



Version  
**6.13**

Financial Crime Compliance Suite

# PLATFORM

## User Guide (On- Prem)

Revision A



**Legal Notice:**

This document is proprietary & confidential and may only be used by persons who have received it directly from ThetaRay LTD. ("ThetaRay") and may not be transferred to any other party without ThetaRay's express written permission. The document provides preliminary and general information only and is not intended to be comprehensive or to address all the possible issues, applications, exceptions or concerns relating to ThetaRay. All information contained in this document is confidential and shall remain at all times the sole property of ThetaRay. The recipient of this document has no right to disclose any or all of its contents or distribute, transmit, reproduce, publicize or otherwise disseminate this document or copies thereof without the prior written consent of ThetaRay, and shall keep all information contained herein strictly private and confidential. The information is intended to facilitate discussion and is not necessarily meaningful or complete without such supplemental discussion. Please note that the information procedures, practices, policies, and benefits described in this document may be modified or discontinued from time to time by ThetaRay without prior notice. As such, ThetaRay provides the document on an "As-Is" basis and makes no warranties as to the accuracy of the information contained therein. In addition, ThetaRay accepts no responsibility for any consequences whatsoever arising from the use of such information. ThetaRay shall not be required to provide any recipient with access to any additional information or to update this document or to correct any inaccuracies herein which may become apparent.

1. Document History .....	14
2. User Guide Overview .....	16
2.1. Introduction .....	16
2.2. The JupyterLab-Based IDE .....	16
2.3. The Files Browser side panel .....	17
2.4. The phases of a detection flow .....	18
2.5. Customize your detection flow .....	19
2.6. Airflow Integration .....	19
2.6.1. DAG composition example .....	19
2.6.2. DAG scheduling .....	20
2.7. Setting Context .....	21
3. Dataset Management .....	22
3.1. Configuring Datasets .....	22
3.1.1. Dataset Metadata Objects Example .....	22
3.1.2. Dataset Attributes .....	23
3.1.3. Field list Attributes .....	24
3.1.4. Validator Attributes .....	25
3.1.5. Set Ingestion Modes .....	26
3.1.6. Enabling Low Latency Queries for Datasets .....	26
3.1.7. Enable Publishing Dataset to the CDD .....	27
3.1.8. Enabling Data Publishing for Parquet Files Interactive Querying .....	27
3.1.9. Writing to Datasets .....	29
3.1.10. Writing Data in the Overwrite and Update Modes .....	29
3.1.11. Writing data in the Append ingestion mode .....	30
3.1.12. Publishing Datasets .....	30
3.1.13. Read/Write to Public / Private Database Area .....	31
3.1.14. Reading from Datasets .....	31
3.1.15. Reading Data in the Append Ingestion Mode .....	32
3.2. Data Encryption .....	32
3.3. Data Tiering .....	33
3.3.1. Overview .....	33
3.3.2. Purpose .....	34

3.3.3. Overview of Tiers .....	34
3.3.4. How Data Tiering Works .....	35
3.3.5. Configuration Details .....	35
3.3.6. Audit .....	41
3.4. Data Purging .....	42
3.4.1. Overview .....	42
3.4.2. Configuring Airflow DAG - Script and Scheduling .....	42
3.4.3. Implementing the Purging Notebook .....	43
3.4.4. Configuring Retention Policy per Metadata .....	44
3.4.5. Audit .....	46
4. Data Acquisition .....	48
4.1. Configuring a Flat File Connector .....	48
4.1.1. FlatFileConnector Metadata Object Example .....	48
4.1.2. Connector Attributes .....	49
4.1.3. Set the Connector Timestamp Configuration Type .....	50
4.1.4. Set the Connector Targets .....	50
4.1.5. Receiving Encrypted Files .....	51
4.2. Configure the REST connector .....	52
4.3. Configuring a SWIFT MT File Connector .....	54
4.3.1. SwiftConnector Metadata Object Example .....	54
4.3.2. Connector Attributes .....	55
4.4. Configuring an ISO 20022 XML File Connector .....	56
4.4.1. Iso20022Xml Connector Metadata Object Example .....	57
4.4.2. Connector Attributes .....	58
4.5. Configuring a SWIFT MX File Connector .....	58
4.5.1. MX Connector Metadata Object Example .....	59
4.5.2. Connector Attributes .....	60
4.6. Configuring a Sepa File Connector .....	60
4.6.1. SepaConnector Metadata Object Example .....	60
4.6.2. Connector Attributes .....	61
4.7. Upload Data into Datasets .....	62
4.8. Uploading Datasource(s) Via Minio UI .....	63

4.9. Configure the REST Connector .....	65
4.9.1. Upload Data into Datasets .....	66
4.10. Uploading Datasource(s) Via Minio UI .....	66
5. Model Training .....	70
5.1. Configuring Feature Extractor .....	70
5.2. Configuring ThetaRay Detector .....	72
5.2.1. Class Definition .....	73
5.2.2. Class Parameters: .....	73
5.2.3. Methods .....	75
5.3. Configuring ThetaRay Optimizer .....	76
5.3.1. About ThetaRay Optimizer .....	76
5.3.2. Continuing the Configuration .....	77
5.3.3. Class Definition .....	77
5.3.4. Class Parameters: .....	77
5.3.5. Methods .....	79
5.4. Configuring ThetaRay Classifier .....	81
5.4.1. Class Definition .....	81
5.4.2. Class Parameters .....	81
5.4.3. Methods .....	82
5.5. Fitting the Feature Extractor Model .....	84
5.6. Training the Detection Model .....	85
5.6.1. Save reference dataset .....	86
5.6.2. Review model in MLflow .....	86
6. The Evaluation process .....	89
6.1. What is the EvaluationFlow entity? .....	89
6.1.1. EvaluationFlow metadata object example .....	89
6.1.2. EvaluationFlow attributes .....	90
6.1.3. TraceQuery Attributes .....	91
6.2. Run the Evaluation process .....	92
6.3. Publish Evaluated Activities to CDD .....	92
6.4. Publishing Analysis Results to Parquet Indexes .....	93
6.5. Load Evaluated Activities .....	93

6.6. Enriching Evaluation Flow Data - Rule Builder Configuration Fields .....	94
7. Decisioning .....	96
7.1. Identifying Risks .....	96
7.2. Creating a risk .....	97
7.3. Setting the risk metadata .....	98
7.3.1. Dynamic Risk Templates .....	99
7.4. Formulating a basic condition .....	101
7.4.1. Parameters .....	101
7.4.2. Variables .....	102
7.4.3. Conditions .....	102
7.4.4. Rating- type display settings .....	103
7.4.5. Condition name and description .....	103
7.5. Formulating a child risk .....	103
7.5.1. Set the child risk parent .....	104
7.5.2. Variables .....	104
7.5.3. Conditions .....	104
7.6. Formulating a risk with Enrichment .....	105
7.6.1. Parameters .....	106
7.6.2. Enrichment .....	106
7.6.3. Variables .....	107
7.6.4. Conditions .....	107
7.6.5. Threshold-type display settings .....	107
7.7. Dynamic Risk Templates - Upgrade and T/shooting .....	108
7.7.1. Upgrading .....	108
7.7.2. Troubleshooting .....	108
8. Network Visualizations (NV) .....	109
8.1. Manual Alerts in IC Network Visualizations & Updating Decisioning Model .....	109
8.1.1. Overview .....	109
8.1.2. Sample Implementation .....	109
8.1.3. Adding Manual Alerts to Network Visualizations .....	110
8.1.4. Feedback Alert Closure Data - Enhance Decisioning .....	110
8.1.5. Creating a REST Connector .....	111

8.1.6. Storing the Connector Data in MinIO Storage .....	112
8.1.7. Creating an Upload Notebook .....	112
8.1.8. Alerts Feedback Dataset .....	113
8.2. Upgrading NV from pre 6.8 Deployments .....	115
8.2.1. Phase #2 Upgrade Guidelines .....	115
8.2.2. Pre - upgrade Manual Information / Directions .....	115
8.2.3. New Mappings .....	116
8.2.4. Manual Upgrade Steps in Detail .....	117
9. Distribution .....	119
9.1. The distribute_alerted_activities function .....	119
9.2. Define a distribution target .....	120
10. Real Time Transaction Monitoring .....	121
10.1. Overview .....	121
10.2. Evaluation Flow .....	121
10.2.1. Evaluation Flow Example .....	121
10.3. Evaluation Models and Histograms .....	122
10.4. REST Connector Configurations (Customer Facing API) .....	122
10.4.1. Example of API Integration (to be done by the customer) .....	123
10.4.2. Connector Configuration Example .....	124
10.5. Dataset Metadata .....	124
10.6. Close Alert Callback .....	125
10.6.1. Target URL .....	125
10.6.2. Authentication .....	125
10.6.3. Redis configuration .....	126
11. Rule Builder - Documentation .....	128
11.1. Deployment .....	128
11.2. Configuration .....	128
12. CRA - Customer Risk Assessment .....	131
12.1. Overview .....	131
12.2. Data Ingestion .....	132
12.2.1. Transaction Data .....	132
12.2.2. KYC & Auxiliary Data .....	132

12.2.3. Screening Feedback Loop .....	132
12.3. Evaluation Flow .....	133
12.4. Decisioning Risk .....	134
12.4.1. Decisioning Enhancements .....	134
12.4.2. Identify Risks Enhancement .....	135
12.4.3. Pre Evaluation Rules .....	136
12.4.4. Annotations .....	137
12.4.5. Risk Example .....	141
12.5. CRA Reports .....	160
12.5.1. Generating Automatic Reports .....	160
12.5.2. Report Notification API .....	161
12.5.3. Classification Changes API .....	162
13. CRA Algorithm .....	164
13.1. Overview .....	164
13.2. Evaluation Flow .....	165
13.3. Risk Engine .....	168
13.3.1. Classification Types .....	170
13.4. Deterministic Rules .....	171
13.4.1. P0 Exclusion Rules .....	171
13.4.2. P1 Deterministic Rules .....	172
13.4.3. Algorithm Classification .....	176
13.4.4. System & Effective Classifications .....	177
13.4.5. CRA Case Rules .....	179
13.4.6. Risk Metadata .....	180
13.5. Annotations .....	208
13.6. Dataset Metadata .....	213
13.7. Example - Date True Code .....	214
13.8. Data Model .....	219
13.9. Manual Reclassification .....	220
13.10. Reference Implementation .....	220
13.10.1. Datasets .....	221
13.10.2. Customer Type Table (aux) .....	222

13.10.3. FATF (aux) .....	222
13.10.4. EU (aux) .....	222
13.10.5. Data Generation .....	223
13.10.6. Exclusive Rules (P1) .....	224
13.10.7. Non-Exclusive Rules (P2) .....	225
13.10.8. Algo Classification .....	225
13.10.9. DAGS .....	226
13.10.10. Case Rules .....	226
13.10.11. Reports .....	227
13.11. Automatic Reports .....	227
13.11.1. Description .....	227
13.11.2. Important Notes: .....	227
13.11.3. Naming convention .....	228
13.11.4. Retrieval Location .....	229
13.11.5. Layout & Field Order .....	229
13.12. Reclassification Report .....	232
13.12.1. Description .....	232
13.12.2. Naming Convention .....	232
13.12.3. Retrieval Location .....	233
13.12.4. Layout & Field Order .....	233
14. Workflow Automation .....	236
14.1. The DAG object .....	236
14.2. What are the DAG object attributes? .....	237
14.2.1. The RunNotebookOperator object .....	237
14.2.2. Inject parameters from DAG .....	238
14.3. Monitoring Processes in Airflow .....	240
14.3.1. Review DAGs .....	240
14.3.2. Review the DAG execution history .....	241
14.3.3. Inspect a task run log .....	242
14.4. Rerun a task .....	243
14.5. Trigger a DAG in Airflow Manually .....	244
15. Entity Resolution .....	246

15.1. Overview .....	246
15.2. How to Start .....	246
15.2.1. Step #1 Configure Graph .....	246
15.2.2. Example Graph Configuration .....	247
15.2.3. Step #2 Review Default Settings for ER .....	251
15.2.4. Step #3 Prepare your Data .....	253
15.2.5. Example: .....	253
15.2.6. Step #4 Run Entity Resolution .....	253
15.2.7. Step 5. Publish Results .....	254
15.2.8. Step #6 Party Evaluation Flow .....	256
16. Manage Code in Gitlab .....	262
16.1. The Gitlab Plugin .....	262
16.1.1. Commit new code to the staging branch .....	262
16.1.2. Push the committed code to the Staging branch .....	264
16.2. Working in Gitlab .....	265
16.2.1. View in GitLab .....	265
16.2.2. Create a merge request .....	265
17. Exporting Solution to Another Environment .....	266
17.1. Prerequisites .....	266
17.2. Step 1: Create IC Configuration File .....	266
17.3. Step 2: Export MLflow models (optional) .....	267
17.4. Step 3: Run the export script .....	268
17.4.1. An example of a script execution command .....	268
17.5. Step 4: Run the import script .....	268
18. Review Operational Metrics in MLflow .....	270
19. Publishing Data as a Graph (Support for Network Visualizations) .....	272
19.1. Defining Graph Metadata .....	272
19.2. Associating a Graph with an Evaluation Flow .....	277
19.3. Publishing Nodes & Edges .....	278
20. Customer Insights Setup .....	282
20.1. IC Settings .....	282
20.2. Customer Insights Platform Configuration .....	282

20.2.1. Evaluation Flow Metadata .....	282
20.2.2. InsightWidget Class .....	286
20.2.3. InsightField Class .....	286
20.2.4. Field Types and Validations .....	287
20.2.5. For KYC Widget (InsightType.KYC) .....	287
20.2.6. For Geographical Activity Widget (InsightType.GEOGRAPHICAL_ACTIVITY) .....	288
20.2.7. InsightPeriod Class .....	291
20.3. Dataset Metadata .....	291
20.4. ETL Process for Customer Insights Dataset .....	291
20.4.1. Calculating Relevant Fields .....	292
20.4.2. Accessing Historical Alerts Data .....	292
20.5. Analysis Method in Risk .....	293
20.6. Feature Explainability Configuration .....	294
20.6.1. Overview .....	294
20.6.2. Example of such Field metadata .....	294
20.6.3. Explainabilities .....	295
20.6.4. ExplainabilityValueProperties Class .....	297
20.6.5. Examples of Widget Configurations .....	299
20.7. Dynamic Description .....	305
20.7.1. Overview .....	305
20.8. Explainability Configuration for Risk Based on RULE .....	309
20.9. Display Condition and Business Units .....	309
20.10. Risk Explainabilities .....	310
20.11. Reference Implementation .....	311
20.11.1. Implementation Utils .....	312
20.12. OpenAI Summarization .....	314
21. Enabling Monitoring Reports .....	315
21.1. Enabling Monitoring Reports Notifications .....	316
21.1.1. Define New External Notification Target .....	316
22. Calendar SLA .....	318
22.1. Overview .....	318
22.2. Getting Started .....	318

22.2.1. Steps to Configure .....	318
22.2.2. Scheduling the DAG .....	321
22.3. On Prem Specific Instructions .....	321
22.3.1. Option #1 - Whitelisting the Calendarific API .....	322
22.3.2. Option #2 - POST to our IC API .....	322
23. Drift Management .....	324
23.1. Drift Introduction and Overview .....	324
23.1.1. Logging Info Stage .....	324
23.1.2. Statistics Stage .....	324
23.2. How does Drift Detection Work in your Deployment? .....	325
23.2.1. Drift Notebook, its Application in the Drift Management Process .....	325
23.2.2. The Importance of Setting Insightful Metric Variables .....	336
23.2.3. Addition of Drift Detection into the BAU Flow .....	336
23.2.4. When Drift is Indicated, what should the User do Next? .....	338
23.3. Productized Metrics - Information .....	339
23.3.1. Metrics - Transaction level .....	339
23.3.2. Metrics - Feature level .....	339
23.3.3. Metrics - Algorithm level .....	339
23.4. Drift in Machine Learning (CRA) .....	340
23.4.1. Introduction .....	340
23.4.2. Drift Identification .....	340
23.4.3. Input Data .....	341
23.4.4. Drift Monitoring and Test Examples .....	341
24. Appendix A - Cross Platform Instance Concurrency Management .....	346
24.1. Overview .....	346
25. Appendix B - Encrypting Existing Data .....	348
25.1. Postgres Data Encryption .....	349
25.2. Minio Data Encryption .....	353
25.2.1. encrypt_parquet_datasets: .....	353
25.2.2. encrypt_parquet_evaluated_activities: .....	354
25.2.3. publish_to_parquet .....	355
25.3. IC Data Encryption .....	355

25.3.1. encrypt_ic .....	355
25.4. Reference Notebooks .....	357

## 1. Document History

HB	Author	Description of revision content
February 2022	DF	First version
June 2022	DF	Major additions include the following topics: Overview, Workflow Automation, Working in Gitlab, Environment Export/Import, MLflow metrics
June 2022	DF	Export/Import correction
July 2022	HB	Publishing Data as a Graph (Network Visualizations)
August 2022	DF	Distribution: alerted activities specification ref.
September 2022	HB	Monitoring reports
December 2022	HB, RA	New parameter added to dataset in order to protect PII
January 2023	HB, NP	Revision B - Added two sub chapters Data Tiering and Purging (Not GA)
January 2023	TS, NP	Adding manual alerts to NV & Enhancing Decisioning Model
February 2023	HB, AV, NP	Add Phase 1 Entity Resolution content (not GA)
	MD, HB, NP	Added Cross platform Instance Concurrency Management (Appendix A)
March 2023	NP, GB HB	Parquet Queries related content
April 2023	DV, HB	CDD version 2 with Database reporting (self serve)
May 2023	HB AG and GH	added SLA Calendar and Drift management
October 2023	DF, NP	Semi supervised, hyper parameters optimizer updates
November 2023	DF, NP	Updates to Dataset Management Chapter 1) & Published Activities to CDD (Chapter 6)
November 2023	AG, AV	Updates to ER and Publishing Data as a Graph chapters
January 2024	GB, HB	Added - Risk Rating Engine (CRA)
May 2024	GB, HB	Real Time Transaction Monitoring
July 2024	NP, DF, HB	Encrypt Decrypt functionality
September 2024	NP LA HB	Appendix B Encrypt API content

HB	Author	Description of revision content
December 2024	NP, IG, HB	Generating Key Files from Notepad
March 2025	DV, HB	Pre-algo rules documentation

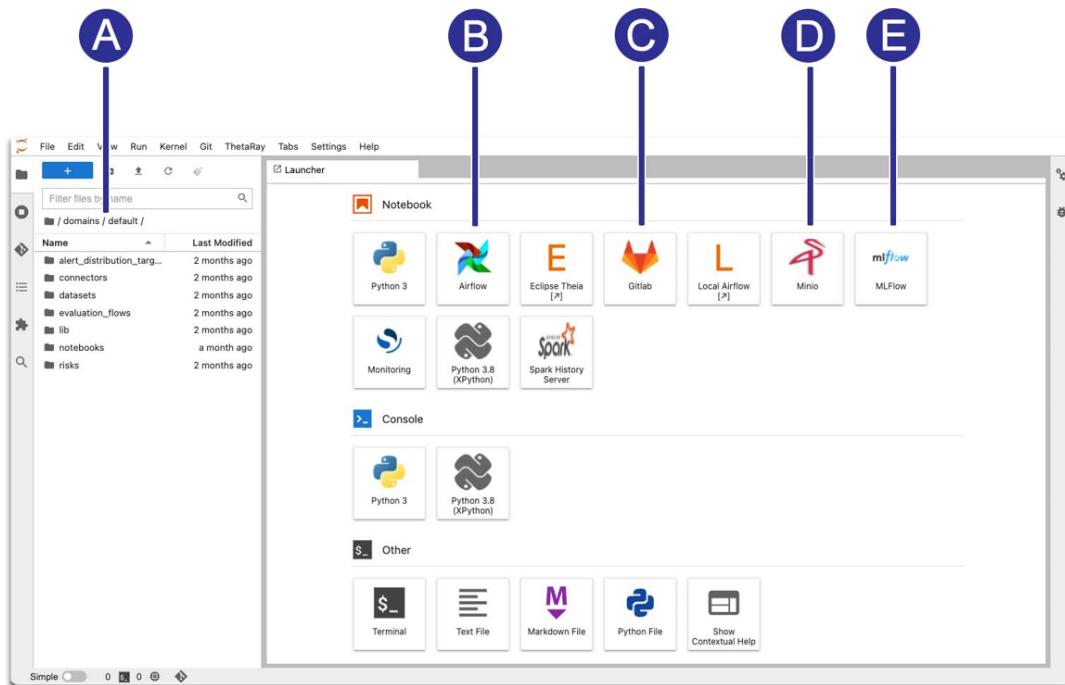
## 2. User Guide Overview

### 2.1. Introduction

The ThetaRay Platform provides a development environment for authoring, testing, managing, and operating end-to-end analysis flows using a combination of machine learning-based scoring and rules. The platform leverages ThetaRay's proprietary machine learning algorithms, rules evaluation, and data management, combined with best-of-breed opensource technologies to provide an integrated environment for implementing, detection, and alert-generation flows.

### 2.2. The JupyterLab-Based IDE

The ThetaRay Platform integrates *JupyterLab* with various powerful tools: *Airflow* is defined and configured from *JupyterLab* to provide process and workflow automation and scheduling. *MLflow* is also started and controlled from your *JupyterLab* app using ThetaRay API to allow you to save and manage your ML models. In addition, *MinIO* and *PostgreSQL* databases are transparently activated and managed to provide data integrity and persistence using the API's read and write functionality. Finally, *GitLab*, the source code management application, is integrated to save and maintain your code and artifacts.



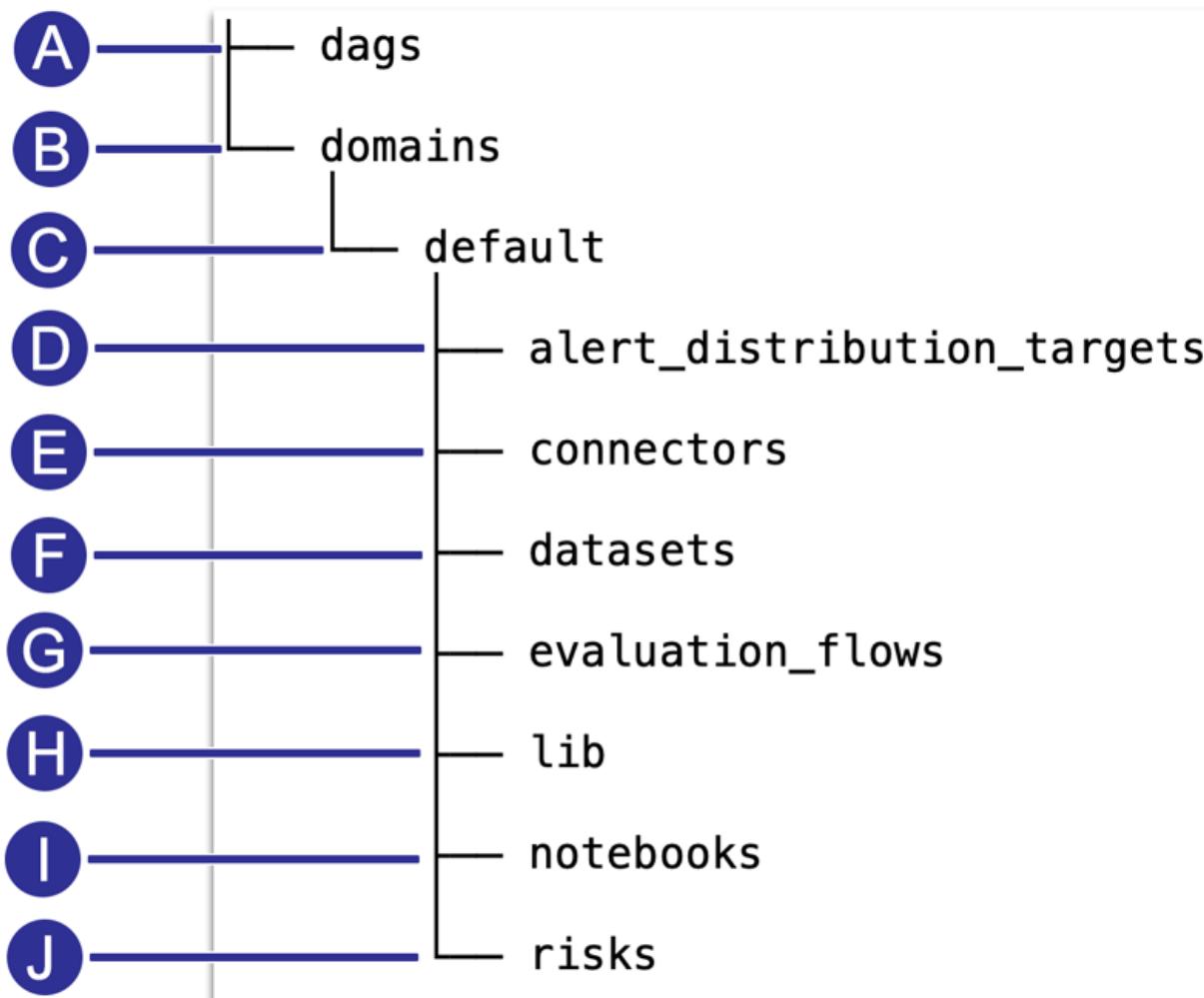
- The *JupyterLab* Files side panel provides access to the Notebooks and other code files.
- The *airflow* workflow manager orchestrates detection flows execution and scheduling.

- c. *GitLab*: a source code and version management application for storing and maintaining source code and artifacts.
- d. *MinIO*: An object data store for data storage and archiving.
- e. *MLflow*: ML model repository application for saving and managing anomaly detection ML models used in the alert generation process.

## 2.3. The Files Browser side panel

The *JupyterLab* File Browser side panel organizes the notebooks and configuration files in a predefined hierarchy containing your solution's code under the Solutions folder.

- Do not change the Files side panel predefined hierarchy.



The Browse Files side panel includes the following folders under the Solutions folder:

- a. The *dogs* folder contains the DAG definition objects; see Workflow Automation.
- b. The *domains* folder is a parent folder for all domains in the platform. Multiple domains enable ThetaRay Platform to provide an end-to-end solution to different areas within the organization.
- c. The *default* folder includes all the components included in a ThetaRay Platform solution.
- d. The *alert\_distribution\_targets* folder contains the metadata objects for the solution's distribution targets, see Distribution.
- e. The *connectors* folder contains the connectors metadata objects; see Data Acquisition.
- f. The *datasets* folder contains the dataset metadata objects; see Dataset Management.
- g. The *evaluation\_flows* folder contains the evaluation flow metadata object; see The Evaluation process.
- h. The *lib* folder contains auxiliary, 3rd party libraries, and additional functionality files.
- i. The *notebooks* folder contains the notebooks that are executed as tasks by *Airflow*.
- j. The *risks* folder contains risk YAML files that define conditions determining in the decisioning stage which data records should raise alerts, see Decisioning.

## 2.4. The phases of a detection flow

ThetaRay Platform provides a powerful API enabling you to build a custom end-to-end detection flow. Depending on your requirements and resources, your detection framework can consist of several flows running at different schedules and intervals. However, regardless of the numerous configurations and setups that are possible, a typical ThetaRay detection flow consists, from beginning to end, of the following sequence of phases:

- **Data acquisition:** Import data from data sources and upload it to the platform's datasets; see Data Acquisition
- **Data preparation:** Transform data acquired from an external source to a structure ready for processing by ThetaRay's machine learning algorithms and rules and interactive exploration. The data preparation phase includes data normalization, aggregation, and feature engineering
- **Model training:** Train a model on a sample dataset using ThetaRay's algorithms, and save it in MLflow
- **Evaluation:** Enrich the dataset records by applying a pre-trained machine learning model. Future versions may include additional evaluation strategies
- **Decisioning:** Determine if the evaluated data records should trigger one or more alerts, each associated with business risk. The process applies rules to each record and uses the evaluation process results or other available data

- **Distribution:** Distribute the alerts to their target Investigation Center instances or other case management applications

## 2.5. Customize your detection flow

You can implement the activities above in the JupyterLab notebooks environment in any layout that fits your design and organization. For example, you can place each activity in a separate JupyterLab notebook or divide the activities across two notebooks. However, it's essential to understand that each notebook is executed as a separate task by the Airflow process manager - a task that, together with the other tasks, is part of an Airflow DAG, which is the Airflow term for a process.

The above tasks can be chained to one continuous **DAG**, but it makes more sense to break them into two, three, and more **DAGs** that can be executed on different schedules. For example, **data\_acquisition** and **data\_preparation** tasks can be run on one **DAG** daily; the **evaluation**, **decisioning**, and **distribution** tasks can yet run on another **DAG** monthly, while the Model training tasks can be yet run on a third **DAG** on-demand without automatic scheduling.

## 2.6. Airflow Integration

Airflow views each *JupyterLab* notebook as one task in a **DAG** consisting of a sequence of tasks executed automatically in a predefined order and schedule. A ThetaRay integration entity, also called **DAG**, maps the *JupyterLab* notebook to the *Airflow* task. *Airflow* accesses the entity, learns which notebooks to execute and in what order, and runs the **DAG**.

### 2.6.1. DAG composition example

```
dataprep >> check_drift >> evaluation >> decisioning

dataprep = \
    RunNotebookOperator(task_id="account_monthly_dataprep",
                        domain="default",
                        notebook_name="account_monthly_dataprep",
                        container_cpu="1000m",
                        container_memory="2Gi",
                        dag=dag)

check_drift = \
    RunNotebookOperator(task_id="check_drift",
                        domain="default",
                        notebook_name="account_monthly_drift",
                        container_cpu="600m",
                        container_memory="24Gi",
```

```
dag=dag)
evaluation = \
    RunNotebookOperator(task_id="evaluation",
                         domain="default",
                         notebook_name="account_monthly_evaluation",
                         container_cpu="2000m",
                         container_memory="4Gi",
                         dag=dag)
decisioning = \
    RunNotebookOperator(task_id="decisioning",
                         domain="default",
                         notebook_name="account_monthly_decisioning",
                         container_cpu="1000m",
                         container_memory="2Gi",
                         dag=dag)
```

## 2.6.2. DAG scheduling

```
default_args = {
    'owner': 'Airflow',
    'depends_on_past': False,
    'retries': 1,
    'retry_delay': datetime.timedelta(minutes=5)
}
dag = DAG(
    dag_id='monthly_analysis',
    catchup=False,
    default_args=default_args,
    schedule_interval=None,
    start_date=datetime.datetime(1993, 1, 1)
    end_date=datetime.datetime(1993, 12, 1)
)
```

The **DAG** entity includes the schedule **start\_date** and **schedule\_interval**. Airflow calculates the dates for each **DAG** execution and uses it as a unique parameter identifying the run.

In addition, *Airflow* injects the execution date into the context of each task as the task **execution\_date** parameter value. The **execution\_date** value is then used by the dataset **read** and **write** functions to timestamp the data records processed by the task, thus establishing a connection between the data and the particular *Airflow* **DAG** run processing it. The context of each text is defined at the beginning of each task notebook; see: Setting Context.

## 2.7. Setting Context

The task context holds the task runtime settings and is alive for the task's duration. The context settings consist of the task execution date, Spark resources configuration and allocation, and other runtime attributes.

The context settings are defined in the **init\_context** function executed at the beginning of each task notebook. However, the same settings can be set on the **DAG** running the task in the context of an automatic *Airflow* process. In this case, the **DAG** settings take precedence and override the corresponding settings defined within the **init\_context** function. However, settings defined in **init\_context** and not in the **DAG** are not deleted but merged with the **DAG** settings.

```
from thetaray.api.context import init_context
import datetime

spark_conf={

    "spark.executor.memory": "8g",
    "spark.executor.cores": "2",
    "spark.executor.instances": "1",
    "spark.sql.adaptive.enabled": "true",
    "spark.dynamicAllocation.enabled": "true",
    "spark.dynamicAllocation.initialExecutors": "1",
    "spark.dynamicAllocation.maxExecutors": "10",
    "spark.dynamicAllocation.minExecutors": "0",
    "spark.dynamicAllocation.shuffleTracking.enabled": "true"
}

context = init_context(execution_date=datetime.datetime(1970,1,1), spark_conf=spark_conf)
```

The **init\_context** function includes the following parameters:

- **execution\_date**: The notebook execution date. The read and write functions use the **execution\_date** value to decide which data records are read from a given dataset. When *Airflow* automatically runs the task, the DAG run **execution\_date** overrides this local value; see DAG scheduling.
- **spark\_conf**: A list of Spark runtime configuration properties.
- **spark\_master**: Determines whether Spark will run the task on a distributed Kubernetes cluster (the default option) or a local cluster on your machine (**spark\_conf="local[\*]"**). It's recommended to select the latter option when your datasets are relatively small to conserve resources.

## 3. Dataset Management

A Dataset in the ThetaRay platform is used to manage structured, tabular data containing either raw data ingested from external sources or derived data computed in the platform through ETL (extract, transform, load) of feature engineering activities.

### 3.1. Configuring Datasets

- Datasets are configured under the *root/domains/<your-domain>/datasets* folder.

#### 3.1.1. Dataset Metadata Objects Example

```
from pyspark.sql import Column
from thetaray.api.solution import DataSet, Field, DataType, IngestionMode
from thetaray.api.solution.dataset import Validator

def account_dataset():
    def account_id_validation(c: Column) -> Column:
        return c > 0 & ~c.isin(22222)

    return DataSet(
        identifier="account",
        display_name="account",
        field_list=[
            Field(identifier="account_id",
                  display_name="account_id",
                  data_type=DataType.LONG,
                  array_indicator=False,
                  validators=[Validator(name="account_id_validator",
action=Validator.Action.REJECT, validate_logic=account_id_validation, enable=False)])
        ],
        Field(identifier="district_id", display_name="district_id", data_
type=DataType.LONG, array_indicator=False),
        Field(identifier="frequency", display_name="frequency", data_
type=DataType.STRING, array_indicator=False),
        Field(identifier="date", display_name="date", data_type=DataType.TIMESTAMP,
date_format='%Y%m%d', array_indicator=False)
    ],
    ingestion_mode=IngestionMode.OVERWRITE,
    publish=True, primary_key=["account_id"],
    num_of_partitions=4,
    num_of_buckets=7,
)
def entities():
```

```
return [account_dataset()]
```

### 3.1.2. Dataset Attributes

Datasets metadata objects include the following attributes:

- **identifier**: A unique and immutable name per dataset (string).
- **display\_name**: The title of GUI boxes displaying the field values (string).
- **field\_list**: The fields comprising the dataset (array of field definition objects):
  - **data\_permission**: The user group that can have access to the dataset from the Investigation Center. User groups are defined in Keycloak.
  - **ingestion\_mode**: How data is written to the dataset and read from it. There are three modes: OVERWRITE, UPDATE, and APPEND.
  - **primary\_key**: Names of the fields serving as primary keys. Since historical versions are maintained, a dataset can contain multiple data records with the same primary key.
  - **num\_of\_partitions**: Number of parallel Spark partitions working on the dataset during the write operation.
  - **num\_of\_buckets**: Sets the number of buckets created for each partition. Data records can be sorted into buckets according to their primary keys. Bucketing is performed when the dataset is in the OVERWRITE or UPDATE ingestion modes.
  - **occurred\_on\_field**: The dataset column whose values signify the time and date of the data record. The field is mandatory for the APPEND ingestion modes and optional for the OVERWRITE and UPDATE ingestion modes.
  - **validate\_enable**: Enable validators.
  - **validators**: A list of data-record-level validator definition objects. A validator definition object checks the overall validity of the data record by applying a specified validation rule that usually takes into account multiple fields.
  - **publish (Bool)**: The publish flag determines whether the dataset can be published to the Postgres database that serves as the platform Central Database Depo, or in short, CDD. The CDD is used for interactive SQL querying by the Investigation Center and during the data enrichment process in the decisioning phase; see Publishing Datasets.
  - **parquet\_indexes**: List of *ParquetIndex* definitions in case the Parquet files are exposed for low latency queries.
  - **table\_indexes**: for Dataset marked as 'published', allows custom Postgres indexes to be defined on the underlying table. Custom indexes can be used to accelerate trace

queries performance, as well as batch / real-time queries issued by the decisioning processes. The property accepts a list of TableIndex objects that include:

- **Identifier:** the index name
- **columns:** a list of column names to index the data on (column order affect index behavior and typically an index will include primary keys and time as the first columns)
- **include:** a list of additional columns to include within the database index data, but are not indexed. 'Included' columns should be used to optimize queries by allowing query results to be served directly instead of reading the full row content.

---

**Note:** A field serving as the identifier across multiple datasets (such as customer id or account id) should remain consistent and encrypted across all datasets to avoid unresolvable references (primarily when accessing the data from the Investigation Center). For example, if the Customer ID column is encrypted as part of an Input Dataset to an Evaluation Flow, the Customer ID column should be encrypted on the Transactions Dataset to enable consistent references between the Dataset, thus allowing the Transactions Tab of the Investigation Center to function correctly.

---

### 3.1.3. Field list Attributes

Every record in a dataset conforms to a schema defined in the **field\_list** array. Each field in the field\_list is set by attributes that determine its name, data type, validation rules, and other characteristics.

```
field_list=[  
    Field(display_name='account_id', data_type=DataType.LONG, identifier="account_id", category="c1-10", description="c1-description"),  
    Field(display_name='party_id', data_type=DataType.STRING, identifier="party_id", description="c1-description"),  
    Field(display_name='party_accounts', data_type=DataType.STRING, identifier="party_accounts", category="c1-10", description="c1-description"),  
    Field(display_name='district_id_bank', data_type=DataType.LONG, identifier="district_id_bank", category="c1-10", description="c1-description"),  
    Field(display_name='frequency', data_type=DataType.STRING, identifier="frequency", category="c1-10", description="c1-description"),  
],
```

The field object properties are:

- **identifier:** A unique and immutable name per dataset (String).
- **display\_name:** The title of GUI boxes displaying the field values (String).
- **data\_type:** The field data type (INT, DOUBLE, LONG, STRING, BOOLEAN, TIMESTAMP).

- **date\_format**: Date/Time patterns for formatting and parsing dates. The pattern follows Spark conventions, e.g., %Y-%m-%d %H-%M-%S-%f. See spark documentation on the topic.
- **array\_indicator**: The field accepts arrays as values (Boolean).
- **indexed**: When set to True and the Dataset is marked as 'published', will result in a Postgres index being created for the column corresponding to the field.
- **validators**: A list of validator objects. A validator object checks the field values validity by applying a specified validation rule.
- **description**: free text (String)
- **category**: An optional attribute that can be used in the dynamic risk templates (see Decisioning chapter for more information)
- **encrypted**: For deployments with data at rest encryption enabled, setting this parameter to true indicates that the field should be encrypted when stored on disk (either in object storage or in a relational database). Only fields of type string can be encrypted.
- **audited**: For deployments with data at rest encryption enable, setting this parameter to true on an encrypted field indicates that values of the field read by an end user (by explicit decryption from Jupyter through the 'decrypt' API or through the Investigation Center) will be audited to OpenSearch and optionally exported through a bucket on Object Storage as part of the audit export functionality.

---

**Note:** A field serving as a identifier across multiple datasets (such as customer id or account id) should be consistently encrypted across all datasets to avoid unresolvable references (primarily when accessing the data from the Investigation Center).

For example, if the Customer ID column is encrypted as part of an Input Dataset to an Evaluation Flow, the Customer ID column should be encrypted on the Transactions Dataset to enable consistent references between the Dataset allowing the Transactions Tab of the Investigation Center to function correctly.

---

### 3.1.4. Validator Attributes

In our example, the **account\_id** field has a validation rule called **account\_id\_validator**. Its logic is defined in a PySpark expression in the **account\_id\_validation** function.

```
def account_id_validation(c: Column) -> Column:  
    return c > 0
```

A validator object checks the fields or data records validity by applying a specified validation rule. You can set a validation rule for a single field or a full data record containing multiple fields.

- **name**: The validation rule name

- **action:** The outcome of a failed validation rule. rejection of the invalid field value or a warning message. (WARN/REJECT)
- **validate\_logic:** the logic of the validation rule described by a function or an SQL string.
- **enable:** disable the validation rule (TRUE/FALSE)
- **validate\_blacklist:** d

### 3.1.5. Set Ingestion Modes

An essential property is the dataset **IngestionMode** which defines how to read data from the dataset and how to write data to the dataset. When determining which of the three ingestion modes is appropriate for your dataset, you must consider how the data gets released by the data owner.

```
ingestion_mode=IngestionMode.OVERWRITE
```

- Select **APPEND** mode If the owner releases only the new data records since the last upload. This mode is standard when a large volume of new records is constantly added over time, and the data record never changes, e.g., payment or transaction tables.
- Select **OVERWRITE** mode If the owner releases all the data records as a single snapshot on every scheduled upload. This mode is expected when the volume of records increases slowly, and their contents do not frequently change, e.g., master data tables such as the account table or auxiliary reference tables such as risky\_countries.
- Select **UPDATE** mode: If the owner releases only records that have been changed since the last upload on every scheduled upload. This mode is typical for a master or auxiliary reference tables like the OVERWRITE mode.

### 3.1.6. Enabling Low Latency Queries for Datasets

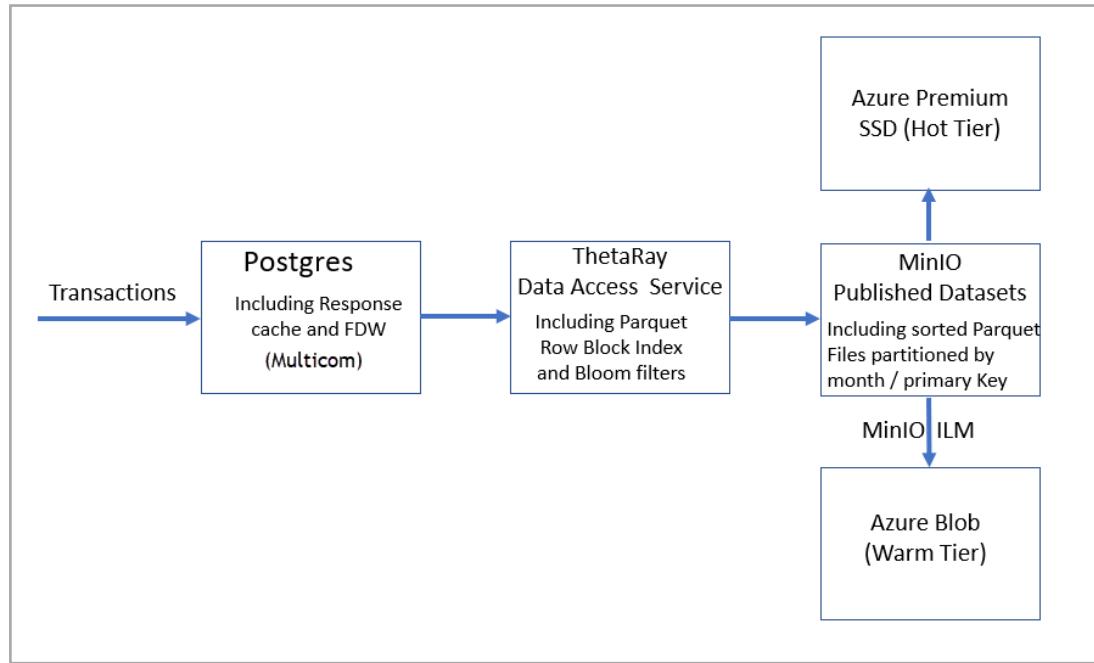
The default storage mechanism for Datasets is based on Parquet files written to object storage a mechanism which is optimized for batch processing and analysis required during analysis of data and alert generation phases. Low latency queries of data within datasets may optionally be required, with the two main use cases being queries issued by the Investigation Center to fetch data related to an altered entity and enrichment queries issued by the decisioning step.

The ThetaRay platform provides two mechanisms for creating additional persistent representation of the information which targets low latency queries -

- Storing data within a Postgres database, an approach suitable for small / medium sized data which is up to a few terabytes in size
- Direct queries of Parquet files stored in Object Storage – functionality that builds one or more secondary representations of the dataset within Object Storage (Parquet Indexes) each index by a primary identifier (such as a client or account identifier) and time

(occurred on field). The mechanism enables low latency queries over big datasets (100s of billions of rows) and is integrated with the data tiering functionality to enable cost effective storage on public cloud environments (Azure).

### 3.1.6.1. Process Flow - High Level



**Figure 1: Interactive Parquet Queries - Process Flow High Level**

### 3.1.7. Enable Publishing Dataset to the CDD

The **publish** flag determines whether the dataset can be published to the Postgres database that serves as the platform Central Database Depo, or in short, CDD. The CDD is used for interactive querying in SQL.

The actual data publishing is carried out by the `publish()` function. But, for now, setting the indicator to true creates an empty table in **PostgreSQL** with the relevant schema.

```
publish=True,
```

If **Publish** is set to **false**, the dataset will only be stored in the *MinIO* object storage server better suited for extensive batch processing involving big data.

### 3.1.8. Enabling Data Publishing for Parquet Files Interactive Querying

Interactive Parquet Querying enables low latency querying of data stored in Parquet files on Object Storage through standard SQL queries issued from Postgres. The mechanism relies on a

separate representation of the data which is indexed by a user configured identifier (such as the account or customer identifiers) as well as the value of the 'occurred on' field.

Setting up the system for Parquet querying involves configured one or more *ParquetIndexes* for a dataset (each indexed by a specific identifier) and later publishing the data stored in the primary dataset storage to these Parquet indexes through a dedicated API.

Parquet Indexes are configured as part of a Dataset metadata definition as follows:

### Parquet Indexes definition for Datasets

```
return Dataset(  
....  
    parquet_indexes=[  
        ParquetIndex(  
            identifier='query_by_account_id',  
            identifier_field='account_id',  
            block_size=2*1024*1024,  
            num_identifier_bins=10,  
            identifiers_sample_size=2000,  
            ilm_setting=ILMSetting(DataTier.HOT,  
                local_storage_retention_days=1000)  
        )  
    ]
```

### Parquet Index definition for an Evaluation Flow

```
return EvaluationFlow(  
....  
    parquet_indexes=[  
        ParquetIndex(  
            identifier='query_by_account_id',  
            identifier_field='account_id',  
            block_size=2*1024*1024,  
            num_identifier_bins=10,  
            identifiers_sample_size=2000,  
            ilm_setting=ILMSetting(DataTier.HOT,  
                local_storage_retention_days=1000)  
        )  
    ]
```

---

**Limitation:** Array fields cannot be used in Datasets which utilize Parquet Indexes.

---

### 3.1.9. Writing to Datasets

Writing to datasets is carried out by the **dataset\_functions.write** function. The function accepts the source dataframe and the target dataset name and writes the source data to the target dataset.

```
from thetaray.api.dataset import dataset_functions

dataset_functions.write(context, source_dataframe, 'account_monthly_train', data_
environment=DataEnvironment.PUBLIC)
```

The **write** function accepts the following arguments:

- **context**: The context object includes the execution\_date parameter, from which the function learns what the execution month is. Since all tables are partitioned into chunks according to the data records execution month, the function can tell from the execution\_date parameter the target partition, see Read/Write to Public/Private Database Area.
- **source\_dataframe**: A Pyspark DataFrame object containing the data records written to the target dataset.
- **target\_dataset**: The identifier of the dataset into which the data records will be written.
- **data\_environment**: Whether the data environment is PRIVATE or PUBLIC; see Set Ingestion Modes.

The **write** function executes a different writing procedure for each **ingestion\_mode** type.

### 3.1.10. Writing Data in the Overwrite and Update Modes

In the **OVERWRITE** ingestion mode, the **write** function has to handle an entire snapshot of the data source; see Set Ingestion Modes.

However, it cannot simply replace the entire old data with the new snapshot on every upload since it has to keep historical records log to recreate the data states on particular past dates.

For this reason, the **overwrite** mode does not overwrite the old copy with the new copy but instead calculates the differences between them and updates the dataset accordingly. So, if for example, when writing accounts to the account dataset, a new account appears in the last snapshot, it is simply added to the account dataset.

However, if an account was updated in the last snapshot, its old data record is tagged "updated," and the updated new data record is also added to the dataset.

For the same reason, when an account disappears from the last snapshot, its record is not erased from the dataset but tagged "deleted" instead. To allow this merging of the new with the old snapshots **Overwrite** and Update ingestion modes tables are required to support primary keys.

In the **UPDATE** ingestion mode, the **write** function handles only data records that changed since the last data upload. It has a similar behavior as the **OVERWRITE** mode: adding the new and

updated data records in the dataset and tagging deleted records and old versions of the updated records as "deleted" and "updated," respectively.

### 3.1.11. Writing data in the Append ingestion mode

In the **APPEND** ingestion mode, the **write** function consumes the data records in their original order and appends them to the end of the dataset.

### 3.1.12. Publishing Datasets

#### 3.1.12.1. Overview

Data within Datasets can be stored in secondary data stores (with respect to the primary representation as Parquet files within Object Storage) to enable low latency querying during activities such as alert investigation. As detailed earlier in the document, datasets can be either published to Postgres tables (the Central Data Depot) for small / medium size datasets or as Parquet Indexes for datasets consisting of potentially 100s of billions of records.

The operation of storing a secondary representation of data in a Dataset is referred to as 'publish'. Prior to performing the operation, the Dataset definition must be configured to enable publishing as detailed in [Enabling Low Latency Queries for Datasets](#) above.

#### 3.1.12.2. Publishing Datasets data to Postgres (CDD)

Publishing data to Postgres is performed by calling the 'publish' function. Following is an example that illustrates the process.

```
from thetaray.api.dataset import dataset_functions
dataset_functions.publish(context, "account",
data_environment = DataEnvironment.PUBLIC)
```

#### 3.1.12.3. Publishing Datasets Data To Parquet Indexes

To write datasets data to Parquet Indexes the system provides a 'publish\_to\_parquet' API. Following is an example:

```
from thetaray.api.dataset.dataset_functions import publish_to_parquet
...
publish_to_parquet(context,
    <dataset_identifier>,
    <parquet_index_identifier>,
    data_environment=DataEnvironment.PUBLIC
```

)

### 3.1.13. Read/Write to Public / Private Database Area

The **MinIO** and *PostgreSQL* data stores dedicate for each user a private area where their data is accessible only to them. By default, the **read** and **write** functions operate in the user's private area when running from an interactive *PostgreSQL* environment. Still, you can change that by setting the **data\_environment** argument to *DataEnvironment.PUBLIC*. Consequently, the data will be written or read from the environment public area and be accessible to all users.

