREATED\_USER], [CREATED\_DATE], [LAST\_EDITED\_USER], convert(date,[LAST\_EDITED\_DATE]) [LAST\_EDITED\_DATE], [SHAPE\_AREA], [SHAPE\_LEN], [OBJECTID], [DEFAULTUSER]

from openrowset(bulk 'abfss://bronze@fedsaentanalytics.dfs.core.windows.net/devsqlfarm/ESRIVEG01D/sde/CORRIDOR\_EVW/', format='DELTA')

with ([Shape] varchar(max),[GDB\_GEOMATTR\_DATA] varbinary(max),[GlobalID] varchar(60),[CORRIDORNAME] varchar(max),[VMSOPCO] varchar(max),[HISTCORRNAME] varchar(max),[CORRIDORDESCRIPTION] varchar(max),[CORRIDORMILEAGE] decimal(38,8),[SPANQUANTITY] int,[VMSID] int,[CORRIDORSEQID] int,[CREATED\_USER] varchar(max),[CREATED\_DATE] datetime2,[LAST\_EDITED\_USER] varchar(max),[LAST\_EDITED\_DATE] datetime2,[SHAPE\_AREA] decimal(38,8),[SHAPE\_LEN] decimal(38,8),[OBJECTID] int,[DEFAULTUSER] varchar(max)) as v;

Or maybe you want to filter like so:

create view [db].[sde\_CORRIDOR\_EVW] as

SELECT [Shape], [GDB\_GEOMATTR\_DATA], convert(uniqueidentifier,[GlobalID]) GlobalID, [CORRIDORNAME], [VMSOPCO], [HISTCORRNAME], [CORRIDORDESCRIPTION], [CORRIDORMILEAGE], [SPANQUANTITY], [VMSID], [CORRIDORSEQID], [CREATED\_USER], [CREATED\_DATE], [LAST\_EDITED\_USER], convert(date,[LAST\_EDITED\_DATE]) [LAST\_EDITED\_DATE], [SHAPE\_AREA], [SHAPE\_LEN], [OBJECTID], [DEFAULTUSER]

from openrowset(bulk 'abfss://bronze@fedsaentanalytics.dfs.core.windows.net/devsqlfarm/ESRIVEG01D/sde/CORRIDOR\_EVW/', format='DELTA')

with ([Shape] varchar(max),[GDB\_GEOMATTR\_DATA] varbinary(max),[GlobalID] varchar(60),[CORRIDORNAME] varchar(max),[VMSOPCO] varchar(max),[HISTCORRNAME] varchar(max),[CORRIDORDESCRIPTION] varchar(max),[CORRIDORMILEAGE] decimal(38,8),[SPANQUANTITY] int,[VMSID] int,[CORRIDORSEQID] int,[CREATED\_USER] varchar(max),[CREATED\_DATE] datetime2,[LAST\_EDITED\_USER] varchar(max),[LAST\_EDITED\_DATE] datetime2,[SHAPE\_AREA] decimal(38,8),[SHAPE\_LEN] decimal(38,8),[OBJECTID] int,[DEFAULTUSER] varchar(max)) as v

where year([LAST\_EDITED\_DATE])>=2020;

Once your happy with the views you created over the delta table in the serverless sql pool. You can now begin to access the data and model it within the reporting later of power bi. To do that open power bi desktop select “Get data”

A screenshot of a computer

Description automatically generated

Navigate to Azure and Azure Synapse Analytics SQL.

A screenshot of a computer

Description automatically generated

Enter your serverless sql endpoint address. If your unsure what this is. You’ll see it within the synapse portal under manage tab (briefcase icon) and selecting “built-in”. This should then open a window with the endpoint address. Like, fe-d-syn-enterpriseanalytics-ondemand.sql.azuresynapse.net. It’s always going to be your synapse workspace name + ondemand.sql.azuresynapse.net. You’ll never really want to use direct query with ADLS delta table (bad performance). You’ll want to import and create a dataset within powerbi. If direct query is needed some data engineering will have to occur to get it within another relational database engine like dedicated sql pool or Azure SQL Database.

A screenshot of a computer

Description automatically generated

Authenticate to serverless sql pool via Microsoft account (your sapid@fenetwork.com)

A screenshot of a computer

Description automatically generated

Select tables to import and click load.

