Open Ended Capstone – Market Research

Yeh, Margaret

Dataset

American Community Survey 5-Year Data (2009-2022)

- https://www.census.gov/data/developers/data-sets/acs-5year.html
- https://api.census.gov/data/2022/acs/acs5/profile/variables.html

Groups: https://censusreporter.org/topics/table-codes/

- Group DP02 – Citizenship, Education, Household composition
- Group DP03 - Selected Economic Characteristics: County level economic characteristics, income of various households, retirement, social security, health insurance
- Group DP04 – Rent and Housing information
- Group S0804 – Means of Transportation to Work by Selected Characteristics for Workplace Geography
- Group S0701 – Geographic Mobility by Selected Characteristics in the United States
- Group S2303 – Work status in the past 12 months
- Group CP03 - Comparative Economic Characteristics

Target Questions to Answer

- Where would it be good to invest a new retail business?
  - Locations of rapid increase in income, increased people moving in, consider housing costs. Employment statistics
- Where are the areas that can benefit from improved public transport?
  - Locations of high population and lots of people commuting by car.
- Is there anything interesting that can correlate with increasing or decreasing housing prices?

Data Exploration

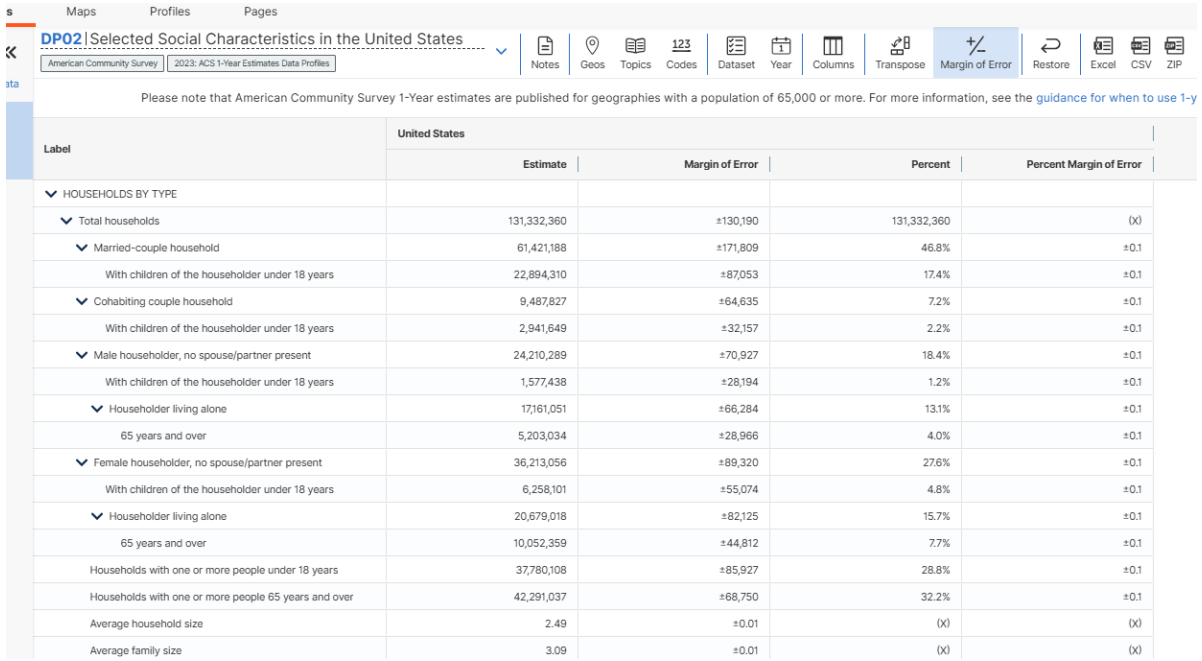
- For some groups, there are years that the API request failed to get. It seems to happen at random
- Sample image of group DP03 (Rent and Housing information)

[12]:

	DP03_0001E	DP03_0001EA	DP03_0001M	DP03_0001MA	DP03_0001PE	DP03_0001PEA	DP03_0001PM	DP03_0001PMA	DP03_0002E	DP03_0002EA	...	DP03_(
0	1367247	None	837	None	1367247	None	-888888888	(X)	922490	None	...	-888
1	1084	None	187	None	1084	None	-888888888	(X)	575	None	...	-888
2	34850	None	147	None	34850	None	-888888888	(X)	16238	None	...	-888
3	178693	None	318	None	178693	None	-888888888	(X)	103769	None	...	-888

Data Exploration

- Data.census.gov hosts their data and shows some of the variables in table format on their website. It is useful for initial exploration and checking for interesting datasets to add. For example, for the group DP02: https://data.census.gov/table/ACSDP1Y2023.DP02?q=dp02



Maps Profiles Pages

DP02 | Selected Social Characteristics in the United States

American Community Survey | 2023: ACS 1-Year Estimates Data Profiles

Notes | Geos | Topics | Codes | Dataset | Year | Columns | Transpose | Margin of Error | Restore | Excel | CSV | ZIP

Please note that American Community Survey 1-Year estimates are published for geographies with a population of 65,000 or more. For more information, see the guidance for when to use 1-y

Label	United States			
	Estimate	Margin of Error	Percent	Percent Margin of Error
HOUSEHOLDS BY TYPE				
Total households	131,332,360	±130,190	131,332,360	(X)
Married-couple household	61,421,188	±171,809	46.8%	±0.1
With children of the householder under 18 years	22,894,310	±87,053	17.4%	±0.1
Cohabiting couple household	9,487,827	±64,635	7.2%	±0.1
With children of the householder under 18 years	2,941,649	±32,157	2.2%	±0.1
Male householder, no spouse/partner present	24,210,289	±70,927	18.4%	±0.1
With children of the householder under 18 years	1,577,438	±28,194	1.2%	±0.1
Householder living alone	17,161,051	±66,284	13.1%	±0.1
65 years and over	5,203,034	±28,966	4.0%	±0.1
Female householder, no spouse/partner present	36,213,056	±89,320	27.6%	±0.1
With children of the householder under 18 years	6,258,101	±55,074	4.8%	±0.1
Householder living alone	20,679,018	±82,125	15.7%	±0.1
65 years and over	10,052,359	±44,812	7.7%	±0.1
Households with one or more people under 18 years	37,780,108	±85,927	28.8%	±0.1
Households with one or more people 65 years and over	42,291,037	±68,750	32.2%	±0.1
Average household size	2.49	±0.01	(X)	(X)
Average family size	3.09	±0.01	(X)	(X)

Data Exploration

- There are some missing columns, and the headers are coded. Descriptions of the codes are included in the dataset documentation but not in the dataset itself
- 1. Is the data homogenous in each column?
  - Yes, within the column. If there are empty values, then the whole column is empty.
- 2. How do you anticipate this data will be used by data analysts and scientists downstream?
  - Data analysts can draw correlations between the data of different groups. For example, with the population group, people can separate out demographics in each county and see how they correlate with the target they are studying.
- 3. Does your answer to the last question give you an indication of how you can store the data for optimal querying speed and storage file compression?
  - It hints that people might want to be able to access datasets from multiple groups at the same time. Maybe storing with a cloud service and using cloud analytics is the best method of optimizing both the speed and storage.
- 4. What cleaning steps do you need to perform to make your dataset ready for consumption?
  - Removal of empty columns and columns that provide no information. Replace the coded headers or put the description of the coded headers somewhere it is easy to look up.
- 5. What wrangling steps do you need to perform to enrich your dataset with additional information?
  - The datasets can be joined so that not so many files need to processed. Maybe in each group, join all the years into one table.