- All users can read from the public area, but only users with a manager role can write to it.

### 3.1.14. Reading from Datasets

Reading from datasets is carried out by the **dataset\_functions.read** function.

```
from thetaray.api.dataset import dataset_functions
ds_transaction = dataset_functions.read(context, source_dataset, to_job_ts =
context.execution_date, data_environment=DataEnvironment.PUBLIC)
```

The **read** function accepts the following arguments:

- **context**: the context object includes the `execution_date` parameter used for selecting which data records to read from the `source_dataset` table.
- **source\_dataset**: The identifier of the dataset containing the data records to be read.
- **to\_job\_ts**: The upper bound of a job execution date range. A data record `job_ts` identifies the job that generated the data. Data records having a timestamp of later execution dates will not be read.
- **from\_job\_ts**: The lower bound of a job execution date range. A data record `job_ts` identifies the job that generated the data. Data records having a timestamp of earlier execution dates will not be read.
- **data\_environment**: Whether the data environment is PRIVATE or PUBLIC; see *Read/Write to Public / Private Database Area*.

The **read** function returns a Pyspark DataFrame containing the read data records.

#### 3.1.14.1. Reading Data in the Overwrite and Update Ingestion Modes

The **read** function performs the following two consecutive steps when reading in the **UPDATE** and **OVERWRITE** ingestion modes:

1. Reads records whose timestamps are equal or smaller than the context execution date.
2. Find data records with the same primary key and return the data record with the latest timestamp.

In case Data at Rest Encryption is enabled within the environment, reading a dataset consisting of encrypted fields from a Jupyter Notebook environment will result in a Spark Dataframe consisting of encrypted data in columns marked as 'encrypted'. The user may either perform exploratory data analysis over the encrypted data as is or use the 'decrypt' API to explicitly decrypt the relevant rows of data.

### 3.1.15. Reading Data in the Append Ingestion Mode

By default, the **read** function in all ingestion modes doesn't read all the data records in the dataset but only those whose timestamps are equal to the context **execution\_date**). If you want the **read** function to read data records whose timestamp is before or equal to the execution\_date, use the **to\_job\_ts** argument. Use the **from\_job\_ts** argument to read data records after and equal the **execution\_date**.

## 3.2. Data Encryption

Optionally, the ThetaRay system can be configured to support data at rest encryption for selected dataset fields. When reading data within such Datasets from a Jupyter environment that for the relevant columns will be returned in encrypted form.

To allow operating on encrypted data within the Jupyter environment, authorized user can perform on-demand decryption / encryption of data using dedicated APIs. These APIs are available only to authorized users and any access to encrypted information is automatically audited.

The following APIs are available ->

```
def decrypt(context,  
df: DataFrame,  
dataset_identifier: str, reason: str)
```

Decrypt the content of a Spark Dataframe, previously obtained through a Dataset read operation (only field marked as encrypted will appear in encrypted form in the data frame). The API expects the user to pass a ThetaRay 'context' object, a Spark dataframe consisting of the read data, the identifier of the read dataset and a reason string which will be used for auditing purposes.

The function returns a new Spark Dataframe consisting of decrypted data.

```
dataset_functions.encrypt(context, df: DataFrame,  
dataset_identifier: str, reason: str)
```

Encrypt the fields marked as encrypted within a Spark Dataframe consisting of data in non encrypted form. The API expects the user to pass a ThetaRay context object, a Spark Dataframe consisting of non encrypted data, the identifier of the associated Dataset and a reason string for auditing purpose.

The function returns a new Spark Dataframe with columns set as encrypted consisting of data in encrypted form.

```
encryption.encrypt(context, data: List[Dict[str, Any]] = None, df: DataFrame  
=  
None)
```

Encrypt the content of a Python list of dictionary or a Spark Dataframe (if the list is not set). All fields will be encrypted and the data is not assumed to be associated with a concrete Dataset within the platform `encryption.decrypt(context, data: List[Dict[str, Any]], df: DataFrame = None)`

Decrypt the content of a Python list of dictionary or a Spark Dataframe (if the list is not set). All fields will be decrypted and the data is not assumed to be associated with a concrete Dataset within the platform.

---

**Note:**

A typical data scientist exploratory workflow will invoke decrypting a subset of a read dataset based on a filter criteria. This can be achieved in the following manner:

1. Use `encryption.encrypt` API to encrypt a selected customer names / account IDS that are typically stored in encrypted form at rest.
  2. Read the target dataset using `dataset_functions.read`.
  3. Filter out rows through Spark by comparing the encrypted values in (2) to the ones in (1).
  4. Decrypt the resulting Dataframe using `dataset_functions.decrypt` API.
- 

## 3.3. Data Tiering

### 3.3.1. Overview

The ThetaRay system leverages Object Storage based on Minio which is deployed into a Kubernetes / Openshift environment and utilizes by default local SSDs for persistent data storage. Public cloud environments (for this current release, Microsoft Azure is supported) provide managed object storage facilities which provides a more cost effective way for storing data. Data Tiering is a functionality that enables data to be dynamically moved from local disks to remote object storage based on user configurable policies, while still being able to access the data as if it was locally stored in Minio.

This overview section includes the following topics:

- Purpose
- Overview of tiers (cost versus availability)
- How it works
- What comes pre-configured (tiers configuration in Minio, ILM policy)

**Note:** Once data has transitioned to a specific tier, it is not possible to revert the data back to the original storage, nor is it possible to transition the data to a different tier.

### 3.3.2. Purpose

Taking data that sits in local disks in MinIO and transition it to managed object storage on a public cloud. The 6.3 release supports Azure BLOB Storage only.

### 3.3.3. Overview of Tiers

Tier Breakdown per Data Storage Device Type

Tier #‐Level	Storage Device	Storage Purpose
Local Storage	Local SSD	
Hot Tier	Azure Premium BLOB	Low latency random access to parquet files (available only starting from version 6.4)
Warm Tier	Azure Standard BLOB	Batch processing

### 3.3.4. How Data Tiering Works

Data Tiering works on the following data types:

1. Datasets of different types.
  - a. Append (transactions).
  - b. Overwrite.
  - c. Update.
2. Raw Data - connectors - uploaded files (csv).
3. Evaluation flows.

Minio remains the primary application storage layer, making the actual tier in which the data is held, transparent on an application level.

The following configurations should be set on the metadata in order for the appropriate data to be transitioned into tiers:

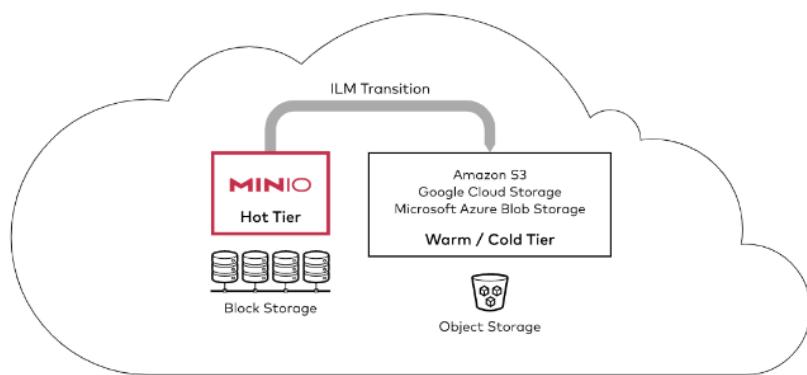
- Target tier - indicates to which tier the data should transition to
- Local Storage Retention Days - indicates how many days should the data remain in the local storage before transitioning to the target tier

The second configuration (local storage retention days) is relevant only for datasets of type APPEND and evaluation flows

### How it works

I

Minio has the ability to “move” data objects into storage tiers via ILM (Information Lifecycle Management) transition. Minio remains the primary application storage layer, making the actual tier in which the data is held transparent on an application level.



**Figure 2: Public Cloud Data Tiering Storage on MinIO - High level**

### 3.3.5. Configuration Details

In order to enable Data Tiering, this following needs to be implemented:

- Configuring Metadata
- Enabling periodic transitioning to remote tiers
- Transitioning existing data to remote tiers

**Note:** For detailed installation instructions and prerequisites please see the Platform Installation Guide

### 3.3.5.1. Configuring Metadata - Tiering Retention Policy

The Tiering Retention Policy is defined using ILMSetting class. It is used to define tiering for Datasets, Connectors and EvaluationFlows. It contains the following attributes:

- target\_tier attribute of type DataTier that can be either NONE, HOT or WARM
  - NONE - keeps the data in minIO local storage
  - HOT - transitions the data to the Hot tier
  - WARM - transitions the data to the Warm tier
- local\_storage\_retention\_days attribute that indicates the length of time the data should remain in the minIO local storage, before moving to the target tier. This attribute is relevant only for Datasets with IngestionMode APPEND and EvaluationFlows.

#### 3.3.5.1.1. Enable Storage Tiering for Datasets

Tiering a dataset requires configuring the optional field `ilm_setting` of type ILMSetting. It contains the target tier. Datasets with the IngestionMode override or update are tagged immediately. For a data set with the IngestionMode APPEND, you can also set `local_storage_retention_days`, which indicate the data should remain in the minio local storage for this period before moving it to its tier. If you wish it to be transferred immediately, you can set it as 0, or do not set it at all.

#### Example Dataset

```
from thetaray.api.solution import DataSet, Field, DataType, IngestionMode, ILMSetting, DataTier, 

def transaction_dataset():
    return DataSet(
        identifier="transaction",
        ...
        ilm_setting=ILMSetting(DataTier.WARM, local_storage_retention_days=30),
    )
```

```
def entities():
    return [transaction_dataset()]
```

### 3.3.5.1.2. Enable Storage Tiering for Evaluated Activities

Evaluation Flow tiering metadata is similar to Datasets with IngestionMode APPEND

#### Example Evaluation Flow

```
from thetaray.api.solution import EvaluationFlow, AlgoEvaluationStep,
ILMSetting, DataTier,
from thetaray.api.solution.evaluation import ModelReference

def evaluation_flow():
    return EvaluationFlow(
        identifier="account_monthly",
        ...
        ilm_setting=ILMSetting(DataTier.WARM, local_storage_retention_days=30),
    )
def entities():
    return [evaluation_flow()]
```

### 3.3.5.1.3. Enable Storage Tiering for Raw Data

Regarding the tiering connector, there is an optional field `ilm_setting` of type `ILMSetting` in the source section. It contains the target tier. Connector files are transferred immediately according to the settings.

#### Example Connector

```
from thetaray.api.connector import FlatFileConnector

from thetaray.api.connector.flat_file_connector.flat_file_connector import Source, Target,
TimestampConf, TimestampConfType
from thetaray.api.solution import ILMSetting, DataTier

def transaction_connector():
    """Build transaction FFC"""
    return FlatFileConnector(
        identifier="transaction",
        source=Source(
            ...
            ilm_setting=ILMSetting(DataTier.WARM),
```

```
),
...
)

def entities():
    return [transaction_connector()]
```

### 3.3.5.2. Enabling Periodic Transitioning to Remote Tiers

ThetaRay provides a set of APIs to enable periodic tagging of data with its target tier (a process that instructs Minio to move the data from local storage to remote object storage), as well as APIs which support migration of data existing in the system prior to enablement of the tiering functionality to remote storage based on the user configured policies.

Calls to these APIs can be automated by including them in Jupyter notebooks which are launched periodically as part of Airflow DAGs.

An example of a DAG required to execute periodically, is the tag\_tiered\_data.py script which is included in the solutions.reference.dags. In the reference implementation, it is a DAG that is set to run daily. It is recommended to set it (or the replacement that will execute its logic) to execute daily, at off-peak hours.

The DAG executes a query that finds data that has met the criteria for tagging, then tags it. The criteria for tagging is that the current datetime equals the occurred on date plus the local\_storage\_retention\_days.

#### 3.3.5.2.1. API Information

##### **thetaray.api.dataset.dataset\_functions.tag\_tiered\_partitions**

tag\_tiered\_partitions(context, data\_environment: DataEnvironment = DataEnvironment.get\_default(), forceTagging: bool = False)

Tag all relevant partitions of relevant datasets

##### Parameters

Name	Description	Mandatory?
context	Execution context details	Yes
data_environment	Environment mode (Public or Private), defining storage area to write into, default is set based on the execution environment	No
forceTagging	default is false Indicate whether to tag according to events in DB (false) or to scan and tag the entire bucket (true). Uses of forceTagging = True is mostly for upgrade scenarios	No

**thetaray.api.datatier.datatier\_functionstag\_tiered\_evaluation\_flows**

```
def tag_tiered_evaluation_flows(context, data_environment: DataEnvironment = DataEnvironment.get_default()):
    Tag all relevant partitions of relevant evaluation_flows
```

**Parameters**

Name	Description	Mandatory
Context	Execution context details	Yes
data_environment	Environment mode (Public or Private), defining storage area to write into, default is set based on the execution environment	No

[3.3.5.2.2. Notebooks Information](#)

1. tag\_tiered\_datasets\_and\_evaluation\_flows: tag dataset and evaluation flows according to metadata.
2. migrate\_to\_data\_tiering: should be run when upgrading to 6.3 after setting the metadata. It will tag all the existing data according to the metadata. It should also be used if the metadata has changed for the entity.

**Note:** A notebook or similar one to the following example should be scheduled periodically in order to tag dataset and evaluation flows according to metadata.

[3.3.5.2.3. Example Notebook Script](#)

```
from thetaray.api.context import init_context
from datetime import datetime

spark_conf={
    "spark.executor.memory": "8g",
    "spark.executor.cores": "4",
    "spark.executor.instances": "2"
}

context = init_context(execution_date=datetime(1970,1,1), spark_conf=spark_conf)
#%%
from thetaray.api.dataset.dataset_functions import tag_tiered_partitions
from thetaray.api.datatier.datatier_functions import force_tag_tiered_connectors, force_
tag_tiered_evaluation_flows
from thetaray.common.data_environment import DataEnvironment
#Tag existing datasets according to the metadata definition
#Run this in cases:
#1. System upgrade to 6.3
#2. Add ILM settings to dataset that hasn't had ILM settings.
```

```
tag_tiered_partitions(context=context, data_environment = DataEnvironment.PUBLIC,
forceTagging=True)

#Tag existing evaluation flows according to the metadata definition
#Run this in cases:
#1. System upgrade to 6.3
#2. Add ILM settings to evaluation flow that hasn't had ILM settings.
force_tag_tiered_evaluation_flows(context=context, data_environment =
DataEnvironment.PUBLIC)

#Tag existing connectors according to the metadata definition
#Run this in cases:
#1. System upgrade to 6.3
#2. Add ILM settings to connector that hasn't had ILM settings.
force_tag_tiered_connectors(context)
#%%
context.close()
```

### 3.3.5.3. Transitioning Existing Data to Remote Tiers

When upgrading existing metadata, you need to decide on the tiering policy for each entity of type Dataset, Connector and EvaluationFlow. After setting it, you should run the Jupyter notebook that is provided as a reference "migrate\_to\_data\_tiering.ipynb" or a similar notebook that can use the following APIs:

#### API 1 - tagging tiered datasets:

```
thetaray.api.dataset.dataset_functions.tag_tiered_partitions
(context=context, data_environment = DataEnvironment.PUBLIC,
forceTagging=True)
```

#### API 2 - Tagging tiered connectors:

```
thetaray.api.datatier.datatier_functions.force_tag_tiered_connectors(context)
```

### API 3 - Tagging tiered EvaluationFlows:

```
thetaray.api.datatier.datatier_functions.force_tag_tiered_evaluation_flows  
(context=context, data_environment = DataEnvironment.PUBLIC)
```

Run these APIs also if you change metadata for the existing entity, and also if you want to tier the historical existing data and not only affect new data.

**Note:** This changes the effect only on historical data that was not tiered. If it was tiered, it is impossible to change the tier for existing data and changes will affect only new data.

You can also use the following APIs to handle only a specific entity:

```
thetaray.api.dataset.dataset_functions._force_tag_dataset_tiered_partitions  
(context=context, ds_identifier=dataset identifier, data_environment =  
DataEnvironment.PUBLIC)
```

```
thetaray.api.datatier.datatier_functions._force_tag_evaluation_flow(context=context, ef_  
identifier=evaluation flow identifier, data_environment = DataEnvironment.PUBLIC)
```

```
thetaray.api.datatier.datatier_functions._force_tag_connector(context=context, sc_  
identifier=<connector identifier>)
```

### 3.3.6. Audit

MinIO publishes audit events regarding objects that been transferred to a tier. You can view these in OpenSearch by querying the "events" index with:

```
"event": " ilm:transition"  
(note the space character in  
" ilm:transition")
```

## 3.4. Data Purging

Topics covered in this chapter include:

- Overview
- Configuring Airflow DAG - Script and Scheduling
- Configuring Notebook
- Configuring Retention Policy (per Metadata)
- Audit

### 3.4.1. Overview

Efficient data management systems require the definition and implementation of an efficient data retention policy. The implemented policy needs to take into account that dataset types can vary across the spectrum and such requirements as corporate policy and regulatory needs.

Data purging supports two data types:

- a. Transactional data -> Datasets /Activities configured in APPEND ingestion mode and are stored in Minio and optionally in Postgres. Data is stored in monthly partitions in Minio and is partitioned by month in Postgres for non real time datasets.
- b. Raw Data - uploaded CSV files

#### 3.4.1.1. How does Purging Work?

For data to be purged, a retention policy needs to be set on the metadata with an indication of how many months to keep the data before it is purged. The retention policy is set on the dataset metadata for transactional data, and on the connector metadata for raw data.

A dedicated Airflow DAG needs to be set up and scheduled. The DAG will run according to schedule and purge the relevant data types according to the retention policy configured.

The referenced dates for purging for each data type are:

1. Transactional data - occurred on date, meaning the date the transactions occurred on. For example, setting a retention policy set at 24 months will purge all transaction data that occurred on a date previous to 2 years from the current date.
2. Uploaded CSV files - execution date, meaning the date the files were uploaded into Thetaray. For example, a retention policy set at 24 months will purge all of the csv files that were uploaded more than 2 years ago.

### 3.4.2. Configuring Airflow DAG - Script and Scheduling

This section of the Platform User Guide provides details on how to setup and schedule the DAG or DAGS to run and purge the various dataset types as and when required. Also included are example DAG scripts.

Datasets are purged via a specific configured and created Airflow DAG. The airflow purging DAG runs a notebook according to a schedule to scan and parse purging configured datasets, locate all matching partitions and purge the data held there.

### 3.4.2.1. Example Airflow DAG Script

```
import datetime

from airflow.models import DAG
from thetaray.api.airflow import RunNotebookOperator, ExecutionDateMode

default_args = {
    'owner': 'Airflow',
    'depends_on_past': False,
    'start_date': datetime.datetime(2019, 6, 13),
    # 'schedule_interval': '@daily',
    'retries': 1,
    'retry_delay': datetime.timedelta(minutes=5),
    'execution_date_mode': ExecutionDateMode.PERIOD_END,
}

dag = DAG(
    dag_id='purging',
    catchup=False,
    default_args=default_args,
    schedule_interval='0 0 1 * *' # At 00:00 on day-of-month 1
)

purge_data = \
    RunNotebookOperator(task_id="purging",
                        domain="default",
                        notebook_name="purging",
                        dag=dag)

purge_data

if __name__ == "__main__":
    dag.cli()
```

### 3.4.3. Implementing the Purging Notebook

A notebook needs to be created and configured in order to execute the purging scheduled in the DAG. The notebook reads the data and finds the partitions/folders suitable for purging, then purges.

**Note:**

- For data stored in Minio, the system purges entire monthly partitions - i.e. data will be purged once all data within the partition is associated with a timestamp the exceeded the retention policy.
- For data stored in Postgres ->
  - For non real time datasets / evaluated activities that are partitioned by month - as with Minio entire partitions are purged
  - For Datasets / Evaluation Flows that are marked as 'Realtime' , no partitioning is used and individual rows that are expired will be deleted.
- When purging data associated with 'Evaluated Activities' -> only rows that are not associated with alerts is purged. The rest of the data is retained in the system

### 3.4.3.1. Example Notebook Script

```
from thetaray.api.context import init_context
from thetaray.api.purging import purge
from datetime import datetime

spark_conf={
    "spark.executor.memory": "8g",
    "spark.executor.cores": "4",
    "spark.executor.instances": "2"
}

context = init_context(execution_date=datetime(1970,1,1), spark_conf=spark_conf)
purge(context)
```

**Note:** The purge API accepts an optional parameter 'rt\_ef\_month\_to\_purge' that controls how many months back from the expiration date to scan Postgres table for expired activities for real time Evaluation Flows. E.g. if the value is set to 3 months and the retention is 6 months - activities from 9 months ago to 6 months ago will be checked.

### 3.4.4. Configuring Retention Policy per Metadata

This section provides useful information about what exactly a Retention policy is, and includes useful guidelines on the key features that as a best practice, should be included in your custom retention policy.

Creating a retention policy determines how long the data will be maintained before purging. The retention policy is configured through a `RetentionPolicy` class that contains a field that holds the integer variable:

**months\_to\_keep:** the number of months the data is maintained before purging.

**Note:** A retention policy configured on a dataset not of type APPEND will fail validation.

### 3.4.4.1. Retention policy - Connector Example

```
from thetaray.api.connector import FlatFileConnector
from thetaray.api.solution import RetentionPolicy

from thetaray.api.connector.flat_file_connector.flat_file_connector import Source, Target,
TimestampConf, TimestampConfType

def transaction_connector():
    """Build transaction FFC"""
    return FlatFileConnector(
        identifier="transaction",
        display_name="transaction",
        source=Source(
            folder="transaction",
            has_header=True,
            delimiter=",",
            timestamp_conf=TimestampConf(type=TimestampConfType.EXECUTION_DATE, filename_pattern="%Y%m%d %H:%M:%S"),
            dataset_schema_ref="transaction",
            retention_policy=RetentionPolicy(months_to_keep=24)
        ),
        targets=[Target(
            dataset_name="transaction"
        )],
        default_target_dataset_name="transaction"
    )

def entities():
    return [transaction_connector()]
```

### 3.4.4.2. Retention policy - Dataset Example

```
from thetaray.api.solution import DataSet, Field, DataType, IngestionMode, RetentionPolicy

def transaction_dataset():
    return DataSet(
```

```
        identifier="transaction",
        display_name="transaction",
        field_list=[
            Field(identifier="trans_id", display_name="trans_id", data_type=DataType.LONG,
array_indicator=False),
            Field(identifier="account_id", display_name="account_id", data_
type=DataType.LONG, array_indicator=False),
            Field(identifier="date", display_name="date", data_type=DataType.TIMESTAMP,
date_format='%Y%m%d', array_indicator=False),
            Field(identifier="type", display_name="type", data_type=DataType.STRING, array_
indicator=False),
            Field(identifier="operation", display_name="operation", data_
type=DataType.STRING, array_indicator=False),
            Field(identifier="amount", display_name="amount", data_type=DataType.DOUBLE,
array_indicator=False),
            Field(identifier="balance", display_name="balance", data_type=DataType.DOUBLE,
array_indicator=False)
        ],
        ingestion_mode=IngestionMode.APPEND,
        publish=True,
        primary_key=["trans_id"],
        occurred_on_field="date",
        data_permission="dpv:public",
        retention_policy=RetentionPolicy(months_to_keep=24)
    )

def entities():
    return [transaction_dataset()]
```

### 3.4.5. Audit

Audit of the purging operation is saved to the events\_<solution> Opensearch index.

The following details are logged for each purge event as follows:

- a. Data Type Dataset / Connector
- b. Data Name: File name / Partition Name
- c. Partition Date ( execution date for raw data)
- d. Purging Datetime
- e. File size

### 3.4.5.1. Example of Dataset Audit Message:

#### Audit Message example - Dataset

Example of connector audit message:

```
{  
    "@version" : "1",  
    "partition_name" : "tr_year='1970',tr_month='1',tr_day='1',tr_date='1970_01_01_00_00_00',tr_partition='1994_08'",  
    "event_type" : "purge_dataset_data",  
    "dataset_identifier" : "transaction",  
    "timestamp" : "2022-11-01T00:00:00.000000Z",  
    "component" : "platform",  
    "host" : "10.24.19.25",  
    "partition_row_count" : 7600,  
    "@timestamp" : "2022-11-22T07:55:51.057111Z",  
    "partition_execution_date" : "1970_01_01_00_00_00",  
    "solution" : "ig"  
}
```

### 3.4.5.2. Example of Connector Audit Message:

#### Example Connector Audit Message

```
{  
    "@version" : "1",  
    "event_type" : "purge_raw_data",  
    "timestamp" : "2022-11-01T00:00:00.000000Z",  
    "file_full_name" : "upload/transaction/1970/01/01/1970_01_01_00_00_00/trans_En.csv",  
    "file_size" : 78333560,  
    "component" : "platform",  
    "host" : "10.24.19.25",  
    "@timestamp" : "2022-11-22T07:50:39.528175Z",  
    "file_execution_date" : "1970_01_01_00_00_00",  
    "solution" : "ig",  
    "connector_identifier" : "transaction"  
}
```

## 4. Data Acquisition

The connector's objective is to connect the dataset to its data source and facilitate data transfer from the latter to the former. The platform supports multiple connectors depending on the format and type of the external data source.

Currently, the platform supports the following connectors:

- FlatFileConnector supports delimited text files (e.g. CSV)
- SWIFT file connector supports reading RJE files consisting of SWIFT MT and MX formatted messages.
- SEPA File Connector supports reading XML formatted files consisting of SEPA messages.

 We configure connectors by setting connector metadata objects that we save in our `root/domains/<domain_name>/connectors` directory.

### 4.1. Configuring a Flat File Connector

The **FlatFileConnector** metadata object enable you to configure connecting and uploading data from a CSV data source.

#### 4.1.1. FlatFileConnector Metadata Object Example

```
from thetaray.api.connector import FlatFileConnector

from thetaray.api.connector.flat_file_connector.flat_file_connector import Source, Target, TimestampConf, TimestampConfType

def transaction_connector():

    return FlatFileConnector(
        identifier="transaction",
        display_name="transaction",
        source=Source(
            folder="transaction",
            has_header=True,
            delimiter=",",
            timestamp_conf=TimestampConf(type=TimestampConfType.EXECUTION_DATE,
            filename_pattern="%Y%m%d %H:%M:%S"),
            infer_schema=False,
```

```
        dataset_schema_ref="transaction"
    ),
    targets=[Target(
        dataset_name="transaction"
    ]),
    unique_target_dataset=False,
    default_target_dataset_name="transaction",
    rest_connector=RestConnector(
        enable=True,
        max_buffered_records=1000000,
        flash_interval=60
    )
)

def entities():
    return [transaction_connector()]
```

#### 4.1.2. Connector Attributes

- **identifier**: A unique and immutable name per dataset.
- **display\_name**: The title of GUI boxes displaying the field values.
- **source**: The configuration properties of the uploaded CSV file.
- **folder**: the name of the MinIO folder where the CSV file is stored.
- **has\_header**: Has\_header – the first line of the file includes a comma-separated list of column names.
- **delimiter**: The CSV field values separator.
- **timestamp\_conf**: The timestamp extraction mode; see Set the connector timestamp configuration type.
- **dataset\_schema\_ref**: The dataset whose schema is used for all datasets populated by the connector.
- **targets**: The target dataset list. Each target is dedicated to one of the target datasets and may include a dataset entry condition that data records must match.
- **default target dataset name**: The default dataset for records not matched to any target.
- **rest\_connector**: The configuration details of the rest connector enabling a client to send data to ThetaRay Platform's landing area; see Configure the REST connector.

#### 4.1.3. Set the Connector Timestamp Configuration Type

The connector appends a timestamp value to each data record to enable read and write activities in datasets with **OVERWRITE** and **UPDATE** ingestion modes. In these ingestion modes, the timestamp is necessary for deciding which data records to expire and which to keep active in case of data records of the same primary keys.

```
timestamp_conf=TimestampConf(type=TimestampConfType.EXECUTION_DATE, filename_pattern="%Y%m%d")
```

You can determine how the timestamps are set by selecting from the following **TimestampConfType** values:

- **EXECUTION\_DATE**: Set the timestamp value to the Airflow execution date.
- **FILENAME**: Set the timestamp execution date to the date appearing in the CSV file name. Parse the timestamp according to the pattern set in the filename\_pattern argument.
- **COLUMN**: Set the timestamp execution date to the record timestamp value specified in the data record under the column\_name argument. Parse the timestamp according to the pattern set in the column\_pattern argument.
- **FILENAME\_LAST MODIFY**: Set the timestamp execution date to the CSV modified date.

#### 4.1.4. Set the Connector Targets

The connector can upload data to multiple target datasets with the same schema or columns. The target object contains the dataset name and a dataset entry condition formulated in Spark SQL. For example, the condition '**country="spain"**' filters out all data records whose country value is not equal to spain.

```
targets=[  
    Target(  
        dataset_name="spain transaction"  
        condition='country="spain"'  
    ),  
    Target(  
        dataset_name="france transaction"  
        condition='country="france"'  
    )  
,
```

Data records not matching any conditions are routed to the dataset specified in the **default\_target\_dataset\_name** attribute.

```
default_target_dataset_name="spain transaction"
```

#### 4.1.5. Receiving Encrypted Files

The ThetaRay system optionally supports consumption of encrypted files arriving from the customer. The technical details of the encryption algorithm and setup of encryption keys are detailed in the Application Integration Interfaces document.

To handle encrypted files, an encryption element should be provided as part of the Connector configuration:

```
return FlatFileConnector( identifier="transaction", display_
name="transaction", source=Source(
...
),
encryption=Encryption(
chunk_rows_size = 10000,
parallelism = 4, key_file=suffix = ".DEK"
)
...
)
```

The Encryption configuration section sets the following parameters:

**chunk\_rows\_size** Number of rows that will be loaded into memory when parsing flat files

size

**parallelism** Number of Spark executors that will be spun up to handle file decryption and parsing

**key\_file\_suffix** The suffix of the file name of files which consist of the encrypted Data Encryption Key (See Application Integration Interfaces).

---

**Note:** When dealing with encrypted files decryption, parsing of an individual file takes place on a single CPU core. To parallelize the work, data should be partitioned into multiple files with parallelism set to control the level of concurrency when uploading multiple files.

---

##### 4.1.5.1. Generating Key Files from Notes

In some cases encryption of CSV files needs to happen from within the ThetaRay environment prior to invocation of the Flat File Connector. This is required in case files are uploaded to Minio in plain form, are encrypted by a custom notebook and deleted after being consumed by the connector.

This requires generation of a JSON file which includes an encrypted data encryption key and a reference to the Key Encryption Key. The JSON structure can be generated through, the following Python API -

**API -Function name:** encrypt\_dek

```
Import: from thetaray.api.encryption import encrypt_dek
```

### Usage example: encrypt\_dek(dek\_str)

Response JSON example:

```
1  {'key_id': 'https://name.vault.azure.net/...',  
2   'encrypted_key': 'XszyyJ398nq00MLtV9hym1Bm/  
3   ZeJrK1WTi2DqG3VnfWGd6gx5PYy7nPxxDFW1TXeBSxa4Ea2acvSgR1sGieJKq/  
4   tNw8ktvh6shh/dzj8gfcP3JdihpD3h4SqhRT6QNJoaZHnfc56uwqvsW18SI/  
5   zmHgdvFYIZFnna2iFCM/SPjxV7jiabPT1GeFCMDII8p1wPgsY4+14Bf3zmeHpCr5v/  
6   sAmRQ0T3QoclgXV1nV2J3iErZbe8F11nq26TtBDiEtWA95Ervp2vXza1cZSoqIwQQQ  
7   GowWuefyUj0YzM951Ks3EIkj3YDQ+c09zfuPqpAX5P3Q55EZ61cN/tz0hPsn5g=='}
```

Full usage example can be found in the Reference branch in the notebook:

```
domains/default/notebooks/encrypt_csv.ipynb
```

## 4.2. Configure the REST connector

A client can transmit data to the ThetaRay Platform using a REST API. The REST connector pulls the transmitted JSON data and converts it to a CSV file after the data has reached a configurable record number limit or a configurable period has elapsed. The file is then moved to the landing area in MinIO, where it can be uploaded to a dataset by a **FlatFileConnector**

The record number and size limits are configured in the *RestConnector* configuration object which is part of *FlatFileConnector* configuration object.

```
rest_connector=RestConnector(  
    enable=True,  
    max_buffered_records=1000000,  
    flash_interval=60  
)
```

The RestConnector attributes for flat connectors:

- **enable:** Enable the REST connector.
- **max\_buffered\_records:** The number of records allowed in a single CSV file.
- **flash\_interval:** The duration after which the transmitted records are moved to a single CSV file
- **nested:** true/false, default false, indicate if the message format will be flat or not

- **field\_mapping**: map between the expected json message format to dataset field names (mandatory if nested attribute is set to true)

### Example Script with Nested Connector

```
from thetaray.api.connector.flat_file_connector.flat_file_connector import Source, Target, TimestampConfType, \
    TimestampConf, RestConfig

def canonical_connector():
    return FlatFileConnector(
        identifier="canonical",
        display_name="canonical",
        source=Source(
            folder="canonical",
            has_header=True,
            delimiter=",",
            timestamp_conf=TimestampConf(type=TimestampConfType.EXECUTION_DATE, filename_pattern="%Y%m%d"),
            dataset_schema_ref="canonical",
            escape=""
        ),
        targets=[Target(
            dataset_name="canonical"
        )],
        default_target_dataset_name="canonical",
        rest_config=RestConfig(enable=True,
            max_buffered_records=6,
            flush_interval_minutes=5,
            read_chunk_size=2,
            nested=True,
            field_mapping=[("instructing_agent1.address.street", "agent1_address_street"),
                          ("instructing_agent1.address.number", "agent1_address_number"),
                          ("party.contact.name", "party_contact_name"),
                          ("id", "id"),
                          ("amount", "amount"),
                          ("notes", "notes")]
        )
    )

def entities():
    return [canonical_connector()]
```

For more information on the **REST API**, see the documentation for the Data Ingestion API (Application Integration Interfaces).

## 4.3. Configuring a SWIFT MT File Connector

The ThetaRay system support consumption of RJE files consisting of SWIFT MT messages by configuring the SwiftConnector metadata object. MT message parsing and conversion into a Dataset record format is enabled through Trace Transformer, a bundled parsing engine that enables flexible mapping between multiple MT message types and a ThetaRay dataset. Mapping logic customization is handled internally by ThetaRay and is made available as a JAR file that can be imported to the solution's GIT project. The SwiftConnector configuration includes references to pre-configured mapping operations to be performed per supported message.

### 4.3.1. SwiftConnector Metadata Object Example

```
from thetaray.api.connector.connector import TimestampConfType
from thetaray.api.connector.fin_connector.fin_connector import Target, TimestampConf
from thetaray.api.connector.swift_connector.swift_connector import SwiftMessageId,
SwiftSource, SwiftConnector, SwiftMapping

def swift_rje_connector():
    return SwiftConnector(
        identifier="swift_rje",
        display_name="swift_rje",
        source=SwiftSource(
            folder="swift_rje",
            timestamp_conf=TimestampConf(type=TimestampConfType.EXECUTION_DATE,
filename_pattern="%Y%m%d"),
            dataset_schema_ref="swift_transactions",
            validate_enable=True,
            mapping=[
                SwiftMapping(
                    message_ids=[SwiftMessageId("103", "CORE"), SwiftMessageId("103",
"STP")],
                    project_key="com.tr.swiftmapping",
                    service_name="SwiftMapping",
                    operation_name="MapMT103",
                    mapping_filename="/thetaray/git/solutions/domains/cbpack/swift_
default_mapping/swiftmapping.jar",
                    mapping_sheetname="Mapping",
                    mapping_default=True,
                    csv_has_header=True,
                    csv_quote='''',
                    csv_delimiter="|"
                ),
                SwiftMapping(
                    message_ids=[SwiftMessageId("202", "")],
                    project_key="com.tr.swiftmapping",
                    service_name="SwiftMapping",

```

```
        operation_name="MapMT202",
        mapping_filename="/thetaray/git/solutions/domains/cbpack/swift_
default_mapping/swiftmapping.jar",
        mapping_sheetname="Mapping",
        mapping_default=False,
        csv_has_header=True,
        csv_quote='"',
        csv_delimiter="|"
    ),
    SwiftMapping(
        message_ids=[SwiftMessageId("202", "COV")],
        project_key="com.tr.swiftmapping",
        service_name="SwiftMapping",
        operation_name="MapMT202COV",
        mapping_filename="/thetaray/git/solutions/domains/cbpack/swift_
default_mapping/swiftmapping.jar",
        mapping_sheetname="Mapping",
        mapping_default=False,
        csv_has_header=True,
        csv_quote='"',
        csv_delimiter="|"
    )
),
targets=[Target(
    dataset_name="swift_transactions"
)],
default_target_dataset_name="swift_transactions"
)
```

#### 4.3.2. Connector Attributes

- **identifier**: A unique and immutable name per dataset.
- **display\_name**: The title of GUI boxes displaying the field values.
- **source**: The configuration properties of the uploaded CSV file.
- **folder**: the name of the MinIO folder where the CSV file is stored.
- **has\_header**: Has\_header – the first line of the file includes a comma-separated list of column names.
- **delimiter**: The CSV field values separator.
- **timestamp\_conf**: The timestamp extraction mode; see Set the connector timestamp configuration type.

- **dataset\_schema\_ref:** The dataset whose schema is used for all datasets populated by the connector.
- **targets:** The target dataset list. Each target is dedicated to one of the target datasets and may include a dataset entry condition that data records must match.
- **default target dataset name:** The default dataset for records not matched to any target.
- **rest\_connector:** The configuration details of the rest connector enabling a client to send data to ThetaRay Platform's landing area; see [Configure the REST connector](#).

## 4.4. Configuring an ISO 20022 XML File Connector

The ThetaRay system support consumption of XML files consisting of SWIFT MX ISO 20022 messages by configuring the MXConnector Iso20022 Xml Connector metadata object.

MX ISO 20022 message parsing and conversion into a Dataset record format is enabled through Trace Transformer, a bundled parsing engine that enables flexible mapping between multiple MX ISO 20022 message types and a ThetaRay dataset.

Mapping logic customization is handled internally by ThetaRay and is made available as a JAR file that can be imported to the solution's GIT project.

The MXConnector Iso20022XmlConnector configuration includes references to pre-configured mapping operations to be performed per supported message type.

CBPR+:

- pacs.008.001.08
- pacs.009.001.08 Core
- pacs.009.001.08 COV
- pacs.009.001.08 ADV

Base MX:

- Camt.056.001.08
- pacs.002.001.10
- pacs.003.001.02
- pacs.003.001.08
- pacs.004.001.09
- pacs.007.001.09
- pacs.008.001.02
- pacs.008.001.10
- pacs.009.001.08 Core
- pacs.009.001.08 COV

- pain.001.001.03

For more information on the xml file specifications, please refer to the *Application Integration Interfaces* document.

#### 4.4.1. Iso20022Xml Connector Metadata Object Example

```
from thetaray.api.connector.connector import TimestampConfType
from thetaray.api.connector.iso_20022_xml_connector.iso_20022_xml_connector import
Iso20022XmlConnector, Iso20022XmlSource, Iso20022XmlMapping
from thetaray.api.connector.fin_connector.fin_connector import Target, TimestampConf

def mx_pacs_full_connector():
    return Iso20022XmlConnector(
        identifier="mx_pacs_full_connector",
        display_name="mx_pacs_full_connector",
        source=Iso20022XmlSource(
            folder="mx_pacs_full",
            timestamp_conf=TimestampConf(type=TimestampConfType.COLUMN, column_name='date',
column_pattern="%Y/%m/%d"),
            dataset_schema_ref="mx_pacs_full",
            validate_enable=True,
            mapping=[
                Iso20022XmlMapping(
                    message_names=[ "pacs.008.001.08"],
                    project_key="MXToCSV",
                    service_name="MXToCSVService",
                    operation_name="MXToCSV",
                    mapping_filename="/thetaray/git/solutions/domains/default/swift_mx_
mapping/MXToCSV.jar",
                    mapping_default=True,
                    csv_has_header=True,
                    csv_quote='"',
                    csv_delimiter="|"
                )
            ]
        ),
        targets=[Target(
            dataset_name="mx_pacs_full"
        )],
        default_target_dataset_name="mx_pacs_full"
    )

def entities():
    return [mx_pacs_full_connector()]
```

#### 4.4.2. Connector Attributes

- identifier: A unique and immutable name per connector.
- display\_name: The title of GUI boxes displaying the field values.
- source: The configuration properties of the uploaded ISO 20022 XML files. The configuration settings of the Iso20022XmlSource include:
- folder: the name of the MinIO folder under the public bucket datasource
  - folder where the XML file is stored
  - timestamp\_conf: The timestamp extraction mode; see Set the connector timestamp configuration type.
  - dataset\_schema\_ref: The dataset whose schema is used for all datasets populated by the connector.
  - validate\_enable: Indicates whether the schema should be validated or not
- A list of Iso20022XmlMapping objects that contain keys to call a specific mapping. Currently only one Iso20022XmlMapping object is supported

#### Platform

- targets: The target dataset list. Each target is dedicated to one of the target datasets and may include a dataset entry condition that data records must match.
- default target dataset name: The default dataset for records not matched to any target.

### 4.5. Configuring a SWIFT MX File Connector

---

**Note:** Not for use for new implementations! For new implementations please use ISO 20022 XML connector only.

---

The ThetaRay system support consumption of XML files consisting of SWIFT MX messages by configuring the **MXConnector** metadata object. MX message parsing and conversion into a Dataset record format is enabled through Trace Transformer, a bundled parsing engine that enables flexible mapping between multiple MX message types and a ThetaRay dataset. Mapping logic customization is handled internally by ThetaRay and is made available as a JAR file that can be imported to the solution's GIT project. The **MXConnector** configuration includes references to pre-configured mapping operations to be performed per supported message type.

The MX connector currently supports the following message types:

- pacs.008.001.x
- pacs.009.001.x / COV

For more information on the xml file specifications, please refer to the [\*\*Application Integration Interfaces\*\*](#) document.

#### 4.5.1. MX Connector Metadata Object Example

```
from thetaray.api.connector.connector import TimestampConfType
from thetaray.api.connector.mx_connector.mx_connector import MXConnector, MXSource,
MXMapping
from thetaray.api.connector.fin_connector.fin_connector import Target, TimestampConf

def mx_pacs_full_connector():
    return MXConnector(
        identifier="mx_pacs_full_connector",
        display_name="mx_pacs_full_connector",
        source=MXSource(
            folder="mx_pacs_full",
            timestamp_conf=TimestampConf(type=TimestampConfType.COLUMN, column_name='date',
column_pattern='%Y/%m/%d'),
            dataset_schema_ref="mx_pacs_full",
            validate_enable=True,
            mapping=[
                MXMapping(
                    message_names=["pacs.008.001.08"],
                    project_key="MXToCSV",
                    service_name="MXToCSVService",
                    operation_name="MXToCSV",
                    mapping_filename="/thetaray/git/solutions/domains/default/swift_mx_
mapping/MXToCSVNonFinal.jar",
                    mapping_default=True,
                    csv_has_header=True,
                    csv_quote='''',
                    csv_delimiter="|"
                )
            ]
        ),
        targets=[Target(
            dataset_name="mx_pacs_full"
        )],
        default_target_dataset_name="mx_pacs_full"
    )

def entities():
    return [mx_pacs_full_connector()]
```

#### 4.5.2. Connector Attributes

- ***identifier***: A unique and immutable name per connector.
- ***display\_name***: The title of GUI boxes displaying the field values.
- ***source***: The configuration properties of the uploaded MX XML files. The configuration settings of the MXSource include:
  - ***folder***: the name of the MinIO folder under the public bucket datasource folder where the XML file is stored
  - ***timestamp\_conf***: The timestamp extraction mode; see Set the connector timestamp configuration type.
  - ***dataset\_schema\_ref***: The dataset whose schema is used for all datasets populated by the connector.
  - ***validate\_enable***: Indicates whether the schema should be validated or not
  - ***mapping***: A list of MXMapping objects that contain keys to call a specific mapping. Currently only one MXMapping object is supported
- ***targets***: The target dataset list. Each target is dedicated to one of the target datasets and may include a dataset entry condition that data records must match.
- ***default target dataset name***: The default dataset for records not matched to any target.

### 4.6. Configuring a Sepa File Connector

---

**Note:** Not for use for new implementations! For new implementations please use ISO 20022 XML connector only.

---

The ThetaRay system support consumption of XML files consisting of ***Sepa*** messages by configuring the ***SepaConnector*** metadata object. Sepa message parsing and conversion into a Dataset record format is enabled through Trace Transformer, a bundled parsing engine that enables flexible mapping between multiple ***Sepa*** message types and a ThetaRay dataset. Mapping logic customization is handled internally by ThetaRay and is made available as a JAR file that can be imported to the solution's GIT project. The ***SepaConnector*** configuration includes references to pre-configured mapping operations to be performed per supported message type.

#### 4.6.1. SepaConnector Metadata Object Example

```
from thetaray.api.connector.connector import TimestampConfType
from thetaray.api.connector.sepa_connector.sepa_connector import SepaConnector, SepaSource,
SepaMapping
```

```
from thetaray.api.connector.fin_connector.fin_connector import Target, TimestampConf

def sepa_connector():
    return SepaConnector(
        identifier="sepa",
        display_name="sepa",
        source=SepaSource(
            folder="sepa",
            timestamp_conf=TimestampConf(type=TimestampConfType.EXECUTION_DATE, filename_pattern="%Y%m%d"),
            dataset_schema_ref="sepa",
            validate_enable=True,
            mapping=[
                SepaMapping(
                    message_names=["pain.001.001.03"],
                    project_key="com.tr.sepamapping",
                    service_name="SepaMapping",
                    operation_name="MapPain00100103",
                    mapping_filename="/thetaray/git/solutions/domains/default/sepa_default_mapping/sepamapping.jar",
                    mapping_sheetname="Mapping",
                    mapping_default=True,
                    csv_has_header=True,
                    csv_quote='"',
                    csv_delimiter="|"
                )
            ]
        ),
        targets=[Target(
            dataset_name="sepa"
        )],
        default_target_dataset_name="sepa"
    )
```

#### 4.6.2. Connector Attributes

- **identifier**: A unique and immutable name per dataset.
- **display\_name**: The title of GUI boxes displaying the field values.
- **source**: The configuration properties of the uploaded MX XML files. The configuration settings of the SepaSource include :
  - **folder**: the name of the MinIO folder where the CSV file is stored.

- **timestamp\_conf**: The timestamp extraction mode; see Set the connector timestamp configuration type.
- **dataset\_schema\_ref**: The dataset whose schema is used for all datasets populated by the connector.
- **mapping** :A list of **SePaMapping** objects that reference the Trace Transformer configuration to be invoked in order to perform the message parsing (as provided by ThetaRay). The information includes a reference to the JAR that includes the custom mapping and the name of the mapping service / operation to be invoked for performing the parsing operation. A separate **SePaMapping** object should be provided per supported message type.
- **targets**: The target dataset list. Each target is dedicated to one of the target datasets and may include a dataset entry condition that data records must match.
- **default target dataset name**: The default dataset for records not matched to any target.

## 4.7. Upload Data into Datasets

The **upload** function activates the connector and triggers the data upload process from the data source to the datasets. Like all dataset write activities, the dataset ingestion mode determines the **upload** function behavior.

```
upload(context, "account", data_environment = DataEnvironment.PUBLIC)
```

The **upload** function accepts the following arguments:

**context**: The context object includes the execution\_date parameter, from which the function learns what the execution month is. Since all tables are partitioned into chunks according to the data records execution month, the function can tell from the execution\_date parameter the upload target partition.

- **target\_dataset**: The dataset into which the data is uploaded.
- **data\_environment**: Whether the data environment is PRIVATE or PUBLIC; see *Read/Write to Public/Private Database Area*.
- **fail\_on\_no\_data**: Trigger an exception if no data is available to upload. False by default
- **use\_control\_files**: Avoids moving data to the 'upload' folder in the private bucket and instead uses a control file within the data source folder to indicate that a file has already been consumed by a job. This must be used along with purging to limit the number of files retained in the data source folder. False by default.
- **schema\_padding**: - If set to true, missing columns in the input file will be set to null values in the target dataset. Defaults to false.

- `upload_by_execution_date`: Upload files from a sub-folder of the 'connector folder' associated with the context's execution date - the folder name should be in the format '`%Y_%m_%d_%H_%M_%S`' (year, month, day, hour, minutes, second). This allows historical data to be gradually uploaded through Airflow DAG instances, each associated with a different execution date.

## 4.8. Uploading Datasource(s) Via Minio UI

If the preferred way to upload a datasource is manually via the MinIO UI.

### Prerequisites

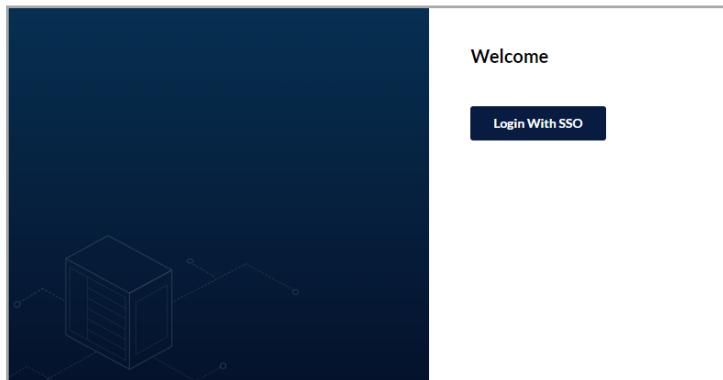
Data source(s) stored on your local computer.