A screenshot of a computer

Description automatically generated

Now start your powerbi development.

## Access via PowerBI using SQL Server

A screenshot of a computer

Description automatically generated

Click get data…

A screenshot of a computer

Description automatically generated

Navigate to Azure->Azure SQL Database

A screenshot of a computer

Description automatically generated

Enter your server name for your sql server database. You can use direct query if you want to query each time the report is run (data as current as what’s in the database… but could be slower if complex model) or import (create dataset within power bi).

A screenshot of a computer

Description automatically generated

Login using Microsoft [sapid@fenetwork.com](mailto:sapid@fenetwork.com) account

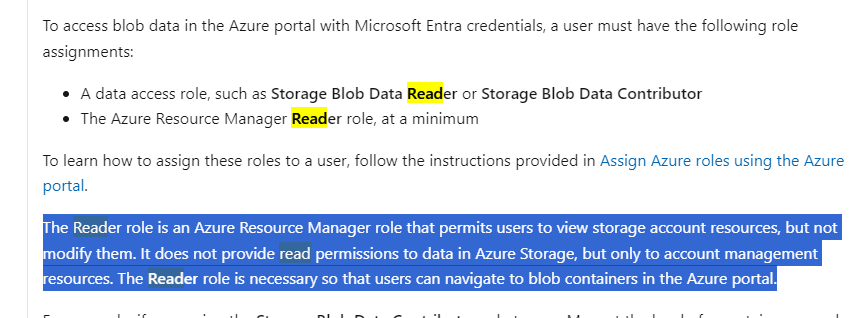
A screenshot of a computer

Description automatically generated

Select tables for inclusion in power bi model and begin development of report.

# Security For Domains access to Enterprise ADLS

First the group must be given access to the storage account via the “Reader” role. This will grant them access to list items only…



Then user/group must have execute to container and any other paths above where they need read access.

# A screenshot of a computer Description automatically generated

Finally, folder below is where read access is given and contains vgms delta tables.

A screenshot of a computer

Description automatically generated

**Qlik to PBI Project Handoff from Earnst & Young**

A document per line of business. Contents at the report level for that line of business:

* primary business contact
* report name
* document qlik report location
* document qlik qvd’s and qvw’s involved
* report status (complete, uat, in flight, not started, sunset)
* if complete, uat, or in flight
  + inventory of raw source systems (secrets) and tables
  + inventory of consumption tables/views and transformation steps involved
  + SLA
  + Pbi datasets
  + Pbi report names
  + Pbi direct query tables involved
* If not started
  + Complexity
  + inventory of raw source systems (secrets) and tables

A screenshot of a computer screen

Description automatically generated

# Synapse Integrator Framework

There are two frameworks that are under development. Both are in beta as of 1/2/2024.

Ingestion framework takes data from on prem and moves it to azure. It consists of two main tables; fe\_config\_ingest and fe\_config\_ingest\_part. The \_part table doesn’t ever require to be touched. It is more of a log table to document and create the partitions/shards of a table per how it is configured in fe\_config\_ingest. There are two log tables that contain all key watermark date columns (fe\_config\_ingest\_log and fe\_config\_ingest\_part\_log). Sources currently available to load from are mssql (Sql Server), oracle, teradata, ss (singlestore), and excel. Data can be landed in either ADLS landing zone (temp) or DW staging table (dedicated sql pool).

Transformation framework takes data already in azure and transforms it. It consists of two main tables; fe\_config\_transform and fe\_config\_transform\_ref. There’s one log table that contain all key watermark date columns (fe\_config\_transfom\_log). The framework currently has the ability to use spark sql to materialize delta tables or t-sql (stored procs) to materialize dedicated sql pool tables.

## Current Framework ToDos

* + add dynamic annotations to pipelines for filtering of triggered runs.. see

https://learn.microsoft.com/en-us/azure/data-factory/concepts-annotations-user-properties   
https://stackoverflow.com/questions/62929329/add-dynamic-content-to-annotations-in-azure-data-factory