Data Exploration: Entity-Relationship Model

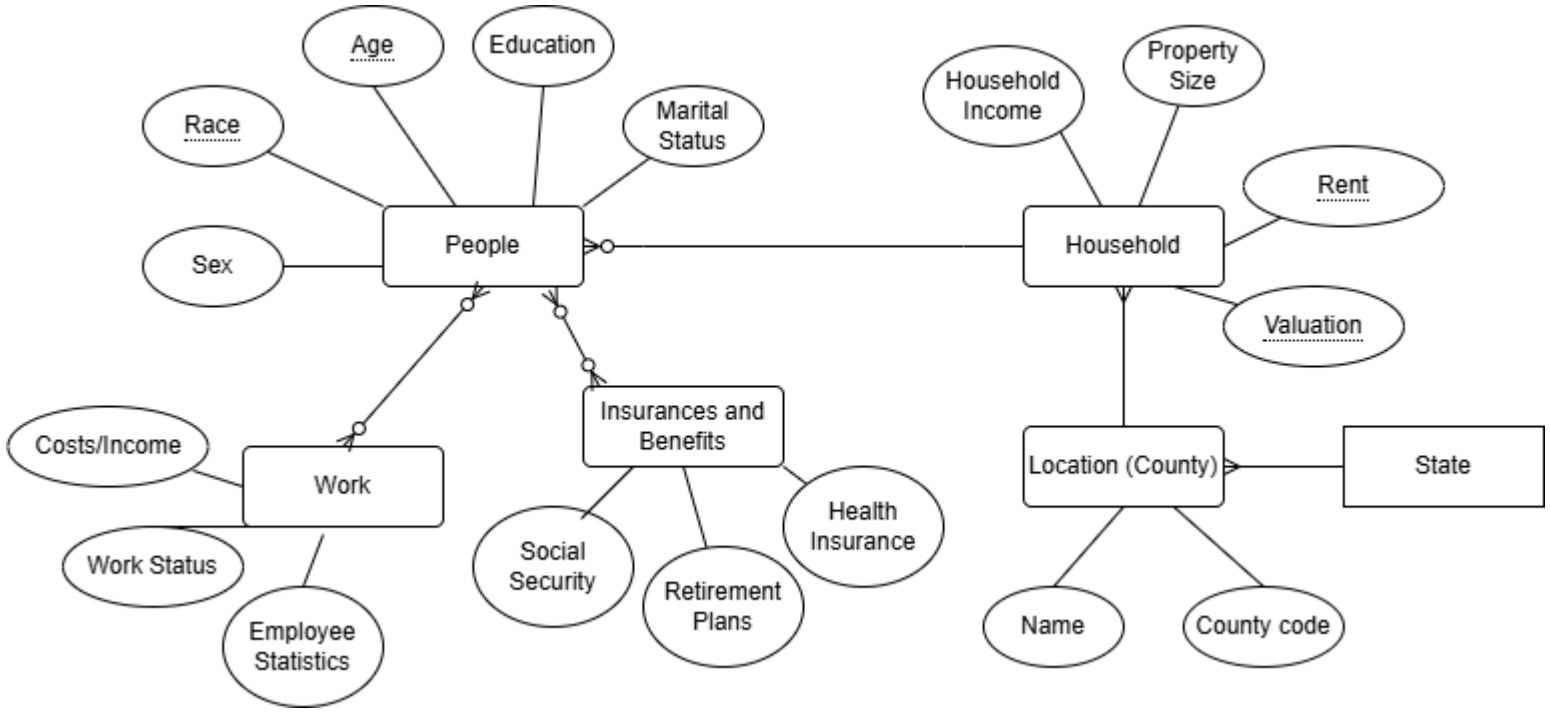


Figure made using Draw.io

Data Exploration:
Goals with transforming the dataset

- Keep most of the information untouched.
- Combine most of the separate datasets together, see if its possible to have all the years in one table (sliding window? Originally the data for each year is a pooling of the previous 5 years). I want to be able to graph the data for a specific county over time.
- Maybe include the json that holds information that corresponds the header code to descriptions, put that in a table for easy lookup

Pipeline Prototype

- Original, testing with DP02: 3221 rows x 1196 columns
- First try was to combine all the same year jsons into one database, but this caused the program to hang. Change to try and clean null values first, then add each json to the dataframe one by one.
- After dropping null columns, testing with DP02, went from 1196 columns down to 721 columns. Added a column for year
- After dropping rows that are all null except for ['GEO_ID','NAME','state','county’], 3143 x 721
- There are many values that don’t make sense like (x), I dropped all non numeric columns other than ['GEO_ID','NAME','state','county’], now there are 576 columns

Pipeline Prototype

- Sample data after cleaning: There are still some zero columns, but overall it is much cleaner than the start. Zero columns might not have been removed because of non-zero values that are outside of this observed window. I decided not to replace the header codes for now, may change it if it becomes a problem later on.

	DP02_0001E	DP02_0001M	DP02_0001PE	DP02_0001PM	DP02_0002E	DP02_0002M	DP02_0002PE	DP02_0002PM	DP02_0003E	DP0
0	7322	189	7322	0	6101	195	83.3	1.7	3156	
1	6691	189	6691	0	4915	271	73.5	3.3	2435	
2	52146	537	52146	0	36164	770	69.4	1.3	16444	
3	14200	400	14200	0	8268	332	58.2	2.4	3668	
4	3329	207	3329	0	2176	198	65.4	4.9	671	

Pipeline Prototype

- Metrics are logged to census_data.log
- Results from logger: The first line of transformer means DP02 has been combined into a single dataframe of 43997 rows and 576 columns
- Currently, the final dataframe is saved locally as csv for easier examination. But this probably will be changed into the future, maybe will need to change it to save as parquet into cloud storage

```
226  2024-12-16 15:10:24,936 - __main__ - INFO - Running main
227  2024-12-16 15:10:51,633 - transformer - INFO - Final combined DataFrame for acs_acs5_profile_DP02: (43997, 576)
228  2024-12-16 15:11:16,327 - transformer - INFO - Final combined DataFrame for acs_acs5_profile_DP03: (45089, 439)
229  2024-12-16 15:11:42,003 - transformer - INFO - Final combined DataFrame for acs_acs5_profile_DP04: (45089, 556)
230  2024-12-16 15:11:47,352 - __main__ - INFO - Saved combined DataFrame for acs_acs5_profile_DP02 to data\acs_acs5_profile_DP02_combined.csv
231  2024-12-16 15:11:51,985 - __main__ - INFO - Saved combined DataFrame for acs_acs5_profile_DP03 to data\acs_acs5_profile_DP03_combined.csv
232  2024-12-16 15:11:57,311 - __main__ - INFO - Saved combined DataFrame for acs_acs5_profile_DP04 to data\acs_acs5_profile_DP04_combined.csv
233
```

Pipeline Prototype

- Slide deck updates:
- Choices I had to make about cleaning/transforming this data in the prototype
  - Some of the things I wanted to do (replace all large negative numbers with n/a) ended up taking an extremely long time to run on each dataframe. I ended up making choices to reduce runtime.
  - I decided against combining information across different groups in this prototype to keep things simple.
- Choices I made about the automation of the data pipeline that impact its performance or reliability
  - Each part (collector and transformer) can be run separately as shown in driver.py
  - I chose to hardcode the year in this implementation (After which it will collect all data up to that given year), but there is a part where if uncommented, it will automatically use the current year instead of the hardcoded year. If using the uncommented current year, then driver.py can be run every year (or every year with a US census release) and it will automatically grab all data up to the current year, transform it, and then save the combined dataframes as individual CSVs.
  - The Collector class is also able to grab a single year (Collector.getyear()), and the Transformer class is able to combine data from a single year to an existing dataframe (Transformer.combine()). If adding just one new year, it is possible to use these methods to update an existing dataframe with the new incoming json. This makes it faster if only updating with the new year. My current implementation assumes that the user starts from nothing, so it will download all years from 2009 every time.
  - The desired groups are listed in GROUPLIST in configs.ini and commented out if they need to be ignored

Pipeline Scaling

- Refactoring the code to work in Spark
- Initial problems:
  - Spark cluster does not run locally on my computer, so I ported the transformer.py and collector.py to Databricks Community Edition and did testing on a sample (DP02, 2009).
  - The json downloaded by collector.py was a list of lists, which would read into spark as a single corrupted string. I needed to change collector.py to use spark and directly parse the response into a spark dataframe before saving.
  - The manipulations take longer to do in Spark than on pandas, some things like null column removal needed to consider run time (A couple minutes on Spark, nearly instant in Pandas). Any sort of iteration through columns can’t be done in a reasonable timeframe.
  - Plan is to remove Null columns, remove null rows, and remove columns containing non-numerical data.

Pipeline Scaling

- Try 1: Originally, to avoid the corrupt string error, data needed to be read through pandas.read_json, and then the Spark dataframe was made from the pandas dataframe. This defeats the purpose of using Spark, so to fix this, Collector.py needed changes to parse the API response correctly. I also changed collector.py to save in a parquet format rather than json
- Original had column names in the first row, changed collector.py to fix this before saving as parquet
- Dropping the null columns results in 720 columns. This matches the previous test with pandas, minus the added year column
- Dropping null rows also matches previous test with pandas, 3221 row

Pipeline Scaling: Thoughts

- Pandas is less restrictive, fixes many problems automatically. When converting to Spark, some things like detection of non-numerical columns would not work unless the schema was determined beforehand. Manually checking each column to convert takes too long. When first reading in the file, all columns are cast as StringType. Pandas is able to ignore the schema while doing manipulations, but Spark needs to add an extra step and correctly pick the data type before saving as parquet.