**From your environment, login to the MinIO console as follows:**

1. In your browser, enter the MinIO url, replacing the placeholder elements with your domain etc., as shown in the following example:

```
https://minio-<suffix><tenant>.thetaray.cloud
```

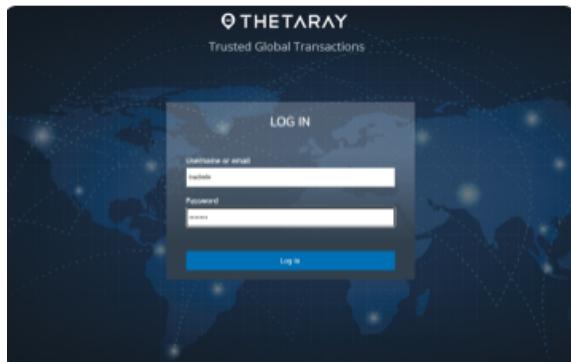
The Console Login with SSO welcome page is displayed as shown in the following figure:



**Figure 3: MinIO Console Login Page**

2. Click the Login with SSO button.

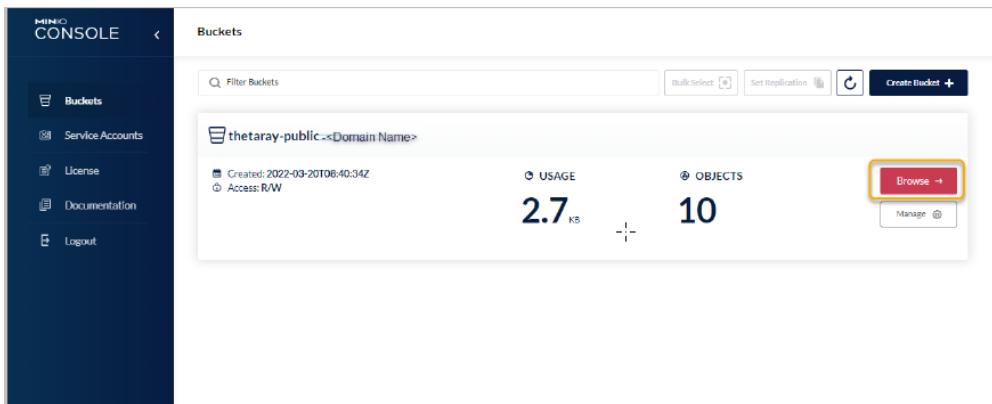
The Login page is displayed as shown below.



**Figure 4:** MinIO Credentials Login Screen

3. Enter the required username and password credentials (available from ThetaRay Customer Success).
4. Click the Login button.

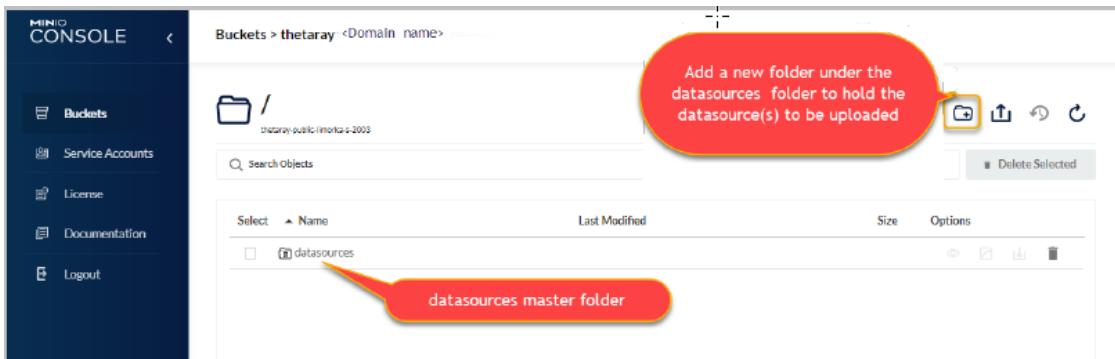
The MinIO CONSOLE screen is displayed as shown in the following example.



**Figure 5:** Example MinIO CONSOLE Display

5. Click the 'Browse' button, as highlighted in the figure above.

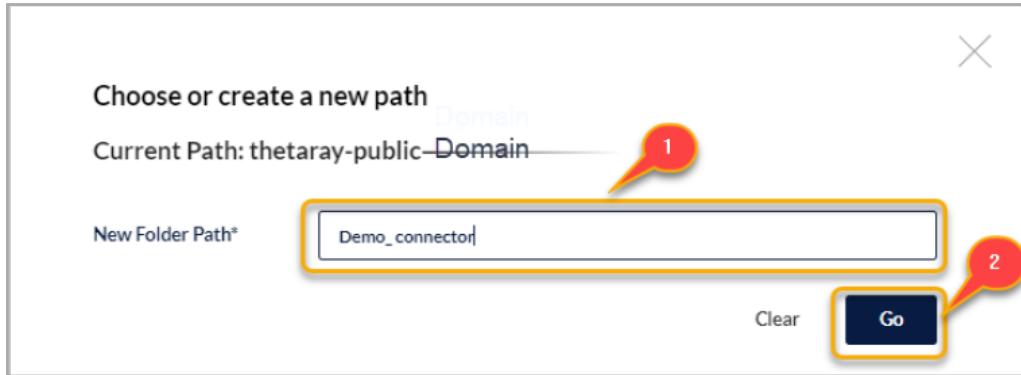
The following main datasources display screen is displayed.



**Figure 6:** Bucket Path, Add Folder and Datasources Master Folder

6. Select the 'datasources' folder.

7. Click the highlighted 'Add Bucket Folder' icon. to display the following folder path popup:



**Figure 7:** New Folder Path Definition Field

8. Define New Folder Path.

9. Click the 'Go' button.

The newly created folder is created and displayed as shown below.



**Figure 8:** Example of Created New Folder - Confirmed

10. Click the Upload Icon as highlighted.
11. Navigate to the folder holding your Datasource(s) for upload.
12. Double click on the .csv or .rje to upload to the bucket folder.
13. Finally, as a best practice, verify the data source upload by displaying the content of the 'datasources' master folder.

## 4.9. Configure the REST Connector

A client can transmit data to the ThetaRay Platform using a REST API. The REST connector pulls the transmitted JSON data and converts it to a CSV file after the data has reached a configurable record number limit or a configurable period has elapsed. The file is then moved to the landing area in MinIO, where it can be uploaded to a dataset by a FlatFileConnector.

The record number and size limits are configured in the **RestConnector** configuration object which is part of **FlatFileConnector** configuration object.

```
rest_connector=RestConnector(  
    enable=True,  
    max_buffered_records=1000000,  
    flash_interval=60  
)
```

The RestConnector attributes:

- **enable**: Enable the REST connector.
- **max\_buffered\_records**: The number of records allowed in a single CSV file.
- **flash\_interval**: The duration after which the transmitted records are moved to a single CSV file.

For more information on the REST API, see the documentation for the Date Ingestion API.

#### 4.9.1. Upload Data into Datasets

The **upload** function activates the connector and triggers the data upload process from the data source to the datasets. Like all dataset write activities, the dataset ingestion mode determines the **upload** function behavior.

```
upload(context, "account", data_environment = DataEnvironment.PUBLIC)
```

The **upload** function accepts the following arguments:

context: The context object includes the execution\_date parameter, from which the function learns what the execution month is. Since all tables are partitioned into chunks according to the data records execution month, the function can tell from the execution\_date parameter the upload target partition.

- target\_dataset: The dataset into which the data is uploaded.
- data\_environment: Whether the data environment is PRIVATE or PUBLIC; see *Read/Write to Public/Private Database Area*.

### 4.10. Uploading Datasource(s) Via Minio UI

If the preferred way to upload a datasource is manually via the MinIO UI.

#### Prerequisites

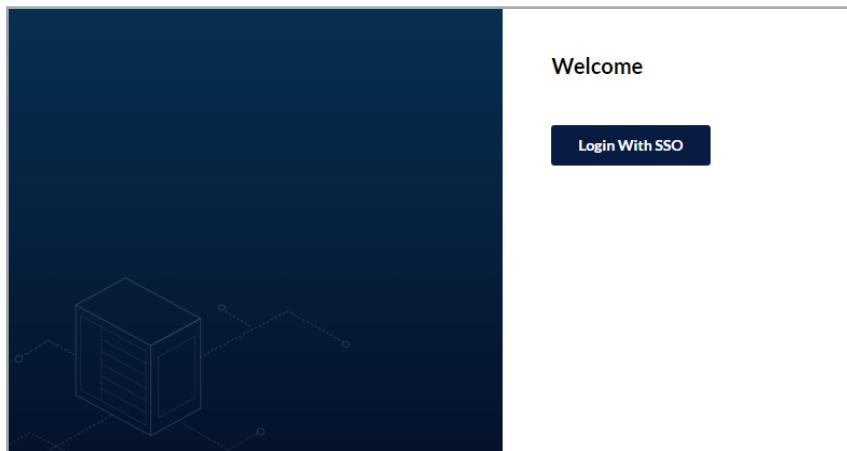
Data source(s) stored on your local computer.

**From your environment, login to the MinIO console as follows:**

1. In your browser, enter the MinIO url, replacing the placeholder elements with your domain etc., as shown in the following example:

```
https://minio-<suffix><tenant>.thetaray.cloud
```

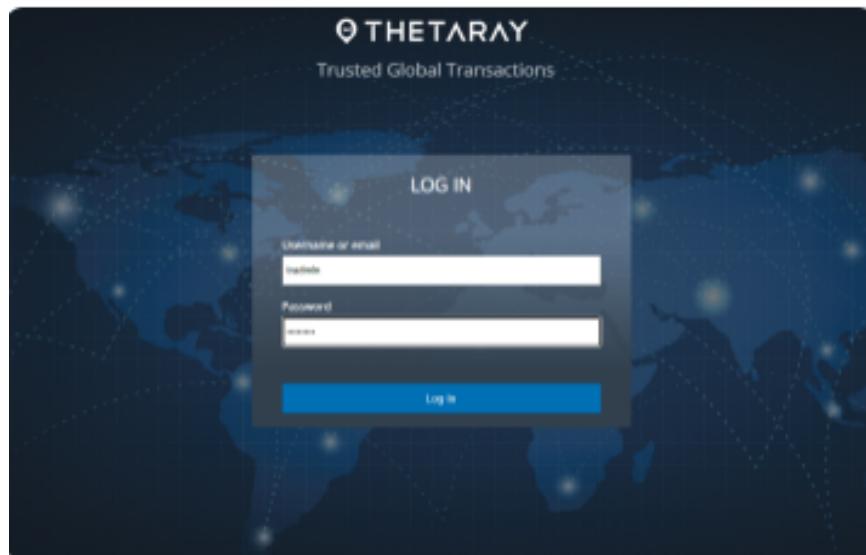
The Console Login with SSO welcome page is displayed as shown in the following figure:



**Figure 9:** MinIO Console Login Page

2. Click the Login with SSO button.

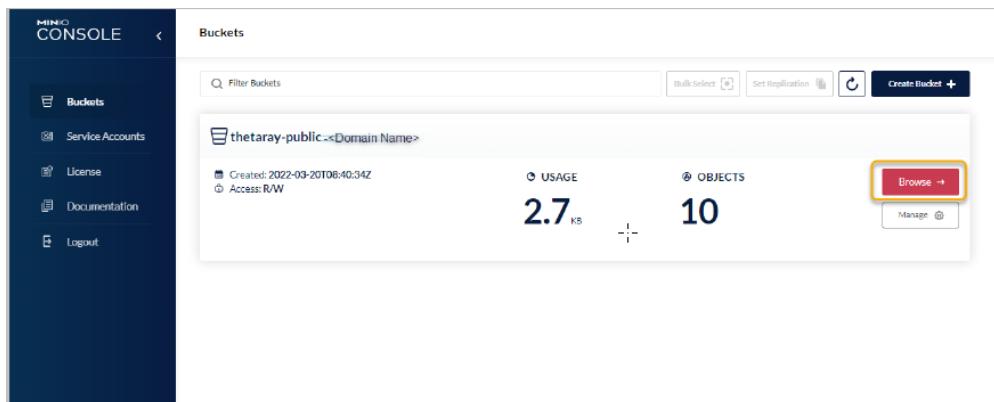
The Login page is displayed as shown below.



**Figure 10:** Minio Credentials Login Screen

3. Enter the required username and password credentials (available from ThetaRay Customer Success).
4. Click the Login button.

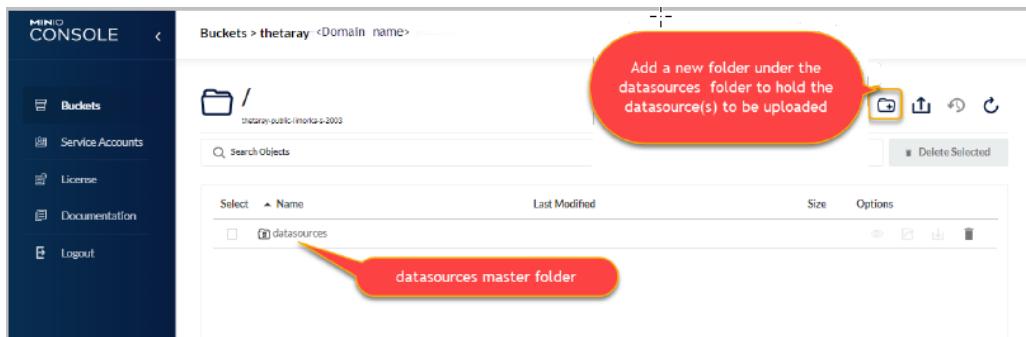
The MinIO CONSOLE screen is displayed as shown in the following example.



**Figure 11:** Example MinIO CONSOLE Display

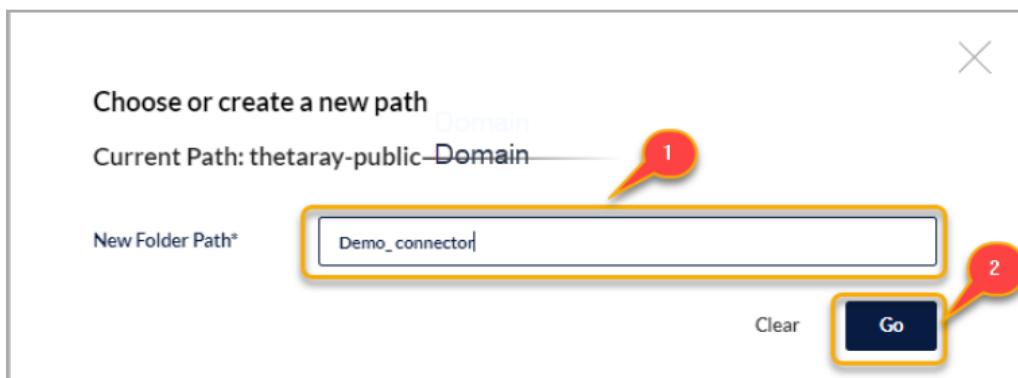
5. Click the 'Browse' button, as highlighted in the figure above.

The following main datasources display screen is displayed.



**Figure 12:** Bucket Path, Add Folder and Datasources Master Folder

6. Select the 'datasources' folder.
7. Click the highlighted 'Add Bucket Folder' icon. to display the following folder path popup:



**Figure 13:** New Folder Path Definition Field

8. Define New Folder Path.
9. Click the 'Go' button.

The newly created folder is created and displayed as shown below.



**Figure 14:** Example of Created New Folder - Confirmed

10. Click the Upload Icon as highlighted.
11. Navigate to the folder holding your Datasource(s) for upload.
12. Double click on the .csv or .rje to upload to the bucket folder.
13. Finally, as a best practice, verify the data source upload by displaying the content of the 'datasources' master folder.

## 5. Model Training

The ThetaRay Platform enables you to train a machine learning model for detecting anomalies based on a sample of data. The model can be later used for evaluating large volumes of new data by automatic evaluation processes running on predefined schedules.

ThetaRay's algorithms operate by first establishing data normality using historical data sets. Later the algorithms detect anomalies by this information.

Establishing data normality involves training multiple machine learning models using APIs provided by ThetaRay. These APIs contracts are based on interfaces defined by the widely used Scikit Learn Python libraries ->, i.e., using a fit method to train a model.

The training process culminates in a machine learning model stored in ThetaRay's embedded MLFlow-based Model Repository.

Each trained model is assigned a unique name and version within the repository, allowing traceability to the specific model version used for detection. The version name consists of one or more user-defined tags and includes an internal reference to the code version used during the training process. This reference enables tracing the code that computed the features for reproducibility and explainability.

In addition, the platform automatically attaches the data used during training to the model, thus allowing enhanced auditability and automated drift detection processes to use the data.

The main output of the model training process consists of two artifacts: feature extractor-transformer and anomaly detection model. So our first step is preparing the ground for building the two artifacts by setting their respective configuration objects: **FeaturesExtractor** and **ThetaRayDetector**. The two objects store the configuration settings.

### 5.1. Configuring Feature Extractor

The goal of a feature extractor is threefold:

- First, selecting a subset of features to train the model.
- In numerical features: Setting the rules for transforming null values to non-null values according to a selected imputation strategy.
- In categorical (non-numerical) features: Setting the rules for transforming categorical (non-numerical) values to numerical values.

#### 1. Setting Numerical Feature Transformer

Let's start with selecting the columns of the numerical features that need to be extracted.

```
numerical_features = [  
    "sum_amount",
```

```
"zscore3",  
"zscore6",  
"zscore12"  
]
```

Next, we configure the strategy for filling in missing values and nulls by setting the **NumericFeaturesTransformer** configuration object. The strategy selected here is the mean strategy whereby the system replaces nulls and missing values with the calculated average of all feature values.

- **features**: A list of the Categorical features/columns.
- **strategy**: How to replace nulls or missing values (imputation).
  - **mean**: Replace missing value with the average of all column values.
  - **median**: Replace missing value with the median of all column values.
  - **constant**: Replace the missing value with the fill\_value value.
  - **most\_frequent**: Replace missing value with the column's most frequent value. The smallest one will be used if there is more than one value.
- **fill\_value**: The constant value used when selecting the constant value in the strategy argument.

## 2. Setting Categorical Feature Transformer

Now let's continue with categorical features. But, first, we single out the columns of the categorical features that need to be extracted.

```
categorical_features = [  
    "col5",  
    "col6",  
    "col7",  
    ""  
]
```

Next, we configure the strategy for transforming categorical values to numerical values by setting the **CategoricFeaturesTransformer** configuration object.

```
from thetaray.api.evaluation.preprocess.categorical_features import  
CategoricFeaturesTransformer  
  
cft = CategoricFeaturesTransformer(features=categorical_features, strategy=none, fill_  
value=None)
```

The CategoricFeaturesTransformer has the following arguments:

- **features**: A list of the Categorical features/columns.
- **mapping**: a mapping of the categorical to a numerical value (optional). The dictionary contains the keys 'col' and 'mapping'. The value of 'col' should be the feature name. The value of 'mapping' should be a dictionary of '**original\_label**' to '**encoded\_label**'. For example:

```
mapping=[  
    {'col': 'col1', 'mapping': {None: 0, 'a': 1, 'b': 2}},  
    {'col': 'col2', 'mapping': {None: 0, 'x': 1, 'y': 2}}  
]
```

- **strategy**: Strategy of replacing missing values.
  - **constant**: Replace the missing value with the fill\_value parameter value.
  - **most\_frequent**: Replace missing value with the column's most frequent value. The smallest one will be used if there is more than one value.
- **fill\_value**: The constant value used when selecting the constant value in the strategy argument

### 3. Setting Feature Extractor

We then instantiate a **FeatureExtractor** object using our categorical and numerical feature transformers.

```
from thetaray.api.evaluation.preprocess.features_extractor import FeaturesExtractor  
  
features_extractor = FeaturesExtractor([nft, cft])
```

## 5.2. Configuring ThetaRay Detector

After we have configured the **FeatureExtractor**, we can continue to configure **the ThetaRayDetector**. Essentially, we set the types of algorithms that will be used to detect anomalies.

```
from thetaray.api.anomaly_detection import ThetaRayDetector  
tr_detector = ThetaRayDetector(*parameters)
```

## 5.2.1. Class Definition

```
Class ThetaRayDetector(*,
    learning_method=1,
    algo_type=['Ny', 'RL', 'NF', 'NK', 'Pg', 'HB', 'GC', 'GL'],
    algo_type_for_semisupervised=None
    normalization_type=1,
    Fusion_threshold=0.5,
    Rating_percentile=1.0,
    perf_metric='Fbeta',
    beta_for_F=2.0,
    thread_mode=1,
    max_wrk=None
    set_zero_rating=0
)
```

## 5.2.2. Class Parameters:

learning\_method: integer, default = 1 (unsupervised learning)

- 1 – unsupervised
- 4 – semi-supervised

```
algo_type: string, default=['Ny', 'RL', 'NF', 'NK', 'Pg', 'HB', 'GC', 'GL']
```

The list of unsupervised algorithms to run, if learning method is set to 1. The description of each algorithm appears in a separate document Product Desc - Survey of Algorithms – ThetaRay). It should be one or more of the algorithms from the list above. If none is provided, all the 8 algorithms above apply.

```
algo_type_for_semisupervised: string, default=None
```

The list of semi-supervised algorithms to run, if learning method is set to 4. The description of each algorithm appears in a separate document Product Desc - Survey of Algorithms – ThetaRay). It should be one or more of the algorithms from the following list: ['NNS', 'GCS', 'GLS', 'DWS', 'EMS']. If none is provided, all the 5 algorithms above apply.

```
normalization_type: int, default=1
```

Type of normalization (scaling) used and applied to the input data set. The parameters used are first learned from the training data set and then applied to the detection data set. Possible values are 0 - no normalization, 1 - min-max, 3 - sigmoid, 4 - zscore.

```
Fusion_threshold: positive float, default=0.5
```

**Fusion\_threshold** is a probability and determines if an entry is an anomaly or not. An anomaly is an entry with a probability (fused score) above the Fusion\_threshold. Fusion\_threshold must be in the range [0.0, 1.0].

```
Rating_percentile: positive float, default=1.0
```

By default, feature rating is calculated automatically for each anomaly, that is for each entry with a fused score above **Fusion\_threshold**. By definition, the entries with scores below the Fusion threshold are **normal**. Rating is then calculated also for a percentage Rating\_percentile of those normal entries with the highest score. Rating\_percentile must be in the range [0.0,5.0]. This enables a “below-the- line” analysis of the alerts found.

```
perf_metric: string, default='Fbeta'
```

perf\_metric is the metric that is being maximized during the optimization of the semi-supervised algorithms hyperparameters. By default, the metric is set to 'Fbeta', the weighted harmonic mean of precision and accuracy, with beta\_for\_F (see below) as a weight on recall. It can be set to one of the following strings: 'Fbeta', 'F1', 'AUC', 'Accuracy', 'Precision' or 'Recall'. It should be mentioned that for all the metrics above, except 'AUC', the fusion threshold is also modified during the optimization process.

```
beta_for_F: positive float, default=2.0
```

Parameter used in conjunction with Fbeta. It represents the ratio of recall importance to precision importance. beta\_for\_F > 1 gives more weight to recall, while beta\_for\_F < 1 favors precision. With beta\_for\_F = 1 we recover F1.

```
thread_mode: int, default=1
```

When set to 1 (default), ThetaRay's algorithms will utilize multiple threads to parallelize model fitting and prediction operations in areas that are not natively parallelized by the underlying libraries such as Tensorflow. When set to 0, these areas of the computation will run on a single CPU core.

```
max_wrk: String, default='None'
```

If thread\_mode is set to 1, this controls the number of threads that will be launched to parallelize work. If set to 'None', the number of threads is based on the number of CPU cores available to the

Kubernetes Container hosting the job. If an explicit numeric value is provided, this would be the number of launched threads.

```
set_zero_rating: int, default=0
```

If `set_zero_rating` is set to 1, an algorithm determines if the value of an anomaly for a specific feature is equal to that of the normal entries. If this is the case it sets the ratings of those features to zero. `set_zero_rating` is 0 by default.

### 5.2.3. Methods

```
Definition: fit(X, Y=None)      Purpose: Fit the model with X
```

#### Parameters

**X: array-like of shape (n\_samples, n\_features)**

Training data, where `n_samples` is the number of samples and `n_features` is the number of features.

**Y: vector of labels (n\_samples,)**

Labels

- a. not used if unsupervised algorithms
- b. If semi-supervised algorithms: Integer,
  - 1 anomaly (mandatory)
  - 0 unknown (mandatory)
  - -1 normal (if available)

#### Returns

**self: object**

Returns the instance itself.

```
Definition: predict(X)      Purpose: Perform prediction on X
```

#### Parameters

**X: array-like of shape (n\_samples, n\_features)**

Detection data, where `n_samples` is the number of samples and `n_features` is the number of features.

#### Returns

**array of shape (n\_samples, )**

Scores or probabilities, ranges between 0.0 (normal) and 1.0 (abnormal). Similar to predict\_proba in scikit-learn.

Definition: feature_rating(Sc)	Purpose: Calculate feature ratings, given the scores Sc
--------------------------------	---

## Parameters

Sc: array of shape (n\_samples,)

Scores or probabilities – the results of the predict() method

## Returns

**array of shape (N, 2\*n\_features + 1)**

Feature ratings table, where:

- N is the number of anomalies + Rating\_percentile\*(n\_samples-number of anomalies)/100
- The first column of the rating array includes the pks of the anomalies in the order they appear in the initial detect data set.

The rest of the rating array is ordered according to:

- | Feature index | Rating | Feature index | Rating | ...
- The ratings appear in decreasing order from the left to the right for each anomaly.
- The ratings are multiplied by 100 and rounded to integers.
- The sum of the ratings equal 10000.

## Example:

PK	Feature index	Rating	Feature index	Rating	Feature index	Rating	Feature index
3526	24	1244	4	1141	21	723	...
7658	8	5301	4	824	31	667	...
...	...	...	...	...	...	...	...

## 5.3. Configuring ThetaRay Optimizer

### 5.3.1. About ThetaRay Optimizer

The ThetaRayOptimizer class enables auto tuning of hyper parameters used in ThetaRay's Semi Supervised algorithms. The class returns both a trained semi supervised model that can be saved into the model repository directly as well as the hyper parameters used to train it that can be

passed as inputs at a later phase to the ThetaRayDetector class described earlier in this document.

### 5.3.2. Continuing the Configuration

After we have configured the **FeaturesExtractor**, we can continue to configure the **ThetaRayOptimizer**. Essentially, we set the types of algorithms that will be used to detect anomalies.

```
from thetaray.api.theta_optimize
import Optimizer tr_optimizer =
ThetaRayOptimizer(*parameters)
```

### 5.3.3. Class Definition

```
Class ThetaRayOptimizer(
    algo_type_for_semisupervised=['NNS', 'GCS', 'GLS', 'DWS', 'EMS']
    normalization_type=1,
    Fusion_threshold=0.5,
    Rating_percentile=1.0,
    N_bayesian_itr=30,
    thread_mode=1,
    max_wrk=None,
    pef_metric = 'Fbeta' ,
    beta_for_F=2.0
    thread_mode=1,
    max_wrk=None,
    set_zero_rating=0
)
```

### 5.3.4. Class Parameters:

```
algo_type_for_semisupervised: string, default==['NNS', 'GCS', 'GLS', 'DWS', 'EMS']
```

The list of semi-supervised algorithms to run. The description of each algorithm appears in a separate document Product Desc - Survey of Algorithms – ThetaRay). It should be one or more of the algorithms from the following list: ['NNS', 'GCS', 'GLS', 'DWS', 'EMS']. If none are provided, all the 5 algorithms above apply.

```
normalization_type: int, default=1
```

Type of normalization (scaling) used and applied to the input data set. The parameters used are first learned from the training data set and then applied to the detection data set. Possible values are 0 - no normalization, 1 - min-max, 3 - sigmoid, 4 - zscore.

```
Fusion_threshold: float, default=0.5
```

**Fusion\_threshold** is a probability and determines if an entry is an anomaly or not. An anomaly is an entry with a probability (fused score) above the Fusion\_threshold. Fusion\_threshold must be in the range [0.0, 1.0].

```
Rating_percentile: float, default=1.0
```

By default, feature rating is calculated automatically for each anomaly, that is for each entry with a fused score above **Fusion\_threshold**. By definition, the entries with scores below the Fusion threshold are **normal**. Rating is then calculated also for a percentage **Rating\_percentile** of those normal entries with the highest score. Rating\_percentile must be in the range [0.0, 5.0]. This enables a “below-the- line” analysis of the alerts found.

```
N_bayesian_itr: int, default=30
```

Number of iterations for the Bayesian optimizer. At each iteration, a cross-validation is performed for each set of hyper parameters chosen, according to the maximum of the expected improvement criterium.

```
thread_mode: int, default=1
```

When set to 1 (default), ThetaRay’s algorithms will utilize multiple threads to parallelize model fitting and prediction operations in areas that are not natively parallelized by the underlying libraries such as Tensorflow. When set to 0, these areas of the computation will run on a single CPU core.

**perf\_metric** is the metric that is being maximized during the optimization of the semi-supervised algorithms hyperparameters. By default, the metric is set to ‘Fbeta’, the weightedharmonic mean of precision and accuracy, with beta\_for\_F (see below), as a weight on recall. It can be set to one of the following strings: ‘Fbeta’, ‘F1’, ‘AUC’, ‘Accuracy’, ‘Precision’ or ‘Recall’. It should be mentioned that for all the metrics above, except ‘AUC’, the fusion threshold is also modified during the optimization process.

```
beta_for_F: positive float, default=2.0
```

Parameter used in conjunction with Fbeta. It represents the ratio of recall importance to precision importance. beta\_for\_F > 1 gives more weight to recall, while beta\_for\_F < 1 favors precision. With beta\_for\_F = 1 we recover F1.

```
thread_mode: int, default=1
```

When set to 1 (default), ThetaRay's algorithms will utilize multiple threads to parallelize model fitting and prediction operations in areas that are not natively parallelized by the underlying libraries such as Tensorflow. When set to 0, these areas of the computation will run on a single CPU core.

```
max_wrk: String, default='None'
```

If `thread_mode` is set to 1, this controls the number of threads that will be launched to parallelize work. If set to 'None', the number of threads is based on the number of CPU cores available to the Kubernetes Container hosting the job. If an explicit numeric value is provided, this would be the number of launched threads.

```
set_zero_rating: int, default=0
```

If `set_zero_rating` is set to 1, an algorithm determines if the value of an anomaly for a specific feature is equal to that of the normal entries. If this is the case it sets the ratings of those features to zero. `set_zero_rating` is 0 by default.

### 5.3.5. Methods

```
Definition: fit(X, Y=None)      Purpose: Fit the model with X
```

#### Parameters

##### **X: array-like of shape (n\_samples, n\_features)**

Training data, where `n_samples` is the number of samples and `n_features` is the number of features.

##### **Y: vector of labels (n\_samples),**

Labels

- a. not used if unsupervised algorithms
- b. If semi-supervised algorithms: Integer,
  - 1 anomaly (mandatory)
  - 0 unknown (mandatory)
  - -1 normal (if available)

#### Returns

### Results, opt\_model

**Results** contains the values of the hyper-parameters after optimization, the number of samples, and the names of the hyper-parameters optimized.

**Opt\_model** is the model after fit performed with the optimized hyper-parameters.

## 5.4. Configuring ThetaRay Classifier

After we have configured the FeaturesExtractor, we can continue to configure the ThetaRayClassifier.

```
from thetaray.api.theta_classifier import Classifier
tr_classifier = Classifier(*parameters)
```

### 5.4.1. Class Definition

```
Class Classifier(*,
    normalization_type=1,
    HP_Auto_Tuning_Semi=0,
    N_bayesian_itr=30,
    perf_metric='AUC',
    beta_for_F=2.0,
    set_zero_rating=0
)
```

### 5.4.2. Class Parameters

```
normalization_type: int, default=1
```

Type of normalization (scaling) used and applied to the input data set. The parameters used are first learned from the training data set and then applied to the detection data set. Possible values are 0 - no normalization, 1 - min-max, 3 - sigmoid, 4 - zscore.

```
normalization_type: int, default=1
```

If set to 1, performs Bayesian optimization of the semi-supervised classifier hyper-parameters. At each iteration, a cross-validation is performed for each set of hyper parameters chosen, according to the maximum of the expected improvement criterium.

```
normalization_type: int, default=1
```

Number of iterations for the Bayesian optimizer.

```
perf_metric: string, default='AUC'
```

perf\_metric is the metric that is being maximized during the optimization of the semi-supervised algorithms hyperparameters. By default, the metric is set to 'AUC', as the

“areaunder the curve”. It can be set to one of the following strings: ‘Fbeta’, ‘F1’, ‘AUC’, ‘Accuracy’, ‘Precision’ or ‘Recall’. It should be mentioned that for all the metrics above, except ‘AUC’, the fusion threshold is also modified during the optimization process.

```
beta_for_F: positive float, default=2.0
```

Parameter used in conjunction with `perf_metric = 'Fbeta'`. It represents the ratio of recall importance to precision importance. `beta_for_F > 1` gives more weight to recall, while `beta_for_F < 1` favors precision. With `beta_for_F = 1` we recover F1.

```
set_zero_rating: int, default=0
```

If `set_zero_rating` is set to 1, an algorithm determines if the value of an anomaly for a specific feature is equal to that of the normal entries. If this is the case it sets the ratings of those features to zero. `set_zero_rating` is 0 by default.

### 5.4.3. Methods

```
Definition: fit(X, Y=None) Purpose: Fit the model with X
```

#### Parameters

##### **X: array-like of shape (n\_samples, n\_features)**

Training data, where `n_samples` is the number of samples and `n_features` is the number of features.

##### **Y: vector of labels (n\_samples,**

Labels

Integer, should be 0,1,2,3,...,N

- 0 unknown (mandatory)
  - 1,2,3,...,N (mandatory) the class the entry belongs to
- N: number of classes
- Each entry MUST have its associated label.

#### Returns

##### **self: object**

Returns the instance itself.

```
Definition: predict(X) Purpose: Perform prediction on X
```

## Parameters

### **X: array-like of shape (n\_samples, n\_features)**

Detection data, where n\_samples is the number of samples and n\_features is the number of features.

## Returns

### **array of shape (n\_samples, N)**

Scores or probabilities, ranges between 0.0 and 1.0. Similar to predict\_proba in

scikit-learn. N is the number of classes. Each row sums up to 1.0.

Definition: evaluation\_metric(Sc, Y) Purpose: Evaluate the results with respect to scores Sc and labels Y (if provided)

## Parameters

Sc: array-like of shape (n\_samples, N)

Scores- ranges between 0.0 and 1.0. Obtained from the predict() method.

Y: vector of labels (n\_samples, )

Class labels – 1,2,3...

## Returns

### **Dictionary valuation[Fu]={‘AUC’,‘F1’,‘Fbeta’,‘Accuracy’,‘Precision’,‘Recall’}**

Evaluation metrics

Definition: feature\_rating\_multi\_labs(Sc, Cl=None) Purpose: Calculate feature ratings

## Parameters

Sc: array-like of shape (n\_samples, N)

Scores or probabilities - ranges between 0.0 and 1.0. Obtained from the predict() method

Cl: array-like of shape (n\_samples, )

Class label (if available) – 1,2,3...provided by the user – not mandatory. If not provided, the class label will be automatically determined based on the score Sc provided.

## Returns

### **array of shape (n\_samples, 2\*n\_features + 3)**

**Feature ratings table, where:**

- n\_samples is the number of samples in the dataset, n\_features is the number of features (columns) in the dataset
- The first column of the rating array includes the pks of the anomalies in the order they appear in the initial detect data set
- The second column is the class the entry belongs to
- The third column is the maximum of the scores in each row

The rest of the columns are:

- | Feature index | Rating | Feature index | Rating | ...
- The ratings appear in decreasing order from left to right
- The ratings are multiplied by 100 and rounded to integers
- The sum of the ratings equal 10000

**Example:**

PK	Class	Max score	Feature index	Rating	Feature index	Rating	Feature index	Rating
3526	3	0.874	24	1244	4	1141	21	723
7658	1	0.689	8	5301	4	824	31	667
...	...	...	...	...	...	...	...	...

## 5.5. Fitting the Feature Extractor Model

We can finally get to actual model training after configuring the Feature Extractor and the ThetaRay Detector.

```
import mlflow
from thetaray.api.models import save_model
from thetaray.api.drift import save_reference_dataset
from thetaray.api.evaluation import evaluate_reference_dataset

with mlflow.start_run(nested=True):
    features_extraction_model = features_extractor.fit(X_train)
    save_model('monthly_account_fe', features_extraction_model, MODEL_TAGS)
    features = features_extraction_model.transform(X_train)
```

First, we start an *MLflow* run context to enable saving the model once it's created.

```
1 | with mlflow.start_run(nested=True):
```

We use the `features_extractor` we created to configure the build of the feature extraction model, and we do it by executing the **features\_extractor's** `fit` method on the training dataset records.

```
features_extraction_model = features_extractor.fit(X_train)
```

We then save the model in *MLflow* since we're running under an *MLflow* context.

```
save_model('model_fe', features_extraction_model, MODEL_TAGS)
```

After creating the feature extraction model, we can apply it to the training dataset records by passing it to the **transform** method.

```
features = features_extraction_model.transform(X_train)
```

The returned value of the method is a Pandas dataframe containing the transformed features according to the settings of the feature extractor (only numerical values without nulls and missing values).

## 5.6. Training the Detection Model

We now have a dataframe containing the data records in the desired form, and we can get to training the detection model.

```
detection_model = tr_detector.fit(X=features)  
save_model('monthly_account_ad', detection_model, MODEL_TAGS)
```

The **tr\_detector's** `fit` method gets the dataframe as the value of the `X` parameter and returns the detection model.

```
detection_model = tr_detector.fit(X=features)
```

Suppose the detection model is configured to work with a semi-supervised, augmented, or hybrid learning method. In that case, we pass it also the labeled data records as the value of the **y** parameter:

```
detection_model = tr_detector.fit(X=features, y=labels)
```

Labels are a Pandas Series consisting of "1" for rows considered an anomaly and "0" for rows whose classification is unknown.

Finally, we save the model in *MLflow* with the specified tags.

```
save_model('monthly_account_ad', detection_model, MODEL_TAGS)
```

You can also save histograms for all training dataset features.

```
from thetaray.api.histograms import save_histograms

save_histograms(context, X_train, numerical_features)
```

The **save\_histogram** method saves the number of occurrences of each value for every feature in the dataset. The Investigation Center then uses the information to present a histogram that shows the value of each feature in the context of the data distribution during training .

### 5.6.1. Save reference dataset

Finally, we let the new model calculate anomaly scores for the dataset on which it was trained and then save the evaluated data as a reference dataset. The reason for saving a reference dataset is to enable a data drift test in the future. The check will confirm that the model is relevant for new data and eliminate the possibility that it's obsolete.

We perform the data drift test by comparing the reference dataset to future evaluations of the same model on new data.

The **evaluate\_reference\_dataset** function uses the detection and features extraction models to calculate anomaly scores for the training dataset.

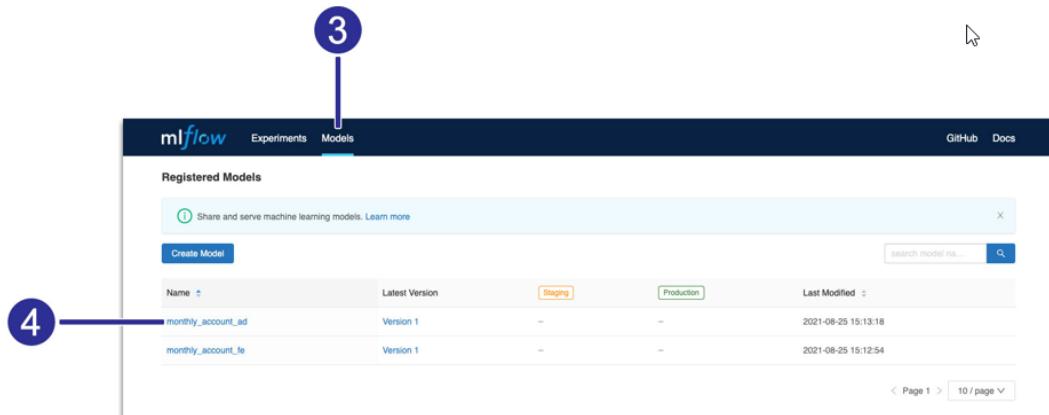
```
spark = context.get_spark_session()
ref_df = spark.createDataFrame(X_train)
evaluated_sample = evaluate_reference_dataset(
    context, EVALUATION_FLOW, EVALUATION_STEP,
    ref_df, features_extraction_model, detection_model
)
```

The **save\_reference\_dataset** function saves the evaluated data in *MLflow* as it runs within the *MLflow* context discussed previously.

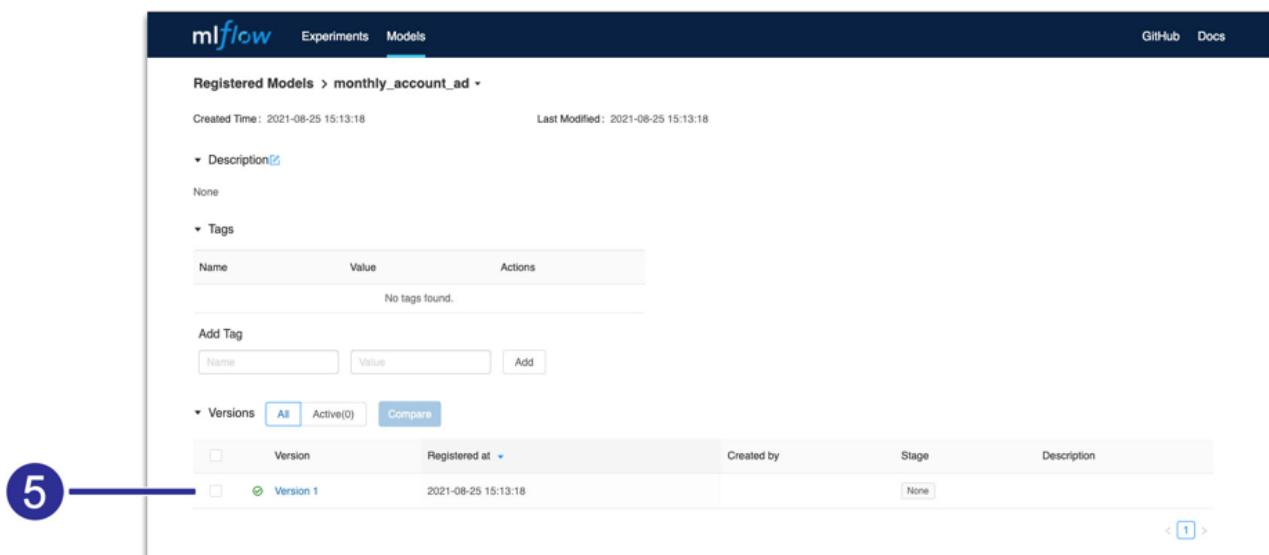
### 5.6.2. Review model in *MLflow*

The feature extraction model, the anomaly detection model, the histograms, and the reference dataset are all saved in *MLflow*. To review those artifacts, perform the following:

1. In the *JupyterLab* menu bar, select File -> New Launcher
2. Click the *mlflow* button in the launcher tab.



3. In *MLflow*, click *Models* on the top bar.
4. On the *Registered Models* page, click *monthly\_account\_ad* to view the detection model.



5. In the *monthly\_account\_ad* page, click *Version1*.

6

Registered Models > monthly\_account\_ad > Version 1

Registered At: 2021-08-25 15:13:18 Creator: Stage: None

Last Modified: 2021-08-25 15:13:18 Source Run: Run 17931813faab41a3af2959cc46eb1f58

▼ Description [🔗](#)  
None

▼ Tags

Name	Value	Actions
_commit_id	dirty_2021-08-25T12:13:16_b4b622a00082 93b35f399b739563878c59670 8fd7	<a href="#">🔗</a> <a href="#">📄</a>
version	release	<a href="#">🔗</a> <a href="#">📄</a>

Add Tag   [Add](#)

▼ Schema

Name	Type
No Schema.	

6. On the *version1* page, click the *Source Run* link.

7

Default > Run 17931813faab41a3af2959cc46eb1f58

Date: 2021-08-25 15:12:53 Source: [ipykernel\\_launcher.py](#) User: manager

Duration: 58.8s Status: FINISHED

▼ Notes [🔗](#)  
None

▼ Parameters

▼ Metrics

▼ Tags

▼ Artifacts

- ▼ models
  - ▼ monthly\_account\_ad
    - MLmodel
    - conda.yaml
    - model.pkl
  - monthly\_account\_fe
  - reference
    - reference\_dft.csv
    - anomalies distribution - PCA.png
    - anomalies distribution - TSNE.png
    - anomalies.html

Full Path: s3://thetaray-nir/mlflow/0/17931813faab41a3af2959cc46eb1f58/artifacts/models/monthly\_account\_ad/MLmodel  
Size: 369B

```

artifact_path: models/monthly_account_ad
flavors:
python_function:
  env: conda.yaml
  loader_module: mlflow.sklearn
  model_path: model.pkl
  python_version: 3.8.5
sklearn:
  pickled_model: model.pkl
  serialization_format: cloudpickle
  sklearn_version: 0.24.1
run_id: 17931813faab41a3af2959cc46eb1f58
utc_time_created: '2021-08-25 12:13:18.321144'

```

7. On the *Run* page, review the model parameters and artifacts.

## 6. The Evaluation process

In the Evaluation phase, the ThetaRay Platform analyzes data records for anomalies based on anomaly detection models. The **evaluate** function accepts the input dataframe and an **EvaluationFlow** entity. The **evaluate** function receives an **EvaluationFlow** object as an argument to learn what model to use for anomaly detection.

```
from thetaray.api.evaluation import evaluate

results_df = evaluate(context, EVALUATION_FLOW, input_df)
```

### 6.1. What is the EvaluationFlow entity?

The EvaluationFlow entity is a metadata object that configures the evaluation process.

Each **EvaluationFlow** instance specifies the input dataset for the process and a sequence of algorithmic enrichment steps used for augmenting the original record with scores produced by previously trained ML models.

In this release, the only step supported is the **AlgoEvaluationStep** step which enables evaluation through ThetaRay's anomaly detection algorithms. The attributes of **AlgoEvaluationStep** refer to the specific instances of the Feature Extraction model and the Detection model involved in the evaluation flow.

#### 6.1.1. EvaluationFlow metadata object example

```
from thetaray.api.solution import EvaluationFlow, AlgoEvaluationStep
from thetaray.api.solution.evaluation import ModelReference
def evaluation_flow():
    return EvaluationFlow(
        identifier = "account_monthly",
        displayName = "account_monthly",
        input_dataset = "account_monthly_analysis",
        data_permission= "dpv:public",
        evaluation_steps= [
            AlgoEvaluationStep(
                Identifier = 'algo',
                name = 'Algo',
                feature_extraction_model =
                    ModelReference('monthly_account_fe', tags={"version": "release"}),
                detection_model =
                    ModelReference('monthly_account_ad', tags={"version": "release"}),
```

```
        pattern_length = 3
    )
],
max_workers=8,
trace_queries=[
    TraceQuery(
        identifier='round_amounts_tracequery',
        features=['round_amounts'],
        dataset='tracequery_round_amounts',
        sql=''''
            SELECT * FROM {dataset_table}
            WHERE ai_agent_id_f = {activity.agent_id}
            AND bi_created_date_time >= {activity.weekly_level_date} - interval '1
week'
            - ({occurred_before} || 'DAY')::INTERVAL
            AND bi_created_date_time <= {activity.weekly_level_date} + ({occurred_
after} ||
            'DAY')::INTERVAL
            AND bj_round_amount_indicator != '0'
        '''
    )
]
```

### 6.1.2. EvaluationFlow attributes

The AlgoEvaluationStep is currently the only evaluation step type supported by the ThetaRay Platform and, therefore, the only one that can be listed in the evaluation\_steps attribute. The step sets the feature extract and the anomaly detection models used in the evaluation process.

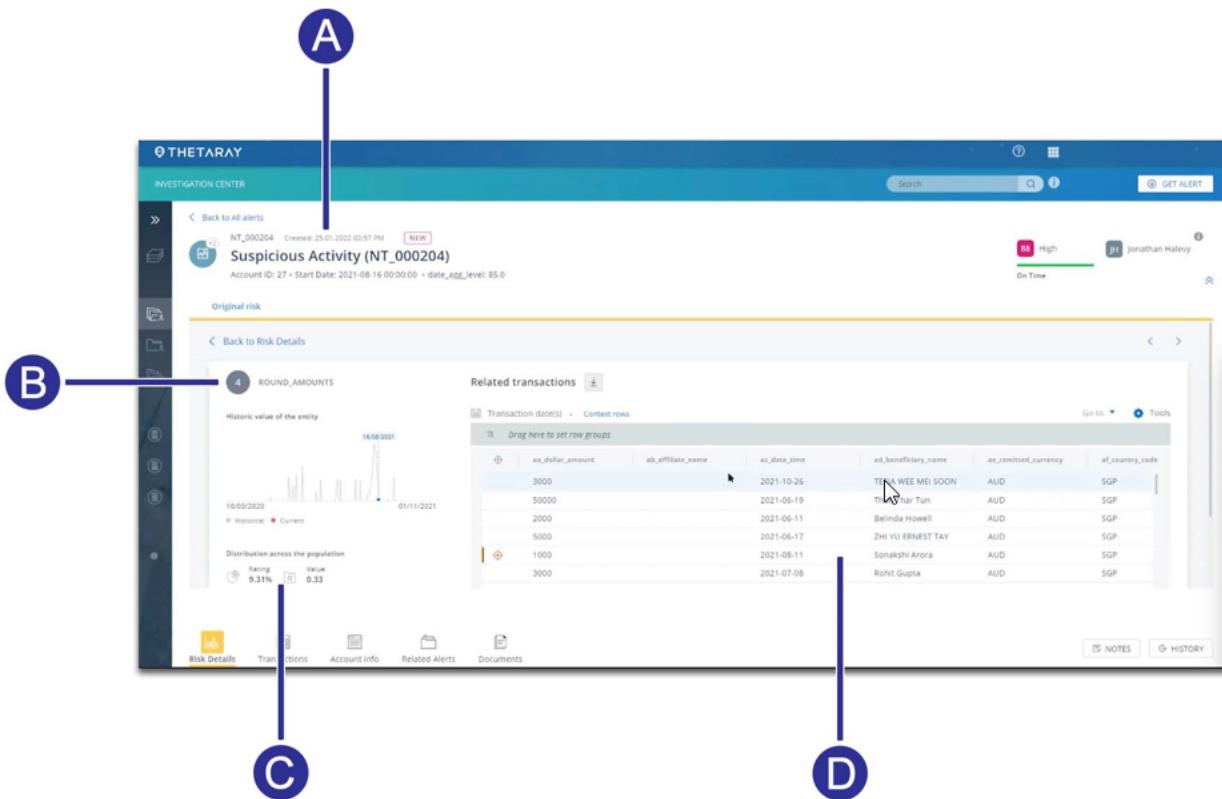
- **Identifier:** The unique step identifier.
- **name:** The title of the step.
- **feature\_extraction\_model:** The ModelReference metadata object includes the feature extraction model's instance name and tags as they appear in MLflow.
- **detection\_model:** The ModelReference metadata object includes the detection model instance name and tags as they appear in MLflow.
- **pattern\_length:** The number of highest-ranking features to chain in a grouping pattern. Alerts from the same account and grouping patterns are consolidated into one alert in the Investigation Center.