* + Create policy for X day retention in temp container... Bob’s team needs to set this up so data gets deleted out of storage container after so many days
  + Implement parameterized ingestions… both query driven and static. Was going to create a parameter table (az\_id, param\_name, param\_value, query\_driven\_flag) and then do string replace in query being passed (‘{@param\_name}’). Query driven make another table with queries in it… (az\_id, source\_system\_conn\_str, query).
  + Implement parameterized transformations… both query driven and static. Was going to create a parameter table (az\_id, param\_name, param\_value, query\_driven\_flag) and then do string replace in query being passed (‘{@param\_name}’). Query driven make another table with queries in it… (az\_id, source\_system\_conn\_str, query).
  + Test flyway for versioning dedicated sql pool tables schema versioning <https://flywaydb.org/>
    - Instead of using Microsoft database projects… cause then can use for any database vendor
  + If ingestion to parquet is empty … remove file so not needing to process into delta. So add functionality in nb\_ingestion\_by\_config. See comment in notebook “TO DO: check file size”
  + Build out src\_col\_def for use where schema isn’t known at all… ie. Excel file with no header… meaning you don’t know column names. Then allow for a config to have column definitions in the config. Would add a new column much like the src\_col\_def column already in table (json field with col\_name and col\_type)
  + Log analytics enabled for dedicated sql pool, spark pool, and serverless. Get with Bob’s team to implement log analytics.
    - <https://learn.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-azure-log-analytics>
    - <https://bradleyschacht.com/log-analytics-with-dedicated-sql-pools-formerly-sql-dw/>
  + In frameworks add column to config for ability to set spark session (# of executors) and priority. Then in pipelines that call looping notebook in both transform and ingest (nb\_transform\_by\_config and nb\_ingest\_by\_config) … add parameter in pipeline and make that parameter drive those execution / session settings
  + In transform … add logic to detect if ref tables have changed and if it actually needs run based on timestamps in transform config.. see “TO DO: add one more watermark column” in nb\_transform\_by\_config
  + Devops alert for when moving into master from feature branch to alert team they need to pull latest into their feature branch
  + Devops release pipeline for moving to production.
  + Devops release pipeline for dedicated sql pool objects
  + Devops release pipeline for config settings (transform and ingest delta tables) changes
  + Devops release pipeline for only when transformation framework changes (transform folders for both
  + pipelines and notebooks; as well as common utility functions notebook) to domains that use it
  + Add force\_init timestamp in \_log delta tables so as to be able to insert a row into framework watermark timestamps… and then add logic to framework if present then delete table and start over as if dw\_last\_load and/or dt\_last\_delta\_init is not present
  + So can effectively schedule data quality tests… Create a testing framework that runs queries that are in a config file against dw or delta expecting 0 rows… otherwise throws an error. First addition would be to check for any primary key violations… see “TO DO: if primary key“ in nb\_parquet\_to\_delta and nb\_delta\_to\_delta
  + Implement z-order in transformation framework and other delta optimizations (vacuum, optimize, etc). See “TO DO: add logic to do optimization” in both nb\_delta\_to\_delta and nb\_parquet\_to\_delta
  + Purview integration into framework using atlas api to get lineage of ingestion and transformation

### Modes

* truncate+fill
  + Deletes all data from the table using overwrite. Inserts the rest of the data from source. NOTE: if table is partitioned this option will result in only 1 partition being saved if source is also partitioned. If truncate+fill was the mode in the config and set partition\_by… then it auto-switches mode to partition+overwrite. As that wouldn’t delete the entire table… it would only remove the partition the source had in it.
* upsert
  + This will insert and update rows based on the primary key. The key must exist for this mode. If table is partitioned the key/join on target includes the partition keys to improve performance of the upsert.
* del+insert
  + This mode will use the config setting del\_filter to first delete the data from delta table give a filter like (“year=2023”). And then will insert the source data. Believe this was called overlapping insert in memmapper.
* Append
  + Just an insert of source columns to table…
* partition+overwrite
  + This will overwrite a partition within the delta table. So, if source data includes year=2024 then it will overwrite the “folder” under delta table with contents of source data. This option could be leveraged also for snapshot information. Meaning if you add a date to source data of when it was pulled. You could partition on the date and each day overwrite that partition.
* Merge
  + Merge is not complete… the version of pyspark that synapse is on does not have the ability to run whennotmatchdelete. Or sql when not match by source delete… work around will be coding in the mode to do the upsert logic + a delete where not exists.