Pipeline Scaling: Difficulties

- Databricks Community Edition has difficulties importing classes from multiple notebooks (Collector.py, Transformer.py). It needs to %run the location of the imported notebook before importing classes and functions, but this also runs the imported notebook as if it was __main__. I wasn't able to figure out how to let it only let me import into the driver.
- Instead of using the driver to import both then create Collect and Transform objects to collect/transform, I altered collector and transformer to work without the driver. Later on I can schedule them to run directly in databricks on Azure one after the other. The driver is kept for local development but won't be used when running on Databricks. This makes it so each of them needs their own SparkSession and logger file which isn't ideal. Will try to figure out a better way in the future.

Pipeline Scaling: Difficulties

- Overall… it is slow, much slower than my pandas prototype (took 23 min for 3 groups, 3 years). I only worked with years 2009-2011 when confirming my code worked in Databricks. Maybe it is slow because I didn’t properly clean the data when making the prototype. There are a lot more steps included in this part of the project to make sure the schema is correct. Also to simplify things, I made all numbers into floats instead of int types.

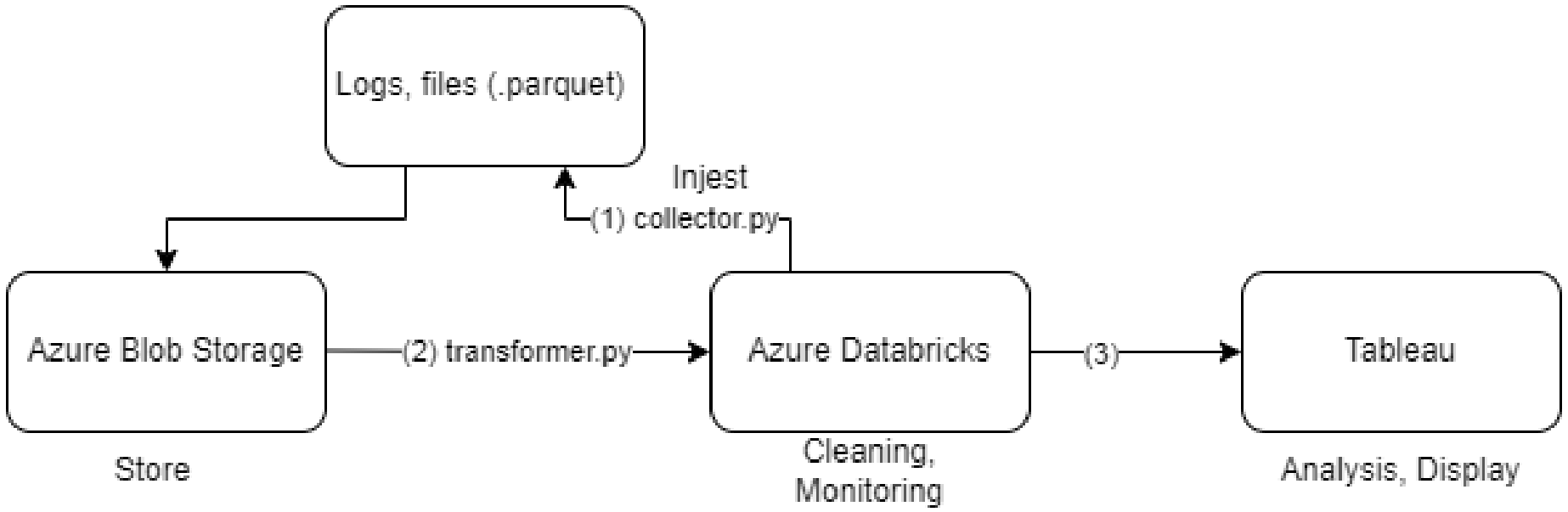
Pipeline Scaling: Results

- Final count of rows and columns for combined with 3 years of DP02. 576 columns corresponds to my prototype done in pandas
- Rest of the examinations and testing were done in the test notebook

Pipeline Scaling: Further Changes

- In transformer, I removed the checking for null columns since that should be covered in the checking for non-numerical values. This sped up the run a lot, 20 minute process turned into ~3-4 minutes.
- The cleaning process is now:
  - Parses and sorts the filenames by Prefix – Year
  - Loops through each prefix, for the first one does:
    - Remove null rows
    - Remove columns with non numerical data, except for state names, county names
    - Adds a column with the year
  - The second one onwards:
    - Remove null rows
    - Adds column with the year
    - Takes the rest of the columns from the first dataframe
    - Union with the first dataframe.