For example, the **sum\_amount#zscore12#zscore6** pattern length is 3. The pattern consists of the three features with the highest feature-ranking in descending order. All alerts from the same account and having the **sum\_amount#zscore12#zscore6** pattern will be displayed together on the Investigation Center.

### 6.1.3. TraceQuery Attributes

The trace\_query object encapsulates the metadata required to carry out trace queries. The objective of trace queries is to enable the Investigation Center to display the transactions contributing to the computation of an alert raising feature.

For example, in the following screen capture, you can see a section of an IC Alert page showing a trace query- obtained transactions contributing to the computation of the **ROUND\_AMOUNTS** feature.



The Alert page above includes the following sections:

- The alert name
- The feature name
- The feature contribution to the alert and the feature value
- The transactions on the basis of which the feature was computed

The trace query that returned the **ROUND\_AMOUNT** transactions in the example above is specified here:

```
TraceQuery(
    identifier='round_amounts_tracequery',
    features=['round_amounts'],
```

```
dataset='tracequery_round_amounts',
sql='''SELECT * FROM {dataset_table}
      WHERE ai_agent_id_f = {activity.agent_id}
            AND bi_created_date_time >= {activity.weekly_level_date} - interval '1 week'
      - ({occurred_before} || 'DAY')::INTERVAL
            AND bi_created_date_time <= {activity.weekly_level_date} + ({occurred_after} || 'DAY')::INTERVAL
            AND bj_round_amount_indicator != '0' '''
)
```

The trace queries include the following attributes:

- **identifier**: The trace query identifier
- **features**: The feature whose transactions are queried
- **dataset**: The dataset containing the trace query results.
- **sql**: The SQL trace query statement
- **parquet\_index**: If Parquet Indexes are enabled for the dataset and should be used as the source for the trace query, the name of the index to use (please make sure that the identifier column of the Parquet Index is aligned with the criteria used in the Trace Query sql statement)

## 6.2. Run the Evaluation process

By passing an **EvaluationFlow** instance as an argument to the `evaluate` function, we tell the evaluation process what models to use for anomaly detection. The **evaluate** function returns a dataframe containing the output of the evaluation process. Each data record is now enriched with an anomaly score and feature ratings for each data record.

The **write\_evaluated\_activities** function takes the evaluated results dataframe and writes it into a MinIO table carrying the EvaluationFlow instance's name.

```
from thetaray.api.evaluation import write_evaluated_activities

results_df = evaluate(context, EVALUATION_FLOW, detect_df)
write_evaluated_activities(context, results_df, EVALUATION_FLOW)
```

## 6.3. Publish Evaluated Activities to CDD

In addition to *MinIO*, the evaluated activities table should be written to the CDD to enable SQL querying from the *Investigation Center* and enable execution of the Decisioning process. The **publish\_evaluated\_activities** function accepts the **EvaluationFlow** instance as a parameter and

uses it to find the name of the **evaluated\_activities** table in *MinIO*, which is the source for the published data. Then, the function copy the table from MinIO to the CDD.

```
from thetaray.api.evaluation import publish_evaluated_activities
publish_evaluated_activities(context, EVALUATION_FLOW, parallelism = <4>)
```

**parallelism:** the parallelism parameter value is optional. Any value up to `total_work_slots` can be used, but the value `-1` is the default value, and if used, the parallelism mechanism will be disabled.

## 6.4. Publishing Analysis Results to Parquet Indexes

Analysis results stored in Postgres can be offloaded into Parquet Indexes to reduce relational database size. To enable the offloading process a Parquet Index needs to be registered on the Evaluation Flow as detailed earlier in the document and the 'publish\_to\_parquet' API should be invoked for the given Evaluation Flow. Here is an example:

```
from thetaray.api.evaluation import publish_to_parquet
from thetaray.common.data_environment import DataEnvironment
...
publish_to_parquet(context, <evaluation flow id>, <parquet index identifier>, data_
environment = DataEnvironment.PUBLIC)
```

**Note:** Publishing 'Evaluated Activities' as Parquet files does not replace the need to call `publish_evaluated_activities` to Postgres as described in section 6.3. The operation allows shortening the retention period of data stored in Postgres to reduce the database size, but doesn't completely eliminate the need to perform the publishing process as downstream processes (decisioning) require evaluated activities to be available in Postgres in order to run.

## 6.5. Load Evaluated Activities

The **load\_evaluated\_activities** function returns a data frame containing the evaluated data records that match the context execution date. This method is primarily used for experimentation through a notebook environment and is not required in a typical operational flow.

```
from thetaray.api.evaluation import load_evaluated_activities
df = load_evaluated_activities(context, EVALUATION_FLOW, parallelism = <4>)
```

**Note:** Regarding the parallelism parameter value detailed in the above sample code, be aware that using the parallelism parameter is optional and the value shown is an example placeholder only. Any value up to `total_work_slots` can be used, but the value `-1` is the default value, and if this is used, the parallelism mechanism will be disabled.

## 6.6. Enriching Evaluation Flow Data - Rule Builder Configuration Fields

The ThetaRay Rule Builder is a self-service tool enabling configuration of alerting rules that are applied over data which either originates from the 'Input Dataset' of the Evaluation Flow or calculated as part of the Evaluation Process based on user configured aggregation schemes.

Integration of the enrichment step that computes user configured aggregations requires calling the '`enrich_with_custom_fields`' API prior to invoking the '`evaluate`' method. The API augments the Spark Dataframe that corresponds to the input data with additional columns based on user configuration. The resulting Spark Dataframe should be passed as input to the '`evaluate`' method, which performs further data enrichment through ThetaRay anomaly detection and classification algorithms.

### API signature

```
def enrich_with_custom_fields(  
    context: JobExecutionContext,  
    evaluation_identifier: str,  
    dataframe: DataFrame,  
    data_environment: DataEnvironment = DataEnvironment.get_default(),  
)
```

### DataFrame:

- `context` - job execution context
- `evaluation_identifier` - Evaluation Flow Identifier
- `dataframe` - Spark Dataframe originating from the Dataset read operation over the input
- `data_environment` - public / private data environment

**Example usage:**

```
# 1. Evaluation dataset read
detect_df = dataset_functions.read(context, "wrangling")
# 2. Enrich with custom fields call
detect_df = enrich_with_custom_fields(
    context=context,
    evaluation_identifier="tr_analysis_uk",
    dataframe=detect_df,
)
# 3. Evaluation
enriched_df = evaluate(
    context=context,
    evaluation_identifier="tr_analysis_uk",
    dataframe=detect_df,
)
```

## 7. Decisioning

The Decisioning process determines whether the Evaluated Activities generated through the Evaluation Process should trigger alerts. The process categorizes alerts into risks, consisting of conditions used to determine when the alert will be triggered.

Risks can access the data available within the Evaluated Activity, including features, forensic information, and output of the evaluation process that include scores and feature rating/ranking. Additionally, the Decisioning process supports enriching risk conditions with data from arbitrary published datasets through a dynamic querying mechanism.

### 7.1. Identifying Risks

The **identify\_risks()** function activates the risk identification process for a specified evaluation flow by looking at each record's score and feature rating in the **evaluated\_activities** table. The process essentially consists of matching the record values to Risks.

```
from thetaray.api.decisioning.api import identify_risks

identify_risks(context,
                evaluation_flow_identifier: str,
                from_job_ts: datetime = None,
                to_job_ts: datetime = None,
                data_environment: DataEnvironment =
                    DataEnvironment.get_default(),
                parallelism: int =
                    Constants.PARALLELISM_DEFAULT_VALUE,
                ignore_block: bool = False,
                chunk_size=1000,
                num_of_cores=os.cpu_count())
```

The function accepts the following parameters:

- **context:** the context object includes the **execution\_date** parameter used for selecting which data records to read from the **evaluated\_activities** table.
- **evaluation\_flow\_identifier:** the identifier of the evaluation flow is used to find the **correct evaluated\_activities** table in the database.
- **from\_job\_ts:** Specify the timestamp of the earliest data record selected for identification. This parameter is useful for identifying or re-identifying risks in a subset of historical data records. You use this parameter to set the time range lower bound.
- **to\_job\_ts:** Specify the timestamp of the latest data record selected for identification. This parameter is useful for identifying or re-identifying risks in a subset of historical data records. Use this parameter to set the time range's higher bound.

- **data\_environment:** Sets whether the data environment is PUBLIC or PRIVATE(default); see Read/Write to public/private database area. When the data environment is private, the function will not execute
- **parallelism:** the parallelism parameter value is optional. Any value up to `total_work_slots` can be used, but the value `-1` is the default value, and if used, the parallelism mechanism will be disabled.
- **chunk\_size:** The number of rows to process in bulk, default 1000.
- **num\_of\_cores:** Maximum cores to use in parallel, default max available cores.

## 7.2. Creating a risk

The **identify\_risks** function tries to match each record in the **evaluated\_activities** table to a risk.

But what is a risk, and how do we define it?

A risk specifies a condition that a data record must satisfy before it is considered an alert and consequently distributed to an Investigation Center. For example, a risk condition can state that a data record will be sent to the *Investigation Center* if its **algo\_score** is greater than 0.55.



We formulate risk criteria in YAML files that must be stored in `root/domains/<domain_name>/risks`.

```
identifier: anomaly
display_name: anomaly
description: anomaly
category: algo
severity: 87
suppression_period_unit: DAY
suppression_period_size: -1
evaluation_flows:
- account_monthly_france
- account_monthly_uk
- account_monthly_spain
parameters:
- name: fusion_threshold
  datatype: DOUBLE
  value: "0.50"
variables:
- name: is_anomaly
  display_name: Is Anomaly
  description: Is Anomaly
  expression: "activity.algo_score > {parameters.fusion_threshold}"
  datatype: BOOLEAN
conditions:
```

```
- variable: is_anomaly
  value: "true"
  display_settings:
    - type: "Rating"
      evaluation_step: "algo"
  title: "Is anomaly?"
  description: "Algo score is greater than the fusion threshold"
```

## 7.3. Setting the risk metadata

Soon, we will look at some examples of different risk conditions, but we begin by explaining the metadata that identifies and classifies the risk:

```
identifier: anomaly
display_name: anomaly
description: anomaly
category: algo
severity: 87
suppression_period_unit: DAY
suppression_period_size: -1
evaluation_flows:
- account_monthly_france
- account_monthly_uk
- account_monthly_spain
```

- **identifier:** The unique risk identifier
- **displayName:** The title of the risk
- **description:** A free text description of the risk.
- **category:** The risk category
- **severity:** The risk severity (HIGH, MEDIUM, LOW, UNCLASSIFIED)
- **suppression\_period\_size:** The suppression period duration in days or months. Use -1 to disable suppression.
- **suppression\_period\_unit:** The unit by which suppression is measured (HOUR, DAY, MONTH).

A suppression period is an interval between alerts originating from the risk and belonging to the same investigated entity (account, customer). The suppression period begins when issuing the first alert and ends after the specified hours, days or months. This risk will create no alerts in the Investigation Center for the same account during the suppression period. By default, the suppression functionality relies on the unique risk identifier. In order to refer to a different identifier for suppressing the risk, please see section below on

## Dynamic Risk Templates

- **evaluation\_flows**: Names of the evaluation flows to which the risk can be applied. Multiple evaluation flows can be applied to a single risk if their evaluated activities tables include the fields referred to by the risk condition
- **active**: A boolean indicator for whether the risk will be active for the evaluation flow. Can also be set on the variable level, in order to deactivate only specific variables within the risk

### 7.3.1. Dynamic Risk Templates

#### 7.3.1.1. Risk – Dynamic Attributes

Dynamic Risk Templates refer to the addition of five optional attributes in the risk metadata that can generate dynamic data to be displayed in the alert. The attributes are:

- **dynamic\_display\_name**
- **dynamic\_description**
- **dynamic\_category**
- **dynamic\_severity**
- **dynamic\_suppression\_identifier**

If the dynamic attributes are used, they will be displayed on the alert instead of their corresponding counterparts in the risk metadata. For example, if dynamic\_display\_name is filled, then that will be the name on the alert instead of display\_name.

However, the original attributes (display\_name, description, category and severity) should still be filled for fallback and internal purposes.

An additional attribute added is read-only, which is a Boolean indicator that will remove the risk from the Rule Parameter Editor. The attribute will be set by default to false.

The dynamic\_suppression\_identifier can be filled if a different identifier or expression would need to be used instead of the default risk\_identifier. For example, the suppression can refer to the dynamic\_display\_name and then the alert connected with the name will be suppressed for the configured suppression time period.

```
identifier: dynamic_risk
display_name: dynamic_risk display_name
description: dynamic_risk description
category: dynamic_risk
severity: 90
read_only: True
dynamic_display_name: "{{ tr_evaluation[0].display_name }} -> {{ activity.tr_evaluation_
```

```
pattern }}"
dynamic_description: "{{ tr_evaluation[0].description }},{{ parameters.fusion_threshold1
}}"
dynamic_category: "{{ tr_evaluation[2].category }}"
dynamic_severity: "{{ activity.tr_evaluation_score * 100 }}"
dynamic_suppression_identifier: "{{ activity.tr_evaluation_pattern }}"
suppression_period_unit: DAY
suppression_period_size: -1
evaluation_flows:
  - tr_analysis
  - party_tr_analysis
```

### 7.3.1.2. Accessing Dynamic Data

Dynamic fields values should be populated from the listed options below by using the jinja2 template standard:

- Risk parameter - {{ parameters.PARAMETER\_NAME }}

```
{{ parameters.fusion_threshold1 }}
```

- Evaluation flow activity columns - {{ activity.COLUMN\_NAME }}

```
{{ activity.tr_evaluation_score }}
```

- Feature metadata for a given rank i.e. description/category - {{ ALGO\_STEP\_NAME [RANK].ATTRIBUTE\_NAME }}

Please note, RANK starts from zero

```
{{ tr_evaluation[0].display_name }}
```

- Risk variables and enrichments

```
{{ variables.is_anomaly1 }}
```

```
{{ enrichments.min_amount }}
```

### 7.3.1.3. Generating Dynamic Values

- **dynamic\_display\_name/dynamic\_description/dynamic\_category** - any generation result is valid including None or Empty values .

- **dynamic\_severity** - internally cast to integer. The calculation result must be between 0-99. If it is out of range, the risk severity will be a fixed value taken from the **severity** attribute.
- If the dynamic attribute refers to a nonexistent field, for example, an undefined parameter, then **identify\_risks** will fail with an appropriate message.

## 7.4. Formulating a basic condition

After discussing the attributes that identify and classify the risk, we can now discuss the attributes involved in setting the risk conditions.

Conditions are constructed on top of **variables**, **parameters**, and **enrichments** which have a distinctive meaning in the context of Risk creation. We will learn what they are and how they construct conditions by looking at three examples, each demonstrating a different aspect of Risk creation.

Let's start with a simple Risk condition that defines every data record whose **algo\_score** is greater than **0.55**.

```
parameters:  
- name: fusion_threshold  
  datatype: DOUBLE  
  value: "0.50"  
variables:  
- name: is_anomaly  
  display_name: Is Anomaly  
  description: Is Anomaly  
  expression: "activity.algo_score > {parameters.fusion_threshold}"  
  datatype: BOOLEAN  
conditions:  
- variable: is_anomaly  
  value: "true"  
  display_settings:  
  - type: "Rating"  
    evaluation_step: "algo"  
  title: "Is anomaly?"  
  description: "Algo score is greater than the fusion threshold"
```

### 7.4.1. Parameters

Parameters refer to constant values. In our example, the **fusion\_threshold** parameter refers to the **DOUBLE** value of **0.55**. Parameters can be DOUBLE, LONG, INT, BOOLEAN, TIMESTAMP, or STRING.

```
parameters:  
- name: fusion_threshold  
  datatype: DOUBLE  
  value: "0.50"
```

#### 7.4.2. Variables

Variables are expressions composed of parameters and values extracted from the current data record. In our example, the expression **activity.algo\_score > {parameters.fusion\_threshold}** determines whether the current data record **algo\_score** value is greater than the **fusion\_threshold** parameter.

```
variables:  
- name: is_anomaly  
  display_name: Is Anomaly  
  description: Is Anomaly  
  expression: "activity.algo_score > {parameters.fusion_threshold}"  
  datatype: BOOLEAN
```

The **activity** prefix in **activity.algo\_score** refers to the **evaluated\_activities** table of the current evaluation flow. The **algo\_score** suffix is created dynamically when the **evaluated\_activities** table is generated. The **algo** part in **algo\_score** refers to the current evaluation-flow STEP instance name, and the score part in **algo\_score** refers to the name of the feature or column.

In our example, the variable expression **datatype** is **BOOLEAN**, but it can also be **NUMERIC** or **STRING**.

#### 7.4.3. Conditions

A **condition** specifies the primitive value that a variable must be equal to for the data record to be considered an alert generating risk. In our example, there is only one **BOOLEAN** variable, **is\_anomaly**, and it has to be equal to **true** for the data record to be considered a risk.

```
conditions:  
- variable: is_anomaly  
  value: "true"  
  display_settings:  
    - type: "Rating"  
    evaluation_step: "algo"
```

 Multiple conditions cannot share the same parameters and consequently fail validation by the metadata sync process. To avoid errors create different parameters for different conditions.

#### 7.4.4. Rating- type display settings

The **display\_settings** list tells the Investigation Center what heuristical information concerning the alert to display. The **type** attribute refers to the kind of heuristical information needed, which in this example (**Rating**) is a set of histograms for all the rated features. The **evaluation\_step** attribute indicates that the features in question were rated by the **algo** step. We discuss another type of display setting in Threshold-type display settings.

```
display_settings:
  - type: "Rating"
    evaluation_step: "algo"
```

#### 7.4.5. Condition name and description

The condition **title** and **description** fields describe the condition on the *Rule Parameters Editor UI*.

### 7.5. Formulating a child risk

A child risk inherits the conditions of its parent risk and adds conditions of its own. Since a child risk is inherently more specific than its parent, when the child risk conditions are satisfied, the child risk is fired rather than the parent risk.

In the example below, we define a child to the first example risk according to which a transaction's **algo\_score** must be greater than **0.55**. In addition, we add another condition stating that the **algo\_sum\_amount** feature is rated as the top contributor to the **algo\_score**.

```
identifier: anomaly_high_amount
display_name: anomaly_high_amount
description: anomaly_high_amount
category: algo
severity: 87
suppression_period_unit: DAY
suppression_period_size: -1
parent_identifier: anomaly
evaluation_flows:
  - account_monthly_france
  - account_monthly_uk
  - account_monthly_spain
variables:
  - name: top_sum_amount_rank
```

```
display_name: top_sum_amount_rank
description: top_sum_amount_rank
expression: "activity.algo_sum_amount_rank=1"
datatype: BOOLEAN
conditions:
- variable: top_sum_amount_rank
  value: "true"
  display_settings:
    - type: "Rating"
      evaluation_step: "algo"
```

### 7.5.1. Set the child risk parent

We define risk as a child risk by setting the **parent\_identifier** attribute to the name of the parent risk.

```
parent_identifier: anomaly
```

### 7.5.2. Variables

In our example, the expression **activity.algo\_sum\_amount\_rank=1** determines whether the current data record's **algo\_sum\_amount\_rank** value is equal to 1.

```
variables:
- name: top_sum_amount_rank
  display_name: top_sum_amount_rank
  description: top_sum_amount_rank
  expression: "activity.algo_sum_amount_rank=1"
  datatype: BOOLEAN
```

### 7.5.3. Conditions

The child risk condition listed here is added to the condition specified in the parent risk. The condition states that the **top\_sum\_amount\_rank** variable must be equal to **true** for the data record to be considered a risk.

```
conditions:
- variable: top_sum_amount_rank
  value: "true"
```

```
display_settings:  
- type: "Rating"  
  evaluation_step: "algo"
```



Child risk conditions cannot share the same parameters as their parent conditions and consequently fail validation by the metadata sync process. Create different parameters for parent and child risk conditions.

## 7.6. Formulating a risk with Enrichment

Enrichments enable you to query the CRA to get more information on the current data record by aggregating or retrieving historical values.

In this example, we construct a condition involving **enrichment**. The condition states that a transaction is considered a risk whenever it belongs to an account whose highest transaction amount in the current and previous months is greater than 50000.

```
identifier: high_transaction  
display_name: High transaction  
description: "Transaction amount above threshold"  
category: scenarios  
severity: 20  
suppression_period_unit: DAY  
suppression_period_size: -1  
evaluation_flows:  
- account_monthly_uk  
parameters:  
- name: transaction_threshold  
  datatype: DOUBLE  
  value: "50000"  
enrichments:  
- name: max_transaction  
  sql: |  
    select max(tx.amount) as max_transaction from transaction_uk as tx where  
    tx.account_id = activity.account_id and tx.date between activity.month - INTERVAL  
'1 month' and  
    activity.month  
outputs:  
- name: max_transaction  
  datatype: DOUBLE  
variables:
```

```
- name: high_transaction
  display_name: High transaction
  description: High transaction
  expression: "{enrichments.max_transaction} > {parameters.transaction_threshold}"
  datatype: BOOLEAN
  conditions:
    - variable: high_transaction
      value: "true"
      display_settings:
        - type: "Threshold"
          feature: "{enrichments.max_transaction}"
          parameter: "{parameters.transaction_threshold}"
```

## 7.6.1. Parameters

In our example, the transaction\_threshold parameter refers to the **DOUBLE** value of **50000**. Parameters can be DOUBLE, LONG, INT, BOOLEAN, TIMESTAMP, or STRING.

```
parameters:
- name: transaction_threshold
  datatype: DOUBLE
  value: "50000"
```

## 7.6.2. Enrichment

Enrichments enables you to use SQL and query the CRA to get more information on the current data record by aggregating or retrieving historical values. In our example, the SQL query determines the highest transaction in the transaction sub-set occurring in the current and previous months.

```
enrichments:
- name: max_transaction
  sql: |
    select max(tx.amount) as max_transaction from transaction_uk as tx where
    tx.account_id = activity.account_id and tx.date between activity.month - INTERVAL
    '1 month'
    and activity.month
  outputs:
    - name: max_transaction
      datatype: DOUBLE
```

### 7.6.3. Variables

In our example, the expression `{enrichments.max_transaction} > {parameters.transaction_threshold}` determines whether the highest transaction in the transaction sub-set occurring in the current and previous months is greater than **50000**.

```
variables:  
- name: high_transaction  
  display_name: High transaction  
  description: High transaction  
  expression: "{enrichments.max_transaction} > {parameters.transaction_threshold}"  
  datatype: BOOLEAN
```

### 7.6.4. Conditions

In our example, there is only one **BOOLEAN** variable, **high\_transaction**, and it has to be equal to **true** for the data record to be considered a risk.

```
conditions:  
- variable: high_transaction  
  value: "true"
```

### 7.6.5. Threshold-type display settings

The **display\_settings** list tells the *Investigation Center* what heuristical information concerning the alert to display. The **type** attribute refers to the kind of heuristical information needed, which in this example is **Threshold**. The **Threshold** type refers to a linear graph that tracks the performance of a selected feature for the past 12 months while showing the **parameter** value as a horizontal line on the graph.

```
display_settings:  
- type: "Threshold"  
  feature: "{enrichments.max_transaction}"  
  parameter: "{parameters.transaction_threshold}"
```

## 7.7. Dynamic Risk Templates - Upgrade and Troubleshooting

### 7.7.1. Upgrading

**To upgrade existing implementations to work with dynamic templates instead of multiple child risks:**

1. Add the dynamic attributes to the generic risk.
2. Keep the pattern variable in generic risk. (It is not used anymore, but retain it for backward compatibility for use by IC.)
3. Remove or disable child risks.

### 7.7.2. Troubleshooting

1. If you have a configuration error, for example, a typo in the reference for one of the dynamic attributes, the `identify_risks` invocation will throw an error and will not be able to generate the dynamic values.
2. If you referenced an empty or nonexistent field from dataset metadata, the dynamic attribute will generate a `None` value, that will be visible within the alert. Take care to reference valid fields and parameters from the dynamic attributes.

## 8. Network Visualizations (NV)

This chapter covers two Network Visualization related topics:

- Manual Alerts in IC Network Visualizations & Updating Decisioning Model
- Upgrading Network Visualization from pre 6.8 Deployments

### 8.1. Manual Alerts in IC Network Visualizations & Updating Decisioning Model

#### 8.1.1. Overview

The process of capturing alert state information delivered from the Investigation Platform by the Platform is enabled by instructing the Investigation Center to send HTTP requests consisting of JSON representation of the alert to the underlying Analytics Platform. The platform should be configured to capture these requests through a REST Connector (a Flat File Connector with REST exposure enabled through the `rest_config` setting).

Information arriving from the Connector will include the 'id' of the alert, the alert state, resolution code and JSON representation of the alert triggers (which can include information about more than a single trigger in case of consolidation within the Investigation Center) and will be captured into a dataset.

A custom notebook presented later in this diagram processes the information within this dataset, extracts the alerts details (it should be noted that the notebook should be adapted to the actual investigated entity identifiers - in the example, `account_id` is used to identify the investigated entity) and publishes nodes to the graph encapsulating the manual alerts, as well as connecting them through edges to account nodes.

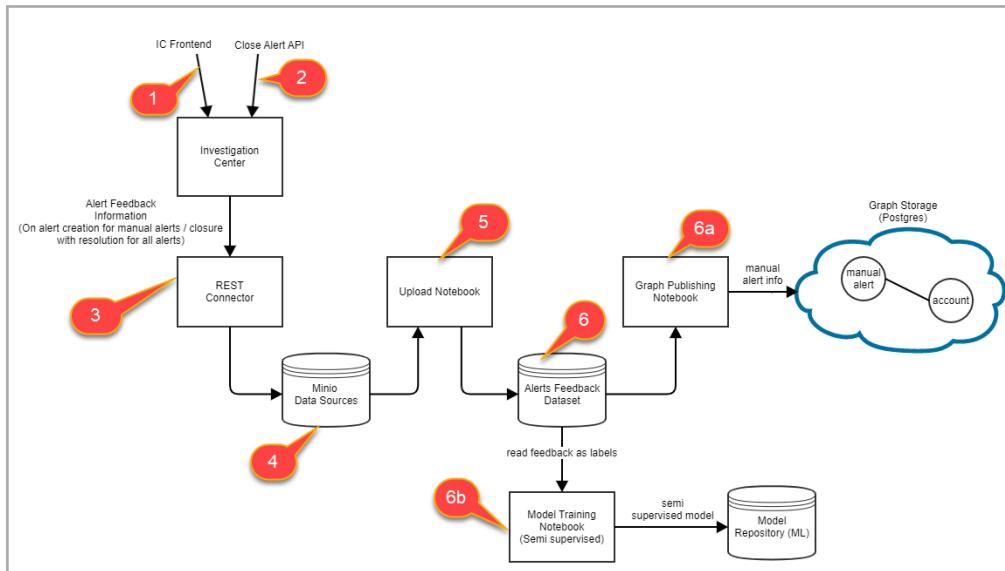
#### 8.1.2. Sample Implementation

The following sections provide a sample implementation illustrating how to capture feedback from the Investigation Center.

The implementation includes the following elements:

- A connector definition, with REST connectivity enabled
- A dataset matching the information that will arrive from the investigation center
- A notebook performing data upload of information arriving from the Investigation Center
- A notebook which processes the feedback information and updates the network visualization graph

The following diagram shows the combined feedback loop flow and how feedback information is used to drive both these processes.



**Figure 15: Data Path - Manual Alert /Alert Closure API to Add alert to NV Graph & update Model**

#### Process Flow:

1. A manual Alert is created in Investigation Center UI.
2. Alert closure data is pushed to Investigation Center via API.
3. A REST connector is used to push related data to MinIO storage.
4. Combined Feedback data in MinIO is used to populate an upload notebook.
5. The upload notebook is run to create a feedback dataset in the database.
6. Dataset data is examined and related data extracted and used to:
  - a. Populate the NV graph notebook that adds manually created alerts to the Network Visualizations graph.
  - b. Populate the model training notebook that updates the machine learning model repository.

#### 8.1.3. Adding Manual Alerts to Network Visualizations

Alerts created manually are added to the **Network Visualizations** network graph via a feedback loop that starts with the data collected from the Investigation Center.

#### 8.1.4. Feedback Alert Closure Data - Enhance Decisioning

Alert closing data from the ThetaRay use case manager or from an external case manager (via an API) is used to enhance the decisioning model.

### 8.1.5. Creating a REST Connector

Data from the manual alert along with the Manual alert data are then used to create a REST connector in the platform Jupyter Lab module.

#### Example code - to create a REST Connector

```
#connectors/tr_analysis_feedback.py
from thetaray.api.connector import FlatFileConnector
from thetaray.api.connector.flat_file_connector.flat_file_connector import Source, Target,
TimestampConfType, TimestampConf, RestConfig
def tr_analysis_feedback_connector():
    return FlatFileConnector(
        identifier="tr_analysis_feedback",
        display_name="tr_analysis_feedback",
        source=Source(
            folder="tr_analysis_feedback",
            has_header=True,
            delimiter=",",
            timestamp_conf=TimestampConf(type=TimestampConfType.EXECUTION_DATE, filename_
pattern="%Y%m%d"),
            dataset_schema_ref="tr_analysis_feedback",
            escape="\\""),
        targets=[Target(
            dataset_name="tr_analysis_feedback"
        ]),
        default_target_dataset_name="tr_analysis_feedback",
        rest_config=RestConfig(enable=True,
            max_buffered_records=1000,
            flush_interval_minutes=5,
            read_chunk_size=1000)
    )
def entities():
    return [tr_analysis_feedback_connector()]

#datasets/tr_analysis_feedback.py
from thetaray.api.solution import DataSet, Field, DataType, IngestionMode
def tr_analysis_feedback():
    return DataSet(
        identifier="tr_analysis_feedback",
        display_name="tr_analysis_feedback",
        field_list =
            Field(identifier='resolutioncode', display_name='Resolution code', data_
type=DataType.STRING),
            Field(identifier='stateid', display_name='State ID', data_
type=DataType.STRING),
            Field(identifier="id", display_name="ID", data_type=DataType.STRING),
            Field(identifier="triggers", display_name="Triggers", data_
```

```
type=DataType.STRING)],
    data_permission="dpv:public",
    ingestion_mode=IngestionMode.OVERWRITE,
    publish=False,
    num_of_partitions=4,
    primary_key=['id'],
    num_of_buckets=7,
)
def entities():
    return [tr_analysis_feedback()]
```

### 8.1.6. Storing the Connector Data in MinIO Storage

The next stage in the process is to move the data processed in the connector and store it temporarily in MinIO data sources storage.

### 8.1.7. Creating an Upload Notebook

The next stage in the process is to upload the data from MinIO source storage to an upload notebook in Platform Jupyter Lab module.

#### Example Code - to upload data to notebook

```
# upload_feedback.ipynb

from thetaray.api.context import init_context
from datetime import datetime

spark_conf={
    "spark.executor.memory": "1g",
    "spark.executor.cores": "1",
    "spark.executor.instances": "1"
}

context = init_context(execution_date=datetime.today(), spark_conf=spark_conf)
spark = context.get_spark_session()
from thetaray.api.connector.upload import upload
from thetaray.common.data_environment import DataEnvironment

upload(context, "tr_analysis_feedback", data_environment=DataEnvironment.PUBLIC)
context.close()
```

### 8.1.8. Alerts Feedback Dataset

The manual alert and decisioning feedback data in the Upload Notebook are then required to create a feedback dataset. The data in the dataset is as explained earlier used to Dataset data is used to populate a graph notebook to update the network visualizations graph and to update the model repository which is used to retrain the current model based on the data in the feedback loop.

#### Example - Create and Run an Update Graph Notebook

```
# update_graph.ipynb

from thetaray.api.context import init_context
from thetaray.api.dataset import dataset_functions
from thetaray.common.data_environment import DataEnvironment
from thetaray.api.graph import publish_nodes, publish_edges
from thetaray.api.evaluation import read_alerted_activities, load_evaluated_activities
from datetime import datetime
import random
from pyspark.sql import functions as f
from pyspark.sql.window import Window
from pyspark.sql.types import *
import pandas as pd
context = init_context(execution_date=datetime.today())
spark=context.get_spark_session()
from thetaray.api.dataset import dataset_functions
from pyspark.sql import functions as f
from thetaray.common.data_environment import DataEnvironment

manual_alerts_df = dataset_functions.read(context, 'tr_analysis_feedback', data_
environment=DataEnvironment.PUBLIC)
manual_alerts_df = manual_alerts_df.withColumn('activity_id', f.get_json_object('triggers',
'${[0].triggerId}'))
manual_alerts_df = manual_alerts_df.withColumn('risk_id', f.get_json_object('triggers',
'${[0].risk.id}'))
manual_alerts_df = manual_alerts_df.withColumn('effective_date', f.to_timestamp(f.get_json_
object('triggers', '${[0].occurredOnDate'})))
manual_alerts_df = manual_alerts_df.withColumn('suppressed', f.lit(False))
manual_alerts_df = manual_alerts_df.withColumn('id', f.concat(f.col('activity_id'), f.lit
('_'), f.col('risk_id'))))
manual_alerts_df = manual_alerts_df.withColumn('account_id', f.get_json_object('triggers',
'${[0].primaryKeySet.account_id}'))
publish_nodes(context,
              manual_alerts_df.select('activity_id', 'risk_id', 'effective_date',
'suppressed', 'id'),
              graph_identifier='public',
              node_type='alert',
```

```
    data_environment=DataEnvironment.PUBLIC)
from thetaray.api.graph.search import get_nodes_by_properties
from itertools import chain

manual_alerts_account_ids = [record['account_id'] for record in manual_alerts_df.select
('account_id').collect()]

account_id_node_id_map = {account_id: get_nodes_by_properties(context,
                                                               'public',
                                                               'account',
                                                               properties={'account_
number': account_id},
                                                               data_
environment=DataEnvironment.PUBLIC)[0]['id']
                           for account_id in manual_alerts_account_ids}

manual_alerts_df = manual_alerts_df.withColumn('account_id_node_id_map', f.create_map
([f.lit(i) for i in chain(*account_id_node_id_map.items())]))
manual_alerts_df = manual_alerts_df.withColumn("node_id", f.col('account_id_node_id_map')
[f.col('account_id')])
al_edges_df = manual_alerts_df.withColumn('source_node', f.col('id'))
al_edges_df = al_edges_df.withColumn('target_node', f.col('node_id'))
publish_edges(context,
              al_edges_df.select('id', 'effective_date', 'source_node', 'target_node'),
              'public',
              edge_type='alerted',
              source_node_type='alert',
              target_node_type='account',
              data_environment=DataEnvironment.PUBLIC, parallelism = <4>)
context.close()
```

**Note:** Regarding the parallelism parameter value detailed in the above sample code, be aware that using the parallelism parameter is optional and the value shown is an example placeholder only. Any value up to total\_work\_slots can be used, but the value -1 is the default value, and if this is used, the parallelism mechanism will be disabled.

## 8.2. Upgrading NV from pre 6.8 Deployments

### 8.2.1. Phase #2 Upgrade Guidelines

#### 8.2.1.1. Overview

With Phase 2 of the Network Visualization feature, essential improvements in performance and storage optimization have been undertaken. Specifically, both node and edge types, as well as their properties, have been renamed to 2-3 letter names. This modification results in enhanced performance and reduced disk space usage.

While we strive to automate and streamline as much of the upgrade process as possible, the unique nature of this upgrade necessitates some manual interventions. Specifically, the described changes impact customer-specific implementations.

### 8.2.2. Pre - upgrade Manual Information / Directions

Before proceeding with the upgrade steps process, it is advisable to read through the following additional information and new mappings tables data.

#### 8.2.2.1. Metadata

- **Graph Metadata:** Update according to the new mapping guidelines.
- **Related Graph in Evaluation Flow:** Ensure references to the graphs in Evaluation Flows (aka Related Graph) are updated and aligned with the new mapping.

#### 8.2.2.2. Publishing and Usage of Graph Data

Make necessary adjustments for:

- Publishing of nodes/edges.
- Reading of nodes/edges (in such areas as ERP, party-level evaluations, and others).

#### 8.2.2.3. Scripts

- **Notebook upgrade\_nodes.ipynb:** This is designed to upgrade node data.
- **Notebook upgrade\_edges.ipynb:** This script upgrades edge data.
- **Cleanup Notebook (upgrade\_drop\_backups.ipynb):** After successfully upgrading, run this script to clean up the old data

### 8.2.3. New Mappings

For detailed mapping conversions from old node/edge types and properties to the new 2-3 letter names, please refer to the provided tables detailed below.

#### 8.2.3.1. Node Mappings

Old Identifier	New Identifier
account	AC
party	PR
alert	AL

#### 8.2.3.2. Property Mappings for Nodes

Account (AC)		Party (PR)		Alert (AL)	
Old Identifier	New Identifier	Old Identifier	New Identifier	Old Identifier	New Identifier
account_number	AN	party_uuid	PI	activity_id	AI
Name	NM	name	NM	risk_id	RI
country	CT	address	AD	suppressed	SP
address	AD	country	CT		
		consolidation	CN		
		updated_on	UON		
		updated_by	UBY		
		email	EM		
		address	AD		

#### 8.2.3.3. Edge Mappings

Old Identifier	New Identifier
transaction	TX
alerted	AL
er_match	ER

### 8.2.3.4. Property Mappings for Edges

Transaction (TX)		Alerted (AL)	ER match (ER)	
Old Identifier	New Identifier		Old Identifier	New Identifier
amount	AM	No properties	state	ST
currency	CR		score	SC
count	CT			

### 8.2.4. Manual Upgrade Steps in Detail

#### 8.2.4.1. Metadata Update

1. Graph Metadata: Navigate to the graph metadata configuration section (under domains/<domain>/graphs). Align and rename old properties to their new shorter counterparts based on the mapping tables.
2. Related Graph in the Evaluation Flow: Under domains/<domain>/evaluation\_flows, ensure that all the evaluation flows, using graphs, have the updated names in the RelatedGraph reference. For example, the following original configuration:

```
RelatedGraph(
    identifier="public",
    node_mappings={"account": [PropertyToFieldMapping(field="account_id",
property="account_number")]},
    grouping_node_mappings={"party": [PropertyToFieldMapping(field="party_id",
property="party_uuid")]},
)
```

Should be modified to:

```
RelatedGraph(
    identifier="public",
    node_mappings={"AC": [PropertyToFieldMapping(field="account_id", property="AN")]},
    grouping_node_mappings={"PR": [PropertyToFieldMapping(field="party_id",
property="PI")]},
)
```

### Update Implementations

1. Publishing Nodes/Edges:
  - a. Publish\_nodes and publish\_edges functions accept types as parameters (type for both, source\_node\_type and target\_node\_type for edges). Those references should be renamed according to the mapping tables.
  - b. Those functions accept data frames, containing columns per each property of a node/edge. Those columns should also be renamed according to the new naming.
  - c.

## 2. Reading Nodes/Edges:

Graph data might be used in different areas of the solution. E.g. Entity Resolution Process (ERP) and generating data for a party-level evaluation, read nodes and edges and process them in different ways. This usage should be adjusted, as both types, provided to read\_nodes/read\_edges functions are being changed, and the names of the properties in the returning data frame.

## Data Upgrade Scripts

### 1. Preparation:

Before initiating the data upgrade process, it's important to evaluate the relevance of your existing data in relation to the new capabilities offered by Network Visualization. If you've identified data that will not be compatible with the new features, or if you plan to republish certain graphs post-upgrade, it is highly advised to delete this irrelevant data prior to upgrading. Taking this step will:

- Simplify the upgrade process.
- Reduce the time required for the upgrade.
- Conserve disk space.

### 2. Data upgrade:

Two new notebooks have been added to the Reference branch under domains/default/notebooks - upgrade\_nodes.ipynb and upgrade\_edges.ipynb. Each of them goes over the configured graphs and maps the existing data to the new convention. Old-formatted data remains in the system in the tables, prefixed with "old\_".

### 8.2.4.2. Post Upgrade

After some time, when an upgrade is considered to have been successful, the tables with the old data can be removed. The notebook domains/default/notebooks/upgrade\_drop\_backups.ipynb will remove those backups for all configured graphs.

## 9. Distribution

### 9.1. The distribute\_alerted\_activities function

Once identified, risks can be distributed to one or multiple Investigation Centers and other case managers. To activate the distribution process, call the distribute\_alerted\_activities function. The function sends an HTTP POST request to the distribution target containing a JSON representation of the alerts.

```
from thetaray.api.distribution import distribute_alerted_activities

distribute_alerted_activities(context, EVALUATION_FLOW, 'investigation_center_alert_
distribution_target')
```

The **distribute\_alerted\_activities** accepts the following parameters:

- **context**: the context object includes the execution\_date parameter used for selecting which data records to read from the evaluated\_activities table.
- **distribution\_target\_identifier**: The identifier of the alert distribution target. **The distribution\_target\_identifier** is a wrapper for the **AlertDistributionTarget** object (see Define a distribution target.)
- **evaluation\_flow\_identifier**: the identifier of the evaluation flow is used to find the correct evaluated\_activities table in the database.
- **from\_job\_ts**: Specify the timestamp of the earliest data record selected for identification. This parameter is useful for distributing or re-distributing risks in a subset of historical data records. You use this parameter to set the time range lower bound.
- **to\_job\_ts**: Specify the timestamp of the latest data record selected for identification. This parameter is useful for identifying or re-identifying risks in a subset of historical data records. Use this parameter to set the time range's lower bound.
- **data\_environment**: Set whether the data environment is PUBLIC or PRIVATE(default); see Read/Write to public/private database area. When the data environment is private, the function will not execute.
- **force**: Force distribution execution, even when the DateEnvironment is set to PRIVATE (default: false).
- **batch\_size**: Number of alerts included in the JSON payload of the HTTP POST request (default: 4).

- **parallelism:** The number of Spark workers to use for the distribution, the parallelism parameter value is optional. Any value up to `total_work_slots` can be used, but the value **4** is the default value.
- **retry\_policy:** Retry Policy in case of technical failures.

## 9.2. Define a distribution target

ThetaRay Platform can distribute alerts to the *Investigation Center* or any other 3rd party case management system. When the distribution target is the *Investigation Center*, the integration is automatic, and there is nothing to do except installing the Investigation Center in the environment. But when the distribution target is a 3rd party system, you need to define an **AlertDistributionTarget** object.



Create the destination target on a python file under `domains/[DOMAIN_NAME]/targets/`

```
from thetaray.api.solution.alert_distribution_target import AlertDistributionTarget,  
TargetAuth, AuthenticationMode, RetryPolicy  
  
def Investigation_Center_alert_distribution_target():  
    return AlertDistributionTarget(  
        identifier="cm-1"  
        name="Case Manager"  
        url="https://casemngr.cm.thirdparty.com/cm-be/api/alerts/create/anomaly"  
        auth=TargetAuth(mode=AuthenticationMode.BASIC_AUTH,  
                        configuration={"client id": "idanbe_idanbe_dev3_ic2_be", "client  
secret key": "thetaray"}))  
  
def entities():  
    return [Investigation_Center_alert_distribution_target()]
```

## 10. Real Time Transaction Monitoring

### 10.1. Overview

Real Time Transaction Monitoring refers to the real time analysis of transactions in order to find patterns of risky or irregular behavior. The analysis is done in real time via deterministic rules that are customized to fit the financial use cases and relevant regulations. The real time analysis provides the results of the rules as a match or no match response.

Distinguishing real-time transaction monitoring from post-analysis transaction monitoring is crucial in understanding their respective roles within the regulatory framework. Real-time transaction monitoring involves the continuous surveillance of transactions as they occur, allowing for immediate detection and response to suspicious activities. In contrast, post-analysis transaction monitoring entails the retrospective examination of historical data to identify patterns or anomalies after the fact.

For more information on supported rules please refer to the Real Time Transaction Monitoring Overview document.

### 10.2. Evaluation Flow

A new 'realtime' attribute was added to the evaluation flow metadata which expects the following:

1. `input_connector` - mandatory, the expected input is the reference to the connector identifier of REST type used to store the real time transaction request in both minio/postgres
2. `distribution_target_identifier` - mandatory, the expected input is the reference to the alert target
3. `callback_target_identifier` - identifier of the `RealtimeCallbackTarget` necessary to receive the callback on Real Time alert closure. For more details on `RealtimeCallbackTarget` see 'Close Alert Callback API' section.

#### 10.2.1. Evaluation Flow Example

```
1  from thetaray.api.solution.evaluation import RealtimeEvaluation
2  return EvaluationFlow(
3      identifier="tr_rt_ef",
4      displayName="tr_rt_ef",
5      input_dataset="rt_transaction",
6      data_permission="dpv:public",
7      ef_type=EvaluationType.THETARAY_ANALYSIS,
```

```
8     realtime=RealtimeEvaluation(
9         input_connector="rt_transaction",
10        distribution_target_identifier="ext_rt",
11        callback_target_identifier="callback_rt"
12    ),
13    max_workers=8,
14    evaluation_steps=[
15        AlgoEvaluationStep(
16            identifier="rt_dummy_evaluation",
17            name="Dummy evaluation",
18            feature_extraction_model=ModelReference(name="rt_fem_dummy", tags={
19                "version": "release"
20            }),
21            detection_model=ModelReference(name="rt_dm_dummy", tags={
22                "version": "release"
23            })
24        ],
25    )
26)
```

## 10.3. Evaluation Models and Histograms

Histograms are the part of the data, required to display alerts in the Investigation Center. As histograms are coupled with the models, a “dummy” model should be created and defined for the Real-Time Evaluation Flow.

Implementation steps:

1. Train and save models (Feature Extraction and Detection), based on the input dataset of the Evaluation Flow. Same as in the batch processing, histograms should be saved using the `save_histograms` function under the same Mlflow run.
2. Add a model reference to the Evaluation Flow and push the change to the Staging branch. After that, the Metadata Sync process will retrieve the histograms and publish them to be used by further Real-Time evaluations.

---

**Note:** If the Evaluation Flow already contains a reference to the model, then a new version should be saved with the different tags, which should also be updated in the Evaluation Flow metadata. This would allow the Metadata Sync process to identify the change and perform needed operations.

---

## 10.4. REST Connector Configurations (Customer Facing API)

Once the “real time” attribute is defined on the Evaluation Flow, a REST endpoint is configured automatically.

## 1. Endpoint format:

"{gw\_url}/evaluate/v1/{solution}/{ef\_identifier}"  
for example -

```
1 | https://gateway-my-shared.development.thetaraydev.com/evaluate/v1/mysolution/tr_rt_ef'
```

2. Request Body - the request body is a dynamic JSON message that is required to have the same structure as the input dataset.
3. Automatically supported validations:
  - a. Date-time format
  - b. Data type mismatch
4. A new 'publish' attribute was added to RestConfig, set by default to TRUE for the real time connector, which publishes every incoming request to the target dataset table immediately.

#### 10.4.1. Example of API Integration (to be done by the customer)

```
1 import requests
2 import os
3 gw_url=f"https://gateway-{os.environ['SHARED_NAMESPACE']}.{os.environ['CLUSTER_DOMAIN']}"
4 def get_token():
5     body = {
6         "clientId": f"platform-ingestion-{os.environ['SOLUTION']}",
7         "clientSecret": "[PUT_SECRET]"
8     }
9     header = {
10         'Content-type': 'application/json',
11         'Accept': 'application/json'
12     }
13     response = requests.post(f"{gw_url}/auth", json=body, headers=header)
14     token = response.json()['result']['token']
15     return token
16 header = {
17     'Content-type': 'application/json',
18     'Accept': 'application/json',
19     'Authorization': f'Bearer {get_token()}'
20 }
21 row={
22     'transaction_id':1,
23     'customer_id':'customer_001',
24     'account_id':'1111111',
25     'date':'2023-12-15',
26     'type':'credit',
```

```
27     'operation':'remittance to another bank',
28     'bank':'bank1',
29     'cnp_account_id':None,
30     'amount':10000
31 }
32 response = requests.post(f"{gw_url}/evaluate/v1/development/tr_rt_ef", json=row,
33 headers=header)
34 print(response.json())
```

#### 10.4.2. Connector Configuration Example

```
1  return FlatFileConnector(
2      identifier="rt_transaction",
3      display_name="rt_transaction",
4      source=Source(
5          folder="rt_transaction",
6          has_header=True,
7          delimiter=",",
8          timestamp_conf=TimestampConf(type=TimestampConfType.EXECUTION_DATE, filename_
pattern="%Y%m%d %H:%M:%S"),
9          dataset_schema_ref="rt_transaction",
10         retention_policy=RetentionPolicy(months_to_keep=24),
11     ),
12     targets=[Target(dataset_name="rt_transaction")],
13     default_target_dataset_name="rt_transaction",
14     rest_config=RestConfig(
15         enable=False,
16         max_buffered_records=1_000_000,
17         flush_interval_minutes=60,
18         read_chunk_size=10_000,
19     ),
20 )
```

### 10.5. Dataset Metadata

A new 'realtime' attribute was added to the dataset metadata.

Important Note: All datasets that are involved in realtime flow should be marked as realtime

## 10.6. Close Alert Callback

**Note:** Please note that in order to enable the Close Alert Callback, a service task needs to be configured in the relevant Workflow file in the IC Settings. For more details, please refer to the current version of the **activiti** application User Guide Support .