## Ingestion

A diagram of a diagram of a diagram

Description automatically generated

### Pipelines

**pipe\_ingest\_fe\_data\_00:** creates run\_id for particular run and calls pipe\_ingest\_fe\_data\_00\_run

parameters:

* config\_db\_name: name of serverless sql pool database that contains config tables / views
* temp\_container: name of container in adls to load data temporarily during run (staging area)
* temp\_root: name of folder/path to where data should be landed in temp\_container
* config\_name: the config\_name in fe\_config\_ingest to be apart of this ingestion run
* tag: the tag in fe\_config\_ingest to be apart of this ingestion run

activities:

* create run\_id: creates a unique identifier that stays with this run of the ingestion framework.
* Run ingest: calls \_00\_run pipeline using parameters

**pipe\_ingest\_fe\_data\_00\_run:** takes a snapshot of config items for tag and config\_name passed in. Loops through those items in config to pull data into temporary staging area in adls and then subsequently call pipeline to push into delta or dw (dedicated sql pool).

parameters:

* config\_db\_name: name of serverless sql pool database that contains config tables / views
* temp\_container: name of container in adls to load data temporarily during run (staging area)
* temp\_root: name of folder/path to where data should be landed in temp\_container
* config\_name: the config\_name in fe\_config\_ingest to be apart of this ingestion run
* tag: the tag in fe\_config\_ingest to be apart of this ingestion run
* run\_id: the unique identifier for the particular run of the ingestion pipeline (created in step \_00)

activities

* copy fe\_config\_ingest: copies table in serverless sql pool out to json adls file for config\_name and tag
* copy fe\_config\_ingest\_part: copies table in serverless sql pool out to json adls file for config\_name and tag
* copy vw\_fe\_config\_ingest: copies table in serverless sql pool out to json adls file for config\_name and tag
* copy vw\_fe\_config\_ingest\_part: copies table in serverless sql pool out to json adls file for config\_name and tag
* Lookup config: reads the json that was just written from “copy vw\_fe\_config\_ingest” (00\_vw\_fe\_config\_ingest.json)
* ForEachTableInConfig: loop throw lookup config items
  + Call \_01\_get\_parts
  + Call \_02\_parts\_to\_parquet

**pipe\_ingest\_fe\_data\_01\_get\_parts:** creates a json file for the az\_id that is passed in that contains the distinct partition for source

parameters:

* az\_id: the unique string for the ingestion config item
* config\_db\_name: name of serverless sql pool database that contains config tables / views
* temp\_container: name of container in adls to load data temporarily during run (staging area)
* temp\_root: name of folder/path to where data should be landed in temp\_container
* config\_name: the config\_name in fe\_config\_ingest to be apart of this ingestion run
* tag: the tag in fe\_config\_ingest to be apart of this ingestion run
* run\_id: the unique identifier for the particular run of the ingestion pipeline (created in step \_00)

activities

* Lookup config file: reads the json that was just written from pipe\_ingest\_fe\_data\_00\_run (00\_vw\_fe\_config\_ingest.json)
* Lookup config: filters down previous “Lookup config file” down to just the particular az\_id
* Switch on source for partitioning:
  + Copy dummy data: activity that creates the 1=1 row for those config items with no partitioning
  + Copy mssql: load each distinct partition from source sql server system in a json file. File will be put in run\_id folder with other temp items. 01\_{az\_id}\_parts.json will be the file.
  + Copy ss: load each distinct partition from source singlestore system in a json file. File will be put in run\_id folder with other temp items. 01\_{az\_id}\_parts.json will be the file.
  + Copy oracle: load each distinct partition from source oracle system in a json file. File will be put in run\_id folder with other temp items. 01\_{az\_id}\_parts.json will be the file.
  + Copy teradata: load each distinct partition from source teradata system in a json file. File will be put in run\_id folder with other temp items. 01\_{az\_id}\_parts.json will be the file.