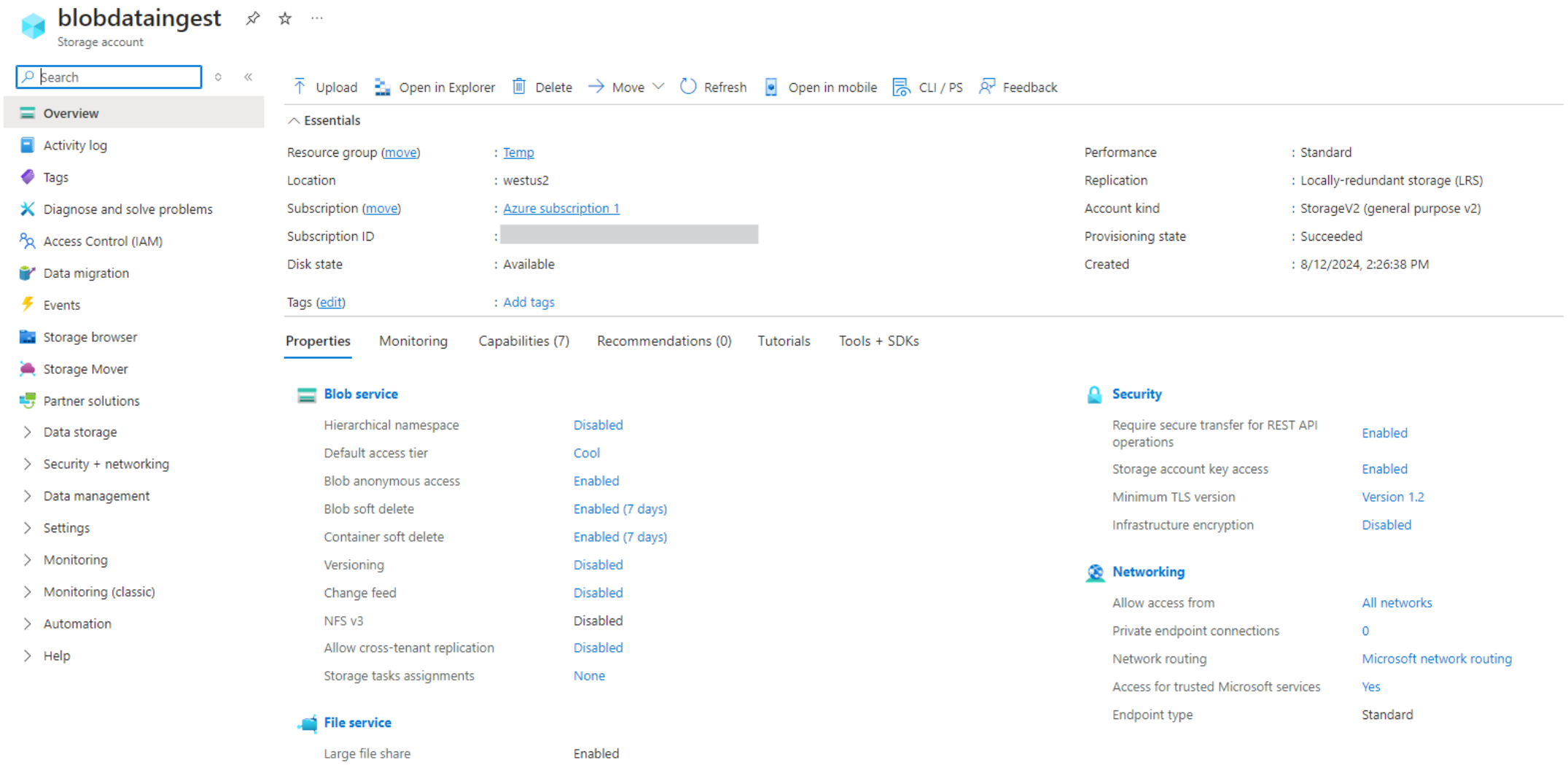
Architecture Planning



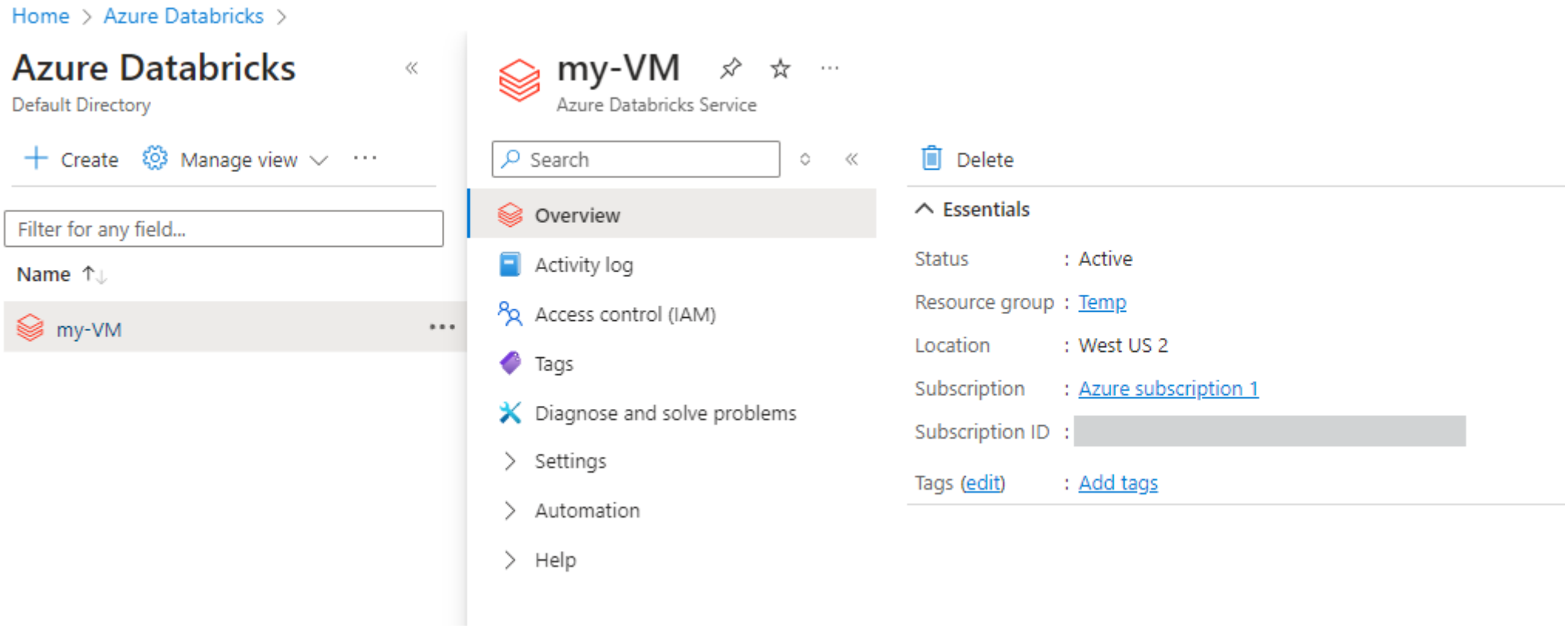
Architecture Planning: Considerations

- I chose to use Azure Blob Storage over Data Lake storage for the cheaper costs. The original version uses hdfs which makes it easier to move to Azure data lake storage – But I plan to change it to use connect to Blob instead.
- Because the data is only collected through the API and the schema is set as soon as it is collected, I don’t think there is a need to use Data Factory here. There isn’t any possibility of data coming in from different sources that will need to be standardized before being stored as parquets.
- Need to change the final form to csv to work with Tableau Public

Architecture Planning: Storage

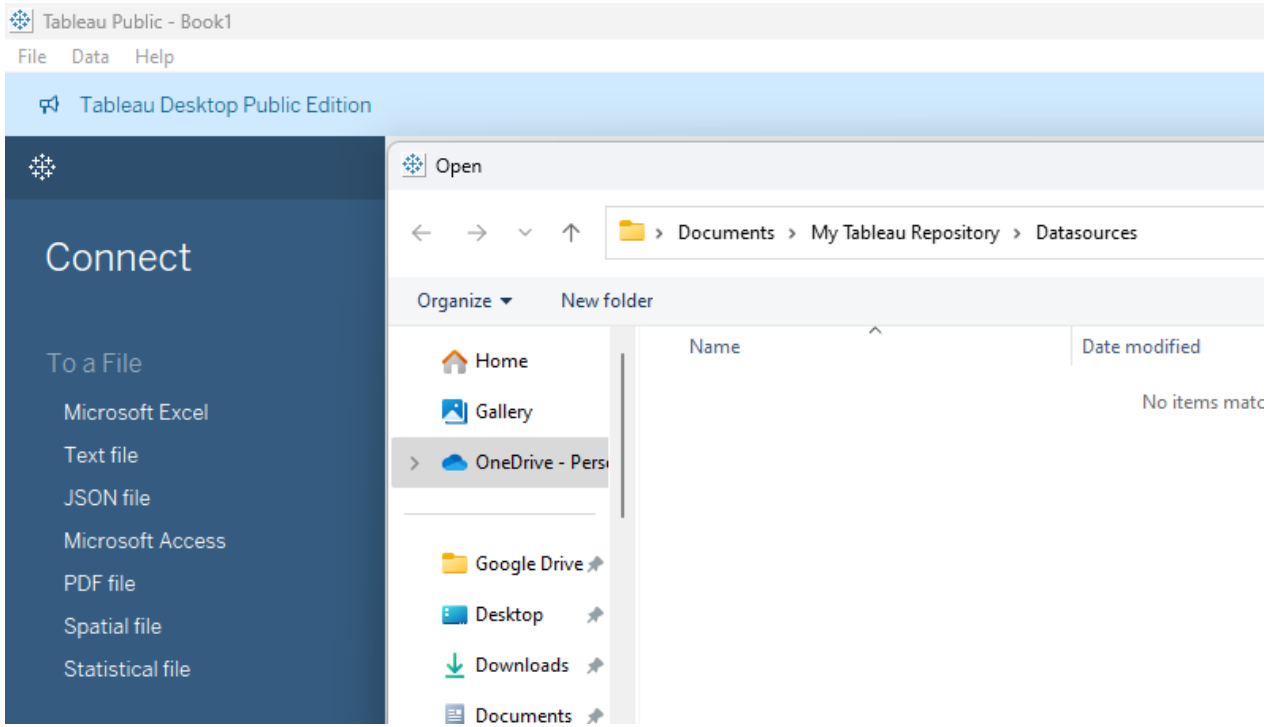


Architecture Planning: Azure Databricks



Architecture Planning: Tableau Public

- Will need to manually upload the CSV file. If this is a problem, may change to using PowerBI or other Tableau option.



Code Testing

- Developed unit tests and tested using pytest and coverage
- Original notebook was already tested cell-by-cell in Databricks during development, but the code was updated so that it could run locally to be tested with pytest.
- Used MagicMock and patch from unittest.mock to do the mocking.
- Most mocking is done by creating a file or Dataframe with certain info, and seeing if the function can read it or transform it correctly.
- Changes:
  - Found some small errors in collector.py (values not initialized in case of failure, causing problems later on)
  - Needed to do extensive mocking to simulate all actions of within dbutils.
  - Also within transformer, needed to mock all the dataframe transformations used (For example, dropna, select, withColumn)

Code Testing

- 11 tests

  pytest test_suite.py –v

```
collected 11 items

test_suite.py::test_get_configs PASSED                                   [  9%]
test_suite.py::test_getdata_success PASSED                               [ 18%]
test_suite.py::test_getdata_failure PASSED                               [ 27%]
test_suite.py::test_get_configs_file_not_found PASSED                    [ 36%]
test_suite.py::test_get_configs2 PASSED                                  [ 45%]
test_suite.py::test_json_to_cleaned_dataframe PASSED                     [ 54%]
test_suite.py::test_clean_df PASSED                                      [ 63%]
test_suite.py::test_process_all PASSED                                   [ 72%]
test_suite.py::test_process_all_empty PASSED                             [ 81%]
test_suite.py::test_remove_null_rows PASSED                              [ 90%]
test_suite.py::test_remove_non_numeric_columns PASSED                    [100%]

============================================ 11 passed in 44.33s ============================================
SUCCESS: The process with PID 23896 (child process of PID 7804) has been terminated.
SUCCESS: The process with PID 7804 (child process of PID 13344) has been terminated.
SUCCESS: The process with PID 13344 (child process of PID 764) has been terminated.
```

Code Testing

- Main tests were:
  - Checking getconfig, mocking a config.ini and checking it was read correctly
  - For collector.py: Check get_data with mocked data to retrieve, also check response if API call fails
  - For transformer.py: Check json_to_cleaned_dataframe, which cleans a json and turns it into a dataframe after running the functions that remove the null rows and non-numeric columns. Mocked everything, including the function responses.
    - Also wrote tests for the functions involved. For the ones that that removed null rows and non-numeric columns, I used mocked dataframes and asserted that results were as expected
    - Also wrote tests to check that process_all worked as intended, with files available as well as in cases without files found.