### 10.6.1. Target URL

A new type of target was added called RealtimeCallbackTarget, used to configure an external endpoint which would receive notifications upon Real Time alert closure. Example:

```
1 def callback_target():
2     return RealtimeCallbackTarget(
3         identifier="callback_rt",
4         name="callback_rt",
5         url="https://outside-url.domain.net",
6         auth=TargetAuth(mode=AuthenticationMode.NO_AUTH),
7         verify=False,
8     )
```

### 10.6.2. Authentication

Supported TargetAuth modes: NO\_AUTH, BASIC\_AUTH.

NO\_AUTH:

```
1 def callback_target_no_auth():
2     return RealtimeCallbackTarget(
3         identifier="callback_rt",
4         ...
5         auth=TargetAuth(
6             mode=AuthenticationMode.NO_AUTH
7         ),
8     )
```

BASIC\_AUTH:

**Note:** username and password configuration should be provided inside TargetAuth object

```
1 def callback_target_basic():
2     return RealtimeCallbackTarget(
```

```
3     identifier="callback_rt",
4     ...
5     auth=TargetAuth(
6         mode=AuthenticationMode.BASIC_AUTH,
7         configuration={
8             "username": "some_username",
9             "password": "some_password"
10        }
11    ),
12 )
```

Encrypted passwords (AES encrypted by same key that's provided in platform setting ENCRYPTION\_KEY) are also supported - should be provided starting with prefix "encrypted:"

```
1 def callback_target_basic():
2     return RealtimeCallbackTarget(
3         identifier="callback_rt",
4         ...
5         auth=TargetAuth(
6             mode=AuthenticationMode.BASIC_AUTH,
7             configuration={
8                 "username": "some_username",
9                 "password": "encrypted:BUxdFK8DI+cjSixBqZcaAVSJAZGf0L2/JgwT/REw6iVd1XjLAG19vTwcIYzy"
10            }
11        ),
12    )
```

### 10.6.3. Redis configuration

Following are the configurations that affect REST connector behavior when interacting with Redis. They are set in commons.yaml configuration file in the "redis:" section:

1. min\_idle\_time - minimum time in milliseconds for the record to stay in the stream in the PENDING status before it would be reclaimed, default 60000.
2. batch\_size - number of records to be read by Rest Connector from Redis in one batch, default 100.
3. autoclaim\_interval\_minutes - defines interval for Rest Connector to check for pending records in the Redis Streams, default 10.
4. stream\_length\_warning - when Redis Stream reaches the defined length, the monitoring alert will be triggered, default 10000.
5. stream\_length\_monitor\_interval - interval in minutes to check Streams length, default 5.
6. read\_block\_millis - defines how much time read operation should wait, if there are no records in the Redis Streams on the reading attempt. Set in milliseconds, default 1000.

---

**Note:** A new 'publish' attribute was added to RestConfig, default False. When set to True, every incoming request will be published to the target dataset table immediately.

---

# 11. Rule Builder - Documentation

## 11.1. Deployment

- common.yaml
- rule builder request /response timeout - default 30 seconds

```
rule_builder:  
  httpx_timeout: 30
```

## 11.2. Configuration

The simulation test is controlled by setting a configuration in the evaluation\_flow metadata file.

A new parameter is added in the evaluation\_flow called 'rule\_builder\_config' which supports the following options:

Attribute	Type	Description	Default Value	Remarks	Example
simulation_spark_config	dict	control spark, supports on any formal spark configuration	"spark.sql.hive.metastore PartitionPruningFallback OnException": "true"	providing this attribute will merge with default value and the project setting	{"spark.executor.memory": "4g"}
chunk_size	int	identify_risks function input parameter	10	providing this attribute will override the default value	3
fetch_size	int	identify_risks function input parameter	10000	providing this attribute will override the default value	500

```

1  from thetaray.api.solution import AlgoEvaluationStep, EvaluationFlow
2  from thetaray.api.solution.evaluation import (
3      EvaluationFlowIdentifiers,
4      EvaluationTimeWindowUnit,
5      ModelReference,
6      EFRuleBuilderConfig,
7  )
8  def evaluation_flow() -> EvaluationFlow:
9      return EvaluationFlow(
10         identifier="tr_analysis_uk",
11         displayName="tr_analysis_uk",
12         input_dataset="wrangling",
13         data_permission="dpv:UK",
14         time_window_size=1,
15         time_window_unit=EvaluationTimeWindowUnit.MONTH,
16         identifiers=.....
17         evaluation_steps=[.....],
18         max_workers=8,
19         rule_builder_config=EFRuleBuilderConfig(
20             simulation_spark_config={"spark.executor.memory": "4g"},
21             chunk_size=3,
22             fetch_size=500
23         ),
24     )
25 def entities() -> List[EvaluationFlow]:
26     return [evaluation_flow()]
27

```

**Note:** Each simulation has his own schema for supporting on various rules settings and storing the simulation test result.

This schema does not contains the full solution tables/functions(e.g. published dataset) but only evaluation flow tables set , which are required by simulation test.

As a result if existing project risk has enrichment with reference to publish dataset , the enrichment must refer to the full dataset path in format of [schema\_name].[dataset\_identifier].

```

1  enrichments:
2      - name: country
3          sql: |
4              select a.country as ac_country, cr.risk as ac_country_risk from my_schema.account
5              as a
6                  join my_schema.country_risk as cr on a.country = cr.country_code
7                  where a.account_id = activity.account_id
8          outputs:
9              - name: ac_country
10                 datatype: STRING

```

```
10 |     - name: ac_country_risk
11 |     datatype: STRING
```

**Security:**

- Same roles as Rule Parameter Editor for admin and developer
- [Rule]-Tester - all operations allowed except from Apply/Delete
- [Rule]-Editor - all operations are allowed

## 12. CRA - Customer Risk Assessment

### 12.1. Overview

The CRA product enables taking various risk factors into consideration when determining the risk levels, such as KYC information, transaction monitoring, Sanction and PEP screening results, etc.. A risk score and a risk classification are calculated from the various risk factors and externalized both to the clients' internal systems and to the ThetaRay Investigation Center.

The CRA product receives a customer or batch of customers and runs the risk calculation model, which is enhanced with additional information, such as the relevant high risk countries. The model also has the option to be enhanced from ThetaRay internal events, such as resolving a screening alert. The risk calculation model is comprised of the risk score calculation and various other rules, and is adjusted to fit the client's specific needs and risk appetite.

In addition to the creation of a dedicated CRA alert in the Investigation Center, the results of the risk calculation model are externalized via API, and are generated into a report. The report is compiled for all the customers classified in the CRA analysis and is available for retrieval via API and through the CRA On Demand Reporting Module. The On Demand Reporting Module has various other capabilities which are elaborated in the corresponding section of this document.

The CRA analysis is built upon existing platform infrastructure, including data ingestion options, evaluation flow and risks. The CRA comes with an extensive Reference Implementation, that can be copied and adjusted to fit the implementation requirements. Details on how to set up and configure the CRA functionalities are found in the following sections.

## 12.2. Data Ingestion

### 12.2.1. Transaction Data

Transactional data can be ingested as any transactions for Transaction Monitoring, including via files, raw transaction formats (RJE and XML) or REST API. The transactions uploaded for Transaction Monitoring analysis can also be reused for CRA analysis.

### 12.2.2. KYC & Auxiliary Data

Similar to transactional data, KYC and auxiliary datasets can be ingested and reused between Transaction Monitoring and CRA analyses. Auxiliary data is crucial for setting the CRA risk factors and their values.

### 12.2.3. Screening Feedback Loop

The Screening Feedback Loop is designed to enrich the CRA analysis with the results of the Screening alerts from the ThetaRay Investigation Center.

Information for the Screening Feedback Loop is gathered in 2 steps, for the first one was extended datasets scrn\_customer and scrn\_transaction by adding a new field - match\_data where stored information about Screena's matches. Another part is adding the new dataset scrn\_feedback\_loop with information from IC there is saved information about approved and blocked alerts. These data in sum enrich the CRA analysis.

Before uploading to the corresponding tables, CSV files that are used for analysis, are stored in the Minio in the public/datasources folder.

For setting it up should be created solution, a custom one or from the Reference branch and merged the Screening branch to the Staging branch. And for applying changes from the branch need to run /domains/screening/notebooks/daily\_data\_upload.ipynb notebook or scrn\_daily\_data\_upload DAG.

## 12.3. Evaluation Flow

A new field has been added to EvaluationFlow metadata entity: ef\_type

The new field accepts the following values:

```
1 class EvaluationType(Enum):  
2     THETARAY_ANALYSIS = 0 #Default  
3     CLASSIFIER = 1  
4     BUSINESS_RULES = 2  
5     CRA = 3
```

### Example:

```
1 from typing import List  
2  
3 from thetaray.api.solution import AlgoEvaluationStep, EvaluationFlow, ParquetIndex, TraceQuery  
4 from thetaray.api.solution.evaluation import (  
5     EvaluationFlowIdentifiers,  
6     ModelReference,  
7     PropertyToFieldMapping,  
8     RelatedGraph,  
9     EvaluationType  
10 )  
11  
12 def cra_evaluation_flow() -> EvaluationFlow:  
13     return EvaluationFlow(  
14         identifier="tr_cra_ef",  
15         displayName="tr_cra_ef",
```

```
16     input_dataset="customer_analysis",
17     data_permission="dpv:public",
18     ef_type=EvaluationType.CRA,
19     max_workers=8
20 )
21
22 def entities() -> List[EvaluationFlow]:
23     return [cra_evaluation_flow()]
```

## 12.4. Decisioning Risk

### 12.4.1. Decisioning Enhancements

Enhancements have been made to the the decisioning process to support CRA analysis:

- Adding “order” within the variables section that determines which variable will be run first - crucial for calculating the risk score
- The attribute 'display setting' is now optional
- Variables have been extended with an “override” attribute - please note that this is relevant for classification variables only at this stage.

---

**Note:** This attribute should always be set to “True” to allow for manual reclassification from the IC.

---

- Variables and Enrichment Output have been extended with the following attributes:
  - valid\_values - list of possible fix values - there is no validation on the platform side to enforce on runtime
  - include\_in\_hash - include in fields hash, used to identify the reclassification effectiveness

- annotations - list of key, value pair , mainly used for the IC alert
- distribute - add to request payload when distributing risk to IC
- A new section called “resources” has been added - a pair of key value, used by the annotations as a reference, e.g. {resources.key} will be replaced by the real value

#### 12.4.2. Identify Risks Enhancement

A new API has been added as a flag to the “identify\_risks” function called “reset\_variable\_overrides”, which is by default set to false.

If true, in case of a rerun on the same execution date all variables data that marked with override: True in the risk yaml will be erased. Please note that this is relevant currently only for “classification” variable, as it can be modified via the alert. This should be used only in the case where a CRA analysis needs to be rerun on the same execution date and provide the results again, and the manual reclassification done by the analysts needs to be overwritten by the new analysis.

##### Example:

```
1 def identify_risks(context,
2                     evaluation_flow_identifier: str,
3                     from_job_ts: datetime = None,
4                     to_job_ts: datetime = None,
5                     data_environment: DataEnvironment = DataEnvironment.get_default(),
6                     parallelism: int = Constants.PARALLELISM_DEFAULT_VALUE,
7                     ignore_block: bool = False,
8                     chunk_size: int = 1000,
9                     num_of_cores: int = os.cpu_count(),
10                    reset_variable_overrides: bool = False):
```

### 12.4.3. Pre Evaluation Rules

Deterministic classification logic can optionally be executed prior to invocation of the 'evaluate' step which applies one or more machine learning models on customer data to determine the classification. This approach is used for implementation that would like to skip model based inferencing for customers that are already rated by deterministic rules.

**To support the above the following steps should be taken -**

Risk variables associated with classification logic that needs to run before 'evaluate' should be tagged configured with a 'phase' equals to 'pre\_algo'.

Example:

```
name: exclusive_rule_1_am1
display_name: exclusive_rule_1_am1
...
phase: PRE_EVALUATE
```

An exclusion filter should be configured on the relevant Evaluation Step, to indicate which rows should be excluded from model based evaluation.

Example:

```
EvaluationFlow(
    identifier="tr_cra_ml",
    displayName="CRA ML Analysis",
    ...
    evaluation_steps=[
        SemiSupervisedClassificationEvaluationStep(
            identifier="aml_ml",
```

```
    name="AML evaluation",
    ...
    exclusion_filter=lambda df: df["rvp_cra_ml_aml_rules_classification"].notnull()
)
```

The 'enrich' API should be called prior to invoking 'evaluate' on the 'input dataset' Spark DataFrame to trigger rules evaluation before running the ThetaRay algorithms.

Example:

```
enriched_df = enrich(context, evaluation_identifier='tr_cra_ml', input_dataframe=detect_df)
evaluated_df = evaluate(
    context=context,
    evaluation_identifier="tr_cra_ml",
    dataframe=enriched_df,
)
```

#### 12.4.4. Annotations

Category	Annotations	Example	Remark
KYC	<ul style="list-style-type: none"><li>• kyc: true</li><li>• kyc.order: &lt;number&gt;</li></ul>	<pre>1   Field( 2       identifier="source_of_funds", 3       display_name="Source of funds", 4       data_type=DataType.STRING,</pre>	<u>NOTE</u> - in practice this is the only annotation used by the dataset

Category	Annotations	Example	Remark
		<pre> 5       annotations={"kyc": True, 6                             "kyc.order": 2}, 7   ), </pre>	
RISK LEVEL	<ul style="list-style-type: none"> <li>• risk_level: true</li> <li>• group_risk_factor</li> <li>• sub_factor</li> <li>• client_data_column</li> </ul>	<pre> 1   annotations: 2     risk_level: True 3     group_risk_factor: "{resources.person}" 4     sub_factor: "Nationality / Jurisdiction" 5     client_data_column: nationality </pre>	client_data_column → name of the column, holding client data value
GROUP SCORE	<ul style="list-style-type: none"> <li>• group_risk_score: true</li> <li>• group_risk_factor</li> <li>• weight_parameter</li> </ul>	<pre> 1   annotations: 2     group_risk_score: True 3     group_risk_factor: "{resources.person}" 4     weight_parameter: "{parameters.person_weight}" </pre>	weight_parameter → value of the factor weight parameter
RISK SCORE	score: true	<pre> 1   annotations: 2     score: True </pre>	
SUB FACTOR - AUTO HIGH	<ul style="list-style-type: none"> <li>• sub_factor_classification: true</li> <li>• group_risk_factor</li> <li>• sub_factor</li> </ul>	<pre> 1   annotations: </pre>	

Category	Annotations	Example	Remark
	<ul style="list-style-type: none"> <li>client_data_column</li> </ul>	<pre> 2       sub_factor_classification: True 3       group_risk_factor: "Compliance diirective" 4       sub_factor: "Decision of the Compliance..." 5       client_data_column: "{variables.compliance_decision}" </pre>	
CLASSIFICATION_TYPE	classification_type: true	<pre> 1   Field( 2       identifier="classification_type", 3       display_name="Classification type (Individual / Company)", 4       data_type=DataType.STRING, 5       annotations={"classification_type": True}, 6   ), </pre>	Individual / Company
CLASSIFICATION	classification: true	<pre> 1   annotations: 2       classification: True </pre>	
ANALYSIS TYPE	tag: analysisType	<pre> 1   annotations: 2       tag: analysisType </pre>	onboarding / ongoing indicator

Category	Annotations	Example	Remark
CUSTOMER ID	customer_id: true	<pre> 1 Field( 2   identifier="customer_id", 3   display_name="Customer ID", 4   data_type=DataType.STRING, 5   annotations={"kyc": True, 6               "kyc.order": 1, 7               "customer_id": True}, 8 ) </pre>	
CUSTOMER NAME	customer_name: true	<pre> 1 Field( 2   identifier="customer_name", 3   display_name="Customer Name", 4   data_type=DataType.STRING, 5   annotations={"customer_name": True}, 6 ) </pre>	
DATE	date: true	<pre> 1 Field( 2   display_name="date", 3   data_type=DataType.TIMESTAMP, 4   date_format="%Y%m%d", 5   identifier="date", 6   annotations={"date": True}, 7 ) </pre>	

#### 12.4.5. Risk Example

The following example can be found in the reference implementation branch, and should provide a good example of the risk factors the CRA product supports.

```
1  identifier: cra_risk
2  display_name: CRA
3  description: CRA
4  category: cra
5  severity: 27
6  dynamic_display_name: "{{ activity.first_name|capitalize }} {{ activity.last_name|capitalize }} {% if variables.cls_change_dynamic_
  display == 'active' %} | {{ enrichments.prev_cls|capitalize }} → {{ variables.classification|capitalize }} {% endif %}"
7  # dynamic_description:
8  # dynamic_category:
9  # dynamic_severity:
10 # dynamic_suppression_identifier:
11 suppression_period_unit: DAY
12 suppression_period_size: -1
13 evaluation_flows:
14   - tr_cra_ef
15 # List of IC group names to group risk factors by
16 resources:
17   person: "Person"
18   jurisdiction: "Jurisdiction"
19   trx: "Transaction"
20   channel: "Channel"
21   products: "Products"
22   auto_high: "Auto-High"
23
24 # List of global parameters
```

```
25 parameters:
26 # List of weights to multiply by with the total score - each main factors group need to have a weight to multiple by
27 - name: person_weight
28   datatype: DOUBLE
29   value: "0.4"
30 - name: trx_weight
31   datatype: DOUBLE
32   value: "0.2"
33 - name: channel_weight
34   datatype: DOUBLE
35   value: "0.05"
36 - name: products_weight
37   datatype: DOUBLE
38   value: "0.1"
39 - name: jurisdiction_weight
40   datatype: DOUBLE
41   value: "0.25"
42 # List of each factor group size - by multiply each with global_max_risk_level we will get the max total for each group
43 - name: num_of_person_rf # rf = risk factors
44   datatype: LONG
45   value: "4"
46 - name: num_of_trx_rf
47   datatype: LONG
48   value: "6"
49 - name: num_of_channel_rf
50   datatype: LONG
51   value: "1"
52 - name: num_of_products_rf
53   datatype: LONG
```

```
54     value: "1"
55     - name: num_of_jurisdiction_rf
56         datatype: LONG
57         value: "1"
58 # List of the classification ranges (between 0-100)
59     - name: low
60         datatype: DOUBLE
61         value: "0"
62     - name: medium
63         datatype: DOUBLE
64         value: "40"
65     - name: high
66         datatype: DOUBLE
67         value: "70"
68 # List of extra global params
69     - name: cash_trx_thrshld
70         datatype: LONG
71         value: "200000"
72     - name: global_max_risk_level # represent the highest value each risk factor can have, lower bound is always 0
73         datatype: LONG
74         value: "10"
75 # List of dataset enrichments
76 enrichments:
77     - name: analysis_type
78         sql: "select customer_cra_analysis_type as analysis_type from customer_analysis_type where customer_id = activity.customer_id and tr_timestamp <= {context.execution_date} order by tr_timestamp desc"
79         outputs:
80             - name: analysis_type
81                 datatype: STRING
```

```
82
83 # risk factors implementations
84 - name: onboarding_channel_risk_level
85   sql: "select risk_level as onboarding_channel from onboarding_channel where keyword_id = activity.onboarding_channel and tr_
86 timestamp <= {context.execution_date} order by tr_timestamp desc"
87   outputs:
88     - name: onboarding_channel
89       datatype: LONG
90       annotations:
91         risk_level: True
92         group_risk_factor: "{resources.channel}"
93         sub_factor: "Onboarding Channel" # represent the risk factor name in IC under the group name
94         client_data_column: onboarding_channel # need to be dataset field
95
96 - name: client_type_risk_level
97   sql: "select risk_level as client_type from client_type where keyword_id = activity.client_type and tr_timestamp <=
98 {context.execution_date} order by tr_timestamp desc"
99   outputs:
100     - name: client_type
101       datatype: LONG
102       annotations:
103         risk_level: True
104         group_risk_factor: "{resources.person}"
105         sub_factor: "Client Type and Structure"
106         client_data_column: client_type
107
108 - name: occupation_risk_level
109   sql: "select risk_level as occupation from occupation where keyword_id = activity.occupation and tr_timestamp <=
110 {context.execution_date} order by tr_timestamp desc"
```

```
108     outputs:
109         - name: occupation
110             datatype: LONG
111             annotations:
112                 risk_level: True
113                 group_risk_factor: "{resources.person}"
114                 sub_factor: "Economic Activity / Occupation"
115                 client_data_column: occupation
116
117         - name: customer_screening_risk_level
118             sql: "select risk_level as customer_screening from customer_screening where keyword_id = activity.customer_screening and tr_timestamp <= {context.execution_date} order by tr_timestamp desc"
119             outputs:
120                 - name: customer_screening
121                     datatype: LONG
122                     annotations:
123                         risk_level: True
124                         group_risk_factor: "{resources.person}"
125                         sub_factor: "Screening Matches"
126                         client_data_column: customer_screening
127
128         - name: trx_screening_risk_level
129             sql: "select risk_level as trx_screening from trx_screening where keyword_id = activity.trx_screening and tr_timestamp <= {context.execution_date} order by tr_timestamp desc"
130             outputs:
131                 - name: trx_screening
132                     datatype: LONG
133                     annotations:
134                         risk_level: True
```

```
135      group_risk_factor: "{resources.trx}"
136      sub_factor: "TRX Screening Matches"
137      client_data_column: trx_screening
138
139      - name: nationality_risk_level
140        sql: "select risk_level as nationality from cra_country_risk where country_code = activity.nationality and tr_timestamp <=
{context.execution_date} order by tr_timestamp desc"
141        outputs:
142          - name: nationality
143            datatype: LONG
144            annotations:
145              risk_level: True
146              group_risk_factor: "{resources.person}"
147              sub_factor: "Nationality (Individual) / Jurisdiction (Company)"
148              client_data_column: nationality
149
150      - name: geo_origin_of_funds_risk_level
151        sql: "select risk_level as geo_origin_of_funds from geo_origin_of_funds where country_code = activity.nationality and tr_timestamp
<= {context.execution_date} order by tr_timestamp desc"
152        outputs:
153          - name: geo_origin_of_funds
154            datatype: LONG
155            annotations:
156              risk_level: True
157              group_risk_factor: "{resources.jurisdiction}"
158              sub_factor: "Geographic Origin of the Funds"
159              client_data_column: nationality
160
161      - name: amount_of_products_risk_level
```

```
162     sql: "select risk_level as amount_of_products from amount_of_products where from_range <= activity.amount_of_products and
activity.amount_of_products <= to_range and tr_timestamp <= {context.execution_date} order by tr_timestamp desc"
163     outputs:
164       - name: amount_of_products
165         datatype: LONG
166         annotations:
167           risk_level: True
168           group_risk_factor: "{resources.products}"
169           sub_factor: "Amount Of Products"
170           client_data_column: amount_of_products
171
172     - name: total_trx_amount_risk_level
173       sql: "select risk_level as total_trx_amount from total_trx_amount where from_range <= activity.total_trx_amount and
activity.total_trx_amount <= to_range and tr_timestamp <= {context.execution_date} order by tr_timestamp desc"
174       outputs:
175         - name: total_trx_amount
176           datatype: LONG
177           annotations:
178             risk_level: True
179             group_risk_factor: "{resources.trx}"
180             sub_factor: "Total Amount in Transactions (Annual - Prev. 365 days)"
181             client_data_column: total_trx_amount
182
183     - name: total_cash_trx_amount_risk_level
184       sql: "select risk_level as total_cash_trx_amount from total_cash_trx_amount where from_range <= activity.total_cash_trx_amount and
activity.total_cash_trx_amount <= to_range and tr_timestamp <= {context.execution_date} order by tr_timestamp desc"
185       outputs:
186         - name: total_cash_trx_amount
187           datatype: LONG
```

```
188     annotations:
189         risk_level: True
190         group_risk_factor: "{resources trx}"
191         sub_factor: "Total Amount in Cash Transactions (Annual - Prev. 365 days)"
192         client_data_column: total_cash_trx_amount
193
194     - name: total trx count risk level
195         sql: "select risk_level as total trx count from total trx count where from_range <= activity.total trx count and activity.total_
196         trx count <= to_range and tr_timestamp <= {context.execution_date} order by tr_timestamp desc"
197         outputs:
198             - name: total trx count
199                 datatype: LONG
200                 annotations:
201                     risk_level: True
202                     group_risk_factor: "{resources trx}"
203                     sub_factor: "Quantity of Transactions (Annual - Prev. 365 days)"
204                     client_data_column: total trx count
205
206             - name: tm_resolution_risk_level
207                 sql: "select risk_level as tm_resolution from tm alerted where keyword_id = activity.tm_resolution and tr_timestamp <=
208                 {context.execution_date} order by tr_timestamp desc"
209                 outputs:
210                     - name: tm_resolution
211                         datatype: LONG
212                         annotations:
213                             risk_level: True
214                             group_risk_factor: "{resources trx}"
215                             sub_factor: "TRX Monitoring Resolution Code"
216                             client_data_column: tm_resolution
```

```
215
216     - name: total_tm_alerts_risk_level
217         sql: "select risk_level as total_tm_alerts from total_tm_alerts where from_range <= activity.total_tm_alerts and activity.total_
218             tm_alerts <= to_range and tr_timestamp <= {context.execution_date} order by tr_timestamp desc"
219         outputs:
220             - name: total_tm_alerts
221                 datatype: LONG
222                 annotations:
223                     risk_level: True
224                     group_risk_factor: "{resources.trx}"
225                     sub_factor: "Amount of TM Alerts Prev. 365 days"
226                     client_data_column: total_tm_alerts
227
228     - name: reclass
229         sql: |
230             SELECT
231                 ac.rv_cra_risk_hash as old_class_hash,
232                 ov.rv_cra_risk_classification as user_cls
233             FROM
234                 activity_tr_cra_ef ac
235             JOIN activity_ovrd_tr_cra_ef ov ON ac.tr_id = ov.tr_id
236             WHERE
237                 customer_id = activity.customer_id
238                 AND created_on <= {context.execution_date}
239                 AND (created_on + interval '1 month') >= {context.execution_date}
240                 ORDER BY created_on desc
241         outputs:
242             - name: user_cls  # reclassification value (NULL if no reclassification)
243                 datatype: STRING
```

```
243     - name: old_class_hash # original system classification hash
244         datatype: STRING
245
246     - name: prev_cls # previous SYSTEM classification only, not USER
247         sql: |
248             select ac.rv_cra_risk_classification as prev_cls
249             from activity_tr_cra_ef as ac join tr_cra_ef as ef on ac.tr_id = ef.tr_id
250             where ef.customer_id = activity.customer_id and ef.tr_job_ts < {context.execution_date}
251             order by ef.tr_job_ts desc
252         outputs:
253             - name: prev_cls
254                 datatype: STRING
255
256
257 # List of calculated variables by order (e.g. factor totals, factor risks after multiple with weights etc)
258 variables:
259     - name: analysis_type
260         display_name: analysis_type
261         description: analysis_type
262         expression: |
263             CASE
264                 WHEN {enrichments.analysis_type} = 'OB' THEN 'Onboarding'
265                 WHEN {enrichments.analysis_type} = 'OG' THEN 'Ongoing'
266                 ELSE 'Undefined'
267             END
268         datatype: STRING
269         order: 0
270         annotations:
271             tag: analysisType # this annotation specify the analysis type of customer
```

```
272     valid_values: ["Ongoing", "Onboarding"]
273     distribute: True
274
275 # Auto high indicators
276 - name: exceed_cash_trx_thrshld_decision
277   display_name: exceed_cash_trx_thrshld_decision
278   description: exceed_cash_trx_thrshld_decision
279   expression: |
280     CASE
281       WHEN activity.total_cash_trx_amount > {parameters.cash_trx_thrshld} THEN 'Yes'
282       ELSE 'No'
283     END
284   datatype: STRING
285   order: 0
286
287 - name: compliance_officer_sar_worthy_decision
288   display_name: compliance_officer_sar_worthy_decision
289   description: compliance_officer_sar_worthy_decision
290   expression: |
291     CASE
292       WHEN activity.compliance_officer_sar_worthy THEN 'Yes'
293       ELSE 'No'
294     END
295   datatype: STRING
296   order: 0
297
298 # List of total scores for every factor's group
299 # sum action is done using coalesce to set defualt value for each enrichment if its value is null
300 - name: person_total
```

```
301     display_name: person_total
302     description: person_total
303     expression: "(COALESCE({enrichments.nationality}, 10) + COALESCE({enrichments.occupation}, 10) + COALESCE({enrichments.client_
304     type}, 10) + COALESCE({enrichments.customer_screening}, 10))"
305     datatype: DOUBLE
306     order: 10
307
308     - name: trx_total
309     display_name: trx_total
310     description: trx_total
311     expression: "(COALESCE({enrichments.total_cash_trx_amount}, 10) + COALESCE({enrichments.total_trx_amount}, 10) + COALESCE
312     ({enrichments.total_trx_count}, 10) + COALESCE({enrichments.trx_screening}, 10) + COALESCE({enrichments.tm_resolution}, 10) + COALESCE
313     ({enrichments.total_tm_alerts}, 10))"
314     datatype: DOUBLE
315     order: 10
316
317     - name: channel_total
318     display_name: channel_total
319     description: channel_total
320     expression: "COALESCE({enrichments.onboarding_channel}, 10)"
321     datatype: DOUBLE
322     order: 10
323
324     - name: products_total
325     display_name: products_total
326     description: products_total
327     expression: "COALESCE({enrichments.amount_of_products}, 10)"
328     datatype: DOUBLE
329     order: 10
```

```
327
328 - name: jurisdiction_total
329   display_name: jurisdiction_total
330   description: jurisdiction_total
331   expression: "COALESCE({enrichments.geo_origin_of_funds}, 10)"
332   datatype: DOUBLE
333   order: 10
334
335 - name: exceed_cash_trx_thrshld_cls
336   display_name: exceed_cash_trx_thrshld_cls
337   description: exceed_cash_trx_thrshld_cls
338   expression: |
339     CASE
340       WHEN {variables.exceed_cash_trx_thrshld_decision} = 'Yes' THEN 'high'
341       ELSE ''
342     END
343   datatype: STRING
344   order: 10
345   include_in_hash: True
346   annotations:
347     risk_level: True
348     sub_factor_classification: True
349     group_risk_factor: "{resources.auto_high}"
350     sub_factor: "Customer with Cash Transactions Totaling More Than USD 200,000 per Year (prev. 365 days)"
351     client_data_column: "{variables.exceed_cash_trx_thrshld_decision}"
352
353 - name: compliance_officer_sar_worthy_cls
354   display_name: compliance_officer_sar_worthy_cls
355   description: compliance_officer_sar_worthy_cls
```

```
356     expression: |
357     CASE
358         WHEN {variables.compliance_officer_sar_worthy_decision} = 'Yes' THEN 'high'
359         ELSE ''
360     END
361     datatype: STRING
362     order: 10
363     include_in_hash: True
364     annotations:
365         risk_level: True
366         sub_factor_classification: True
367         group_risk_factor: "{resources.auto_high}"
368         sub_factor: "Decision of the Compliance Committee or the Compliance Officer"
369         client_data_column: "{variables.compliance_officer_sar_worthy_decision}"
370
371     # List of total risk of every factor's group (total*weight)
372     - name: person_risk
373         display_name: person_risk
374         description: person_risk
375         expression: "{variables.person_total} / CAST({parameters.num_of_person_rf} * {parameters.global_max_risk_level} AS DOUBLE
376 PRECISION) * {parameters.person_weight} * 100"
376         datatype: LONG
377         order: 20
378         annotations:
379             group_risk_score: True
380             group_risk_factor: "{resources.person}"
381             weight_parameter: "{parameters.person_weight}"
382
383     - name: trx_risk
```

```
384     display_name: trx_risk
385     description: trx_risk
386     expression: "{variables trx_total} / CAST({parameters.num_of_trx_rf} * {parameters.global_max_risk_level} AS DOUBLE PRECISION) * {parameters trx_weight} * 100"
387     datatype: LONG
388     order: 20
389     annotations:
390       group_risk_score: True
391       group_risk_factor: "{resources trx}"
392       weight_parameter: "{parameters trx_weight}"
393
394     - name: channel_risk
395       display_name: channel_risk
396       description: channel_risk
397       expression: "{variables channel_total} / CAST({parameters.num_of_channel_rf} * {parameters.global_max_risk_level} AS DOUBLE PRECISION)* {parameters.channel_weight} * 100"
398       datatype: LONG
399       order: 20
400       annotations:
401         group_risk_score: True
402         group_risk_factor: "{resources channel}"
403         weight_parameter: "{parameters.channel_weight}"
404
405     - name: products_risk
406       display_name: products_risk
407       description: products_risk
408       expression: "{variables products_total} / CAST({parameters.num_of_products_rf} * {parameters.global_max_risk_level} AS DOUBLE PRECISION) * {parameters.products_weight} * 100"
409       datatype: LONG
```

```
410     order: 20
411     annotations:
412       group_risk_score: True
413       group_risk_factor: "{resources.products}"
414       weight_parameter: "{parameters.products_weight}"
415
416     - name: jurisdiction_risk
417       display_name: jurisdiction_risk
418       description: jurisdiction_risk
419       expression: "{variables.jurisdiction_total} / CAST({parameters.num_of_jurisdiction_rf} * {parameters.global_max_risk_level} AS
420       DOUBLE PRECISION) * {parameters.jurisdiction_weight} * 100"
421       datatype: LONG
422       order: 20
423       annotations:
424         group_risk_score: True
425         group_risk_factor: "{resources.jurisdiction}"
426         weight_parameter: "{parameters.jurisdiction_weight}"
427
428     # Total score after take into account all factor's groups risks
429     - name: score
430       display_name: score
431       description: score
432       expression: "{variables.person_risk} + {variables.trx_risk} + {variables.channel_risk} + {variables.products_risk} +
433       {variables.jurisdiction_risk}"
434       datatype: LONG
435       order: 30
436       annotations:
437         score: True
438         distribute: True
```

```
437
438 # Auto high classification
439 - name: auto_high
440   display_name: auto_high
441   description: auto_high
442   expression: |
443     CASE
444       WHEN {variables.exceed_cash_trx_thrshld_cls} = 'high' or {variables.compliance_officer_sar_worthy_cls} = 'high' THEN 'high'
445       ELSE ''
446     END
447   datatype: STRING
448   order: 30
449
450 # CRA score rule - End result score classifications after checking score with the ranges (from params section)
451 - name: score_classification
452   display_name: score_classification
453   description: score_classification
454   expression: |
455     CASE
456       WHEN {variables.score} >= {parameters.low} and {variables.score} < {parameters.medium} THEN 'low'
457       WHEN {variables.score} >= {parameters.medium} and {variables.score} < {parameters.high} THEN 'medium'
458       ELSE 'high'
459     END
460   datatype: STRING
461   order: 40
462
463 # Hash of the current activity (only with fields (dataset) or variables (risk) that are marked include_in_hash: True)
464 - name: hash
465   display_name: hash
```

```
466     description: activity hash
467     expression: "{input_hash}" # hash of new activity
468     datatype: STRING
469     order: 40
470
471 # CRA classification rule - new classification after taking into account score & high indicators
472 - name: system_cls
473     display_name: system_classification
474     description: system classification
475     expression: |
476     CASE
477         WHEN {variables.auto_high} = 'high' THEN 'high'
478         ELSE {variables.score_classification}
479     END
480     datatype: STRING
481     valid_values: ["high","medium","low"]
482     distribute: True
483     order: 50
484
485 # CRA classification taking into account reclassification
486 - name: classification
487     display_name: classification
488     description: classification
489     expression: |
490     CASE
491         WHEN {enrichments.user_cls} is not null AND {enrichments.old_class_hash} = {variables.hash}
492         THEN {enrichments.user_cls}
493         ELSE {variables.system_cls}
494     END
```

```
495     datatype: STRING
496     annotations:
497       classification: True
498     valid_values: ["high","medium","low"]
499     distribute: True
500     order: 60
501     override: True
502
503 # CRA alert rule
504 - name: is_cra_alert
505   display_name: is_cra_alert
506   description: is_cra_alert
507   expression: "{variables.classification} = 'high'"
508   datatype: BOOLEAN
509   order: 70
510
511 # Defined the alert display name based on changing in classification
512 - name: cls_change_dynamic_display
513   display_name: cls_change_dynamic_display
514   description: cls_change_dynamic_display
515   expression: |
516     CASE
517       WHEN {variables.is_cra_alert} and {enrichments.prev_cls} IS NOT NULL and {enrichments.prev_cls} != 'high' THEN 'active'
518       ELSE ''
519     END
520   datatype: STRING
521   order: 75
522
523 ### CONDITIONS ###
```

```
524 | conditions:
525 | - variable: is_cra_alert
526 |   value: "true"
```

## 12.5. CRA Reports

### 12.5.1. Generating Automatic Reports

A report is generated automatically for each CRA analysis, and can contain a single customer or the entire customer base, according to the analysis type. The report contains a detailed breakdown of the risk score, rule results, classifications and alert details.

The report is generated using `generate_cdd_analysis_report` function which is added to the relevant notebook as part of the CRA flow.

```
1 | def generate_cdd_analysis_report(context,
2 |                                     evaluation_flow_identifier: str,
3 |                                     data_environment: DataEnvironment = DataEnvironment.get_default()):
4 |
5 | -----
6 |
7 | from thetaray.common.data_environment import DataEnvironment
8 | from thetaray.api.cdd.api import generate_cdd_analysis_report
9 |
10 | generate_cdd_analysis_report(context, "tr_cdd_ef", data_environment= DataEnvironment.PUBLIC)
```

There is a python API that can / should be called from the notebook that generate the report:

```
def notify_cra_report(context, event: str, path: str, analyzed_customers_count: int, notification_identifier: str, retry_policy=RetryPolicy())
```

## 12.5.2. Report Notification API

Once the report is uploaded to a dedicated bucket, a notification is sent to a dedicated endpoint configured by the client to indicate that the report is available for retrieval via API. The notification will include the path to the relevant report.

Notification target should be configured as ExternalNotificationTarget metadata entity, in the targets folder of the solution domain.

### Example:

```
1 from thetaray.api.solution.alert_distribution_target import TargetAuth, AuthenticationMode
2 from thetaray.api.solution.external_notification_target import ExternalNotificationTarget
3
4 def ext_distribution_target():
5     return ExternalNotificationTarget(
6         identifier="cra_report_notification_target",
7         name="cra_report_notification_target",
8         url="....",
9         auth=TargetAuth(mode=AuthenticationMode.NO_AUTH),
10    )
11
12 def entities():
13     return [ext_distribution_target()]
```

### 12.5.3. Classification Changes API

The classification changes API notifies at the end of each CRA analysis of any classification changes that occurred. Any classification change is included in the API call, whether the classification changes to a lower or higher level, in order to keep the KYC system up to date. A manual reclassification from the CRA alert by the analyst will also trigger the classification change API.

The Classification Changes API notifies an external URL on a classification change for the following cases:

1. Customer onboarding
2. Classification was changed after ongoing evaluation
3. As part of the reclassification API:
  - a. For the manual reclassification - need to configure the notification target name in common.yaml.gompl

```
1 | cra:  
2 |   report_rows_limit_csv: 10000  
3 |   report_rows_limit_excel: 10000  
4 |   manual_reclassification_notification_target: classification_change_target
```

The notification target should be configured as ClassificationChangeNotificationTarget metadata entity, in the targets folder of the solution domain. This metadata entity is similar to the existing

ExternalNotificationTarget. It includes additional optional data:

1. annotations - list of annotations names to include their value in the notification request.
2. batch\_size (default 1000) - maximum size of the customers array in the request that is sent to the notification target. If there are more notified customers then this value, multiple requests will be sent.

Example:

```
1 from thetaray.api.solution.alert_distribution_target import TargetAuth, AuthenticationMode
2 from thetaray.api.solution.classification_change_notification_target import ClassificationChangeNotificationTarget
3
4 def ext_distribution_target():
5     return ClassificationChangeNotificationTarget(
6         identifier="classification_change_target",
7         name="classification_change_target",
8         url="....",
9         auth=TargetAuth(mode=AuthenticationMode.NO_AUTH),
10        annotations=["classification_type"]
11    )
12
13 def entities():
14     return [ext_distribution_target()]
```

Triggering classification changes API from the notebook (at the end of the evaluation flow process):

```
1 def notify_classification_changes(context, evaluation_flow_identifier: str, classification_change_notification_identifier: str, batch_
  size=1000, parallelism=4, retry_policy=RetryPolicy())
```

## 13. CRA Algorithm

**Disclaimer** It should be noted that the content of this sub section is draft only and therefore its contents therein is classified as non GA.

**Note:** Please note that the documentation below describes the product supported functionalities of the CRA Risk Engine. All project implementations will be provided in an additional document.

### 13.1. Overview

Customer Risk Assessment (CRA) is a ThetaRay product designed to aid financial institutions in assessing the risk levels of their customers. CRA is a fundamental process that financial institutions and other entities involved in financial transactions perform to understand their customers, assess their risk profiles, and verify their identities. It is a crucial component of anti-money laundering (AML) and counter-terrorist financing (CTF) efforts, and it complements the AML solution by helping to prevent financial crimes and ensure regulatory compliance.

The solution is designed to comply with regulatory requirements and industry standards, ensuring that the Customer Risk Assessment process is robust, transparent, and aligned with organizational goals. The ThetaRay CRA product provides a solution for analyzing customers one by one, as well as continuously monitoring the entire customer database at a selected cadence.

The CRA product receives customer data (single customer data or a batch of customer data) and runs the risk rating engine, which is enhanced with additional information, such as the relevant high risk countries. The risk rating engine incorporates deterministic rules of varying categories and logic, as well as a proprietary risk classification algorithms.

In addition to the creation of a dedicated CRA case in the Investigation Center, the results of the risk rating engine are externalized via API, and are generated into a report. The report is compiled for all the customers classified in the CRA analysis and is available for retrieval via API and through the CRA On Demand Reporting Module.

## 13.2. Evaluation Flow

The following fields have been added or enhanced in the Evaluation Flow Metadata in order to support the CRA Algorithm flavor:

- ef\_type - an additional type has been added called "CRA\_ML"
- evaluation\_steps - two evaluation steps need to be configured, one for each classification type (AML and Sanctions), of type SemiSupervisedClassificationEvaluationStep
- SemiSupervisedClassificationEvaluationStep - is an evaluation step that uses the semi-supervised classifier model and gives as an output scores for each of the evaluation classes (labels) defined and ratings for the features that were used in train in range from 100 to 10000. Please refer to the model documentation for more details. All of the attributes as shown in the example below should be present in the final implementation and should not be changed.
- label\_selector - is a mandatory parameter that serves as a function to enhance the evaluation process with additional logic. This logic is used to select the final evaluation class and retrieve the correct feature rating. In the reference example the logic compares the scores and selects the classification class with a higher priority if the score difference between classes is less than 5%.
- alert\_conf - an attribute that has been added. The supported parameters are:
  - reclassification\_period - number of days that the manual overwrite is valid for

### Evaluation Flow Metadata Example

```
1 from typing import List
2 from thetaray.api.solution.evaluation import (
3     EvaluationFlow,
4     EvaluationType,
5     ModelReference,
6     SemiSupervisedClassificationEvaluationStep,
7     AlertConfiguration,
```

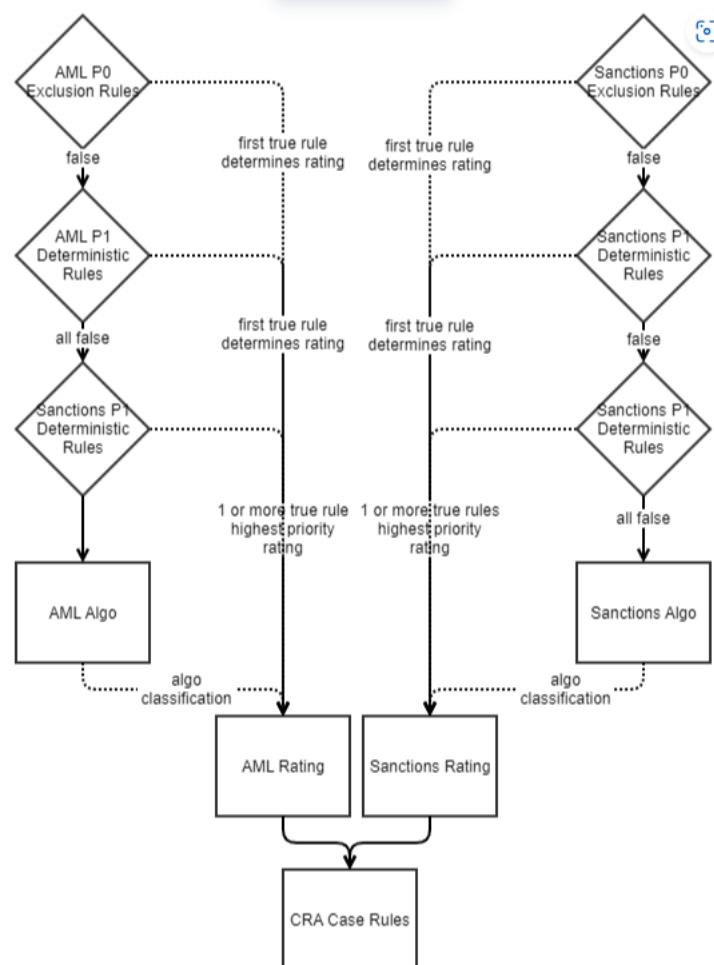
```
8  )
9  def evaluation_flow() -> EvaluationFlow:
10     def labels_selection_logic(scores_array) -> int:
11         score_l, score_m, score_h, score_vh = scores_array
12         if score_vh > max(score_h - score_vh * 0.05, score_m - score_vh * 0.05, score_l - score_vh * 0.05):
13             return 4
14         elif score_h > max(score_vh, score_m - score_h * 0.05, score_l - score_h * 0.05):
15             return 3
16         elif score_m > max(score_vh, score_h, score_l - score_m * 0.05):
17             return 2
18         else:
19             return 1
20     return EvaluationFlow(
21         identifier="tr_cra_ml",
22         displayName="CRA ML Analysis",
23         input_dataset="cra_ml_customer",
24         data_permission="dpv:public",
25         ef_type=EvaluationType.CRA_ML,
26         evaluation_steps=[
27             SemiSupervisedClassificationEvaluationStep(
28                 identifier="aml_ml",
29                 name="AML evaluation",
30                 feature_extraction_model=ModelReference(
31                     name="aml_feature_extraction_model",
32                     tags={"version": "release"},
33                 ),
34                 detection_model=ModelReference(name="aml_detection_model", tags={"version": "release"}),
35                 labels={
36                     1: "L",
```

```
37             2: "M",
38             3: "H",
39             4: "VH",
40         },
41         label_selector=labels_selection_logic,
42     ),
43     SemiSupervisedClassificationEvaluationStep(
44         identifier="scrn_ml",
45         name="Screening evaluation",
46         feature_extraction_model=ModelReference(
47             name="scrn_feature_extraction_model",
48             tags={"version": "release"},
49         ),
50         detection_model=ModelReference(name="scrn_detection_model", tags={"version": "release"}),
51         labels={
52             1: "L",
53             2: "M",
54             3: "H",
55             4: "VH",
56         },
57         label_selector=labels_selection_logic,
58     ),
59 ],
60 max_workers=5,
61 alert_conf=AlertConfiguration(
62     reclassification_period=365 # days
63 ),
64 )
65 def entities() -> List[EvaluationFlow]:
```

```
66 |     return [evaluation_flow()]
```

### 13.3. Risk Engine

#### Risk Engine Flow



### 13.3.1. Classification Types

Each classification type - AML and Sanctions - needs its own variable within the risk YAML

```
1  - name: aml_rules_classification
2    display_name: "AML Rules Classification"
3    description: "AML Rules Classification"
4    expression: "{{ classify_by_priority('aml') }}"
5    datatype: STRING
6    order: 1
7    annotations:
8      rules_group: aml
9      group_classification: true
10     classification_logic: prioritized_rules
11     classification_column: "{variablesaml_effective_classification}"
12     priority: 1
13 - name: san_rules_classification
14   display_name: 'Sanctions Rules Classification'
15   description: 'Sanctions Rules Classification'
16   expression: |
17     {{ classify_by_priority('san') }}
18   datatype: STRING
19   order: 1
20   annotations:
21     rules_group: 'san'
22     group_classification: true
23     classification_logic: prioritized_rules
24     classification_column: '{variables.san_effective_classification}'
25     priority: 1
```

## 13.4. Deterministic Rules

**Note:** All rules described below should be set up for each classification type - AML and Sanctions. Each rule for each classification type needs to have its own variable within the risk YAML.

### 13.4.1. P0 Exclusion Rules

The rules in this category are prioritized by importance, where the first true rule will set the classification and "ignore" the rules that succeed it. The priority parameter in annotations should be used in order to determine the importance of the rule, meaning which one will set the classification first if it is found true.

#### Example

```
1  - name: customer_is_funds
2    display_name: "Customer is Funds"
3    description: "Customer is Funds"
4    output: NC
5    condition: "classification_type = 'Funds'"
6    datatype: STRING
7    order: 0
8    annotations:
9      priority: 1
10     classification_rule: True
11     classification_column: "{variablesaml_effective_classification}"
12     rules_group: aml
```

### 13.4.2. P1 Deterministic Rules

The rules in this category run together, and multiple ones can be found to be true. In such a case where more than one rule is true, the classifications themselves are prioritized, and the highest classification is set.

The priority parameter in annotations should be set to the same value for all of the rules in this category.

```
1 | - name: risky_nationality
2 |   display_name: "Risky Nationality"
3 |   description: "Nationality of the customer is risky"
4 |   output: VH
5 |   condition: "nationality in ('IR', 'KP', 'SY', 'SD', 'YE', 'VE')"
6 |   datatype: STRING
7 |   order: 0
8 |   annotations:
9 |     priority: 100
10 |    classification_rule: True
11 |    classification_column: "{variablesaml_effective_classification}"
12 |    rules_group: aml
```

A new templating feature has been introduced: {{ classify\_by\_priority('group') }} function.

This function was created in order to automatically generate SQL expressions to select the relevant classification amongst the classification rules (P1 Deterministic Rules) defined in risk YAML.