**pipe\_ingest\_fe\_data\_02\_parts\_to\_parquet:** takes previously written json file with all distinct partitions and loops through them. Calling each query/partition and writing to /data subfolder in temp run id adls location.

parameters:

* az\_id: the unique string for the ingestion config item
* config\_db\_name: name of serverless sql pool database that contains config tables / views
* temp\_container: name of container in adls to load data temporarily during run (staging area)
* temp\_root: name of folder/path to where data should be landed in temp\_container
* config\_name: the config\_name in fe\_config\_ingest to be apart of this ingestion run
* tag: the tag in fe\_config\_ingest to be apart of this ingestion run
* run\_id: the unique identifier for the particular run of the ingestion pipeline (created in step \_00)

activities

* Get config data: from serverless sql pool… submit a query against the json created in \_01\_get\_parts. It combines that json file with other information to create the query with the where partition predicate needed. Creates a 02\_{az\_id}\_parts.json file containing this query and other configuration settings.
* Lookup config: reads json file created in “Get config data”
* ForEach config part:
  + Switch on source type:
    - query\_ss: submits either \_init or \_inc query to singlestore for particular partition. Writes to parquet file in run id “data” subfolder with az\_id in name.
    - table\_ss: submits query to singlestore for particular table. Writes to parquet file in run id “data” subfolder with az\_id in name.
    - query\_mssql: submits either \_init or \_inc query to sql server for particular partition. Writes to parquet file in run id “data” subfolder with az\_id in name.
    - table\_mssql: submits query to sql server for particular table. Writes to parquet file in run id “data” subfolder with az\_id in name.
    - table\_excel: Writes to parquet file for excel doc and sheet within configuration. Writes to parquet file in run id “data” subfolder with az\_id in name.
    - query\_oracle: submits either \_init or \_inc query to oracle for particular partition. Writes to parquet file in run id “data” subfolder with az\_id in name.
    - table\_oracle: submits query to oracle for particular table. Writes to parquet file in run id “data” subfolder with az\_id in name
    - query\_teradata: submits either \_init or \_inc query to teradata for particular partition. Writes to parquet file in run id “data” subfolder with az\_id in name.
    - table\_teradata: submits query to teradata for particular table. Writes to parquet file in run id “data” subfolder with az\_id in name.
  + NOTE: decisions on if using \_init or \_inc query are made on weather or not dt\_part\_last\_load exists within log table (fe\_config\_ingest\_log)
* Log success of copy: copies \_parts.json file for az\_id and creates new one with \_SUCCESS appended to end of file name.

**pipe\_ingest\_fe\_data\_03\_parquet\_to\_delta:** poorly named since no longer does this just write to delta. It now will write to dw (dedicated sql pool too). But this pipeline just calls a notebook that will loop through config\_name and tag within spark to eventually get it where it needs to go.

parameters:

* az\_id: the unique string for the ingestion config item
* config\_db\_name: name of serverless sql pool database that contains config tables / views
* temp\_container: name of container in adls to load data temporarily during run (staging area)
* temp\_root: name of folder/path to where data should be landed in temp\_container
* config\_name: the config\_name in fe\_config\_ingest to be apart of this ingestion run
* tag: the tag in fe\_config\_ingest to be apart of this ingestion run
* run\_id: the unique identifier for the particular run of the ingestion pipeline (created in step \_00)

activities

* source parquet to delta: calls nb\_ingestion\_by\_config

Notebooks

**nb\_ingestion\_by\_config**

Code Cells

**nb\_parquet\_to\_delta**

Code Cells

**nb\_parquet\_to\_dw**

Code Cells

### Tables

**fe\_config\_ingest** (Ingestion Config Table)

* uuid: unique Identifier for config object
* config\_name: string to signify grouping of config
* src\_type: mssql, oracle, ss, excel, teradata
* src\_db\_name: string that will determine where the table lives in ADLS as well as be part of the az\_id
* src\_schema\_name: string that will determine where the table lives in ADLS as well as be part of the az\_id
* src\_table\_name: string that will determine where the table lives in ADLS as well as be part of the az\_id
* src\_col\_def: json string that assists with data type conversion for consumption view that is created for each delta table. Ex. [{"col\_name":"some\_col", "sql\_type":"varchar(max)"}]
* src\_query\_init: sql query that is run only when the table hasn’t been initialized within the framework
* src\_query\_inc: sql query that is run after table has been initialized… if null then defaults back to \_init query
* src\_conn\_secret\_name: secret name in KeyVault that is used to create connection to source
* dest\_delta\_table\_container: name of container where the destination delta table will reside
* temp\_stage\_container: name of temporary / staging container is for ingestion of data into Azure adls
* mode: truncate+fill, upsert, del+insert, append, partition+overwrite, merge
* inc\_del\_filter: string with sql filter for del+insert mode only
* primary\_key: comma delimited string of primary key of table so that merge operations can be done using these columns as the predicate
* partition\_by: comma delimited string of how table should be partitioned in adls. If defined this will create a row for each DISTINCT “part” in the fe\_config\_ingest\_part table. If not defined a “stub row” will be put into fe\_config\_ingest\_part where 1=1.
* tag: magic string to filter down config\_name while running framework
* src\_file\_container: used for excel or flat files to be used as source… it’s where the file will be picked up and read from adls
* src\_file\_directory: used for excel or flat files to be used as source… it’s where the file will be picked up and read from adls
* src\_file\_name: used for excel or flat files to be used as source… it’s where the file will be picked up and read from adls
* excel\_sheet\_name: which sheet to use if excel
* has\_header: flat file has header row?
* rows\_to\_skip: where to start consuming data from flat file
* is\_disabled: disable the config item/object from running
* dw\_schema\_name: the dedicated sql pool sql tables schema where data will be staged
* dw\_table\_name: the dedicated sql pool sql table you want the data appended to… if it doesn’t exist it will create table. This table is append only. It will include run id guid for specific run and it will include the time at which it was inserted.