Code Testing

- Final coverage report: 81% on collector.py, 65% on transformer.py

  coverage run -m pytest test_suite.py -v

  coverage report --omit="C:/spark/python/*,dbutils.py"

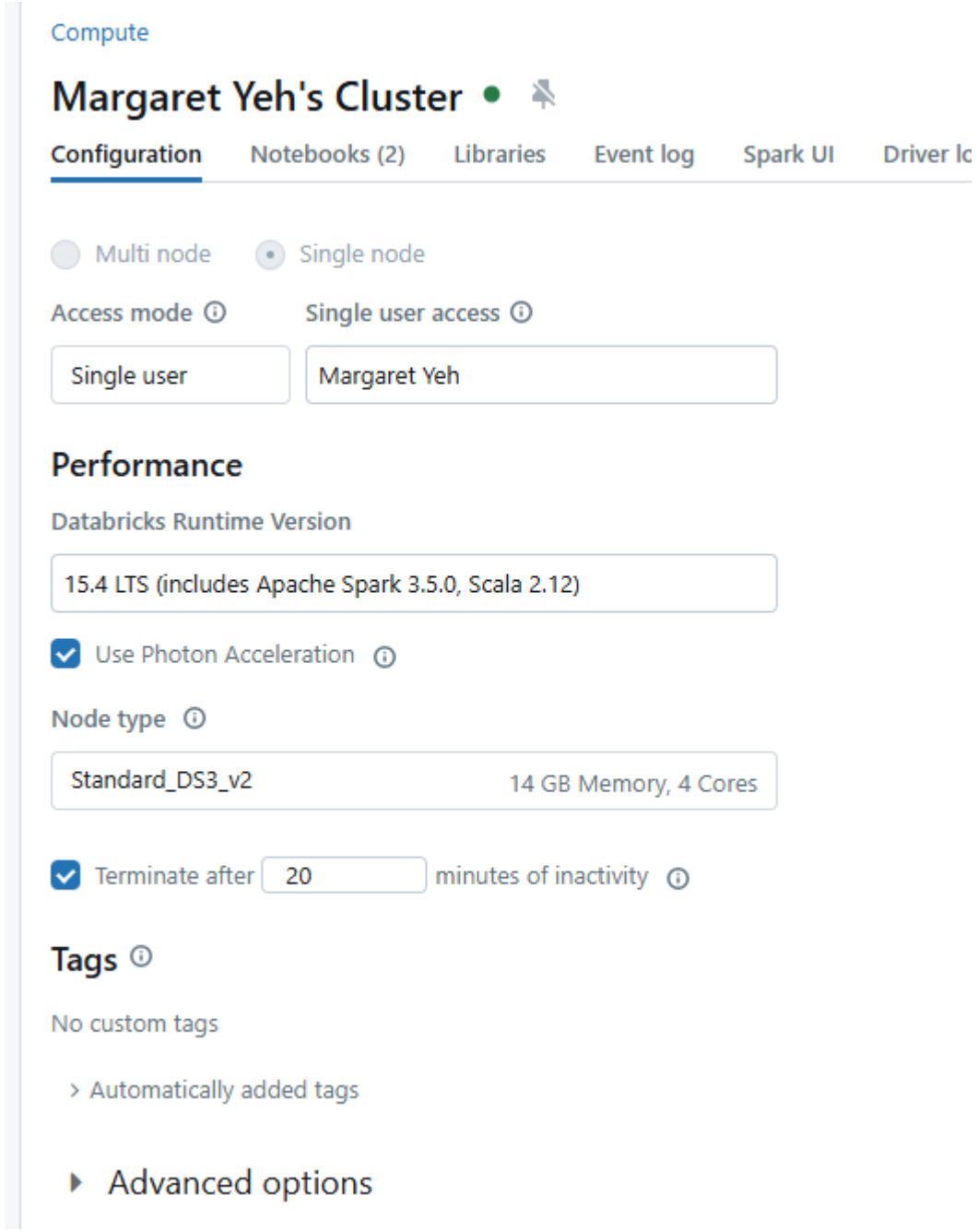
```
$ coverage report --omit="C:/spark/python/*,dbutils.py
Name              Stmts   Miss  Cover
-----------------------------------
collector.py        104     20    81%
test_suite.py       129      0   100%
transformer.py      141     49    65%
-----------------------------------
TOTAL               374     69    82%
```

Deployment

- This project uses Azure Blob Storage and Azure Databricks.
- The original development process was done on Databricks community edition, so porting over to Azure Databricks only involved moving the two notebooks, collector and transformer, and the config.ini file.
- Config.ini is loaded into the DBFS through New -> Add data -> Upload files to DBFS

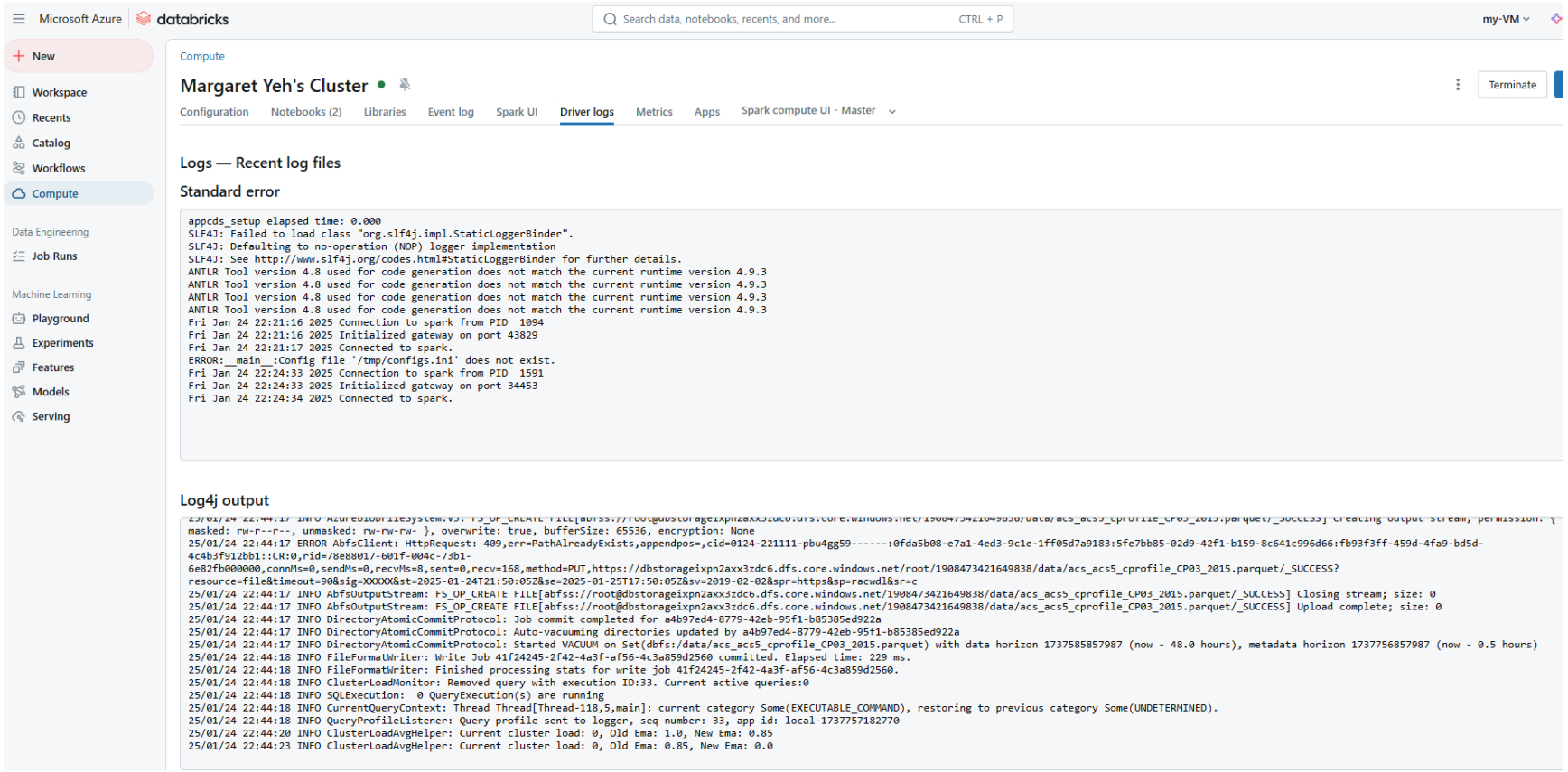
Deployment

- Compute setup
  - Configuration for the compute is the same as that in Databricks Community.