#### Usage example:

```
1 | - name: aml_rules_classification
```

```
2 display_name: "AML Rules Classification"
3 description: "AML Rules Classification"
4 expression: "{{ classify_by_priority('aml') }}"
5 datatype: STRING
6 order: 1
7 annotations:
8   rules_group: aml
9   group_classification: true
10  classification_logic: prioritized_rules
11  classification_column: "{variablesaml_classification}"
12  priority: 1
```

The function goes over all **active** risk variables that have same rules\_group annotation as provided in the function argument and have classification\_rule annotation as true. The function then builds SQL to select the first not NULL value from the risk variables defined as classification rules sorted by priority set in the priority annotation. If the priority is the same it is sorted by the output attribute value by classification priority from highest to lowest.

As an example we have three variables:

```
1 - name: var1
2   output: VH
3   condition: "some condition"
4   datatype: STRING
5   order: 0
6   annotations:
7     priority: 1
8     classification_rule: True
9     classification_column: "{variablesaml_classification}"
```

```
10    rules_group: aml
11  - name: var2
12    output: L
13    condition: "some condition"
14    datatype: STRING
15    order: 0
16    annotations:
17      priority: 2
18      classification_rul
19      e: True
20      classification_column: "{variablesaml_classification}"
21      rules_group: aml
22  - name: var3
23    output: H
24    condition: "some condition"
25    datatype: STRING
26    order: 0
27    annotations:
28      priority: 2
29      classification_rule: True
30      classification_column: "{variablesaml_classification}"
31      rules_group: aml
```

For a setup like the one described above, the generated expression will be:

```
1 | COALESCE(var1, var3, var2, NULL)
```

This function then uses values\_properties - a mapping that allows to enhance the possible classification values with it's priority and display name.

For example:

```
1 values_properties:
2     L:
3         display_name: Low
4         priority: 1
5     M:
6         display_name: Medium
7         priority: 2
8     H:
9         display_name: High
10        priority: 3
11    VH:
12        display_name: Very High
13        priority: 4
14    PRB:
15        display_name: Prohibited
16        priority: 5
17    NC:
18        display_name: Non Client
19        priority: 6
20    TPI:
21        display_name: Third Party Introduction
22        priority: 6
23    GE:
24        display_name: Group Entity
25        priority: 6
```

### 13.4.3. Algorithm Classification

If all rules from all categories were found to be false, then the classification from the relevant group model should be selected.

```
1  - name: aml_ml_classification
2      display_name: "AML ML Classification"
3      description: "AML ML Classification"
4      expression: aml_ml_classification
5      datatype: STRING
6      order: 1
7      annotations:
8          group_classification: true
9          rules_group: aml
10         classification_logic: ml
11         evaluation_step: aml_ml
12         classification_column: "{variablesaml_effective_classification}"
13         priority: 2
14 - name: aml_ml_score
15     display_name: "AML"
16     description: "AML ML Score"
17     expression: |
18     CASE
19         WHEN {variablesaml_rules_classification} IS NOT NULL THEN 100
20         WHEN aml_ml_classification = 'VH' THEN aml_ml_vh_score * 100
21         WHEN aml_ml_classification = 'H' THEN aml_ml_h_score * 100
22         WHEN aml_ml_classification = 'M' THEN aml_ml_m_score * 100
23         WHEN aml_ml_classification = 'L' THEN aml_ml_l_score * 100
24         ELSE 0
```

```
25      END
26      datatype: DOUBLE
27      order: 2
28      annotations:
29          classification_score: true
30          classification_column: "{variablesaml_effective_classification}"
```

#### 13.4.4. System & Effective Classifications

Whenever there is a Risk Engine run the output will be the system classifications for each counterparty analyzed. The system classifications can be determined by either the Deterministic Rules or the AI Model, and each classification type can potentially get a different value. The system classification will be audited in the automatic reports.

The effective classification refers to the final classification that takes into consideration the system classification and the results of the manual reclassification (when an analyst manually overwrites the system classification for a given counterparty). The valid manual reclassification can be set as the effective classification, with additional conditions if required.

```
1 - name: aml_effective_classification
2   display_name: 'Effective_AML_Classification'
3   description: 'Effective AML Classification'
4   expression: |
5     case
6       when
7         {variablesaml_system_classification} <> 'PRB'
8         and
9         {enrichmentsaml_user_classification} is not null
10      then
```

```
11         {enrichmentsaml_user_classification}
12     else
13         {variablesaml_system_classification}
14     end
15     datatype: STRING
16     override: true
17     order: 3
18     valid_values:
19         - L
20         - M
21         - H
22         - VH
23         - PRB
24         - NR
25         - TPI
26         - GE
27     values_properties:
28         L:
29             display_name: Low
30             priority: 1
31         M:
32             display_name: Medium
33             priority: 2
34         H:
35             display_name: High
36             priority: 3
37         VH:
38             display_name: Very High
39             priority: 4
```

```
40     PRB:  
41         display_name: Prohibited  
42         priority: 5  
43     NR:  
44         display_name: Not Rated  
45         priority: 6  
46     TPI:  
47         display_name: Third Party Introduction  
48         priority: 6  
49     GE:  
50         display_name: Group Entity  
51         priority: 6  
52     annotations:  
53         classification: True
```

### 13.4.5. CRA Case Rules

The risk engine supports the configuration of rules will ultimately decide whether to generate a CRA case (FS Opinion Request) in the Investigation Center or not, and can optionally be based on the classifications that were outputted from the engine. Additional deterministic rules can be configured at this stage to take any inherent risks into consideration.

If any of the rules were found to be true, they will be indicated as true in the report view in the On Demand Report Module, in the automatic report and in the OOB Simulation report.

In order to set the variable as an FS Opinion Request rule, the annotation to be used is case\_trigger\_rule: true

```
1 | - name: aml_classification_is_high
```

```
2     display_name: "AML Classification is High"
3     description: "AML Classification is High"
4     expression: |
5         CASE
6             WHEN {variablesaml_effective_classification} in ('H', 'VH', 'PRB') THEN true
7             ELSE false
8         END
9     datatype: BOOLEAN
10    order: 4
11    annotations:
12        case_trigger_rule: true
13    - name: other_trigger_rule
14        display_name: "Other Trigger Rule"
15        description: "Other Trigger Rule"
16        expression: |
17            CASE
18                WHEN classification_type = 'Risky' THEN true
19                ELSE false
20            END
21        datatype: BOOLEAN
22        annotations:
23            case_trigger_rule: true
```

#### 13.4.6. Risk Metadata

New attributes for **Risk Variable** are introduced:

1. condition in combination with output. It can be used instead of expressions in order to automatically generate conditional SQL expression like this:

```
1 | CASE
2 |   WHEN {condition} THEN {output}
3 |   ELSE NULL
4 | END as {variable_name}
```

#### Example of risk variable yaml

```
1 | - name: risky_source_of_funds
2 |   display_name: "Risky Source of Funds"
3 |   description: "Source of funds of the customer is risky"
4 |   output: VH
5 |   condition: "source_of_funds = 'Risky'"
6 |   datatype: STRING
```

2. order - used to handle variables which uses another variables in it's expression. For example

```
1 | - name: var1
2 |   output: {some_value}
3 |   condition: "source_of_funds = 'Risky'"
4 |   datatype: BIGINT
5 |   order: 0
6 | - name: var2
7 |   expression: "{variables.var1}"
8 |   datatype: BIGINT
9 |   order:1
```

3. annotations - set of key value mapping that should be used according to the table found in the "annotations" section, in order to acknowledge decisioning, IC and reporting service about the role of the variable. For example:

```
1 | - name: customer_is_funds
2 |   display_name: "Customer is Funds"
3 |   description: "Customer is Funds"
4 |   output: NR
5 |   condition: "classification_type = 'Funds'"
6 |   datatype: STRING
7 |   order: 0
8 |   annotations:
9 |     priority: 1
10 |    classification_rule: True
11 |    classification_column: "{variablesaml_classification}"
12 |    rules_group: aml
```

4. override - indicates if the risk variable should be included in the activity\_override table. This should be used only for classification variables.

---

**Note:** This attribute should always be set to "True" to allow for manual reclassification from the On Demand Reporting module.

---

5. include\_in\_report - if risk variable value should be included to report.

New attributes for **Risk Enrichment**:

1. output.annotations - set of key value mapping that should be used according to the table found in the "annotations" section, in order to acknowledge decisioning, IC and reporting service about the role of the variable
2. output.valid\_values - list of possible values
3. output.include\_in\_report - indicates if the risk variable value should be included in the automatic report

## Full Risk YAML Example

```
1 identifier: cra_ml_risk
2 display_name: CRA ML Risk
3 description: CRA ML Risk
4 category: cra_ml
5 severity: 1
6 # dynamic_display_name:
7 # dynamic_description:
8 # dynamic_category:
9 # dynamic_severity:
10 # dynamic_suppression_identifier:
11 suppression_period_unit: DAY
12 suppression_period_size: -1
13 evaluation_flows:
14   - tr_cra_ml_ef
15 parameters:
16   - name: parameter
17     datatype: STRING
18     value: 'parameter'
19 enrichments:
20   # ----- Enrichments ----- #
21   - name: customer_id
22     sql: |
23       select activity.customer_id as customer_id
24     outputs:
25       - name: customer_id
26         datatype: STRING
27   - name: customer_type_id_to_str
28     sql: |
```

```
29      select description as customer_description
30          from cra_ml_customer_types
31          where activity.type_id=type_id
32    outputs:
33      - name: customer_description
34          display_name: "Customer Type"
35          datatype: STRING
36          annotations:
37              kyc: true
38              kyc.order: 12
39      - name: user_classification
40          sql: |
41              select
42                  rv_cra_ml_risk_aml_effective_classification as aml_user_classification,
43                  rv_cra_ml_risk_san_effective_classification as san_user_classification
44              from
45                  activity_ovrd_tr_cra_ml_ef
46              where
47                  customer_id = activity.customer_id
48                  and
49                  effective_date >= {context.execution_date}
50              order by
51                  created_on desc
52    outputs:
53      - name: aml_user_classification
54          datatype: STRING
55      - name: san_user_classification
56          datatype: STRING
57  # ----- Enrichments P1 ----- #
```

```
58  - name: p1_rule_1_triggered
59    sql: |
60      select
61        activity.cpy_managed_by_am = true
62        and
63        activity.cpy_category like 'MCG%'
64        and
65        activity.execution = 'Y'
66      as p1_rule_1_triggered
67    outputs:
68      - name: p1_rule_1_triggered
69        datatype: BOOLEAN
70  - name: p1_rule_2_triggered
71    sql: |
72      select activity.relation_establishment = 'third party introduction'
73      as p1_rule_2_triggered
74    outputs:
75      - name: p1_rule_2_triggered
76        datatype: BOOLEAN
77  - name: p1_rule_3_triggered
78    sql: |
79      select activity.cpy_category like 'PCG%'
80      as p1_rule_3_triggered
81    outputs:
82      - name: p1_rule_3_triggered
83        datatype: BOOLEAN
84  # ----- Enrichments P2 -----
85  - name: p2_rule_1_triggered
86    sql: |
```

```
87     select exists (
88         select *
89         from
90             cra_ml_do_not_enter
91         where
92             country_code in (
93                 activity.nationality_incorp_cntry,
94                 activity.residence_cntry,
95                 activity.cntry_of_mailing_address,
96                 activity.cntry_of_tax_address
97             )
98             and
99             tr_timestamp <= {context.execution_date}
100        ) as p2_rule_1_triggered
101    outputs:
102        - name: p2_rule_1_triggered
103            datatype: BOOLEAN
104    - name: p2_rule_2_triggered
105        sql: |
106            select activity.type_id = 3
107            as p2_rule_2_triggered
108        outputs:
109            - name: p2_rule_2_triggered
110                datatype: BOOLEAN
111    - name: p2_rule_3_triggered
112        sql: |
113            select exists (
114                select *
115                from
```

```
116         cra_ml_black_list
117         where
118             country_code in (
119                 activity.nationality_incorp_cntry,
120                 activity.residence_cntry,
121                 activity.cntry_of_mailing_address,
122                 activity.cntry_of_tax_address
123             )
124             and
125                 tr_timestamp <= {context.execution_date}
126         ) as p2_rule_3_triggered
127         outputs:
128             - name: p2_rule_3_triggered
129                 datatype: BOOLEAN
130         - name: p2_rule_4_triggered
131             sql: |
132                 select activity.automatic_class = 'L'
133                 as p2_rule_4_triggered
134             outputs:
135                 - name: p2_rule_4_triggered
136                     datatype: BOOLEAN
137             - name: p2_rule_5_triggered
138                 sql: |
139                     select activity.automatic_class = 'M'
140                     as p2_rule_5_triggered
141             outputs:
142                 - name: p2_rule_5_triggered
143                     datatype: BOOLEAN
144             - name: p2_rule_6_triggered
```

```
145     sql: |
146         select activity.automatic_class = 'H'
147         as p2_rule_6_triggered
148     outputs:
149         - name: p2_rule_6_triggered
150             datatype: BOOLEAN
151     variables:
152         # ----- Variables ----- #
153         - name: analysis_type
154             display_name: 'Analysis Type'
155             description: 'Analysis Type'
156             expression: |
157                 (
158                     select
159                         case
160                             when analysis_type = 'OB' then 'Onboarding'
161                             when analysis_type = 'OG' then 'Ongoing'
162                         end
163                     from
164                         customer_analysis_type
165                     where
166                         customer_id = activity.customer_id
167                         and
168                         tr_job_ts <= {context.execution_date}
169                     order by
170                         tr_job_ts desc
171                     limit 1
172                 )
173             datatype: STRING
```

```
174     valid_values: ['Onboarding', 'Ongoing']
175     annotations:
176       tag: analysisType
177     # ----- Variables P1 ----- #
178     - name: p1_rule_1_aml
179       display_name: p1_rule_1_aml
180       description: p1_rule_1_aml
181       condition: |
182         {enrichments.p1_rule_1_triggered}
183       output: 'NR'
184       datatype: STRING
185       annotations:
186         rules_group: 'aml'
187         classification_rule: True
188         classification_column: '{variables.aml_effective_classification}'
189         report_prefix: 'aml'
190         priority: 1
191     - name: p1_rule_1_san
192       display_name: p1_rule_1_san
193       description: p1_rule_1_san
194       condition: |
195         {enrichments.p1_rule_1_triggered}
196       output: 'NR'
197       datatype: STRING
198       annotations:
199         rules_group: 'san'
200         classification_rule: True
201         classification_column: '{variables.san_effective_classification}'
202         report_prefix: 'san'
```

```
203     priority: 1
204     - name: p1_rule_2_am1
205       display_name: p1_rule_2_am1
206       description: p1_rule_2_am1
207       condition: |
208         {enrichments.p1_rule_2_triggered}
209       output: 'TPI'
210       datatype: STRING
211       annotations:
212         rules_group: 'am1'
213         classification_rule: True
214         classification_column: '{variables.am1_effective_classification}'
215         report_prefix: 'am1'
216         priority: 2
217     - name: p1_rule_2_san
218       display_name: p1_rule_2_san
219       description: p1_rule_2_san
220       condition: |
221         {enrichments.p1_rule_2_triggered}
222       output: 'TPI'
223       datatype: STRING
224       annotations:
225         rules_group: 'san'
226         classification_rule: True
227         classification_column: '{variables.san_effective_classification}'
228         report_prefix: 'san'
229         priority: 2
230     - name: p1_rule_3_am1
231       display_name: p1_rule_3_am1
```

```
232     description: p1_rule_3_aml
233     condition: |
234       {enrichments.p1_rule_3_triggered}
235     output: 'GE'
236     datatype: STRING
237     annotations:
238       rules_group: 'aml'
239       classification_rule: True
240       classification_column: '{variablesaml_effective_classification}'
241       report_prefix: 'aml'
242       priority: 3
243     - name: p1_rule_3_san
244       display_name: p1_rule_3_san
245       description: p1_rule_3_san
246       condition: |
247         {enrichments.p1_rule_3_triggered}
248       output: 'GE'
249       datatype: STRING
250       annotations:
251         rules_group: 'san'
252         classification_rule: True
253         classification_column: '{variables.san_effective_classification}'
254         report_prefix: 'san'
255         priority: 3
256     # ----- Variables P2 ----- #
257     - name: p2_rule_1_san
258       display_name: 'Do Not Enter'
259       description: 'Do Not Enter'
260       condition: |
```

```
261     {enrichments.p2_rule_1_triggered}
262     output: 'VH'
263     datatype: STRING
264     annotations:
265       rules_group: 'san'
266       classification_rule: True
267       classification_column: '{variables.san_effective_classification}'
268       report_prefix: 'san'
269       priority: 10
270   - name: p2_rule_2_aml
271     display_name: 'Natural Person'
272     description: 'Natural Person'
273     condition: |
274       {enrichments.p2_rule_2_triggered}
275     output: 'H'
276     datatype: STRING
277     annotations:
278       rules_group: 'aml'
279       classification_rule: True
280       classification_column: '{variables.aml_effective_classification}'
281       report_prefix: 'aml'
282       priority: 10
283   - name: p2_rule_2_san
284     display_name: 'Natural Person'
285     description: 'Natural Person'
286     condition: |
287       {enrichments.p2_rule_2_triggered}
288     output: 'H'
289     datatype: STRING
```

```
290     annotations:
291         rules_group: 'san'
292         classification_rule: True
293         classification_column: '{variables.san_effective_classification}'
294         report_prefix: 'san'
295         priority: 10
296     - name: p2_rule_3_aml
297         display_name: 'Blacklist'
298         description: 'Blacklist'
299         condition: |
300             {enrichments.p2_rule_3_triggered}
301         output: 'PRB'
302         datatype: STRING
303         annotations:
304             rules_group: 'aml'
305             classification_rule: True
306             classification_column: '{variables.aml_effective_classification}'
307             report_prefix: 'aml'
308             priority: 10
309     - name: p2_rule_3_san
310         display_name: 'Blacklist'
311         description: 'Blacklist'
312         condition: |
313             {enrichments.p2_rule_3_triggered}
314         output: 'PRB'
315         datatype: STRING
316         annotations:
317             rules_group: 'san'
318             classification_rule: True
```

```
319     classification_column: '{variables.san_effective_classification}'
320     report_prefix: 'san'
321     priority: 10
322 - name: p2_rule_4_aml
323     display_name: 'Automatic Low'
324     description: 'Automatic Low'
325     condition: |
326       {enrichments.p2_rule_4_triggered}
327     output: 'L'
328     datatype: STRING
329     annotations:
330       rules_group: 'aml'
331       classification_rule: True
332       classification_column: '{variables.aml_effective_classification}'
333       report_prefix: 'aml'
334       priority: 10
335 - name: p2_rule_4_san
336     display_name: 'Automatic Low'
337     description: 'Automatic Low'
338     condition: |
339       {enrichments.p2_rule_4_triggered}
340     output: 'L'
341     datatype: STRING
342     annotations:
343       rules_group: 'san'
344       classification_rule: True
345       classification_column: '{variables.san_effective_classification}'
346       report_prefix: 'san'
347       priority: 10
```

```
348 - name: p2_rule_5_aml
349   display_name: 'Automatic Medium'
350   description: 'Automatic Medium'
351   condition: |
352     {enrichments.p2_rule_5_triggered}
353   output: 'M'
354   datatype: STRING
355   annotations:
356     rules_group: 'aml'
357     classification_rule: True
358     classification_column: '{variables.aml_effective_classification}'
359     report_prefix: 'aml'
360     priority: 10
361 - name: p2_rule_5_san
362   display_name: 'Automatic Medium'
363   description: 'Automatic Medium'
364   condition: |
365     {enrichments.p2_rule_5_triggered}
366   output: 'M'
367   datatype: STRING
368   annotations:
369     rules_group: 'san'
370     classification_rule: True
371     classification_column: '{variables.san_effective_classification}'
372     report_prefix: 'san'
373     priority: 10
374 - name: p2_rule_6_aml
375   display_name: 'Automatic High'
376   description: 'Automatic High'
```

```
377     condition: |
378         {enrichments.p2_rule_6_triggered}
379     output: 'H'
380     datatype: STRING
381     annotations:
382         rules_group: 'aml'
383         classification_rule: True
384         classification_column: '{variables.aml_effective_classification}'
385         report_prefix: 'aml'
386         priority: 10
387     - name: p2_rule_6_san
388         display_name: 'Automatic High'
389         description: 'Automatic High'
390         condition: |
391             {enrichments.p2_rule_6_triggered}
392         output: 'H'
393         datatype: STRING
394         annotations:
395             rules_group: 'san'
396             classification_rule: True
397             classification_column: '{variables.san_effective_classification}'
398             report_prefix: 'san'
399             priority: 10
400 # ----- Rules Classification ----- #
401     - name: aml_rules_classification
402         display_name: 'AML Rules Classification'
403         description: 'AML Rules Classification'
404         expression: |
405             {{ classify_by_priority('aml') }}
```

```
406     datatype: STRING
407     order: 1
408     annotations:
409       rules_group: 'aml'
410       group_classification: true
411       classification_logic: prioritized_rules
412       classification_column: '{variablesaml_effective_classification}'
413       priority: 1
414     - name: san_rules_classification
415       display_name: 'Sanctions Rules Classification'
416       description: 'Sanctions Rules Classification'
417       expression: |
418         {{ classify_by_priority('san') }}
419     datatype: STRING
420     order: 1
421     annotations:
422       rules_group: 'san'
423       group_classification: true
424       classification_logic: prioritized_rules
425       classification_column: '{variables.san_effective_classification}'
426       priority: 1
427     # ----- ML Classification -----
428     - name: aml_ml_classification
429       display_name: 'AML ML Classification'
430       description: 'AML ML Classification'
431       expression: |
432         activityaml_ml_classification
433     datatype: STRING
434     order: 2
```

```
435     annotations:
436         rules_group: 'aml'
437         group_classification: true
438         classification_logic: ml
439         evaluation_step: aml_ml
440         classification_column: '{variables.aml_effective_classification}'
441         priority: 2
442     - name: san_ml_classification
443         display_name: 'Sanctions ML Classification'
444         description: 'Sanctions ML Classification'
445         datatype: STRING
446         expression: |
447             activity.scrn_ml_classification
448         order: 2
449     annotations:
450         rules_group: 'san'
451         group_classification: true
452         classification_logic: ml
453         evaluation_step: scrn_ml
454         classification_column: '{variables.san_effective_classification}'
455         priority: 2
456     # ----- Result Classification ----- #
457     - name: aml_system_classification
458         display_name: 'AML Classification'
459         description: 'AML Classification'
460         expression: |
461             coalesce(
462                 {variables.aml_rules_classification},
463                 {variables.aml_ml_classification}
```

```
464      )
465      datatype: STRING
466      order: 3
467      annotations:
468        system_classification: true
469        classification_column: '{variables.aml_effective_classification}'
470        report_prefix: 'aml'
471      values_properties: &aml_classification_values_properties
472      L:
473        display_name: Low
474        priority: 1
475      M:
476        display_name: Medium
477        priority: 2
478      H:
479        display_name: High
480        priority: 3
481      VH:
482        display_name: Very High
483        priority: 4
484      PRB:
485        display_name: Prohibited
486        priority: 5
487      NR:
488        display_name: Not Rated
489        priority: 6
490      TPI:
491        display_name: Third Party Introduction
492        priority: 6
```

```
493     GE:
494         display_name: Group Entity
495         priority: 6
496     - name: san_system_classification
497         display_name: 'Sanctions Classification'
498         description: 'Sanctions Classification'
499         expression: |
500             coalesce(
501                 {variables.san_rules_classification},
502                 {variables.san_ml_classification}
503             )
504         datatype: STRING
505         order: 3
506         annotations:
507             system_classification: true
508             classification_column: '{variables.san_effective_classification}'
509             report_prefix: 'san'
510         values_properties: &san_classification_values_properties
511         L:
512             display_name: Low
513             priority: 1
514         M:
515             display_name: Medium
516             priority: 2
517         H:
518             display_name: High
519             priority: 3
520         VH:
521             display_name: Very High
```

```
522     priority: 4
523     PRB:
524         display_name: Prohibited
525         priority: 5
526     NR:
527         display_name: Not Rated
528         priority: 6
529     TPI:
530         display_name: Third Party Introduction
531         priority: 6
532     GE:
533         display_name: Group Entity
534         priority: 6
535 # ----- ML Scores -----
536 - name: aml_ml_score
537     display_name: 'AML'
538     description: 'AML ML Score'
539     expression: |
540     case
541         when {variablesaml_rules_classification} is not NULL then 100
542         when {variablesaml_ml_classification} = 'VH' then activityaml_ml_vh_score * 100
543         when {variablesaml_ml_classification} = 'H' then activityaml_ml_h_score * 100
544         when {variablesaml_ml_classification} = 'M' then activityaml_ml_m_score * 100
545         when {variablesaml_ml_classification} = 'L' then activityaml_ml_l_score * 100
546         else 0
547     end
548     datatype: DOUBLE
549     order: 4
550     annotations:
```

```
551     classification_score: true
552     classification_column: '{variables.aml_effective_classification}'
553     report_prefix: 'aml'
554 - name: san_ml_score
555     display_name: 'SAN'
556     description: 'Sanctions ML Score'
557     expression: |
558     case
559         when {variables.san_rules_classification} is not NULL then 100
560         when {variables.san_ml_classification} = 'VH' then activity.scrn_ml_vh_score * 100
561         when {variables.san_ml_classification} = 'H' then activity.scrn_ml_h_score * 100
562         when {variables.san_ml_classification} = 'M' then activity.scrn_ml_m_score * 100
563         when {variables.san_ml_classification} = 'L' then activity.scrn_ml_l_score * 100
564         else 0
565     end
566     datatype: DOUBLE
567     order: 4
568     annotations:
569         classification_score: true
570         classification_column: '{variables.san_effective_classification}'
571         report_prefix: 'san'
572 # ----- Effective Classifications ----- #
573 - name: aml_effective_classification
574     display_name: 'AML'
575     description: 'AML'
576     expression: |
577     case
578         when
579             {variables.aml_system_classification} <> 'PRB'
```

```
580         and
581         {enrichmentsaml_user_classification} is not null
582     then
583         {enrichmentsaml_user_classification}
584     else
585         {variablesaml_system_classification}
586     end
587     datatype: STRING
588     override: true
589     order: 40
590     valid_values:
591         - L
592         - M
593         - H
594         - VH
595         - PRB
596         - NR
597         - TPI
598         - GE
599     values_properties: *aml_classification_values_properties
600     annotations:
601         classification: True
602         report_prefix: 'aml'
603     - name: san_effective_classification
604         display_name: 'Sanctions'
605         description: 'Sanctions'
606         expression: |
607             case
608                 when
```

```
609         {variables.san_system_classification} <> 'PRB'
610         and
611         {enrichments.san_user_classification} is not null
612         then
613             {enrichments.san_user_classification}
614         else
615             {variables.san_system_classification}
616         end
617     datatype: STRING
618     override: true
619     order: 40
620     valid_values:
621         - L
622         - M
623         - H
624         - VH
625         - PRB
626         - NR
627         - TPI
628         - GE
629     values_properties: *san_classification_values_properties
630     annotations:
631         classification: True
632         report_prefix: 'san'
633     # ----- Variables Case Trigger ----- #
634     - name: aml_classification_is_risky
635     display_name: 'AML Classification is Risky'
636     description: 'AML Classification is Risky'
637     expression: |
```

```
638     {variablesaml_effective_classification} in ('H', 'VH', 'PRB')
639     datatype: BOOLEAN
640     order: 50
641     annotations:
642       case_trigger_rule: true
643     - name: san_classification_is_risky
644       display_name: 'Sanctions Classification is Risky'
645       description: 'Sanctions Classification is Risky'
646       expression: |
647         {variablesan_effective_classification} in ('H', 'VH', 'PRB')
648       datatype: BOOLEAN
649       order: 50
650       annotations:
651         case_trigger_rule: true
652     - name: customer_is_pep
653       display_name: 'Customer is PEP'
654       description: 'Customer is PEP'
655       expression: |
656         activity.pep = true
657       datatype: BOOLEAN
658       order: 50
659       annotations:
660         case_trigger_rule: true
661     - name: nationality_incorp_cntry_fatf
662       display_name: 'Nationality Incorporation Country in FATF'
663       description: 'Nationality Incorporation Country in FATF'
664       expression: |
665         exists (
666           select *
```

```
667         from
668         cra_ml_fatf_grey_black_list
669         where
670             country_code = activity.nationality_incorp_cntry
671             and
672                 tr_timestamp <= {context.execution_date}
673         )
674     datatype: BOOLEAN
675     order: 50
676     annotations:
677         case_trigger_rule: true
678     - name: nationality_incorp_cntry_eu
679         display_name: 'Nationality Incorporation Country in EU'
680         description: 'Nationality Incorporation Country in EU'
681         expression: |
682             exists (
683                 select *
684                 from
685                     cra_ml_eu_no_cooperate
686                 where
687                     country_code = activity.nationality_incorp_cntry
688                     and
689                     tr_timestamp <= {context.execution_date}
690             )
691     datatype: BOOLEAN
692     order: 50
693     annotations:
694         case_trigger_rule: true
695     # ----- Result Case Trigger ----- #
```

```
696  - name: aml_case_trigger
697    display_name: 'AML Case Trigger'
698    description: 'AML Case Trigger'
699    expression: |
700      {variablesaml_classification_is_risky}
701      or
702      {variables.customer_is_pep}
703      or
704      {variables.nationality_incorp_cntry_fatf}
705      or
706      {variables.nationality_incorp_cntry_eu}
707    datatype: BOOLEAN
708    order: 100
709    annotations:
710      case_trigger: true
711  - name: san_case_trigger
712    display_name: 'Sanctions Case Trigger'
713    description: 'Sanctions Case Trigger'
714    expression: |
715      {variables.san_classification_is_risky}
716      or
717      {variables.customer_is_pep}
718      or
719      {variables.nationality_incorp_cntry_fatf}
720      or
721      {variables.nationality_incorp_cntry_eu}
722    datatype: BOOLEAN
723    order: 100
724    annotations:
```

```
725     case_trigger: true
726 conditions:
727 # ----- Conditions -----
728 - variable: aml_case_trigger
729   value: 'true'
730   display_settings:
731     - type: Threshold
732       parameter: '{parameters.parameter}'
733       feature: '{activity.customer_id}'
734 - variable: san_case_trigger
735   value: 'true'
736   display_settings:
737     - type: 'Threshold'
738       parameter: '{parameters.parameter}'
739       feature: '{activity.customer_id}'
```

## 13.5. Annotations

CATEGORY	ANNOTATIONS	EXAMPLE	REMARK
KYC	<ul style="list-style-type: none"><li>• kyc: true</li><li>• kyc.order: &lt;number&gt;</li></ul>	<pre>1 Field( 2   identifier="source_of_funds", 3   display_name="Source of funds", 4   data_type=DataType.STRING, 5   annotations={"kyc": True,}</pre>	NOTE - in practice this is the only annotation used by the dataset

CATEGORY	ANNOTATIONS	EXAMPLE	REMARK
		<pre> 6             "kyc.order": 2}, 7   ), </pre>	
ANALYSIS TYPE	<ul style="list-style-type: none"> <li>• tag: analysisType</li> </ul>	<pre> 1   annotations: 2     tag: analysisType </pre>	onboarding / ongoing indicator
CUSTOMER ID	<ul style="list-style-type: none"> <li>• customer_id true</li> </ul>	<pre> 1   Field( 2     identifier="customer_id", 3     display_name="Customer ID", 4     data_type=DataType.STRING, 5     annotations={"kyc": True, 6                   "kyc.order": 1, 7                   "customer_id": True}, 8   ) </pre>	
CUSTOMER NAME	<ul style="list-style-type: none"> <li>• customer_name true</li> </ul>	<pre> 1   Field( 2     identifier="customer_name", 3     display_name="Customer Name", 4     data_type=DataType.STRING, 5     annotations={"customer_name": True}, 6   ) </pre>	

Category	Annotations	Example	Remark
DATE	<ul style="list-style-type: none"> <li>date true</li> </ul>	<pre> 1   Field( 2     display_name="date", 3     data_type=DataType.TIMESTAMP, 4     date_format="%Y%m%d", 5     identifier="date", 6     annotations={"date": True}, 7   ) </pre>	
CLASSIFICATION RULE RISK VARIABLE	<ul style="list-style-type: none"> <li>priority: &lt;number&gt;</li> <li>classification_rule: true</li> <li>classification_column: variable tpl which defines final classification rules_group: &lt;string&gt;</li> </ul>	<pre> 1   annotations: 2     priority: 100 3     classification_rule: True 4     classification_column: "{variablesaml_effective_classification}" 5     rules_group: aml </pre>	Variable annotated like this will be considered as classification rule - i.e. the core unit of risk engine that is used for classification made by rules
CLASSIFICATION BY RULES RISK VARIABLE	<ul style="list-style-type: none"> <li>classification_logic: prioritized_rules</li> <li>classification_rule true</li> <li>classification_</li> </ul>	<pre> 1   annotations: 2     priority: 100 3     classification_rule: True 4     classification_column: "{variablesaml_effective_classification}" 5     rules_group: aml </pre>	Classification that is defined in result of evaluation classification rules with selecting first not null value by priority.

CATEGORY	ANNOTATIONS	EXAMPLE	REMARK
	<ul style="list-style-type: none"> <li>column: variable tpl which defines final classification</li> <li>rules_group: &lt;string&gt; g</li> </ul>		
CLASSIFICATION BY ML RISK VARIABLE	<ul style="list-style-type: none"> <li>classification_logic: ml</li> <li>evaluation_step: aml_ml</li> <li>classification_column: variable tpl which defines final classification</li> <li>rules_group: &lt;string&gt;</li> <li>group_classification: true</li> </ul>	<pre> 1 annotations: 2   group_classification: true 3   rules_group: aml 4   classification_logic: ml 5   evaluation_step: aml_ml 6   classification_column: "{variables.aml_effective_classification}" </pre>	Classification calculated basing on score returned by semi supervised classifier.
SCORE	<ul style="list-style-type: none"> <li>group_classification: true</li> <li>classification_column: variable tpl which defines</li> </ul>	<pre> 1 annotations: 2   group_classification: true 3   classification_column: "{variables.aml_effective_classification}" 4 </pre>	

CATEGORY	ANNOTATIONS	EXAMPLE	REMARK
	final classification		
CLASSIFICATION	<ul style="list-style-type: none"> <li>classification: true</li> </ul>	<pre> 1   annotations: 2     classification: true </pre>	Final group classification. Rules or ML if classification weren't made by rules. classification_type us a hint for reporting service to understand either classification is effective or system, if not set - classification is considered as system and effective classification in report will be taken directly from reclassification table (activity_ovrd_{ef_id})
SYSTEM CLASSIFICATION	<ul style="list-style-type: none"> <li>system_classification: bool</li> <li>classification_column: variable tpl which defines final classification</li> </ul>	<pre> 1   annotations: 2     system_classification: true 3     classification_column: "{variablesaml_effective_classification}" 4   In order to introduce ability to implement custom logic of effective cla </pre>	In order to introduce ability to implement custom logic of effective classification definition instead of directly using values from override table system classification should be introduced.
CASE TRIGGER RULES	<ul style="list-style-type: none"> <li>case_trigger_rule: true</li> </ul>	<pre> 1   annotations: </pre>	Any calculations that can be made in to use in risk condition expression. Can be classification

CATEGORY	ANNOTATIONS	EXAMPLE	REMARK
		2   case_trigger_rule: true	check of any other logic that can be implemented withing SQL.
CASE TRIGGER	• case_trigger: true	1   annotations: 2   case_trigger: true	Variable that should be used in risk condition

## 13.6. Dataset Metadata

The following attributes have been added to the Field class:

1. include\_in\_report - indicates which fields should be added to the automatic report.
2. annotations - set of key value mapping that should be used according to the table found in the “annotations” section, in order to acknowledge decisioning, IC and reporting service about the role of the field from the dataset.

Available annotations:

- a. kyc: bool
- b. kyc.order: <number>
- c. customer\_id: true
- d. customer\_name: true
- e. date: true

## 13.7. Example - Date True Code

```
1  from typing import List
2  from thetaray.api.solution import DataSet, DataType, Field, IngestionMode
3  def customer_dataset() -> DataSet:
4      return DataSet(
5          identifier="cra_ml_customer",
6          display_name="cra_ml_customer",
7          field_list=[
8              Field(
9                  identifier="customer_id",
10                 display_name="Customer ID",
11                 data_type=DataType.STRING,
12                 annotations={"kyc": True, "kyc.order": 1, "customer_id": True},
13             ),
14             Field(
15                 identifier="customer_name",
16                 display_name="Customer Name",
17                 data_type=DataType.STRING,
18                 annotations={"customer_name": True},
19                 include_in_report=True,
20                 include_in_hash=True,
21             ),
22             Field(
23                 display_name="date",
24                 data_type=DataType.TIMESTAMP,
25                 date_format="%Y%m%d",
26                 identifier="date",
27                 annotations={"date": True},
```

```
28     ),
29     Field(
30         identifier="classification_type",
31         display_name="Classification type (Individual / Company)",
32         data_type=DataType.STRING,
33         annotations={"classification_type": True},
34         valid_values=["Individual", "Company"],
35     ),
36     Field(identifier="client_type", display_name="Client Type and Structure", data_type=DataType.STRING),
37     Field(
38         identifier="nationality",
39         display_name="Nationality (NP) / Jurisdiction (Corporates)",
40         data_type=DataType.STRING,
41         include_in_report=True,
42     ),
43     Field(
44         identifier="residence",
45         display_name="Domicile or Residence (NP) / Operation Country (Corporates)",
46         data_type=DataType.STRING,
47     ),
48     Field(identifier="lifetime", display_name="Time of knowing the client", data_type=DataType.STRING),
49     Field(
50         identifier="world_checklist",
51         display_name="World-Check listed / PEP or PEP Related",
52         data_type=DataType.STRING,
53     ),
54     Field(
55         identifier="occupation",
56         display_name="Economic Activity / Profession / Occupation",
```

```
57         data_type=DataType.STRING,
58     ),
59     Field(identifier="channel", display_name="Onboarding channel / Contact", data_type=DataType.STRING),
60     Field(
61         identifier="source_of_funds",
62         display_name="Source of funds",
63         data_type=DataType.STRING,
64         annotations={"kyc": True, "kyc.order": 2},
65     ),
66     Field(
67         identifier="geo_origin_of_funds",
68         display_name="Geographic origin of the funds",
69         data_type=DataType.STRING,
70         annotations={"kyc": True, "kyc.order": 3},
71     ),
72     Field(
73         identifier="business_location",
74         display_name="Location of the business activity of the client",
75         data_type=DataType.STRING,
76     ),
77     Field(
78         identifier="j_country_aml",
79         display_name="Country of remmitter / senders (AML risk)",
80         data_type=DataType.STRING,
81     ),
82     Field(
83         identifier="j_country_tax",
84         display_name="Country of remmitter / senders (tax evasion risk)",
85         data_type=DataType.STRING,
```

```
86 ),
87     Field(
88         identifier="transaction_total",
89         display_name="Customer with cash transactions total per year",
90         data_type=DataType.LONG,
91     ),
92     Field(
93         identifier="client_is_pep",
94         display_name="Client is PEP/PEP Related (holder, beneficiary or signatory)",
95         data_type=DataType.BOOLEAN,
96         include_in_report=True,
97     ),
98     Field(
99         identifier="opf_high_countries",
100        display_name="With operations, partners or funds from high-risk countries (FATF)",
101        data_type=DataType.BOOLEAN,
102    ),
103    Field(
104        identifier="aml_f_1",
105        display_name="AML Feature 1",
106        data_type=DataType.LONG,
107    ),
108    Field(
109        identifier="aml_f_2",
110        display_name="AML Feature 2",
111        data_type=DataType.LONG,
112    ),
113    Field(
114        identifier="aml_f_3",
```

```
115         display_name="AML Feature 3",
116         data_type=DataType.LONG,
117     ),
118     Field(
119         identifier="aml_f_4",
120         display_name="AML Feature 4",
121         data_type=DataType.LONG,
122     ),
123     Field(
124         identifier="aml_f_5",
125         display_name="AML Feature 5",
126         data_type=DataType.LONG,
127     ),
128     Field(
129         identifier="scrn_f_1",
130         display_name="Screening Feature 1",
131         data_type=DataType.LONG,
132     ),
133     Field(
134         identifier="scrn_f_2",
135         display_name="Screening Feature 2",
136         data_type=DataType.LONG,
137     ),
138     Field(
139         identifier="scrn_f_3",
140         display_name="Screening Feature 3",
141         data_type=DataType.LONG,
142     ),
143     Field(
```

```
144         identifier="scrn_f_4",
145         display_name="Screening Feature 4",
146         data_type=DataType.LONG,
147     ),
148     Field(
149         identifier="scrn_f_5",
150         display_name="Screening Feature 5",
151         data_type=DataType.LONG,
152     ),
153 ],
154 ingestion_mode=IngestionMode.APPEND,
155 publish=True,
156 primary_key=["customer_id"],
157 num_of_partitions=4,
158 num_of_buckets=7,
159 occurred_on_field="date",
160 data_permission="dpv:public",
161 )
162 def entities() -> List[DataSet]:
163     return [customer_dataset()]
164
```

## 13.8. Data Model

### Risk Rating Engine Results

The Evaluation flow results table consists of the following columns:

- Input Dataset Columns

- Rank and rating column for each column of the input dataset with the evaluation step identifier as a prefix  
i.e.  $\text{number\_of\_columns} = n * x * 2$   
 $n$  - number of evaluation steps  
 $x$  - number of input dataset columns  
 $2$  -  $\text{rank}$  and  $\text{rating}$  prefix
- Scores set for each classification class for each evaluation step  
i.e.  $\text{number\_of\_columns} = x * y$   
 $x$  - number of evaluation classes (labels)  
 $y$  - number of evaluation steps

Evaluation flow activity table contains results of decisioning and is built mostly from columns per each risk variable and risk enrichment.

### 13.9. Manual Reclassification

An additional new table is introduced - `activity_ovrd_{evaluation_flow_identified}` which contains the results of the manual reclassification for a certain counterparty.

### 13.10. Reference Implementation

This section currently covers the following topics:

- Data Generation for training as well as P1, P2 and Algo cases
- Risk Engine Analysis (risk.yaml file)
- Reports annotations

- Ongoing logic for all customers
- Automatic Classification Change Notification

### 13.10.1. Datasets

Dataset Identifier	Description
cra_ml_customer	customer kyc
cra_ml_customer_label	used for specifying labels for training
cra_ml_customer_label	used for specifying labels for training
customer_analysis_type	used for determining customer analysis type, however for this reference implementation all customers are marked as OG (Ongoing)
cra_ml_black_list	auxiliary for rules
cra_ml_do_not_enter	auxiliary for rules
cra_ml_eu_no_cooperate	auxiliary for rules
cra_ml_fatf_grey_black_list	auxiliary for rules
cra_ml_customer_types	Different customer types (shown in kyc)

**Notes:**

- According to the specs, 80% of the labels will be 0 (unlabeled) and the rest evenly distributed among the classes - see breakdown below
- For customers, we will be creating N=10 columns to train model for each model type (AML/screening), 8 of which are numerical, 1 boolean and 1 categorical

### 13.10.2. Customer Type Table (aux)

type_id	Description
1	Supranational entities
2	Private
3	Natural Person
4	Funds

### 13.10.3. FATF (aux)

<https://www.fatf-gafi.org/en/countries/black-and-grey-lists.html>

### 13.10.4. EU (aux)

#### Which Countries are Listed?

On 20th February 2024, the Council adopted the EU list of non-cooperative jurisdictions for tax purposes. It is composed of **12 Countries**.

**Listed: These countries do not co-operate with the EU or have not fully met their commitments**



### 13.10.5. Data Generation

Data generation is required as following:

Customer Classification	Percentage	Nominal	Comment
Not rated/GE/TPI	10%	10k	From P1
Automatic Low	25%	25k	From P2
Automatic Medium	25%	25k	From P2
Algo Classification	40%	40k	P1 & P2 conditions not met, we don't control classification output
<b>Total</b>	<b>100%</b>	<b>100k</b>	
<b>From these 100%, we will randomly create:</b>			
PEP True	1%	1k	
FATF or EU	1%	1k	
Do Not Enter Country	1%	1k	
Natural Person	1%	1k	

Customer Classification	Percentage	Nominal	Comment
Blacklist	1%	1k	
All of the customers will be labeled as following:			
Label	Percentage	Nominal	Class
0	80%	80k	Unlabeled
1	5%	5k	Low
2	5%	5k	Medium
3	5%	5k	High
4	5%	5k	Very High

1. Note that we do not control the amount of alerts that will be created by algo classifying customers as H/VH
2. The number of alerts that we expect is over 5k

### 13.10.6. Exclusive Rules (P1)

Possible Classifications:

- NR - Not Rated
- TPI - Third Party Introduction
- GE – Group Entity

Rule no.	cpy_managed_by_a	cpy_category	execution	relation_establishment	AML	Screening
1	True	"MCG%" (% is a wildcard)	"Y"	-	NR	NR
2	-	-	-	"third party introduction"	TPI	TPI
3	-	"PCG%" (% is a wildcard)	-	-	GE	GE

### 13.10.7. Non-Exclusive Rules (P2)

Severity scale (High to low):

1. PRB (Prohibited)
2. VH (Very High)
3. H (High)
4. M (Medium)
5. L (Low)

Rule no.	Description	type_id	country Fields	automatic_class	AML	Screening
1	Do Not Enter	-	Value in aux DS	-	-	VH
2	Natural Person	3	-	-	H	H
3	Blacklist	-	-Value in aux DS	-	PRB	PRB
4	Automatic Low	-	-	Low	L	L
5	Automatic Medium	-	-	Medium	M	M
6	Automatic High	-	-	High	H	H

\* country fields - nationality\_incorp\_cntry, residence\_cntry, cntry\_of\_mailing\_address, cntry\_of\_tax\_address

\* country fields - if **any** of the country fields fall in countries that are part of the list then rules is met

### 13.10.8. Algo Classification

Both models (AML and Screening) will be trained on the same set of data, however, each of them will have its own set of features for training.

The set of features for training and evaluation will span 3 types: integer, boolean, categorical. Each model type (AML/Screening) will have 10 features dedicated to it, out of which: 8 are integers, 1 boolean, one categorical.

### 13.10.9. DAGS

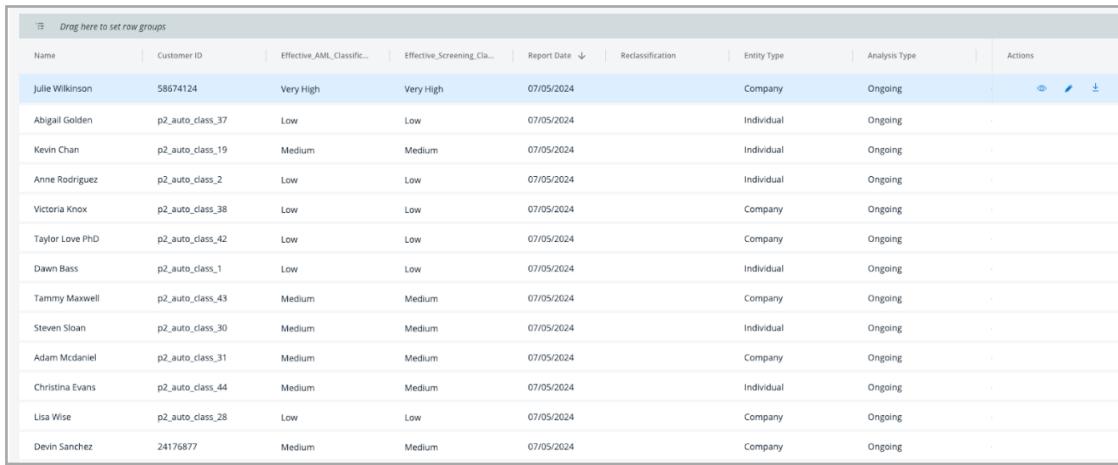
Dags that will be included in the reference implementation:

Name	Description	Comment
cra_ml_generate_data	Generate data for all stages	
cra_ml_prepare_mock_data	Create mock auxiliary data	
cra_ml_upload_and_initialize_analysis_type	Upload data and update onboarding/ongoing	We removed onboarding
cra_ml_ongoing_process	Ongoing logic	Including report and classification change
cra_ml_onboarding_process	Onboarding logic	*not to be used as is, this refimpl supports ongoing only out of the box
cra_ml_train_semi_supervised	Train and save the semi-supervised models	Both for AML and Screening. Should we split them?
cra_ml_generate_reclassification_reports		
cra_ml_demo	One stop shop dag for running everything	also serves as documentation for the processes

### 13.10.10. Case Rules

1. Classification - if AML or Screening classification is High or above (H/VH/PRB) then alert
2. PEP (PEP == True)
3. If the field nationality\_incorp\_cntry is in aux "FATF" or "EU"

### 13.10.11. Reports



Name	Customer ID	Effective_AML_Classific...	Effective_Screening_Cla...	Report Date	Reclassification	Entity Type	Analysis Type	Actions
Julie Wilkinson	58674124	Very High	Very High	07/05/2024		Company	Ongoing	  
Abigail Golden	p2_auto_class_37	Low	Low	07/05/2024		Individual	Ongoing	  
Kevin Chan	p2_auto_class_19	Medium	Medium	07/05/2024		Individual	Ongoing	  
Anne Rodriguez	p2_auto_class_2	Low	Low	07/05/2024		Individual	Ongoing	  
Victoria Knox	p2_auto_class_38	Low	Low	07/05/2024		Company	Ongoing	  
Taylor Love PhD	p2_auto_class_42	Low	Low	07/05/2024		Company	Ongoing	  
Dawn Bass	p2_auto_class_1	Low	Low	07/05/2024		Individual	Ongoing	  
Tammy Maxwell	p2_auto_class_43	Medium	Medium	07/05/2024		Company	Ongoing	  
Steven Sloan	p2_auto_class_30	Medium	Medium	07/05/2024		Individual	Ongoing	  
Adam McDaniel	p2_auto_class_31	Medium	Medium	07/05/2024		Company	Ongoing	  
Christina Evans	p2_auto_class_44	Medium	Medium	07/05/2024		Individual	Ongoing	  
Lisa Wise	p2_auto_class_28	Low	Low	07/05/2024		Company	Ongoing	  
Devin Sanchez	24176877	Medium	Medium	07/05/2024		Company	Ongoing	  

### 13.11. Automatic Reports

#### 13.11.1. Description

- Automatic reports are compiled at the end of each risk assessment, and include all of the counterparties that were analyzed.
- The reports are generated as a zip file, and the extracted report is in csv format.
- Delimiter for the csv file is a comma (,).