**fe\_config\_ingest\_part** (partition table)

* uuid: Unique Identifier for partition config
* az\_id: the az\_id for the fe\_config\_ingest row/object
* part\_filter: value for distinct partition… this will be 1=1 if no partition is defined in config. If partition defined for say “year” column, then you’d see a row for each year in source table.
* is\_disabled: disable partition from being loaded

**fe\_config\_ingest\_log** (ingestion log table)

* uuid: Unique Identifier for log entry
* az\_id: the az\_id for the fe\_config\_ingest row/object
* op: the operation that was completed for logged timestamp.
  + dt\_last\_source\_init: time data was pulled for init query of source system
  + dt\_last\_delta\_init: time data was pushed into delta table using init query… if null then this makes the init query run. If NOT null this makes the inc query run if there is one.
  + dt\_last\_delta\_inc: time data was pushed into delta table using inc query
  + dt\_last\_dw\_load: time data was pushed into dedicated sql pool
  + error: time error was given
* ts: timestamp for the completed operation
* msg: text of operation for logging purposes. Currently only used for logging error text

**fe\_config\_ingest\_part\_log** (ingestion partition log table)

* uuid: Unique Identifier for log entry
* az\_id: the az\_id for the fe\_config\_ingest row/object
* op: the operation that was completed for logged timestamp.
  + dt\_part\_created: time when partition was created in log table
  + dt\_part\_last\_seen: time partition data was last “seen” in source table
  + dt\_part\_last\_load: time partition data was pushed into delta table
  + dt\_part\_dw\_last\_load: time partition data was pushed into dedicated sql pool table
* ts: timestamp for the completed operation
* msg: text of operation for logging purposes. Currently only used for logging error text

## Transformation

A diagram of a diagram of a diagram

Description automatically generated

### Pipelines

**pipe\_transform\_fe\_data\_00:** pipeline that calls a notebook to kickoff a config / tag transformation.

parameters:

* config\_name: passed to notebook to filter down config rows via fe\_config\_transform.config\_name
* tag: passed to notebook to filter down config rows via fe\_config\_transform.tag

activities:

* run transform config: calls notebook nb\_transform\_by\_config passing config\_name and tag

### Notebooks

**nb\_transform\_by\_config**: notebook that loops through configs passed in via parameters.

Code Cells

1. parameter cell; which if in same enterprise analytics synapse workspace only config\_name and tag need to change
2. uses networkx python package to build a dag based on fe\_config\_transform\_ref and fe\_config\_transform config tables. This is to determine what can be batched together and in what order things need to be run.
3. function definitions
   1. run\_load: takes one config row at a time and determines whether it is a delta transform or a dw transform.
   2. run\_dw\_to\_dw\_proc: takes one config row and calls a stored procedure to be run to do transformation step(s) within dedicated sql pool.
   3. run\_delta\_to\_delta: takes on config row and calls a notebook to do delta transformation (nb\_delta\_to\_delta\_transform)
4. Enumerates through config objects in parallel if possible, depending on code cell 2 dag steps (calling run\_load).