Deployment

- Logs are located in Compute -> Driver Logs



Deployment

- The first run collects all files. Subsequent runs can comment this section in collector.py

```
#----
# Use this for the first run, comment out if running automatically yearly
year = 2022
collector.getallupto(int(year))
#----
```

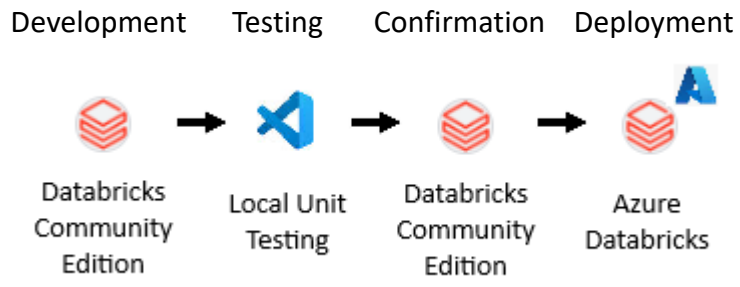
…And uncomment the following section instead.

```
#----
# For updating database, uncomment and run this yearly. It will add the census data for all groups if that year exists, else exits.
#
#current_year = datetime.datetime.now().year
#collector.getyear(int(current_year))
#----
```

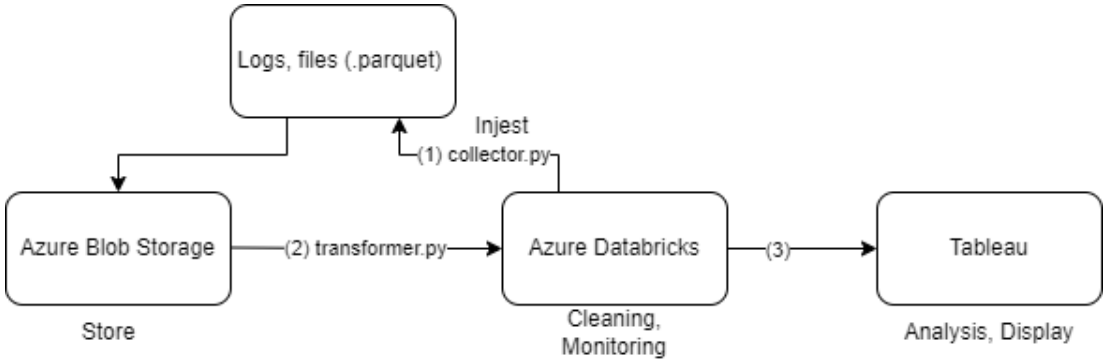
- The first run collects all files. Subsequent runs can attempt to only collect the data for the current year. If it succeeds, then the file will be added to the database. If it fails (no data for current year) then no changes will be made.

Summary (Deployment)

- Development was done through Databricks Community and tested locally with pytest. The resulting code after testing was ported back to Databricks Community for a final checkup, and then moved to Azure Databricks.



Deployment Flowchart



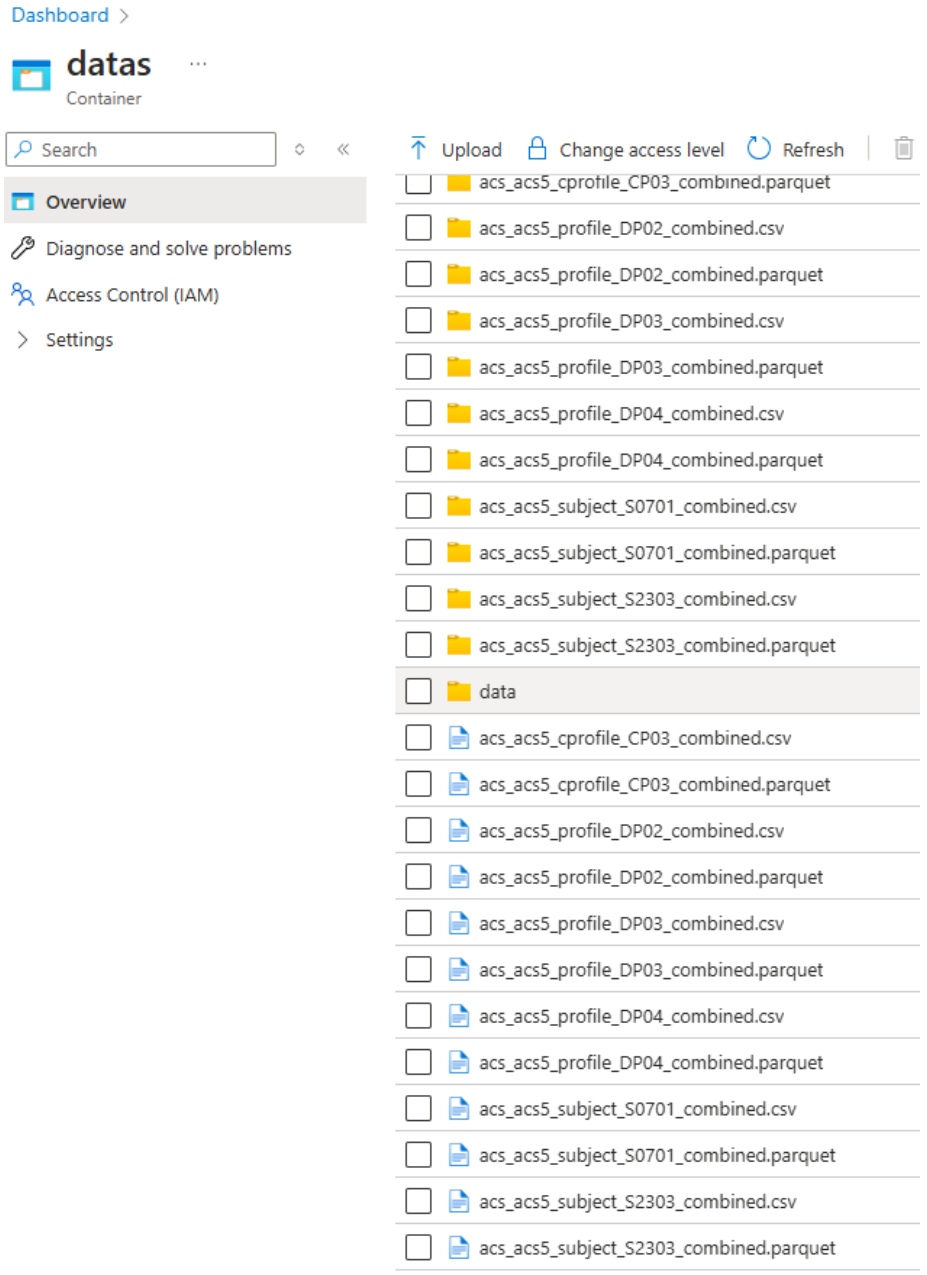
Production Architecture

Summary (Deployment)

- Code overview:
  - The Collector notebook downloads data from the American Community Survey 5-year dataset and saves it as parquet files to DBFS. It can be configured to either download the whole set, part of the set, or all groups of a single year. For this project, the two configurations that are used are the whole-set download and the single year download (for subsequent updates)
  - The Transformer notebook collects the downloaded parquet files and cleans them by removing non-numerical data and empty columns/rows. It also combines the years for each group into a single parquet file per group. This collected parquet is also saved as a CSV for future porting to Tableau Public
  - The final cleaned parquets and CSVs are uploaded to Azure blob storage for future analysis.