#### 13.11.2. Important Notes:

- When no value is applicable for the field it will be **blank**.

- The “rule” fields include only one column for each classification type that include all classification-determining rules and their results (classifications). This is devised to keep a consistent report schema even if the rules were adjusted.
- Similarly, the “feature” fields have been updated to include only one column for each classification type include all features and their results (numerical contribution). This is devised to keep a consistent report schema even if the models were retrained.
- Custom columns can be added via annotation "include in report" on the relevant dataset. These appear immediately after "update\_time" column and before "parent\_id" in the Layout & Field Order table detailed below.

### 13.11.3. Naming convention

**Zip file Name:**

```
CRA_risk_rating_report_{evaluation flow name}  
_YYYY_MM_DD_HH_MM_SS.zip
```

**Example:**

```
CRA_risk_rating_report_cra_ef_2025_03_31_07_58_29.zip
```

**File name:**

```
ar_{evaluation flow name}_YYYY_MM_DD_HH_MM_SS.csv
```

**Example:**

```
ar_cra_ef_2025_03_31_07_58_29.csv
```

#### 13.11.4. Retrieval Location

```
reports/CRA_REPORTS/automatic-reports/dark/cra_ef/{date}
```

**Example:**

```
reports/CRA_REPORTS/automatic-reports/dark/cra_ef/2024_11_21_23_50_00'
```

#### 13.11.5. Layout & Field Order

Column Name	Automatic Report	Automatic Report for Valid Manual Overwrite
customer_id		
evaluation_flow_id	cra_ef	
platform_instance		
dpv	default "dpv:public"	
time	timestamp of analysis	
update_time	will differ from "time" only in reclassification report	
parent_customer_id	filled only in case "inheritance_type" is child	
inheritance_type	"child" or "parent" or empty	
child_ids	filled only in case "inheritance type" is parent	
sibling_ids	filled only in case "inheritance_type" is child	
aml_model_id		
san_model_id		

Column Name	Automatic Report	Automatic Report for Valid Manual Overwrite
analysis_type		
classification_initiator	system	Manual
aml_effective_classification	same as "aml_system_classification" when no valid overwrite	will show reclassification classification when overwrite is approved and valid
san_effective_classification	same as above	same as above – please note that there can be a case where only one classification type is overwritten and valid
case_required	boolean – true if case rule is true (at least one)	
suppression		
case_rules	Case rules and results	
aml_system_classification	aml rating	
san_system_classification	sanctions rating	
aml_system_classification_trigger	"Rules" or "AI Model"	
san_system_classification_trigger	"Rules" or "AI Model"	
aml_rules	list of rules, will always populate rules and their results. If a rule is true then the classification value will appear, and if it is false then "N/A"	
san_rules	same as above	
aml_score	empty according to configuration	
aml_features	list of features – populated only if AI model triggers the rating	

Column Name	Automatic Report	Automatic Report for Valid Manual Overwrite
san_score	empty according to configuration	
san_features	list of features – populated only if AI model triggers the rating	
case_id	empty for all automatic report – relevant only for reclassification report	empty for all automatic report – relevant only for reclassification report
case_type	same as above	same as above
case_resolution	same as above	same as above
case_resolution_notes	same as above	same as above
user_responsible	same as above	same as above
case_resolution_code	same as above	same as above
reclassification	always empty for automatic report when no valid overwrite	true for valid overwrite
aml_reclassification	same as above	classification value of valid overwrite
san_reclassification	same as above	classification value of valid overwrite
reclassification_timestamp	same as above	populated for valid overwrite
reclassification_valid_date	same as above	populated for valid overwrite
reclassification_user_requested	same as above	populated for valid overwrite
reclassification_user_authorized	same as above	populated for valid overwrite
reclassification_notes	same as above	populated for valid overwrite
aml_explainability	shows rule + counterparty data or feature +	shows rule + counterparty data or feature + counterparty data for triggering rules /

Column Name	Automatic Report	Automatic Report for Valid Manual Overwrite
	counterparty data for triggering rules / features	features
san_explainability	shows rule + counterparty data or feature + counterparty data for triggering rules / features	shows rule + counterparty data or feature + counterparty data for triggering rules / features

## 13.12. Reclassification Report

### 13.12.1. Description

- The reclassification report consists of all closed reclassifications in the same day, generated by a daily DAG.
- Both approved and rejected reclassifications are available in the report.
- If there are no reclassifications closed in the day, the report will generate only with the columns but no rows underneath.
- The reports are generated as a zip file, and the extracted report is in csv format.
- Delimiter for the csv file is a comma (,).
- The reclassification report will always have the same format as the automatic report, including any custom fields that were added.

### 13.12.2. Naming Convention

#### Zip file name & file name:

```
reclassification_ar_{evaluation flow name}_YYYY_MM_DD_HH_MM_SS.csv
```

#### Example:

```
reclassification_ar_cra_ef_2025_03_31_00_00_00.csv
```

### 13.12.3. Retrieval Location

```
reports/CRA_REPORTS/reclassification-reports/dark/dpv-public/{date}
```

#### Example:

```
reports/CRA_REPORTS/reclassification-reports/dark/dpv-public/2024_11_21_23_50_00
```

### 13.12.4. Layout & Field Order

**Please note:** All empty fields in the report structure below will be filled with the details relevant to the counterparty at the time the manual overwrite was initiated, in the same formats as the automatic report.

The filled fields indicate:

Column Name	Reclassification Report
customer_id	
evaluation_flow_id	
platform_instance	
dpv	
time	

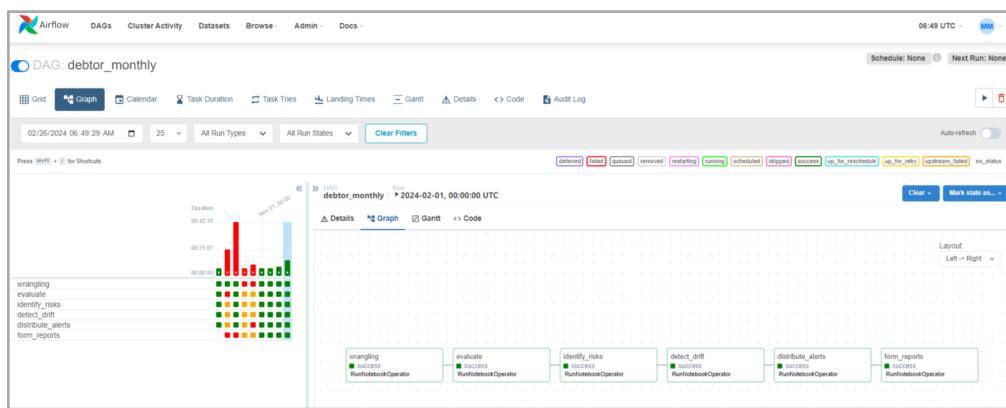
Column Name	Reclassification Report
update_time	
parent_customer_id	
inheritance_type	
child_ids	
sibling_ids	
aml_model_id	
san_model_id	
analysis_type	
classification_initiator	manual
aml_effective_classification	*will show the system classification, not the overwritten
san_effective_classification	*will show the system classification, not the overwritten
case_required	
suppression	
case_rules	
aml_system_classification	
san_system_classification	
aml_system_classification_trigger	
san_system_classification_trigger	
aml_rules	
san_rules	
aml_score	

Column Name	Reclassification Report
aml_features	
san_score	
san_features	
case_id	ID of the reclassification request
case_type	RECLASSIFICATION
case_resolution	APPROVED or REJECTED
case_resolution_notes	notes from the close form of the reclassification request
user_responsible	username who closed the reclassification request
case_resolution_code	resolution selected when closing the reclassification request, for example, "overwrite applied" (customizable from <b>workflow</b> )
reclassification	true
aml_reclassification	classification value of overwrite
san_reclassification	classification value of overwrite
reclassification_timestamp	timestamp of overwrite approval (closing the reclassification request)
reclassification_valid_date	expiration date of the overwrite
reclassification_user_requested	username who initiated the reclassification
reclassification_user_authorized	username that closed the reclassification request
reclassification_notes	notes from the reclassification initiation form
aml_explainability	
san_explainability	

## 14. Workflow Automation

Airflow views each Jupyter Notebook located in `domains/<domain-name>/notebooks` as one task in a sequence that is executed automatically in a predefined order and schedule. This mapping of JupyterLab notebooks to Airflow tasks is achieved by a ThetaRay integration entity inheriting the Airflow API's DAG object. Airflow accesses the entity, learns which notebooks to execute and in what order, and runs the DAG.

The DAG task sequence in Airflow's Graph view:



### 14.1. The DAG object

Every file under the `dags` folder includes a DAG object specifying the configuration and tasks of one or more processes.

 DAG objects are configured under the `root/dags/` folder.

The `schedule_interval` attribute sets whether it's a daily or monthly process. The `start_date` attribute sets the process's first run **date**.

```
default_args = {
    'owner': 'Airflow',
    'depends_on_past': False,
    'retries': 1,
    'retry_delay': datetime.timedelta(minutes=5)
}
dag = DAG(
    dag_id='monthly_analysis',
    catchup=False,
    default_args=default_args,
    schedule_interval=None,
    start_date=datetime.datetime(1993, 1, 1)
```

```
end_date=datetime.datetime(1993, 12, 1)
```

## 14.2. What are the DAG object attributes?

- **dag\_id**: The DAG unique identifier
- **catchup**: Perform a backfill sequence of runs corresponding to the schedule\_interval and the time range between start\_date and end\_date. In the example above, Airflow will perform a sequence of twelve consecutive runs, the first run on the January '93 data records and the last run on the December '93 data records.
- **default\_args**:
  - **depends\_on\_past**: Run the DAG only after any previous DAGS from a scheduling perspective have successfully been completed.
  - **retries**: Number of retries in case of technical workflow failure.
  - **retry\_delay**: The time interval between retries.
- **schedule\_interval**: Run the workflow in a "@monthly", "@weekly", or "@daily" intervals.
- **start\_date**: The date of the workflow's first run in the schedule
- **end\_date**: The date of the workflow's last run in the schedule

### 14.2.1. The RunNotebookOperator object

Every task in the process is defined as a **RunNotebookOperator** object. The object's **notebook\_name** attribute points to the *Jupyter* notebook where the task is defined.

```
dataprep = \
    RunNotebookOperator(task_id="account_monthly_dataprep",
                        domain="default",
                        notebook_name="account_monthly_dataprep",
                        container_cpu="1000m",
                        container_memory="2Gi",
                        dag=dag)

check_drift = \
    RunNotebookOperator(task_id="check_drift",
                        domain="default",
                        notebook_name="account_monthly_drift",
                        container_cpu="600m",
                        container_memory="24Gi",
                        dag=dag)
```

```
evaluation = \
    RunNotebookOperator(task_id="evaluation",
        domain="default",
        notebook_name="account_monthly_evaluation",
        container_cpu="2000m",
        container_memory="4Gi",
        dag=dag)

decisioning = \
    RunNotebookOperator(task_id="decisioning",
        domain="default",
        notebook_name="account_monthly_decisioning",
        container_cpu="1000m",
        container_memory="2Gi",
        dag=dag)
```

The **RunNotebookOperator** has the following attributes:

- **task\_id**: The task name in *Airflow*.
- **domain**: The name of the domain folder under which the Jupyter notebook of the task is located (under the notebooks folder - **domains/<domain\_name>/notebooks**.)
- **notebook\_name**: The name of the task notebook relative to the **<domain\_name>/notebooks** folder without the .ipynb extension suffix.
- **container\_cpu**: The computational power required by the task is measured by millicores (m). One thousand millicores are equivalent to one CPU core. Parallel thread execution or multi-core processing would require a resource of 2000m or more. The resource is released as soon as the task finishes. For more details, see the Resource management documentation for Kubernetes at <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>.
- **container\_memory**: The memory volume required by the task container.
- **dag**: The DAG object to which the task is assigned.
- **parameters**: Value key pairs injected into the executed notebook; see below.

The task order is defined here:

```
dataprep >> check_drift >> evaluation >> decisioning
```

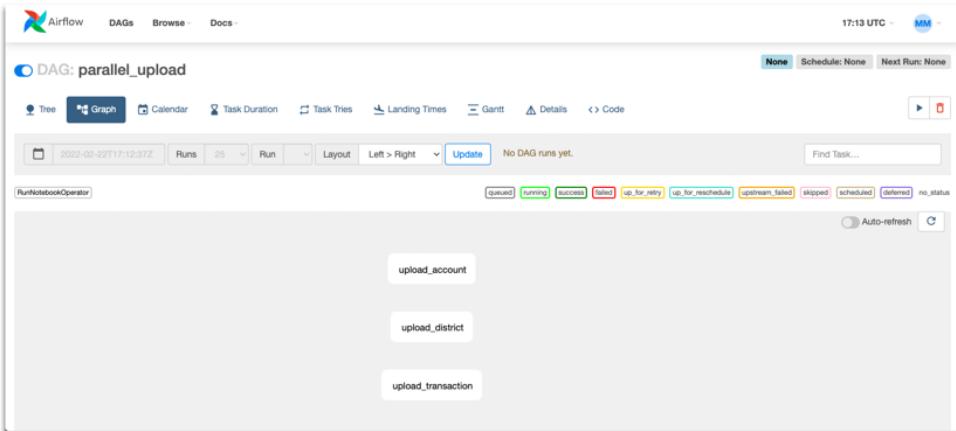
### 14.2.2. Inject parameters from DAG

The parameters attribute enables you to inject parameter values from the dag to the executed notebook.

For example, the **parallel\_upload** DAG **RunNotebookOperator** object contains a **parameter** attribute referring to the key-value pair **connector: c**. Since **RunNotebookoperator** is placed inside an iteration clause, the **c** variable accepts a different value on each iteration.

```
default_args = {  
    'owner': 'airflow',  
    'depends_on_past': False,  
    'retries': 1,  
    'retry_delay': datetime.timedelta(minutes=5)  
}  
  
connectors = ["district", "account", "transaction"]  
  
with DAG(  
    dag_id='parallel_upload',  
    catchup=False,  
    default_args=default_args,  
    schedule_interval=None,  
    start_date=datetime.datetime(1970, 1, 1)  
) as dag:  
  
    for c in connectors:  
        upload_task = RunNotebookOperator(task_id=f"upload_{c}",  
                                            parameters={  
                                                "connector": c  
                                            },  
                                            domain="default",  
                                            notebook_name="parameterized_upload")  
  
if __name__ == "__main__":  
    dag.cli()
```

When the **parallel\_upload** DAG object, in the example, is read by *Airflow*, the following DAG graph is created:



The **parallel\_upload** DAG consists of three separate upload tasks executing the same notebook **parameterized\_upload**. However, the *Airflow* task injects a different **connector** value into the notebook's **context.parameter** attribute in each notebook execution run.

```
from thetaray.api.connector.upload import upload
from thetaray.common.data_environment import DataEnvironment

connector = context.parameters.get("connector", DEFAULT_CONNECTOR)

upload(context, connector, data_environment=DataEnvironment.PUBLIC)
```

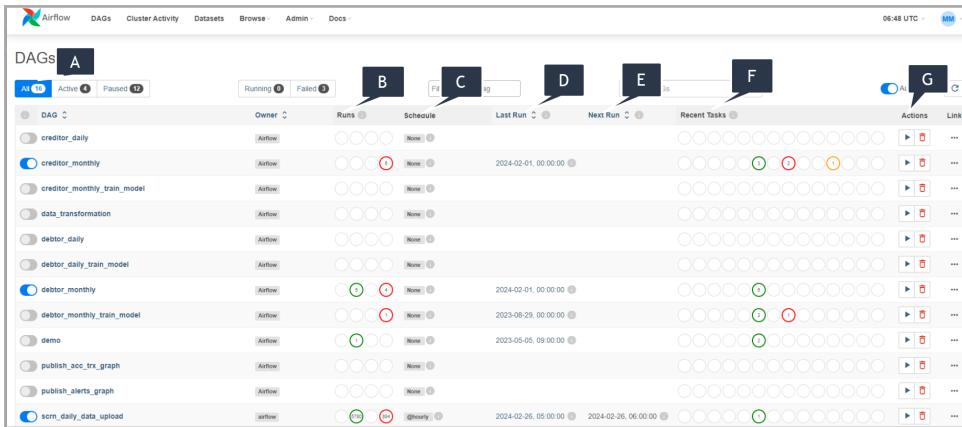
## 14.3. Monitoring Processes in Airflow

*Airflow* periodically checks dag files in the root/dag folder, scans the DAG **start\_date**, **end\_date**, and **schedule\_interval**, and executes the due DAGs. This process is transparent and occurs automatically, but you can open *Airflow* to review and monitor the DAG's execution flow, check their task status, inspect their logs and restart the execution of failed tasks.

### 14.3.1. Review DAGs

The *Airflow* **DAGs** view shows all the different dags defined under root/dags in the JupyterLab environment.

The DAGs view is displayed as shown in the following figure:



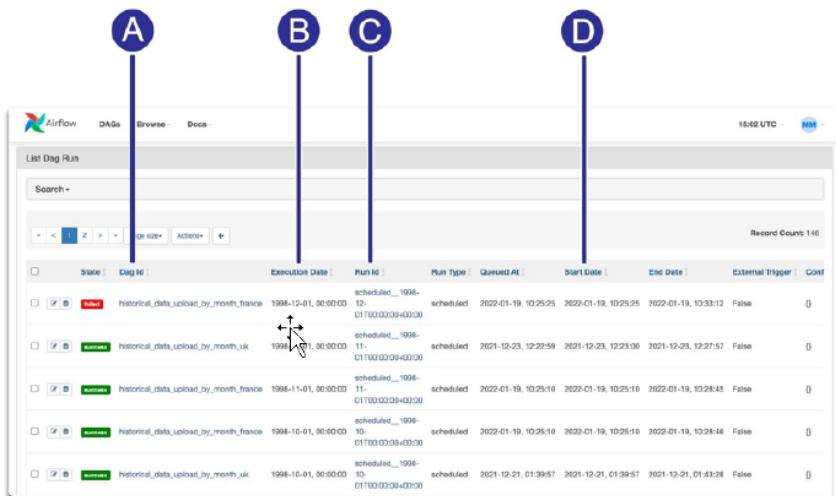
The DAGs view displays the following information:

- Filter DAGs by activity status (All, Active Paused).
- Status of all previous DAG runs.
- DAG Schedule (monthly, weekly, daily)
- Date/Time of the latest DAG run.
- Date/Time of the next DAG run.
- Status of tasks from all active DAG runs or, if not currently active, from the most recent run.
- Run now or delete DAG.

### 14.3.2. Review the DAG execution history

To review the DAG execution history, perform the following:

1. Select Browse -> DAG Runs.



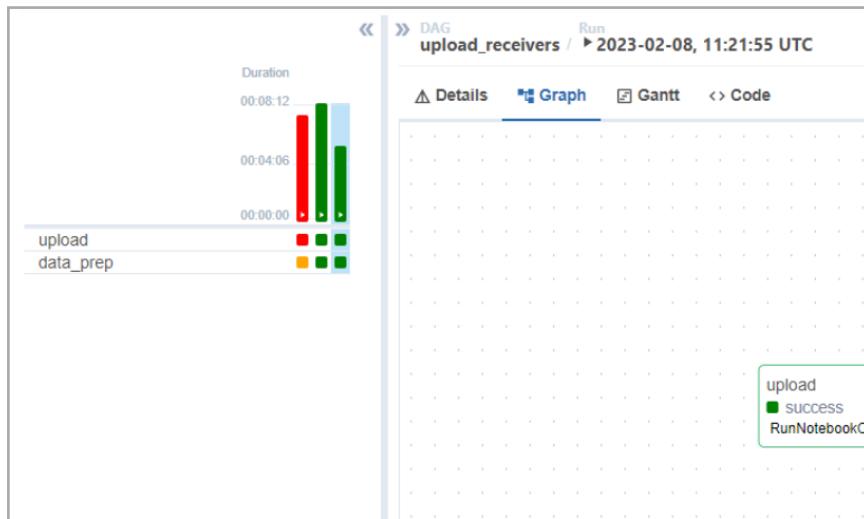
State	Dag ID	Execution Date	Run ID	Run Type	Queued At	Start Date	End Date	External Trigger	Conf
Success	historical_data_upload_by_month_france	1994-12-01, 00:00:00	12-C1700000940030	scheduled...	2022-01-19, 10:25:25	2022-01-19, 10:25:25	2022-01-19, 10:33:12	False	0
Success	historical_data_upload_by_month_uk	1994-10-01, 00:00:00	11-C1700000940030	scheduled...	2021-12-23, 12:22:59	2021-12-23, 12:23:00	2021-12-23, 12:27:57	False	0
Success	historical_data_upload_by_month_france	1994-11-01, 00:00:00	11-C1700000940030	scheduled...	2022-01-19, 10:25:10	2022-01-19, 10:25:10	2022-01-19, 10:28:45	False	0
Success	historical_data_upload_by_month_france	1994-10-01, 00:00:00	10-C1700000940030	scheduled...	2022-01-19, 10:25:10	2022-01-19, 10:25:10	2022-01-19, 10:28:45	False	0
Success	Historical_data_upload_by_month_uk	1994-10-01, 00:00:00	10-C1700000940030	scheduled...	2021-12-21, 01:39:57	2021-12-21, 01:39:57	2021-12-21, 01:43:28	False	0

The DAG RUN list displays the following information for each run:

- A. The ID of the executed DAG.

- B. DAG Logical Date (Execution Date) – denotes the date associated with the specific task execution and can be treated as a batch identifier.
- C. The RUN ID is composed of the DAG Execution Date. The **execution date** determines what data gets overridden in a rerun.
- D. The Run real-time Start Date (in contrast to the DAG Execution Date, which is set on the DAG object and is part of the RUN ID).

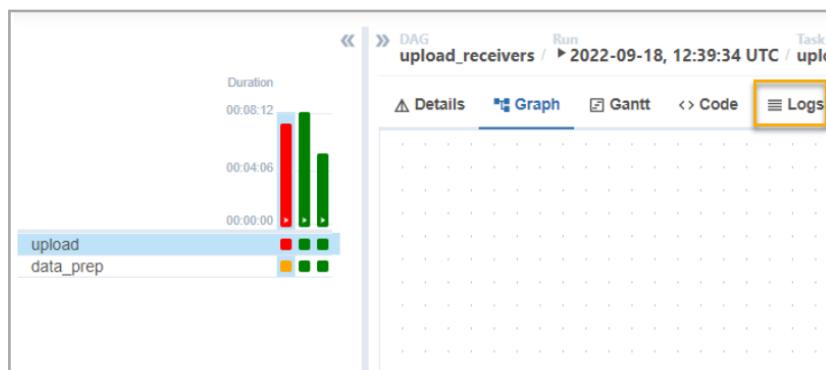
2. Click the DAG RUN you want to view.



### 14.3.3. Inspect a task run log

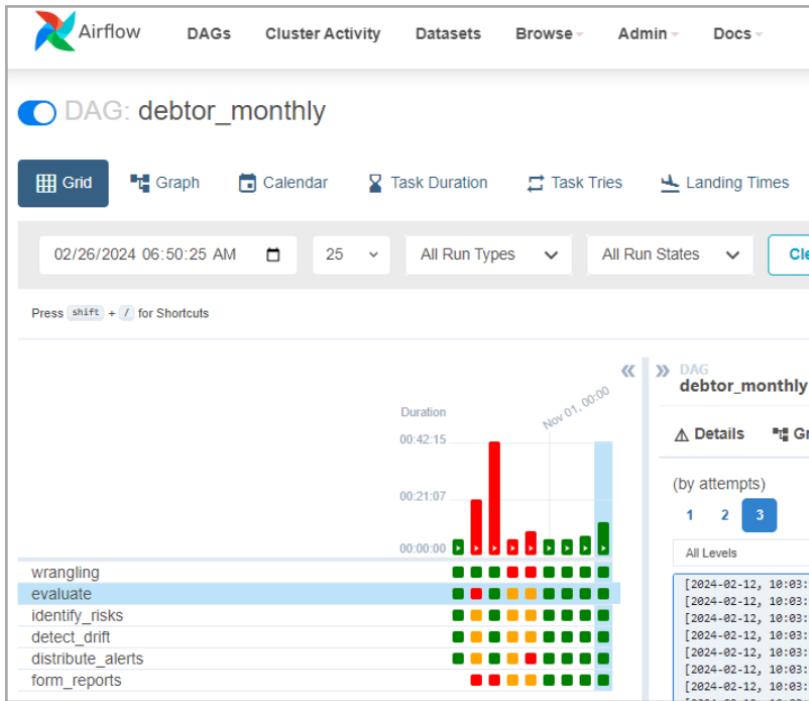
When a particular DAG run has not succeeded because one of the tasks has failed, you can inspect the failed task log to understand the cause of failure:

1. Display the graph view of the DAG.
2. Click on the task for which you'd like to view its log.



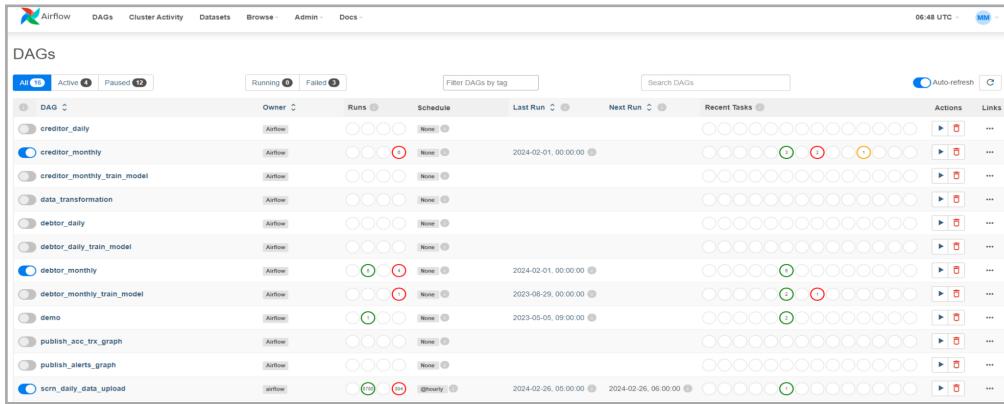
3. Click on 'Logs'.

The scenario depicted in the screenshot below shows an example of the debtormonthly\_analysis DAG task listing all attempts.



4. Select the attempt you'd like to view.

## 14.4. Rerun a task



1. Display the Graph View of a DAG run that you want to rerun.

The screenshot shows the Airflow DAGs page with the following details:

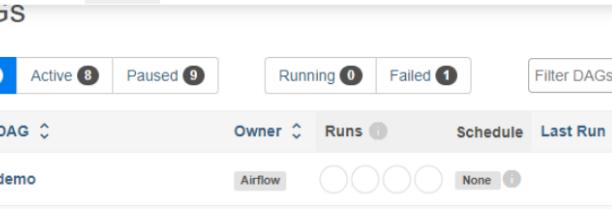
- Header:** Airflow, DAGs, Cluster Activity, Datasets, Browse, Admin, Docs, 06:48 UTC, and a 'More' link.
- Filtering:** Buttons for 'All' (selected), 'Active' (1), and 'Paused' (17). A 'Running' (1) button is highlighted with a red circle.
- Search:** 'Filter DAGs by tag' and 'Search DAGs' input fields.
- Auto-refresh:** A blue button with 'Auto-refresh' and a refresh icon.
- Table Headers:** DAG, Owner (Airflow), Runs, Schedule, Last Run, Next Run, Recent Tasks, Actions, and Links.
- DAG List:**
  - creditor\_daily: Owner Airflow, Runs 0, Schedule None, Last Run 2024-02-01, 00:00:00, Next Run 2024-02-02, 00:00:00, Recent Tasks 0.
  - creditor\_monthly: Owner Airflow, Runs 1 (highlighted with a red circle), Schedule None, Last Run 2024-02-01, 00:00:00, Next Run 2024-03-01, 00:00:00, Recent Tasks 1 (highlighted with a green circle).
  - creditor\_monthly\_train\_model: Owner Airflow, Runs 0, Schedule None, Last Run 2024-02-01, 00:00:00, Next Run 2024-03-01, 00:00:00, Recent Tasks 0.
  - data\_transformation: Owner Airflow, Runs 0, Schedule None, Last Run 2024-02-01, 00:00:00, Next Run 2024-02-02, 00:00:00, Recent Tasks 0.
  - debtor\_daily: Owner Airflow, Runs 0, Schedule None, Last Run 2024-02-01, 00:00:00, Next Run 2024-03-01, 00:00:00, Recent Tasks 0.
  - debtor\_dally\_train\_model: Owner Airflow, Runs 0, Schedule None, Last Run 2024-02-01, 00:00:00, Next Run 2024-03-01, 00:00:00, Recent Tasks 0.
  - debtor\_monthly: Owner Airflow, Runs 1 (highlighted with a green circle), Schedule None, Last Run 2024-02-01, 00:00:00, Next Run 2024-03-01, 00:00:00, Recent Tasks 1 (highlighted with a green circle).
  - debtor\_monthly\_train\_model: Owner Airflow, Runs 1 (highlighted with a red circle), Schedule None, Last Run 2023-08-29, 00:00:00, Next Run 2024-03-01, 00:00:00, Recent Tasks 1 (highlighted with a green circle).
  - demo: Owner Airflow, Runs 1 (highlighted with a green circle), Schedule None, Last Run 2023-05-05, 09:00:00, Next Run 2024-03-01, 00:00:00, Recent Tasks 1 (highlighted with a green circle).
  - publish\_acc\_trx\_graph: Owner Airflow, Runs 0, Schedule None, Last Run 2024-02-01, 00:00:00, Next Run 2024-03-01, 00:00:00, Recent Tasks 0.
  - publish\_alerts\_graph: Owner Airflow, Runs 0, Schedule None, Last Run 2024-02-01, 00:00:00, Next Run 2024-03-01, 00:00:00, Recent Tasks 0.
  - scrm\_daily\_data\_upload: Owner airflow, Runs 1 (highlighted with a green circle), Schedule hourly, Last Run 2024-02-26, 05:00:00, Next Run 2024-02-26, 06:00:00, Recent Tasks 1 (highlighted with a green circle).

2. Click on the task that you want to rerun.
  3. On the *Task Instance* popup, click *Clear*.
  4. Confirm the *Clear* action by clicking *OK*.
  5. Again, click on the task that you want to rerun.

The DAG continues to run downstream from the selected task.

## 14.5. Trigger a DAG in Airflow Manually

In most cases, *Airflow* automatically triggers DAGs according to a predefined schedule, but you



The screenshot shows the Airflow DAGs page. At the top, there are navigation links: DAGs (selected), Cluster Activity, Datasets, Browse, Admin, and Docs. Below the navigation is a search bar with the placeholder 'Filter DAGs by tag'. The main area displays a table of DAGs. The columns are: DAG (with a dropdown arrow), Owner (with a dropdown arrow), Runs (with a dropdown arrow), Schedule (with a dropdown arrow), and Last Run (with a dropdown arrow). The table rows list the following DAGs:

DAG	Owner	Runs	Schedule	Last Run
demo	Airflow	○ ○ ○ ○	None	1
receiver_daily	Airflow	○ ○ ○ ○	None	1
receiver_daily_train_model	Airflow	○ ○ ○ ○	None	1
receiver_monthly	Airflow	○ ○ ○ ○	None	2022-10-06, 09:25:33
receiver_monthly_train_model	Airflow	○ ○ ○ ○	None	2022-10-02, 07:48:18
receiver_weekly	Airflow	○ ○ ○ ○	None	1
receiver_weekly_train_model	Airflow	○ ○ ○ ○	None	1

Annotations at the bottom:

- (1) Active the 'receiver monthly' DAG
- (2) Click on the 'play' button to trigger the DAG Run

can also trigger a DAG manually.

Airflow   DAGs   Cluster Activity   Datasets   Browse   Admin   Docs

</style> <h1>Receiver Train Model</h1> <i--> <p>Use this DAG to run monthly analysis.</p>>> <p>Use this DA

<table class="styled-table"> <thead> <tr> <th>Parameter</th> <th>Default Value</th> <th>Comments</th> </tr>

arbitrarily chosen and should always be changed</td> </tr> <tr> <td>train\_end\_date</td> <td>2019-08-01</td>

<td>sample\_percentage</td> <td>20</td> <td>percentage is a number in the range 0-100 (exclusive)</td> </tr>

DAG conf Parameters

train_start_date:	1993-01-01
train_end_date:	2019-08-01
sample_percentage:	20

Generated Configuration JSON and Dagrun Options

Unpause DAG when triggered

**Trigger** **Cancel**

## 15. Entity Resolution

### 15.1. Overview

Entity resolution refers to the process of identifying and matching entities, such as companies or individuals, across various data sources. The goal is to create a complete and accurate representation of each entity, in order to support risk management, compliance, and other financial operations.

The Entity Resolution module is used to match entities based on customer defined identifiers attributes such as names, addresses, government-issued identification numbers, and others.

The entity resolution matching process involves a combination of manual review and automated techniques, such as algorithms and machine learning.

The matched entities can be viewed and edited from a dedicated area in the Investigation Center. Transaction Monitoring analysis is carried out on the confirmed matched entities, with dedicated alerts.

The Transaction Monitoring alerts are of three types:

1. **Account alert** - alerts on the separate account level, unrelated to Entity Resolution.
2. **Account with Party alerts** - accounts that have a related party, but the alert is on the account level.
3. **Party alerts** - party level alerts, that can optionally consider different features as account level alerts.

All alert types are supported by default and are meant to cover all Entity Resolution functionality. If, for any reason, a specific alert type is required not to appear, the configurations necessary are described in the following sections.

### 15.2. How to Start

This section details the various steps required to configure and run your **Entity Resolution** module.

#### 15.2.1. Step #1 Configure Graph

##### Graph Information and prerequisites

NodeType class is enriched with write\_allowed boolean parameter (default - False), the same is used for EdgeType. It should be set to True for the "party" node.

Property class uses the parameter - reference. It should be set for dynamic party properties to link them to related properties of account node properties

The party node uses mandatory properties. besides party\_uuid from phase 1. Those are party\_uuid name, consolidation, updated\_on, and updated\_by. It can also have up to 3 additional properties (in the example below its address and country).

## Implementation

First step to using **Entity Resolution** (ER) is to configure relevant metadata to enable matching results to be published.

This will create a new data graph in your domain graph folder, or extend an existing graph if one is already configured for Network Visualization.

---

**Note:** In case you have no graph configured you must create a new one.

---

If a graph is already configured, please add the additional metadata needed, which can be copied from a reference branch.

The key elements relevant to Entity Resolution are:

1. Creating "party" NodeType
2. Creating "er\_match" EdgeType
3. Modifying "account" NodeType to include the properties relevant for displaying matching accounts.

### 15.2.2. Example Graph Configuration

```
from typing import List

from thetaray.api.solution import DataType, EdgeType, Graph, NodePropertyReference, NodeType, Property

def graph1_graph() -> Graph:
    return Graph(
        identifier="public",
        nodes=[

            NodeType(
                identifier="AC",
                display_name="Account",
                description="Bank Account",
                properties=[

                    Property(
                        identifier="AN",
                        display_name="IBAN",
                        description="The account IBAN",
                    )
                ]
            )
        ]
    )
```

```
        type=DataType.STRING,
        is_key=True,
    ),
    Property(
        identifier="NM",
        display_name="Full name",
        description="Display name of the account",
        type=DataType.STRING,
    ),
    Property(
        identifier="CT",
        display_name="Country",
        description="The country of the account",
        type=DataType.STRING,
    ),
    Property(
        identifier="AD",
        display_name="Address",
        description="The address of the account",
        type=DataType.STRING,
    ),
],
),
NodeType(
    identifier="AL",
    display_name="Alerted activity",
    description="Alerted activity",
    properties=[
        Property(
            identifier="AI",
            display_name="Alerted activity ID",
            description="Alerted activity ID",
            type=DataType.STRING,
        ),
        Property(
            identifier="RI",
            display_name="Risk identifier",
            description="Alerted activity ID",
            type=DataType.STRING,
        ),
        Property(
            identifier="SP",
            display_name="Suppressed",
            description="Was the alerted activity",
            type=DataType.BOOLEAN,
        ),
    ],
),
```

```
),
NodeType(
    identifier="PR",
    display_name="Party of accounts",
    description="Party of accounts",
    write_allowed=True,
    properties=[
        Property(
            identifier="PI",
            display_name="Party uuid",
            description="Accounts party uuid",
            type=DataType.STRING,
            is_key=True,
        ),
        Property(
            identifier="NM",
            display_name="Party name",
            description="Accounts party name",
            type=DataType.STRING,
        ),
        Property(
            identifier="AD",
            display_name="Party address",
            description="Accounts party address",
            type=DataType.STRING,
            reference=NodePropertyReference(node_identifier="AC", property_
identifier="AD"),
        ),
        Property(
            identifier="CT",
            display_name="Party country",
            description="Accounts party country",
            type=DataType.STRING,
            reference=NodePropertyReference(node_identifier="AC", property_
identifier="CT"),
        ),
        Property(
            identifier="CN",
            display_name="Consolidation flag",
            description="Consolidation flag",
            type=DataType.LONG,
        ),
        Property(
            identifier="UON",
            display_name="Updated on",
            description="Last updated date",
            type=DataType.TIMESTAMP,
        ),
    ],
),
```

```
        ),
        Property(
            identifier="UBY",
            display_name="Updated by",
            description="User updated the party",
            type=DataType.STRING,
        ),
    ],
),
],
edges=[
    EdgeType(
        identifier="TX",
        display_name="Transaction",
        description="The transaction from one account to another",
        properties=[
            Property(
                identifier="AM",
                display_name="Amount",
                description="The transaction amount",
                type=DataType.DOUBLE,
            ),
            Property(
                identifier="CR",
                display_name="Currency",
                description="The transaction currency",
                type=DataType.STRING,
            ),
            Property(
                identifier="CT",
                display_name="Transactions count",
                description="Amount of transactions, represented by the edge",
                type=DataType.LONG,
            ),
        ],
),
    EdgeType(
        identifier="AL",
        display_name="Alerting edge",
        description="Edge, connecting alerted activity to account",
        properties=[],
    ),
    ## Needed for party implementations for solutions that handle CB payments
    EdgeType(
        identifier="ER",
        display_name="EMP Match",
        write_allowed=True,
```

```
        description="Edge from party to matched account",
        properties=[
            Property(
                identifier="ST",
                display_name="State",
                description="State of the candidate",
                type=DataType.LONG,
            ),
            Property(
                identifier="SC",
                display_name="Score",
                description="Score of the match",
                type=DataType.DOUBLE,
            ),
        ],
),
### Needed for party implementations for solutions that handle retail banking
EdgeType(
    identifier="OW",
    display_name="Owner",
    description="Edge from party to account",
    properties=[],
),
],
data_permission="dpv:public",
)

def entities() -> List[Graph]:
    return [graph1_graph()]
```

---

**Note**, that the above example contains only nodes and edges, related to the Entity Resolution feature.

---

### 15.2.3. Step #2 Review Default Settings for ER

Settings are located at /settings/eresolution\_settings.py

Below is short summary listing of settings parameters.

- matching\_fields\_parameters: List of matching fields and their weight to use for matching.  
Each element in the list is a dictionary that has the following keys:
  - "field": name of the field to use for matching
  - "weight": weight to give this field when computing the final match score

- "normalized": (optional) normalized name of the field that will be used for matching instead of field
- Normalization, for example, removes titles (Mr., Mrs., etc.), converts to lowercase, etc. It is recommended to normalize fields that are not casesensitive, in order to ensure better fields matching
- minhash\_lsh\_chunk\_size (int) - Number of records that will be processed by one worker for computing minhash lsh.  
Can be used to tune memory issues / increase speed of computing.
- minhash\_lsh\_matching\_field (str): The name of the field to use for minhash LSH indexing and querying.

**It is mandatory to set a field that will be used for indexing and querying; in Reference Implementation, this field is set to "normalized\_name"**

- minhash\_lsh\_num\_permutations (int): The number of permutations to use in the minhash LSH.

**Higher number of permutations will increase the accuracy of the LSH but will also increase the computational cost.**

- minhash\_lsh\_jaccard\_threshold (float): The Jaccard similarity threshold to use for querying the minhash LSH.

**Lower threshold will increase the number of candidates but also increase the number of false positives.**

- minhash\_lsh\_ngrams (int): The number of grams to use for minhash LSH indexing.

**The number of grams used will affect the granularity of the indexing and thus the recall and precision of the LSH. It is not recommended to change the default setting.**

- match\_threshold (float): The threshold to use when determining if two accounts match.
- match\_batch\_size (int): The batch size for calculating matching score by Damerau-Levenshtein algorithm.
- workers (int): The number of worker processes to use for parallel computation. Workers = processes.
- auto confirmed threshold - with the float setting - auto\_confirmation\_threshold, which defaults to 0.8. EMP creates matches in auto-confirmed status (instead of the candidate) when the match score is higher than the auto\_confirmation\_threshold. To disable auto-confirmation, the auto\_confirmation\_threshold should be set to -1.

The crucial fields for configuration are:

- matching\_fields\_parameters and minhash\_lsh\_matching\_field. These should be set according to the available data.

**Note:** The matching process is incremental, so changing thresholds and settings will affect only new match outputs.

#### 15.2.4. Step #3 Prepare your Data

Please note that all calculations are done in-memory and not distributed with spark, so it is recommended to prepare only the required columns for the input dataset.

Input dataframe should represent all accounts that contain these columns:

- **id** - id of account that can be used as a primary key
- **date** - field that can be used as effective\_date (DateTime, associated with the account)
- other fields should be same as set it in **matching\_fields\_parameters** setting. In case of Reference implementation it's 'name', 'address', 'normalized\_name'

#### 15.2.5. Example:

```
import datetime
from pyspark.sql import functions as f
from thetaray.api.dataset import dataset_functions
from thetaray.common.data_environment import DataEnvironment

accounts = dataset_functions.read(context, "account").drop("tr_timestamp")

accounts = accounts.withColumn("normalized_name", f.lower(accounts["name"]))
accounts = accounts.withColumnRenamed("account_id", "id")
accounts = accounts.withColumnRenamed("date", "effective_date")
accounts = accounts.select("id", "name", "address", "normalized_name", "country",
"effective_date")
accounts = accounts.withColumnRenamed("name", "NM")
accounts = accounts.withColumnRenamed("address", "AD")
accounts = accounts.withColumnRenamed("country", "CT")

accounts.show()
```

#### 15.2.6. Step #4 Run Entity Resolution

'Matching code' is contained in a package in the Reference branch located in the following path /lib/entity\_resolution directory.

**To run it simply import it as shown in the following code example and run.**

**Entity Resolution - run example**

```
from lib.entity_resolution import EntityResolution

er = EntityResolution(context=context, data_
environment=DataEnvironment.PUBLIC)
res = er.resolve(account_df)
```

'res' will be represented as pandas dataframe that should be converted to spark with appropriate schema like on this example:

```
schema = StructType(
[
    StructField("source", StringType()),
    StructField("clique", StringType()),
    StructField("mean_score", DoubleType()),
    StructField("clique_size", LongType()),
    StructField("NM", StringType()),
    StructField("AD", StringType()),
    StructField("CT", StringType()),
    StructField("effective_date", TimestampType()),
]
)
res = spark.createDataFrame(res_pdf, schema=schema)
```

---

**Note:** The only thing required to change is replacing "name" and "address" with the fields from the matching settings.

---

---

**Note:** 'res' represents the number of new matches. To avoid overwriting approve/rejected data, it does not contain records that already exist in the graph.

---

## 15.2.7. Step 5. Publish Results

To ensure valid info is presented in the Investigation Center, "account" nodes, "party" nodes and "er\_match" edges should be published.

### 15.2.7.1. Publishing Accounts

Creating a dataframe is required by providing:

```
existing_accounts = read_nodes(context=context, graph_identifier="public", type="AC")
accounts_nodes_df = accounts.withColumn("effective_date", f.to_timestamp(f.lit
(context.execution_date)))
```

```
accounts_nodes_df = accounts_nodes_df.join(existing_accounts, on="id", how="left_anti")
accounts_nodes_df = accounts_nodes_df.withColumn("AN", f.col("id"))
accounts_nodes_df.printSchema()
publish_nodes(
    context=context,
    nodes_df=accounts_nodes_df,
    graph_identifier="public",
    node_type="AC",
)
)
```

### 15.2.7.2. Publishing Parties

Parties should be extracted from the **Entity Resolution** output process by running the code as shown in the following code snippet.

```
existing_parties = read_nodes(context=context, graph_identifier=graph, type="PR")
# parse graph
parties_nodes_df = res.select("clique")
parties_nodes_df = parties_nodes_df.dropDuplicates(["clique"])
parties_nodes_df = parties_nodes_df.withColumnRenamed("clique", "PI")
parties_nodes_df = parties_nodes_df.withColumn("effective_date", f.lit(context.execution_date))
parties_nodes_df = parties_nodes_df.withColumn("id", f.col("PI"))

# add properties
parties_nodes_df = parties_nodes_df.withColumn("NM", f.lit(None).cast("string"))
parties_nodes_df = parties_nodes_df.withColumn("AD", f.lit(None).cast("string"))
parties_nodes_df = parties_nodes_df.withColumn("CT", f.lit(None).cast("string"))

# add technical columns
parties_nodes_df = parties_nodes_df.withColumn("CN", f.lit(1).cast("long"))
parties_nodes_df = parties_nodes_df.withColumn("UON", f.lit(None).cast("timestamp"))
parties_nodes_df = parties_nodes_df.withColumn("UBY", f.lit(None).cast("string"))
# remove existing parties
parties_nodes_df = parties_nodes_df.join(existing_parties, on="id", how="left_anti")

parties_nodes_df.printSchema()

publish_nodes(
    context=context,
    nodes_df=parties_nodes_df,
    graph_identifier=graph,
    node_type="PR",
)
```

**Note:** As effective date - actual date of processing (context.execution\_date) should be used. To track parties changes by adding new account.

### 15.2.7.3. Edges Publishing

Same as parties, should be extracted from results with usage of context.execution\_date as effective\_date

```
edges_df = res.select("mean_score", "clique", "source")
edges_df = edges_df.withColumnRenamed("source", "id")
edges_df = edges_df.withColumn("effective_date", f.to_timestamp(f.lit(context.execution_date)))
edges_df = edges_df.withColumnRenamed("clique", 'source_node')
edges_df = edges_df.withColumn("target_node", f.col("id"))
edges_df = edges_df.withColumnRenamed("mean_score", "SC")
threshold_col = f.lit(context.solution.eresolution_settings.auto_confirmation_threshold)
edges_df = edges_df.withColumn(
    "ST",
    f.when(
        (f.col("SC") >= threshold_col) & (threshold_col != -1),
        f.lit(MatchState.AUTO_CONFIRM.value)
    ).otherwise(MatchState.CANDIDATE.value).cast("long")
)
edges_df.printSchema()
publish_edges(
    context=context,
    edges_df=edges_df,
    graph_identifier="public",
    edge_type="ER",
    source_node_type="PR",
    target_node_type="AC",
)
```

### 15.2.8. Step #6 Party Evaluation Flow

#### 15.2.8.1. Metadata Changes

In order to enable the consolidation of alerts related to the same party, a new set of IDs was added to the EvaluationFlow as the "identifiers" field, containing the following:

- grouping\_identifier - mandatory, one or more fields of the input dataset, uniquely identifying the entity (Party)
- primary\_identifier - optional, one or more fields of the input dataset, uniquely identifying sub-entity (Account). Must be filled for sub-entity level Evaluation Flow

- primary\_entities\_list - field, containing a JSON-formatted list of identifiers of the sub-entities, associated with the grouping\_identifier (IDs of accounts of the party)

RelatedGraph now has additional grouping\_node\_mappings field to link a grouping node's id property (party) to the evaluation flow's field(s) containing party id.

For the below example to work, "wrangling" (set as input\_dataset) should contain the fields party\_id, account\_id, and party\_accounts.

party\_accounts field, in this case, would contain data in the following format:

```
[{"account_id": "8169"}, {"account_id": "200"}]
```

The full example can be found in the Reference branch.

## Example metadata

```
from typing import List

from thetaray.api.solution import AlgoEvaluationStep, EvaluationFlow, TraceQuery
from thetaray.api.solution.evaluation import (
    EvaluationFlowIdentifiers,
    ModelReference,
    PropertyToFieldMapping,
    RelatedGraph,
)

def evaluation_flow() -> EvaluationFlow:
    return EvaluationFlow(
        identifier="party_tr_analysis",
        displayName="party_tr_analysis",
        input_dataset="party_wrangling",
        data_permission="dpv:public",
        identifiers=EvaluationFlowIdentifiers(
            grouping_identifier=[ "party_id" ],
            primary_entities_list="party_accounts",
        ),
        evaluation_steps=[
            AlgoEvaluationStep(
                identifier="tr_evaluation",
                name="Thetaray evaluation",
                feature_extraction_model=ModelReference(
                    name="tr_feature_extraction_model",
                    tags={"version": "release" },
                ),
                detection_model=ModelReference(name="tr_detection_model", tags={"version": "release" }),
                pattern_length=3,
            ),
        ],
    )
```

```
],
max_workers=8,
global_trace_query=TraceQuery(
    identifier="transactions_global_tq",
    identifier_column="trans_id",
    dataset="transaction",
    is_grouping_query=True,
    sql="""
        SELECT ds.* FROM {dataset_table} ds
        JOIN {primary_identifiers} AS pr_ids
            ON ds.account_id = pr_ids.account_id
        WHERE ds.date >= {activity.date_loan} -interval '1 month' - ({occurred_
before} || ' DAY')::INTERVAL
            AND ds.date <= {activity.date_loan} + ({occurred_after} || 'DAY')::INTERVAL
        """,
),
trace_queries=[

    TraceQuery(
        identifier="transactions_tq",
        identifier_column="trans_id",
        features=[
            "amount",
            "min1",
            "max1",
            "mean1",
            "min2",
            "max2",
            "mean2",
            "min3",
            "max3",
            "mean3",
            "min4",
            "max4",
            "mean4",
            "min5",
            "max5",
            "mean5",
            "min6",
            "max6",
            "mean6",
        ],
        dataset="transaction",
        is_grouping_query=True,
        sql="""
            SELECT ds.* FROM {dataset_table} ds
            JOIN {primary_identifiers} AS pr_ids
                ON ds.account_id = pr_ids.account_id
        """
    )
]
```

```
        WHERE ds.date >= {activity.date_loan} - interval '1 month' - 
        ({occurred_before} || ' DAY')::INTERVAL
            AND ds.date <= {activity.date_loan} + ({occurred_after} || ' 
        DAY')::INTERVAL
        """,
    ),
    TraceQuery(
        identifier="loan_tq",
        identifier_column="loan_id",
        features=["payments", "amount", "duration"],
        dataset="loan",
        is_grouping_query=True,
        sql="""
            SELECT ds.* FROM {dataset_table} ds
            JOIN {primary_identifiers} AS pr_ids
                ON ds.account_id = pr_ids.account_id
            AND ds.date >= {activity.date_loan} - interval '1 month' - ({occurred_
        before} || ' DAY')::INTERVAL
            AND ds.date <= {activity.date_loan} + ({occurred_after} || ' 
        DAY')::INTERVAL
        """,
    ),
    ],
    related_graph=RelatedGraph(
        identifier="public",
        node_mappings={"PR": [PropertyToFieldMapping(field="party_id",
        property="PI")]}},
    ),
)

def entities() -> List[EvaluationFlow]:
    return [evaluation_flow()]
```

### 15.2.8.2. Account Evaluation Flow Updates

#### Input Dataset

As described above, new identifiers for evaluation flow have been introduced in phase 2. This means that new fields should be added to the input dataset of the Evaluation Flow (containing grouping\_identifier and primary\_entities\_list). In order to do so, the graph tables can be used to read party-account pairs and join them to the original data frame. Please, see the example in the "wrangling.ipynb" notebook of the Reference implementation.