**nb\_delta\_to\_delta\_transform**: does delta to delta transformation based on parameters being passed in

Code Cells

1. parameter cell;
   1. sql: is spark sql statement that creates the transformation materialization
   2. storage\_account: shouldn’t need changed if in same synapse workspace. But is used to create path to delta table if parent\_full\_delta\_path isn’t used in configuration.
   3. dest\_table\_name: is used for creating serverless pool view for delta table
   4. dest\_schema\_name: is used to create/add table schema to serverless sql pool for delta table view;
   5. table\_container: is used to determine which serverless sql pool database to put the view overtop of delta table; as well as is where materialized transformed delta table will be created in adls
   6. mode: determines how to apply the data to delta table (ex. truncate+fill, append, etc.)
   7. init\_flag: if true will re/create view and overwrite schema (delta table should maintain history even if already exists)
   8. partition\_by: used to partition delta table by this comma delimited string
   9. table\_path: used as path for materialized delta transformed table in adls
   10. primary\_key: used if mode is merge for instance incrementally load delta table
   11. view\_col\_config: used to change serverless sql pool default data types from spark to sql
   12. refs: json string containing queries table reference information… for instance it will check the sql for az\_id/parent\_id and replace it with parent\_table\_path so a delta.`{path}` can be derived and ran.
2. import nb\_framework\_common definitions
3. define generate\_delta\_abfss\_path function to be applied to refs dataframe (refs json passed in)
4. create refs dataframe based on json passed in. add column parent\_delta\_abfss by calling generate\_delta\_abfss\_path. Then loop through each ref table/object and create a temporary view overtop of that path/delta table. If a last\_delta\_version is apart of the configuration then read that version of the table instead of the default current version (this was for using change data feed… but has not been fully implemented).
5. Check parameters that were passed in or generated with code prior to this cell to make sure we are good to go.
6. If init\_flag then call common notebooks initialize\_delta\_table function to init delta table and create serverless sql pool view.
7. If no init\_flag then call common notebooks incremental\_delta\_table function to incrementally load based on the mode in config.

**nb\_dw\_proc\_transform:** calls dedicated sql pool stored procedure

Code Cell

1. Parameter cell;
   1. sql: t/sql that will run on dedicated sql pool
   2. dest\_dw\_db: shouldn’t need changed if running in current dev synapse workspace, but is database where sql should be run
   3. dest\_dw\_endpoint: is the dedicated sql pool connection endpoint
2. import of nb\_framework\_common notebook definitions
3. gets dedicated sql pool connection and runs sql passed in on it

### Tables

**fe\_config\_transform** (transform config table)

* uuid: unique Identifier for config object
* config\_name: string to signify grouping of config
* dest\_delta\_container: adls container to write delta table to
* dest\_delta\_table\_path: adls path to write delta table to… if null then assumes it’s a stored proc that needs running for dedicated sql pool
* dest\_delta\_schema\_name: is used for the database name in serverless sql pool view
* dest\_delta\_table\_name: is used for view name in serverless sql pool
* dest\_delta\_table\_primary\_key: comma delimited string of primary key of delta table so that merge operations can be done using these columns as the predicate
* dest\_delta\_table\_partition\_by: comma delimited string of how table should be partitioned in adls
* dest\_delta\_table\_z\_order: NOT USED CURRENTLY, plan is to include this though for performance of delta table
* sql\_transform\_init: if delta transformation it should be the spark sql to initialize the table. That spark sql should include az\_id for synapse workspace config item as the table name. For instance, select \* from some\_az\_id would resolve to select \* from delta.`container\path\where\delta\az\_id\lives`. If az\_id is outside of synapse workspace that delta.` container\path\where\delta\az\_id\lives` can be hard coded. Or in fe\_config\_transform\_ref table. The column parent\_az\_id can include the abfss:// path. If dedicated sql pool (dw) transformation. This is the stored procedure or t-sql needed for initial load.
* sql\_transform\_inc: if delta transformation it should be the spark sql to incrementally load the table. The same comments above about sql\_transform\_init (about az\_id and table path) remain. If dedicated sql pool (dw) transformation. This is the stored procedure or t-sql needed for incremental load.
* src\_col\_def: json string that assists with data type conversion for consumption view that is created for each delta table. Ex. [{"col\_name":"some\_col", "sql\_type":"varchar(max)"}]
* tag: magic string to filter down config\_name while running framework
* mode: truncate+fill, upsert, del+insert, append, partition+overwrite
* dest\_dw\_endpoint: dedicated sql pool server endpoint… if null then assumes it’s a delta / spark transformation
* dest\_dw\_db: dedicated sql pool database name
* is\_disabled: disable the config item/object from running