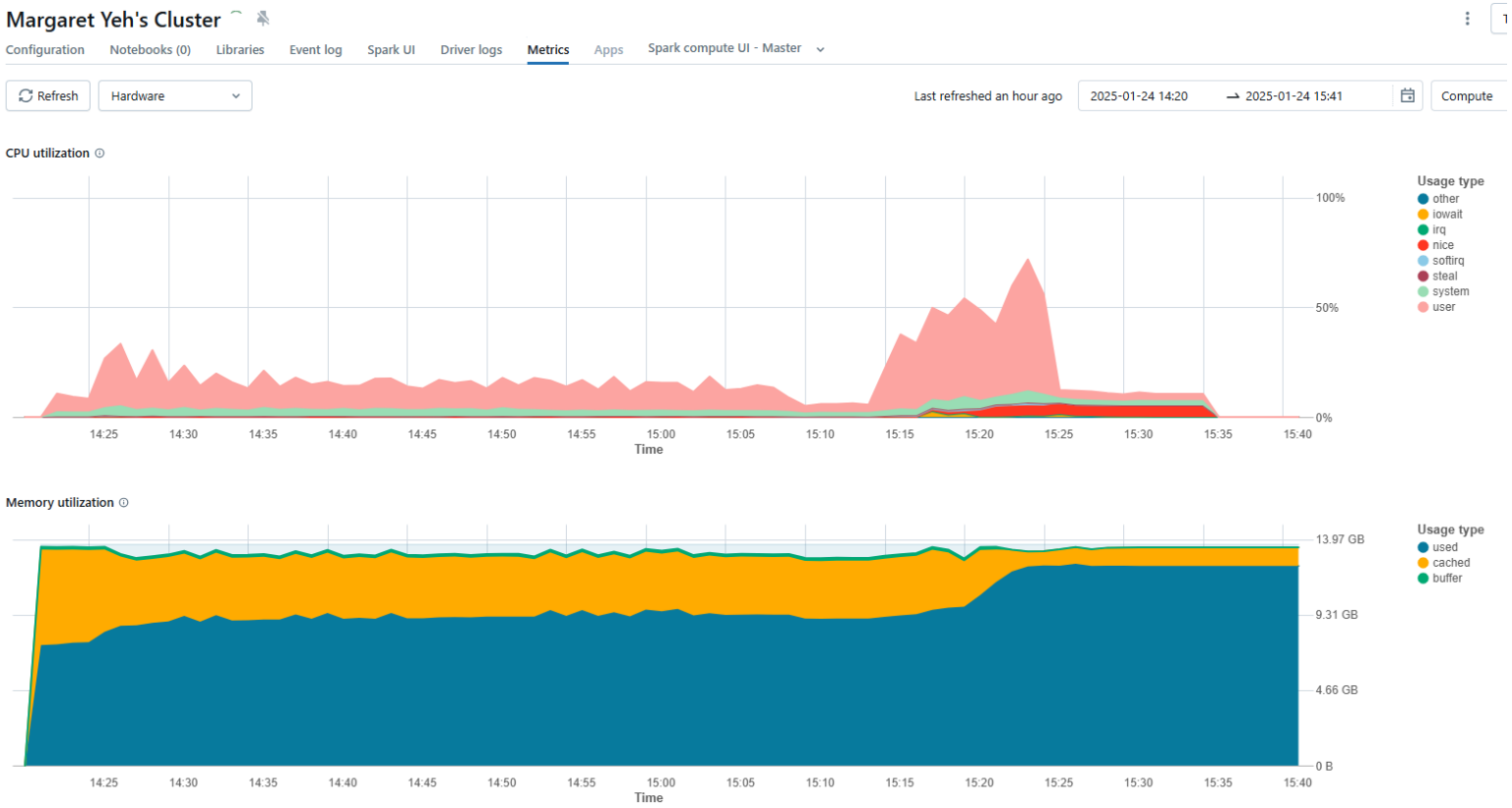
Summary (Deployment)

- Final CSVs and Parquets are downloaded to Blob as shown.
- From here, they can be used for further analysis or imported into Tableau Public



Summary (Deployment)

- Cluster stats for initial run (Need to get these into dashboard for the final part of the project)



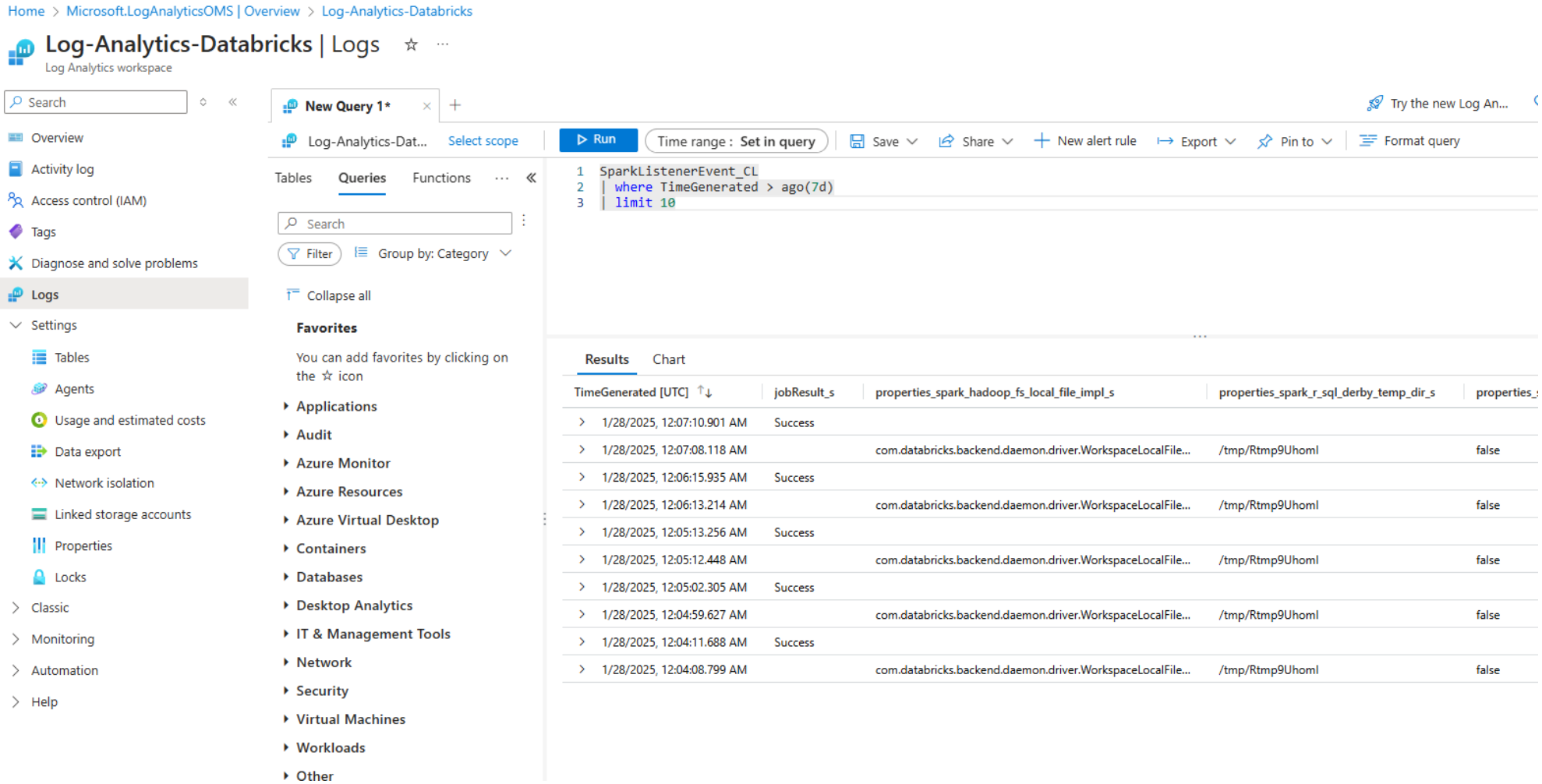
Dashboard Setup

- Follow the instructions at https://learn.microsoft.com/en-us/azure/architecture/databricks-monitoring/application-logs
- Use https://github.com/mspnp/spark-monitoring and follow the README instructions for setup
- Init script needs to be changed to Workspace instead, use this to upload:
  - databricks workspace import --file C:\Users\Margaret\localdir\2024\spark-monitoring\src\spark-listeners\scripts\spark-monitoring.sh /Workspace/spark-monitoring/spark-monitoring.sh --format RAW
  - Source = Workspace, Filepath = /Workspace/spark-monitoring/spark-monitoring.sh

Dashboard Debugging

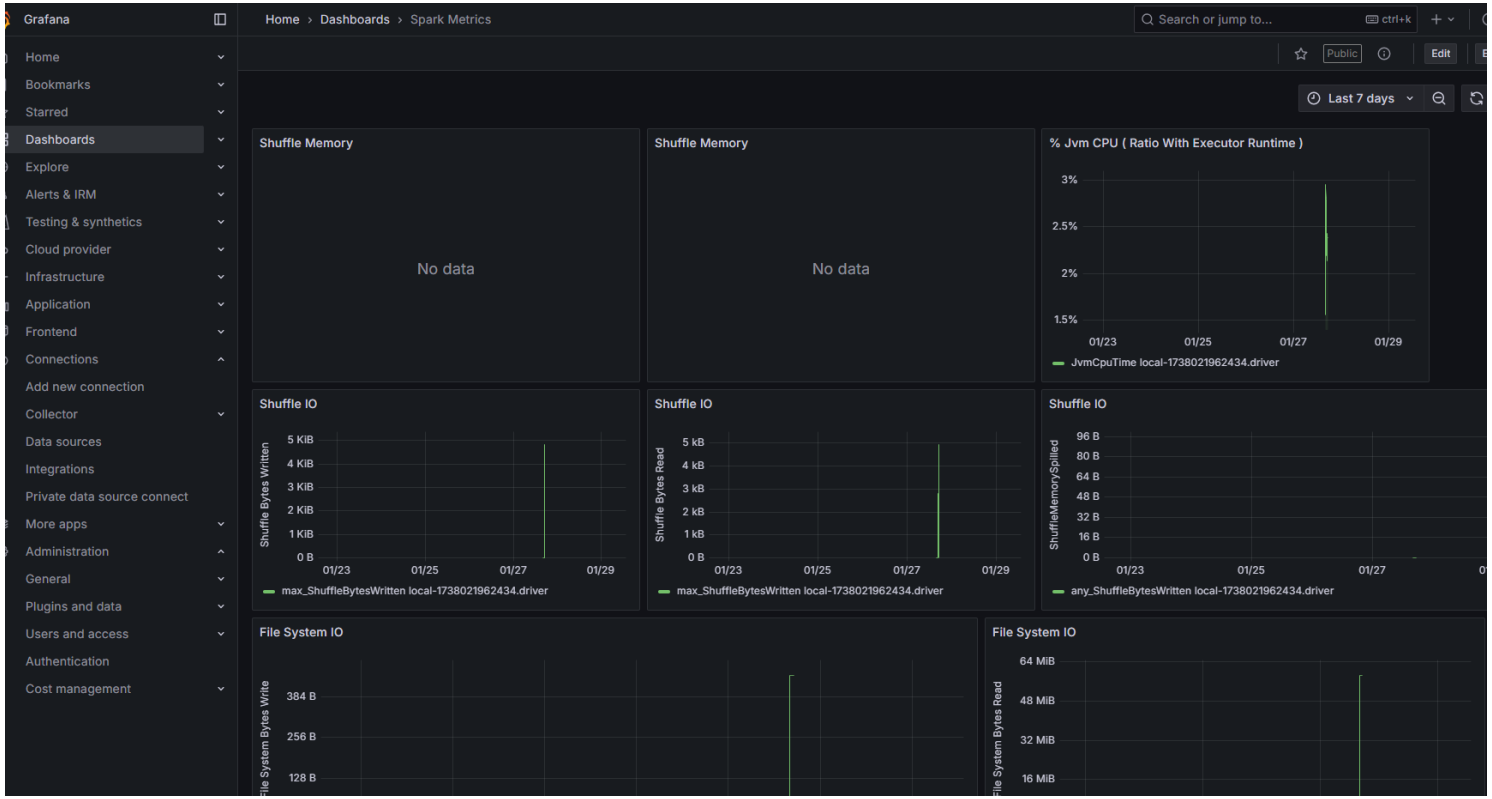
- Init script failure: Cluster scoped init script /Workspace/spark-monitoring/spark-monitoring.sh failed: Script exit status is non-zero
  - Reason: Wrong Cluster setup, need to use instructions here instead: https://github.com/mspnp/spark-monitoring/tree/l4jv2 (for 15.4 LTS, Spark 3.5, Scala 2.12)
  - Should use the “dbr-15.4-lts” profile, create the jar with Docker (Windows instructions). Use this instead of the command on README (Original was not able to find the build).
    - docker run -it --rm -v %cd%:/spark-monitoring -v "%USERPROFILE%/.m2":/root/.m2 mcr.microsoft.com/java/maven:8-zulu-debian10 mvn -f /spark-monitoring/pom.xml clean install -P "dbr-15.4-lts"

After Setup (Must Reprocess the Dataset)



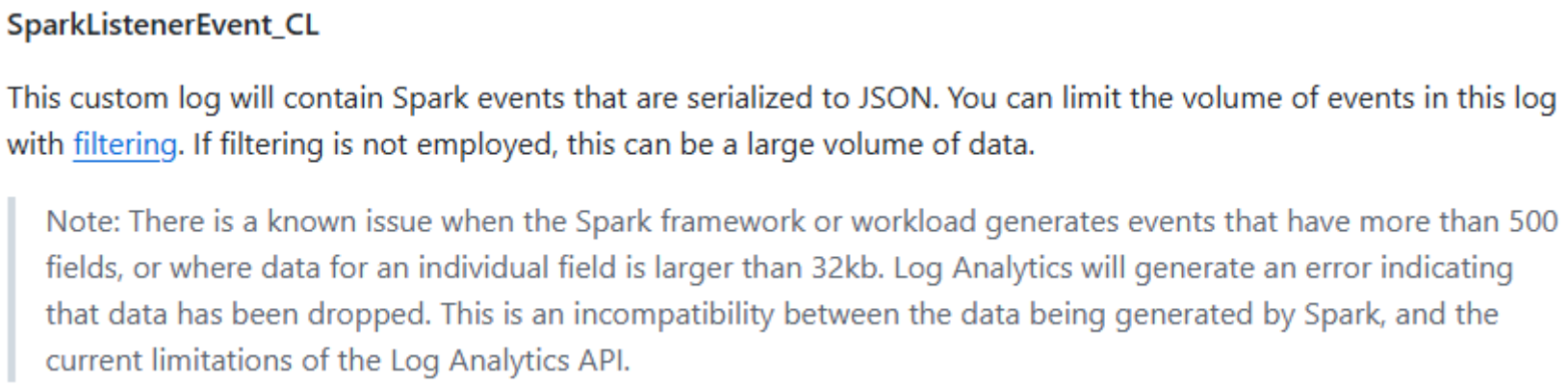
Setting it up to work with Grafana Cloud

Use the instructions here: https://learn.microsoft.com/en-us/azure/architecture/databricks-monitoring/dashboards
But Grafana setup is done in the cloud: https://grafana.com/products/cloud/ instead of in a virtual machine for easier sharing. Use the given instructions to produce the SparkMonitoringDash.json which can be imported into Grafana cloud.



Problems:

- Missing Panels:
  - https://github.com/mspnp/spark-monitoring?tab=readme-ov-file. According to the README file on the spark-monitoring github, there is a limit to how much data can be collected by Log Analytics, and data may be dropped if the events are not filtered. I did not set any filters for the previous run, so this is probably the reason why only half of the dashboard panels show up. Querying the affected columns in Log Analytics doesn't return anything.
  - Instructions for filtering in future fixes: https://github.com/mspnp/spark-monitoring/blob/main/docs/filtering.md#limiting-events-in-sparklistenerevent_cl. After changing the filters, the dataset will need to be reprocessed again.



SparkListenerEvent_CL

This custom log will contain Spark events that are serialized to JSON. You can limit the volume of events in this log with filtering. If filtering is not employed, this can be a large volume of data.

> Note: There is a known issue when the Spark framework or workload generates events that have more than 500 fields, or where data for an individual field is larger than 32kb. Log Analytics will generate an error indicating that data has been dropped. This is an incompatibility between the data being generated by Spark, and the current limitations of the Log Analytics API.

Final Dashboard

- After removing panels with no data, the remaining panels include:
  - % Serialize Time (Ratio with Executor Runtime)
  - % Deserialize Time (Ratio with Executor Runtime)
  - % Jvm CPU (Ratio with Executor Runtime)
  - Cluster Metrics (Sum Shuffle Memory Per Cluster)
  - # Tasks per Executor
  - % Executor CPU (Ratio With Executor Runtime)
  - Shuffle IO (Bytes Written, Bytes Read, Memory Spilled)
  - File System IO (System Bytes Written, System Bytes Read)
  - Disk IO (Shuffle Disk Bytes Spilled, Shuffle Remote Bytes Read to Disk)

Final Dashboard

Link to snapshot:
https://margyeh7.grafana.net/dashboard/snapshot/FoK2brNCSrRFDJ6dr9sOMOSd8qEla2bg