**fe\_config\_transform\_ref** (transform parent child table references)

* uuid: unique Identifier for reference config object
* parent\_az\_id
* az\_id
* last\_delta\_version
* is\_disabled: disable the config item/object from running

**fe\_config\_transform\_log** (transform config log table)

* uuid: unique Identifier for log entry
* az\_id: the az\_id for the fe\_config\_transform row/object
* op: the operation that was completed for logged timestamp.
  + dt\_last\_transform\_init: time data initialized with sql\_transform\_init into delta table
  + dt\_last\_transform\_inc: time data incrementally loaded with sql\_transform\_inc into delta table
  + dt\_dw\_last\_transform\_init: time data initialized with sql\_transform\_init into dedicated sql pool
  + dt\_dw\_last\_transform\_inc: time data incrementally loaded with sql\_transform\_inc into dedicated sql pool
  + error: time error was given
* ts: timestamp for the completed operation
* msg: text of operation for logging purposes. Currently only used for logging error text

## Utility Notebooks

**nb\_framework\_common**: utility notebook that contains many common functions between both frameworks

Functions

**ls\_files\_to\_dataframe**: returns dataframe (path, name, size) for files in directory

* + - path = string abfss path
    - search\_pattern = string regex search pattern
    - match\_case = bool for regex search is it case sensitive

**get\_serverless\_synapse\_conn**: returns connection string and token for serverless sql pool. NOTE: bad name that never got to redo. Not just serverless… can and is used for dedicated too.

* server = string sql endpoint
* dbname = string sql database name

**run\_serverless\_sql**: runs sql against serverless sql pool. NOTE: bad name that never got to redo. Not just serverless… can and is used for dedicated too.

* sql = string for sql
* cnnstr = string the odbc conn string
* token = struct the auth token for serverless sql pool
* server = string serverless sql endpoint
* dbname = string serverless sql database name

**get\_schema\_info:** gets tracked\_cols (name, type) and partitionBy (list of partition column names)

* source\_df = spark dataframe containing data
* view\_col\_def = string valid json containing sql type to create in view
* partition\_by = string comma delimited of partition columns

**initialize\_delta\_table**: initialize a delta table... create a view in serverless sql pool to catalog

* source\_df = the spark dataframe containing data to be loaded into delta
* delta\_table\_path = string abfss path to delta table to write to
* parquet\_file\_path = string abfss path to parquet source... added to meta data of commit
* dest\_db\_name = string name of database in serverless sql pool
* dest\_schema\_name = string name of schema in serverless sql pool
* dest\_view\_name = string name of view in serverless sql pool
* tracked\_cols = pandas dataframe with all columns that are in source table plus a few others
* partitionBy = list of partition columns

**incremental\_delta\_table**: incrementally load a delta table based on mode

* mode = append, partition+overwrite, merge, upsert, truncate+fill, del+insert
* source\_df = the spark dataframe containing data to be loaded into delta
* delta\_table\_path = string abfss path to delta table to write to
* commit\_meta\_data = string to add meta data for commit.. eg. abfss path to parquet source...
* tracked\_cols = pandas dataframe with all columns that are in source table plus a few others
* keys = array of primary keys
* partitionBy = list of partition columns
* del\_filter = deletion filter predicate only used if mode is del+insert

**fn\_table**: databricks like function for table\_changes... which isn't implemented in synapse. gives spark dataframe for version passed in... or just base table if that passed in. creates temp view to be referenced in spark sql.

* tablename = can be managed or external table name... or in format of delta.`<abfss path>`
* temp\_view\_name = name of temporary view to create in spark session for table
* list\_from\_to\_version = list of numbered or or timestamps ... item 0 from and item 1 to
* nb\_framework\_ingest\_import

nb\_framework\_ingest\_import – this imports config file (.xlsx) into fe\_config\_ingest

nb\_framework\_transform\_import – this imports config file (.xlsx) into fe\_config\_transform

nb\_init\_frameworks – used to initialize the framework tables for both transform and ingest frameworks. As well as it creates the necessary views overtop of those tables… in both spark catalog and serverless sql pool.

Installing Framework in Synapse Workspace

Dev Branching and Pull Requests

Releasing to Prod