#### Evaluation Flow

Account Evaluation Flow should be enriched with configurations, described above - a list of new identifiers, and grouping\_node\_mappings for the related\_graph. See evaluation\_flow\_tr from the reference **Account with Party Alert Type** implementation for the example.

The configurations above will result in the two types of account level alerts:

1. Account alerts
2. Account with Party alerts

If Account with Party alerts are not required for analysis, the following guidance can be used to remove them:

The load graph API exposed through the entity resolution class can be used to retrieve confirmed party to account relationships stored in the database. This information can be used to enrich account level transactions with associated party and on the other hand, can be used to filter out accounts with a related party from the Transaction Monitoring analysis.

A few stipulations:

1. Please make sure that the analysis of the party will cover the cases of the account analysis, so no potential suspect activity will be missed.
2. Consolidation between party alerts and account alerts will no longer be supported.

### 15.2.8.3. New Party Evaluation Flow

Entity Resolution allows analyzing data on the party level (in addition to the existing account-level analysis). It can be done by aggregating the data on the party level (using graph data) and creating an evaluation flow based on these new aggregations. In other words, the input dataset of the account-level evaluation flow can be aggregated by party id and used as an input dataset for the new party-level evaluation flow. Of course, the aggregation can be done at earlier stages of the flow in order to use different features for party-level analysis.

A full example of party-level analysis flow can be found in the Reference branch.

Entities, relevant to the flow:

- dataset wrangling\_party.py
- evaluation flow evaluation\_flow\_tr\_party.py
- risk risk\_id2\_party.yaml
- notebooks:
- wrangling\_party.ipynb
- tr-algo-party-train.ipynb
- tr-algo-party-detect.ipynb
- decisioning-party.ipynb
- distribution-party.ipynb

DAG party\_evaluation.py

### Party-level trace query

Data that is usually retrieved by trace queries (e.g. transactions), in most cases would be present in the resolution of an account, while alerts from the party evaluation flow are based on party-level aggregated data. So, in order to support this case (i.g. to retrieve data related to all the accounts of the party), a new flavor of trace query was introduced.

First of all, a new parameter was added to TraceQuery to mark the grouping implementation - is\_grouping\_query (boolean, defaults to False). Also, a new substitutable keyword was introduced for the SQL implementation - primary\_identifiers. When joining it to the query's main dataset, it will filter data by accounts of the party.

```
TraceQuery(  
    identifier='transactions_global_tq',  
    identifier_column="trans_id",  
    dataset='transaction',  
    is_grouping_query=True,  
    sql=''  
        SELECT ds.* FROM {dataset_table} ds  
        JOIN {primary_identifiers} AS pr_ids  
            ON ds.account_id = pr_ids.account_id  
        WHERE ds.date >= {activity.date_loan} -interval '1 month' - ({occurred_before} || 'DAY')::INTERVAL  
            AND ds.date <= {activity.date_loan} + ({occurred_after} || 'DAY')::INTERVAL  
        ...  
)
```

## 16. Manage Code in Gitlab

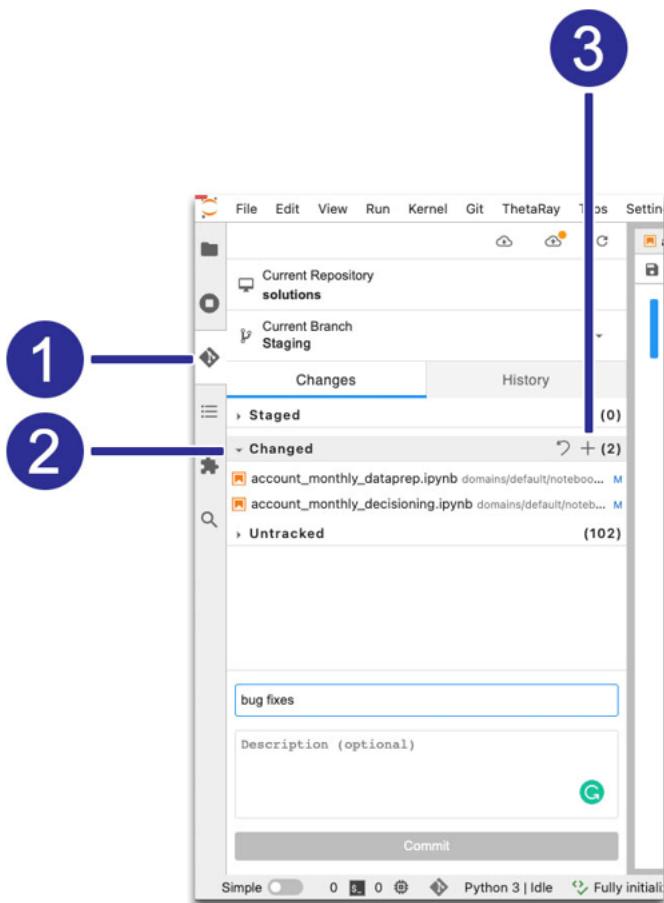
Gitlab manages source code for the ThetaRay Platform in two modes:

- **The GitLab plugin:** An applet accessible from the Jupyter side panel allowing you to commit changes and push them to different branches.
- **GitLab:** An environment-level installation of GitLab where you can create branches and perform all git commands and activities.

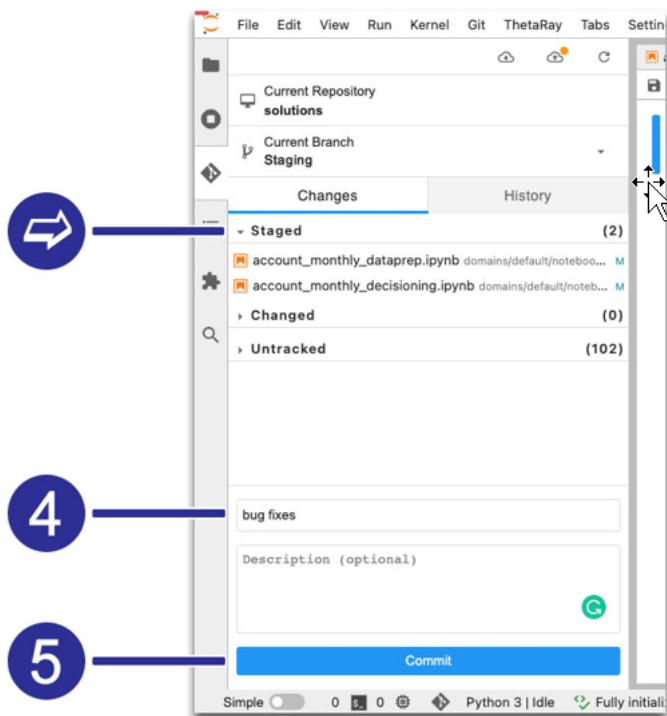
### 16.1. The Gitlab Plugin

Use the *GitLab plugin* to commit code to the work branch or push it to the Staging branch (providing you have adequate permissions).

#### 16.1.1. Commit new code to the staging branch



1. Click  in the sidebar to display the Gitlab plugin.
2. Expand Changed.
3. Click + to stage the changed files.



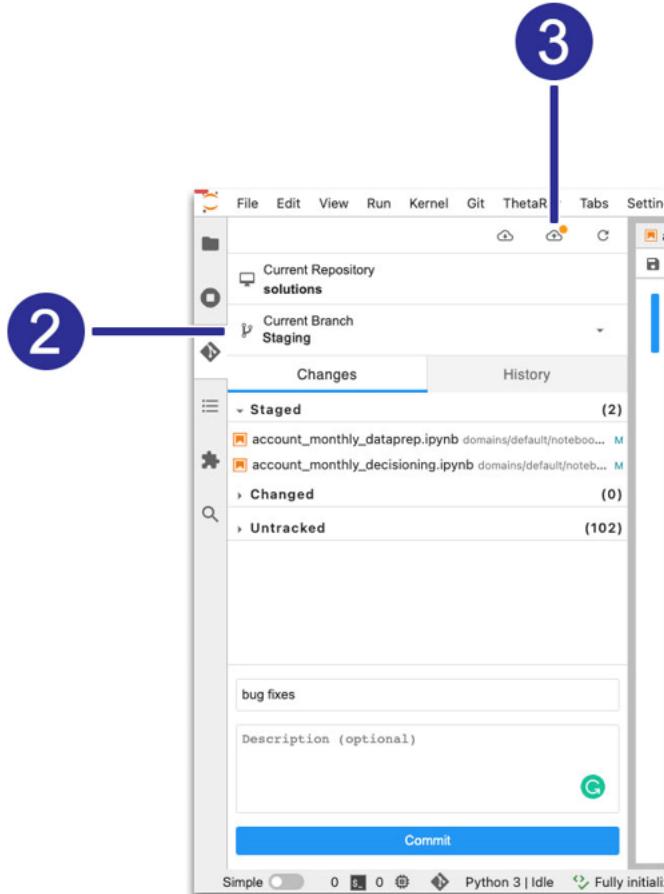
The changed files are in Staged

4. Type the commit message.
5. Click Commit.

The code is committed to the local staging branch.

### 16.1.2. Push the committed code to the Staging branch

1. Click  in the sidebar to display the Gitlab plugin.



2. Select the Staging branch (Only admin users can push to Staging).
3. Click  to push the committed changes and enter your username and password..

After you commit your changes to *Staging*, a metadata sync process will perform the following validation on your code.

- Validate that all mandatory dataset fields in new datasets are included.
- Apply new metadata to data stores schema.
- Merge the content of the staging branch to the operational branch.

An *Airflow* daemon polls the operational branch searching for DAGs and starts to run a scheduled process. If we upgrade during an *Airflow* process run, the process will end on the current version of the database, and the next run will use the new version after the metadata sync has run its course.

## 16.2. Working in Gitlab

An environment-level installation of **GitLab** is accessible from the launcher.

### 16.2.1. View in GitLab

Open the *GitLab* module for any *git* activity you can't perform using the plugin.

1. In the *JupyterLab* menu bar, select *File* -> *New Launcher*
2. Click  the button in the launcher tab.
3. Login with your credentials.
4. Under **Your projects** tab, select the platform instance you're working on.
5. Select the relevant branch in the  box.
6. In the GitLab side panel, click **Repository** and then **Commits** to view the commits history of the branch.
7. In the *GitLab* side panel, click **Repository** and then **Branches** to view the different branches in the environment.
8. In the *GitLab* side panel, click  **Merge requests** to create a merge request.

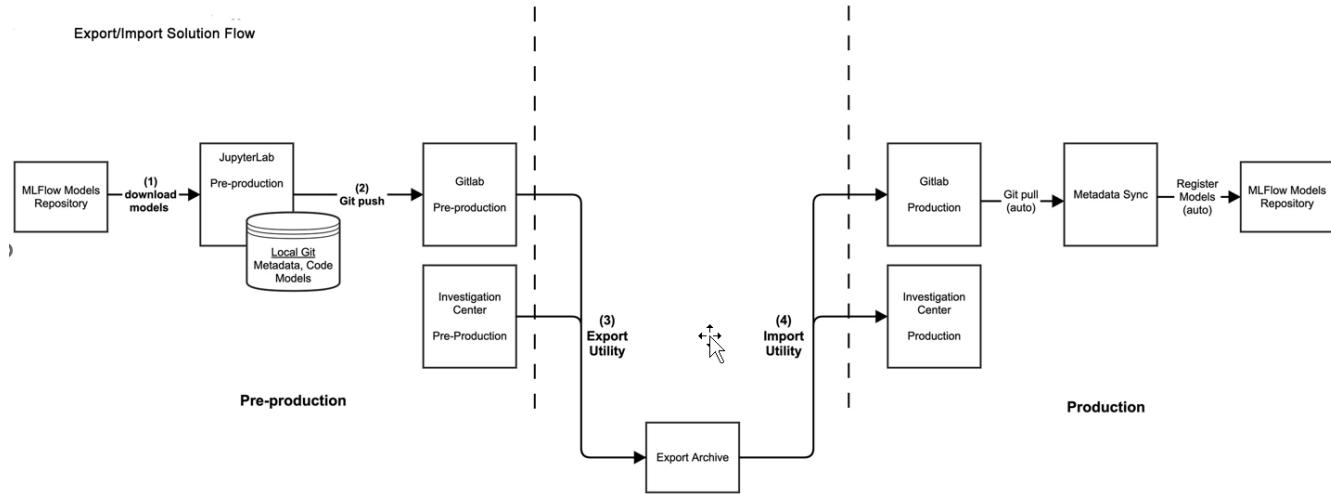
### 16.2.2. Create a merge request

If your user role does not have permission to push new code to the Staging environment, you can issue a merge request in Gitlab

1. In the *GitLab* side panel, click  **Merge requests**.
2. Click **New merge request**.
3. Select the **Source** and **Target** branches on the **New merge** request page.
4. Click **Compare branches** and **continue**.
5. Select the merge request **Reviewer**.
6. Click **Create merge** request.

## 17. Exporting Solution to Another Environment

You can copy a solution from one environment to another by running the export script from the source, transferring the resulting archive to the target environment and then running the import script.



The exported archive includes:

- Jupyter Notebooks, DAGs, Datasets, Connectors, Evaluation flows, and any other file in the instance file hierarchy.
- IC settings.
- *MLflow* models.
- Reference data to be uploaded to target datasets.

**i** The exported archive does not include the platform data stores (MinIO and PostgreSQL).

### 17.1. Prerequisites

The script uses the following CLI commands:

- `mc`: For MinIO operations
- `zip/unzip`

### 17.2. Step 1: Create IC Configuration File

The export script includes the `ic-config-path` option, where you can set the path to a JSON file containing the source IC config parameters; see Step 3: Run the export script.

```
{  
  "importType": "UPDATE",  
  "alertDefinitions": [  
    "1634464539832"  
,  
  "queues": [  
    "1634464540451"  
,  
  "users": [],  
  "exportUserAttributes": true,  
  "exportCustomViews": true,  
  "exportQueueOrder": true  
}
```

To create the IC configuration file, perform the following steps on a *Chrome* browser:

1. Open the Investigation Center (IC) whose setting you want to export.
2. On the top bar, click  and then  Investigation Settings.
3. Click  Export/Import.
4. Set the settings that you want to export.
5. Open *Chrome Developer Tools* and select the **Network** tab.
6. Click **EXPORT**.
7. Select the **Export** process.
8. Select the **Payload** tab.
9. Click **view parsed** and copy the JSON code to a new file.

### 17.3. Step 2: Export MLflow models (optional)

A dedicated function enables you to add *MLflow* model data to the exported GitLab branch. The function saves the models in your environment *GitLab* repository under `git/solutions/models`. After the solution is imported, the models are imported to the target *MLflow* instance by metadata sync process.

To export the *MLflow* models, perform the following:

1. In your Jupyter Notebooks environment, run the **save\_model** function.

```
from thetaray.api.models import download_models  
  
download_models(context, 'tr_analysis', 'tr_evaluation')
```

The function accepts the following parameters:

- **context**: the context object.
- **evaluation\_identifier**: The identifier of the evaluation flow that used the model.
- **evaluation\_step\_identifier**: The identifier of the evaluation flow step that specifies the MLflow model name, tag, and version.

2. Commit the changes to the branch you want to export; see [Manage Code In GitLab](#)

## 17.4. Step 3: Run the export script

1. Run the export script (export-full.sh) with the following options:

- **path**: The export archive destination path.
- **Solution**: The exported ThetaRay platform solution name.
- **git-username**: The exported GitLab repository username.
- **git-password**: The exported Gitlab repository password.
- **Branch**: The exported Branch name (or tag name).
- **git-url**: The exported GitLab Repository URL.
- **ic-url**: The URL of the IC whose settings are exported.
- **ic-username**: The username of the IC whose settings are exported.
- **ic-password**: The password of the IC whose settings are exported.
- **ic-config-path**: The destination path to the IC configuration file.

### 17.4.1. An example of a script execution command

```
./export-full.sh --git-username user --git-password password --ic-username user --ic-  
password password -b Reference --path path/to/destination --git-url gitlab-shared-  
master-20211014.ndp.thetaraydev.com --solution sol --ic-config-path  
path/to/ic/config.json --ic-url ic-app.ndp.thetaraydev.com
```

## 17.5. Step 4: Run the import script

Once the exported archive is finalized, you can transfer it to the destination environment machine.

1. Run the import script (import-full.sh) from the destination environment machine with the following options:

- **path**: The target archive destination path.
- **solution**: The target ThetaRay platform solution name.

- **source-solution**: The source ThetaRay platform solution name.
- **parameters**: Environment parameters in key-value pairs.
- **git-username**: The target *GitLab* repository username.
- **git-password**: The target *Gitlab* repository password.
- **Branch**: The target Branch name (or tag name).
- **git-url**: The target *GitLab* Repository URL.
- **ic-url**: The URL of the IC whose settings are exported.
- **ic-username**: The username of the target IC.
- **ic-password**: The password of the target IC.
- **ic-config-path**: The path to the imported IC configuration file.
- **minio-url**: The target *MinIO* URL
- **minio-access-key**: The target *MinIO* access key
- **minio-secret-key**: The target **MinIO** secret key

```
./import-full.sh --git-username user --git-password password --ic-username user --ic-password password -b test --path /path/to/archive.zip --git-url gitlab-shared-master-20211014.ndp.thetaraydev.com --solution sol --ic-url ic-app.ndp.thetaraydev.com --parameters "distribution_target_identifier='<ic instance uid>'" --source-solution srcsol
```

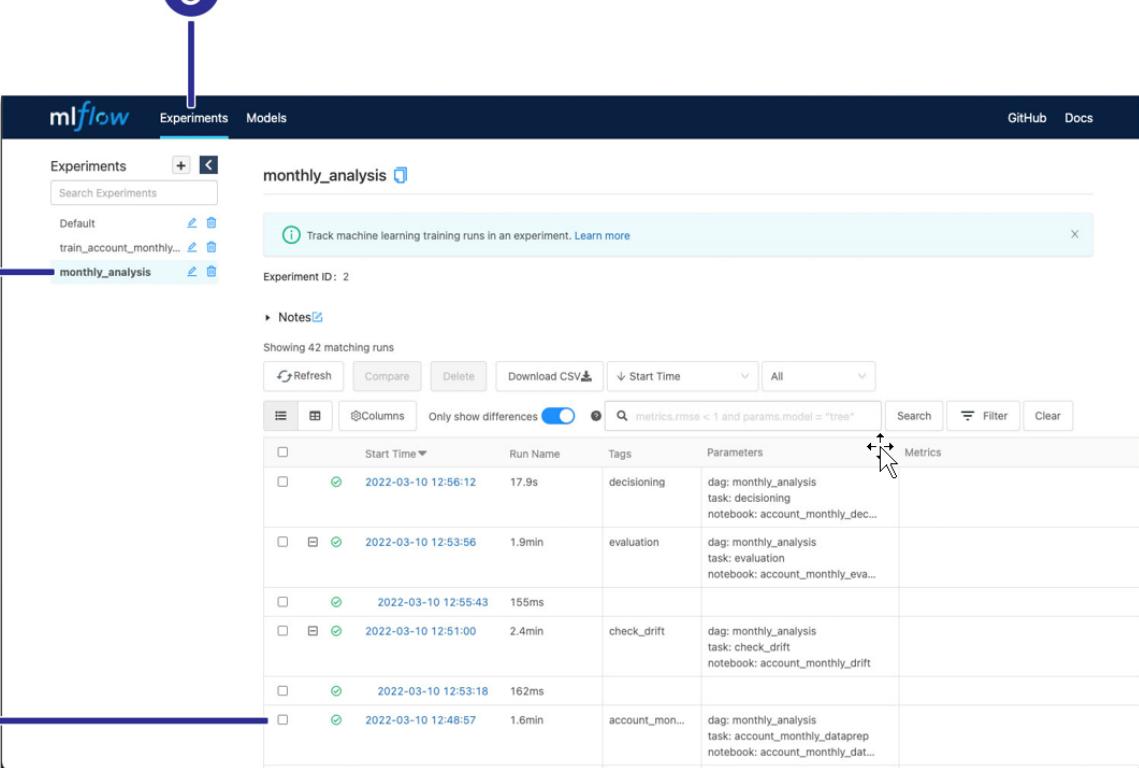
## 18. Review Operational Metrics in MLflow

You can monitor the status and history of the automatic *Airflow* processes in *MLflow*. You can learn the following information:

- The DAG task run status (success/failure), start date, and duration.
- The number of records written, updated or uploaded to each dataset in every *Airflow* task.

An *Airflow* process (the DAG, to use *Airflow* terminology) is represented in *MLflow* as an Experiment and the *Airflow* task as Run

1. In the *JupyterLab* menu bar, select *File* -> *New Launcher*
2. Click the  button in the launcher tab.
3. Select the **Experiments** tab.

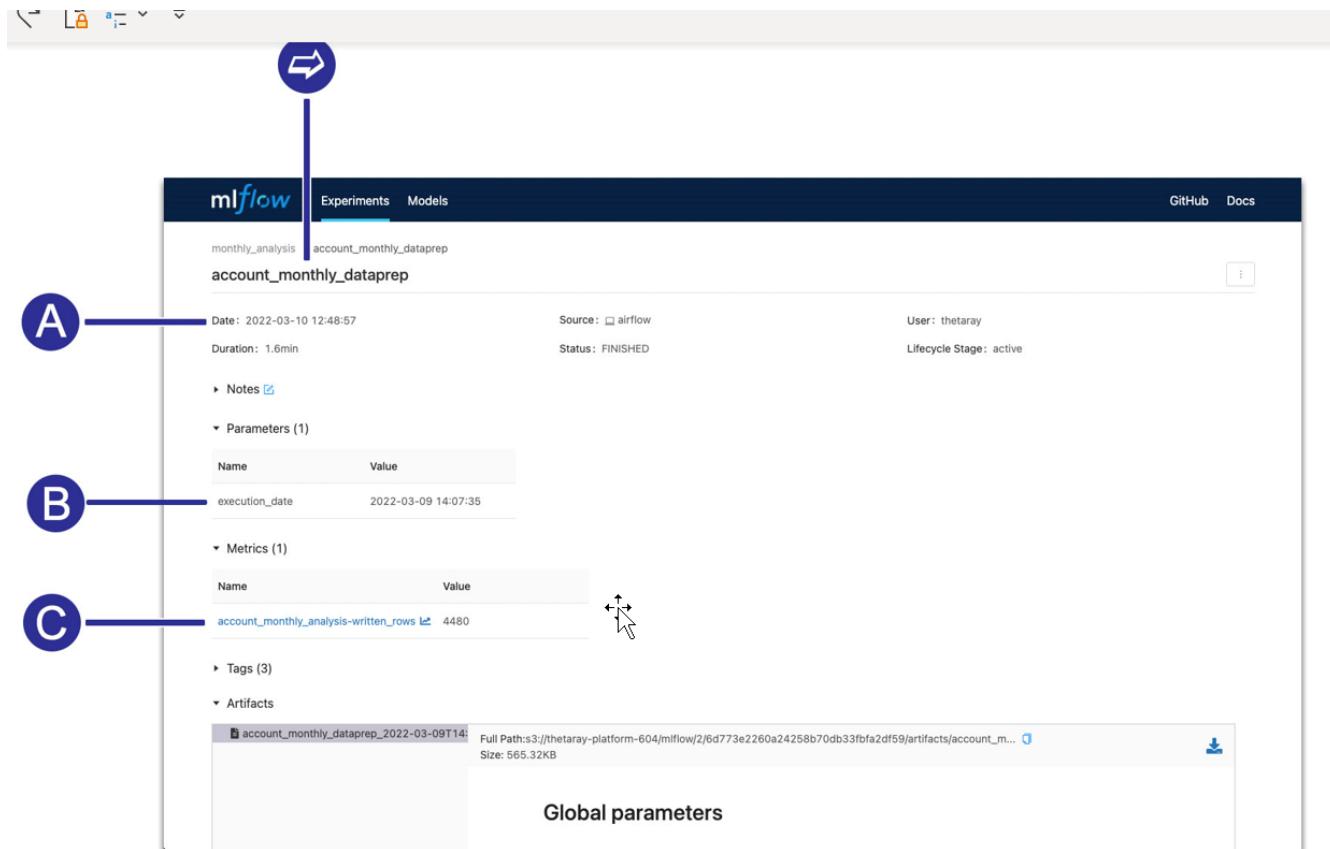


The screenshot shows the MLflow UI with the 'Experiments' tab selected. The main area displays the 'monthly\_analysis' experiment. The table lists 42 matching runs, each with a checkbox, start time, run name, tags, and parameters. A 'Metrics' column is shown on the right, with a cursor pointing at it. The top navigation bar includes 'Experiments' (selected), 'Models', 'GitHub', and 'Docs'.

	Start Time	Run Name	Tags	Parameters	Metrics
<input type="checkbox"/>	2022-03-10 12:56:12	17.9s	decisioning	dag: monthly_analysis task: decisioning notebook: account_monthly_de...	
<input type="checkbox"/>	2022-03-10 12:53:56	1.9min	evaluation	dag: monthly_analysis task: evaluation notebook: account_monthly_eva...	
<input type="checkbox"/>	2022-03-10 12:55:43	155ms			
<input type="checkbox"/>	2022-03-10 12:51:00	2.4min	check_drift	dag: monthly_analysis task: check_drift notebook: account_monthly_drift	
<input type="checkbox"/>	2022-03-10 12:53:18	162ms			
<input type="checkbox"/>	2022-03-10 12:48:57	1.6min	account_mon...	dag: monthly_analysis task: account_monthly_dataprep notebook: account_monthly_dat...	

4. Select the DAG you want to review.  
**i** Each DAG is represented as an MLflow Experiment.
5. Select the task run you want to review.  
**i** Each task is represented as an *MLflow* Run.

The following information is displayed on the *MLflow* run page:



The screenshot shows the mlflow UI for a run named 'account\_monthly\_dataprep'. The run was completed on 2022-03-10 at 12:48:57, with a duration of 1.6min. It was run by airflow and finished successfully. The 'Parameters' section shows an 'execution\_date' parameter set to 2022-03-09 14:07:35. The 'Metrics' section shows a single metric 'account\_monthly\_analysis-written\_rows' with a value of 4480. The 'Artifacts' section lists a single artifact named 'account\_monthly\_dataprep\_2022-03-09T14' with a full path and size information.

**A** Date: 2022-03-10 12:48:57

**B** execution\_date: 2022-03-09 14:07:35

**C** account\_monthly\_analysis-written\_rows: 4480

A. **Date:** The task calendric run date.

B. **execution\_date:** The DAG execution\_date does not represent the calendric run date but is assigned to the DAG run on the run's initial execution and does not change when the run is re-executed.

C. **Metrics:** The number of rows written to the dataset. In addition to this built in metric, user-defined metrics can be added from the Jupyter task notebook using the log\_metric function.

## 19. Publishing Data as a Graph (Support for Network Visualizations)

Data managed in tabular format within the ThetaRay system can be mapped into a graph model to enable visualization and analysis of relations between entities. This task is required when you wish to publish data in graph format to the ***Network Visualizations*** module.

A Graph consists of Nodes connected using directed Edges. A Node encapsulates a certain business entity such as an Account or an Alert Triggered associated with a given account. Relations between nodes such as a monetary transaction between two accounts or the association between an alert and its account are mapped to edges in the graph.

Nodes and Edges within the graph are tagged with a given type (e.g. Account, Transaction, ...) and can be associated with additional information. Moreover, each graph element (node / edge) has an associated 'time' which expresses the time at which KYC information is effective from, for nodes encapsulating an Account or the time at which a transaction occurred for edges.

## 19.1. Defining Graph Metadata

Before data can be published as a graph, the structure of nodes and edges that the graph will consist of should be defined. The information includes the types of nodes and edges that the graph consists of and their associated fields.

Multiple 'Graph Instances' can be defined to enable data segregation between different geographies each associated with a given 'Data Permission' allowing queries done from the Investigation Center to return only data authorized by the user.

The graph metadata should be defined under the 'graphs' directory with the specific domain – below is an example code for a graph definition:

```
from thetaray.api.solution import Graph, NodeType, EdgeType, Property, DataType
from typing import List

from thetaray.api.solution import DataType, EdgeType, Graph, NodePropertyReference, NodeType, Property
def graph1_graph() -> Graph:

    return Graph(
        identifier="public",
        nodes=[
            NodeType(
                identifier="AC",
                display_name="Account",
                description="Bank Account",
                properties=[],
                edges=[],
                subgraphs=[]
            )
        ],
        edges=[]
    )
```

```
encrypted=True,  
audited=True,  
properties=[  
    Property(  
        identifier="AN",  
        display_name="IBAN",  
        description="The account IBAN",  
        type=DataType.STRING,  
        is_key=True,  
        encrypted=True,  
        audited=True,  
    ),  
    Property(  
        identifier="NM",  
        display_name="Full name",  
        description="Display name of the account",  
        type=DataType.STRING,  
    ),  
    Property(  
        identifier="CT",  
        display_name="Country",  
        description="The country of the account",  
        type=DataType.STRING,  
    ),  
    Property(  
        encrypted=True,  
        audit=True,  
    ),  
    Property(  
        identifier="AD",  
        display_name="Address",  
        description="The address of the account",  
        type=DataType.STRING,  
        encrypted=True,  
        audited=False,  
    ),  
],  
,  
NodeType(  
    identifier="AL",  
    display_name="Alerted activity",  
    description="Alerted activity",  
    properties=[  
        Property(  
            identifier="AI",  
            display_name="Alerted activity ID",  
            description="Alerted activity ID",  
        ),  
    ],  
),
```

```
        type=DataType.STRING,
    ),
    Property(
        identifier="RI",
        display_name="Risk identifier",
        description="Alerted activity ID",
        type=DataType.STRING,
    ),
    Property(
        identifier="SP",
        display_name="Suppressed",
        description="Was the alerted activity",
        type=DataType.BOOLEAN,
    ),
],
),
NodeType(
    identifier="PR",
    display_name="Party of accounts",
    description="Party of accounts",
    write_allowed=True,
    properties=[
        Property(
            identifier="PI",
            display_name="Party uuid",
            description="Accounts party uuid",
            type=DataType.STRING,
            is_key=True,
        ),
        Property(
            identifier="NM",
            display_name="Party name",
            description="Accounts party name",
            type=DataType.STRING,
            encrypted=True,
            audited=True,
        ),
        Property(
            identifier="AD",
            display_name="Party address",
            description="Accounts party address",
            type=DataType.STRING,
            reference=NodePropertyReference(node_identifier="AC", property_
identifier="AD"),
            encrypted=True,
            audit=False,
        ),
        Property(

```

```
        identifier="CT",
        display_name="Party country",
        description="Accounts party country",
        type=DataType.STRING,
        reference=NodePropertyReference(node_identifier="AC", property_
identifier="CT"),
    ),
    Property(
        identifier="CN",
        display_name="Consolidation flag",
        description="Consolidation flag",
        type=DataType.LONG,
    ),
    Property(
        identifier="UON",
        display_name="Updated on",
        description="Last updated date",
        type=DataType.TIMESTAMP,
    ),
    Property(
        identifier="UBY",
        display_name="Updated by",
        description="User updated the party",
        type=DataType.STRING,
    ),
],
),
],
edges=[
    EdgeType(
        identifier="TX",
        display_name="Transaction",
        description="The transaction from one account to another",
        properties=[
            Property(
                identifier="AM",
                display_name="Amount",
                description="The transaction amount",
                type=DataType.DOUBLE,
            ),
            Property(
                identifier="CR",
                display_name="Currency",
                description="The transaction currency",
                type=DataType.STRING,
            ),
            Property(
                identifier="CT",
            
```

```
        display_name="Transactions count",
        description="Amount of transactions, represented by the edge",
        type=DataType.LONG,
    ),
],
),
#####
Needed for party implementations for solutions that handle CB payments
EdgeType(
    identifier="AL",
    display_name="Alerting edge",
    description="Edge, connecting alerted activity to account",
    properties=[],
),
# Needed for party implementations for solutions that handle CB payments
EdgeType(
    identifier="ER",
    display_name="EMP Match",
    write_allowed=True,
    description="Edge from party to matched account",
    properties=[
        Property(
            identifier="ST",
            display_name="State",
            description="State of the candidate",
            type=DataType.LONG,
        ),
        Property(
            identifier="SC",
            display_name="Score",
            description="Score of the match",
            type=DataType.DOUBLE,
        ),
    ],
),
#####
Needed for party implementations for solutions that handle retail banking
EdgeType(
    identifier="OW",
    display_name="Owner",
    description="Edge from party to account",
    properties=[],
),
],
data_permission="dpv:public",)

def entities() -> List[Graph]: return [graph1_graph()]
```

There can be two types of implementations to handle parties at Network Visualization

1. CB Payments - in this case you should define ER (Entity Resolution) edge to define matching between account and party. More details can be found in Entity Resolution Chapter of this documentation.
2. Retail Banking - in this case we already know all accounts that our party consist so we don't have to use Entity Resolution. Instead, we just define Owner edge which directs us from Party to relevant Account which Party owns.

Identifiers should be defined using acronyms to save storage space and improve performance.

Currently, identifiers are not flexible in any way and graph should be defined as in example above selection the edge that is relevant for customer implementation. Display Name and Description can be freely edited by customer.

### Encrypting Graph Data

In case application-level data at rest encryption is enabled on the environment, relevant parts of the graph containing sensitive information can be marked as encrypted and optionally audited (indicating that field values are incorporated in encrypted form as part of the data access audit).

Marking a node as encrypted indicates that the node identifier will be stored in encrypted form in the database. Specific properties can be tagged as encrypted, which will in turn store them in encrypted form as part of the JSON representation of properties in the database.

For edges, in a similar manner to nodes, edge identifiers and properties can be encrypted. Moreover, source and target node identifiers will be automatically encrypted if the associated source / target node types are marked as encrypted. It should be noted, that if multiple types of nodes can serve as source / target nodes for an edge, all have to be encrypted or not.

## 19.2. Associating a Graph with an Evaluation Flow

System generated Alerts within the Investigation Center originate from an Evaluation Flow within the platform. As the Network based investigation of an Alert originates from the Graph Node encapsulating the Investigated Entity (e.g. account) – the Evaluation Flow definition is augmented with mapping information that includes a reference to the Graph associated with the Evaluation Flow, as well as mapping between the Primary Key fields of the alert (the investigated entity identifier) and the associated graph node.

The below example maps the account\_id field from the alert to the node having a matching account\_number property within the graph:

```
from typing import List

from thetaray.api.solution import AlgoEvaluationStep, EvaluationFlow, ParquetIndex,
TraceQuery
from thetaray.api.solution.evaluation import (
```

```
EvaluationFlowIdentifiers,  
ModelReference,  
PropertyToFieldMapping,  
RelatedGraph,  
)  
  
def evaluation_flow() -> EvaluationFlow:  
    return EvaluationFlow(  
        identifier="tr_analysis",  
        ...  
        related_graph=RelatedGraph(  
            identifier="public",  
            node_mappings={"AC": [PropertyToFieldMapping(field="account_id",  
property="AN")]},  
            grouping_node_mappings={"PR": [PropertyToFieldMapping(field="party_id",  
property="PI")]},  
        ),  
    )
```

## 19.3. Publishing Nodes & Edges

The following APIs are located in the 'thetaray.api.graph' module to enable data publishing into a graph data model:

### 1. Nodes Publishing

```
def publish_nodes(  
    context: JobExecutionContext,  
    nodes_df: DataFrame,  
    graph_identifier: str,  
    node_type: str,  
    data_environment: DataEnvironment = DataEnvironment.get_default()  
) -> dict
```

The publish\_nodes function publishes the content of a Spark DataFrame to a target graph as graph nodes.

The nodes\_df Spark DataFrame is expected to include the following columns:

- id: textual field with the unique identifier of the node
- effective\_date: DateTime, associated with the node
- column for each property, defined in the NodeType metadata

### 2. Edges Publishing

```
def publish_edges(
    context: JobExecutionContext,
    edges_df: DataFrame,
    graph_identifier: str,
    edge_type: str,
    source_node_type: str,
    target_node_type: str,
    data_environment: DataEnvironment = DataEnvironment.get_default()
) -> dict
```

The `publish_edges` method publishes the content of a **Spark DataFrame** to a target graph as graph edges. In case the source / target nodes referenced by the edge do not exist yet, the method automatically generates nodes within the graph, based on the source / target nodes types configured as part of the graph metadata for the given edge type.

The **`edges_df` Spark DataFrame** is expected to include the following columns:

- `id`: textual field with the unique identifier of the edge
  - `source_node`: textual field with the unique identifier of the source node
  - `target_node`: textual field with the unique identifier of the target node
  - `effective_date`: `DateTime`, associated with the node
  - column for each property, defined in the `EdgeType` metadata
3. Read Nodes

```
def read_nodes(
    context: JobExecutionContext,
    graph_identifier: str,
    type: str,
    data_environment: DataEnvironment = DataEnvironment.get_default(),
    flat_data: bool = False,
)
```

Reads all nodes for given type and returns data stored in Spark DataFrame.

The structure of returned dataframe is:

- `id`: textual field with the unique identifier of the node
- `effective_date`: `DateTime`, associated with the node
- `data`: `Json` represented as `String` which contains all properties stored to the graph

In case if `flat_data` flag is applied to it instead of a `data` column, it returns a `DataFrame` with each property stored in separate column.

4. Read Edges

```
def read_edges(  
    context: JobExecutionContext,  
    graph_identifier: str,  
    type: str,  
    data_environment: DataEnvironment = DataEnvironment.get_default(),  
    flat_data: bool = False,  
)
```

Reads all edges for given type and returns data stored in Spark DataFrame.

The structure of returned dataframe is:

- id: textual field with the unique identifier of the node
- source\_node: textual field with the unique identifier of the source node
- target\_node: textual field with the unique identifier of the target node
- effective\_date: DateTime, associated with the node
- data: Json represented as String which contains all properties stored to the graph

In case if flat\_data flag applied it instead of data column, it returns DataFrame with each property stored in separate column

During publishing make sure you are avoiding duplicates in your graph. Having duplicates will decrease graph performance so it requires handling it on stage of publishing or performing some data governance after. To avoid this problem for APPEND datasets you can simply use correct execution date so only relevant data will be loaded and in case of data in UPDATE/OVERWRITE mode you should filter out nodes/edges that already exists in graph using read\_nodes and read\_edges APIs.

For example:

```
existing_accounts = read_nodes(context=context, graph_identifier="public", type="AC")  
accounts_nodes_df = accounts.withColumn("effective_date", f.to_timestamp(f.lit  
(context.execution_date)))  
accounts_nodes_df = accounts_nodes_df.join(existing_accounts, on="id", how="left_anti")  
accounts_nodes_df = accounts_nodes_df.withColumn("AN", f.col("id"))  
accounts_nodes_df.printSchema()
```

In order to make Network Visualization work only with accounts, you should perform next steps using the above API:

1. Read all accounts, normalize them to expected format and publish as nodes.
2. Read all transactions, define for them source account and target account with assigning relevant properties and publish as edges.

3. Read all alerted and evaluated activities using `read_alerted_activities` and `load_evaluated_activities` APIs and publish alerts nodes and alerted edges assigning relevant properties defined in graph.

To expand implementation with Parties:

1. In case of:
  - a. CB Payments solutions - configure and run Entity Resolution.
  - b. Retail Banking - publish your parties and connect them with accounts using Owner edge.
2. Read and publish all alerted and evaluated activities for party evaluation flow in same way as for account level evaluation flow.

More code examples can be found in reference implementation at `graphing-network-viz.ipynb` notebook.

## 20. Customer Insights Setup

### 20.1. IC Settings

Alert explainability can be enabled in mapper configuration. It can be done in a few steps:

1. Click “Mappers” section in IC settings.
2. Select relevant mapper and go to “Alert Tabs” Section
3. Click “Risk Details”
4. Unwrap “Enable enhanced UI”
5. Turn on “Switch to upgraded view”
6. Configure Customer Insights.

For customer insights you have available widgets configurations which were implemented in platform metadata. Until you don’t implement Evaluation Flow metadata for customer insights you won’t have available widgets to configure.

1. KYC Widget
2. Geographical Activity Widget
3. Transactional Activity Widget
4. Historical Alerts widget

By clicking “Edit” the following can be changed:

1. Name for each widget.
2. Ordering of how widgets will be displayed on alert layout.
3. Select display or hide any of specific widgets.
4. Configure redirect link to any of alert tabs.

Since a new mapper version is created on any configuration change it means that new UI will be applied only to new distributed alerts and old alerts will remain on the original UI.

### 20.2. Customer Insights Platform Configuration

In order to make available customer insights widgets in IC Settings, the metadata and ETL described below should be implemented.

#### 20.2.1. Evaluation Flow Metadata

The relevant evaluation flow of mappers should be enhanced with a new attribute - customer\_insights.

**Python - Full Example:**

```
1  from typing import List
2  from thetaray.api.solution import (
3      AlgoEvaluationStep,
4      EvaluationFlow,
5      ParquetIndex,
6      TraceQuery,
7      CustomerInsights,
8      InsightField,
9      InsightFieldType,
10     InsightPeriod,
11     InsightType,
12     InsightWidget,
13 )
14 from thetaray.api.solution.evaluation import (
15     EvaluationFlowIdentifiers,
16     EvaluationTimeWindowUnit,
17     ModelReference,
18     PropertyToFieldMapping,
19     RelatedGraph,
20 )
21 EvaluationFlow(
22     # all previous attributes...
23     customer_insights=CustomerInsights(
24         dataset="customer_insights",
25         widgets=[
26             InsightWidget(
27                 identifier="kyc",
28                 display_name="KYC Widget",
29                 type=InsightType.KYC,
30                 fields=[
31                     InsightField(
32                         field_ref="kyc_classification",
33                         type=InsightFieldType.CLASSIFICATION,
34                     ),
35                     InsightField(
36                         field_ref="kyc_name", type=InsightFieldType.CUSTOMER_NAME
37                     ),
38                     InsightField(
39                         field_ref="kyc_is_new", type=InsightFieldType.TAG
40                     ),
41                     InsightField(
42                         field_ref="kyc_recently_updated",
43                         type=InsightFieldType.TAG,
44                     ),
45                     InsightField(
46                         field_ref="kyc_newly_incorporation",
47                         type=InsightFieldType.TAG,
```

```
48     ),
49     InsightField(
50         field_ref="kyc_new_customer",
51         type=InsightFieldType.TAG,
52     ),
53     InsightField(
54         field_ref="kyc_occupation",
55     ),
56     InsightField(
57         field_ref="kyc_null_field",
58     ),
59 ],
60 ),
61 InsightWidget(
62     identifier="ga",
63     display_name="Geographical Activity Widget",
64     type=InsightType.GEOGRAPHICAL_ACTIVITY,
65     fields=[
66         InsightField(
67             field_ref="hr_cc", type=InsightFieldType.HIGH_RISK_COUNTRIES
68         ),
69         InsightField(
70             field_ref="mr_cc",
71             type=InsightFieldType.MEDIUM_RISK_COUNTRIES,
72         ),
73         InsightField(
74             field_ref="lr_cc", type=InsightFieldType.LOW_RISK_COUNTRIES
75         ),
76         InsightField(
77             field_ref="director_ad", type=InsightFieldType.ADDRESS
78         ),
79         InsightField(
80             field_ref="company_ad", type=InsightFieldType.ADDRESS
81         ),
82         InsightField(field_ref="some_ad", type=InsightFieldType.ADDRESS),
83     ],
84 ),
85 InsightWidget(
86     identifier="tr",
87     display_name="Transactional Activity Widget",
88     type=InsightType.CUSTOMER_ACTIVITY,
89     fields=[
90         InsightField(field_ref="tr_in", type=InsightFieldType.TRX_IN),
91         InsightField(field_ref="tr_out", type=InsightFieldType.TRX_OUT),
92         InsightField(
93             field_ref="tr_in_count", type=InsightFieldType.TRX_IN_DETAIL
94         ),
```

```
95     InsightField(
96         field_ref="tr_out_count",
97         type=InsightFieldType.TRX_OUT_DETAIL,
98     ),
99     InsightField(
100        field_ref="tr_in_seg",
101        type=InsightFieldType.TRX_POPULATION_IN,
102    ),
103    InsightField(
104        field_ref="tr_out_seg",
105        type=InsightFieldType.TRX_POPULATION_OUT,
106    ),
107    InsightField(
108        field_ref="tr_in_seg_count",
109        type=InsightFieldType.TRX_POPULATION_IN_DETAIL,
110    ),
111 ],
112 period=InsightPeriod(
113     from_ref="trx_from_date",
114     to_ref="trx_to_date"
115 )
116 ),
117 InsightWidget(
118     identifier="alerts",
119     display_name="Historical Alerts Widget",
120     type=InsightType.HISTORICAL_ALERTS,
121     fields=[
122         InsightField(field_ref="tm", type=InsightFieldType.CATEGORY),
123         InsightField(field_ref="scrn", type=InsightFieldType.CATEGORY),
124     ],
125     period=InsightPeriod(
126         from_ref="trx_from_date",
127         to_ref="trx_to_date"
128     )
129     ),
130     ],
131 ),
132 )
```

In the above example, a new class, *CustomerInsights*, which has two key attributes, has been introduced:

**dataset\_reference** - This is the identifier of the dataset that contains the data from which customer insights widgets will be built. This dataset should exist within the solution and contain the same primary key set as the input dataset of the evaluation flow.

Dataset with **UPDATE** or **OVERWRITE** ingestion modes only is supported in current point of time.

A referenced dataset should be assigned to same data permission group (DPV) as the input dataset.

**widgets** - A list of InsightWidget objects. The InsightWidget class allows you to configure specific widgets that will be displayed in the UI.

## 20.2.2. InsightWidget Class

The InsightWidget class represents a widget to be displayed in the customer insights section of the UI. It has the following attributes:

- **identifier (required)**: A unique identifier for the widget within the customer insights configuration.
- **display\_name(required)**: Currently not used.
- **type(required)**: This attribute defines the type of the widget and is specified using the InsightType enum. The type determines the kind of insights the widget will display and how it will be rendered in the UI.

The available InsightType values are:

- **KYC**: Know Your Customer insights.
- **GEOGRAPHICAL\_ACTIVITY**: Geographical Activity insights.
- **HISTORICAL\_ALERTS**: Historical Alerts insights.
- **CUSTOMER\_ACTIVITY**: Customer Activity insights.
- **fields(required)**: A list of InsightField objects. Each InsightField defines a reference to a dataset field specified in the dataset attribute of CustomerInsights. The type of each InsightField defines how the frontend (FE) will interpret and display the relevant field.
- **period (optional)**: An InsightPeriod object that specifies the time period for which the insights are relevant. It contains references to fields in the dataset that define the start (from\_ref) and end (to\_ref) of the period. Note that the period attribute is not applicable for all widget types (e.g., it is not allowed for the KYC widget).

## 20.2.3. InsightField Class

The InsightField class represents a field used in a widget and defines how it should be interpreted and displayed in the UI. It has the following attributes:

- **field\_ref**: A reference to the field in the dataset specified by the dataset\_reference attribute in CustomerInsights.
- **type (optional)**: Specifies how the FE will interpret and display the field. It is defined using the InsightFieldType enum.
- **display\_name (optional)**: Allows overriding the display name inherited from dataset field.
- **description (optional)**: Allows overriding the description inherited from dataset field.

### 20.2.3.1. InsightFieldType Enum

The InsightFieldType enum defines the types available for each field. It associates each field type with an InsightType (i.e., the widget type) and specifies any validations that apply to the field.

### 20.2.4. Field Types and Validations

Each InsightFieldType may have associated validations, which include:

- **REQUIRED:** The field is mandatory and must be provided.
- **UNIQUE:** The field must be unique within its context (i.e., there should not be multiple fields of this type in the same widget).

Below are the available InsightFieldType values, along with their associated InsightType, validations, and expected data formats:

### 20.2.5. For KYC Widget (InsightType.KYC)

- CLASSIFICATION
  - Description: Defines customer classification.
  - Validations: REQUIRED, UNIQUE.
  - Expected Values: "Low", "Medium", "High".
- CUSTOMER\_NAME
  - Description: Defines the customer's name.
  - Validations: REQUIRED, UNIQUE.
- TAG
  - **Description:** Defines tags for the KYC widget (e.g., "New Customer", "Recently Updated").
  - **Validations:** REQUIRED.
  - Text for (I) icon for tag is taken from description.
- **No type**
  - If no type is specified for field in KYC widget it will be simply displayed as key-value pair
    - display\_name: dataset column value

The following image shows an example of how a no data type will be displayed for KYC data:

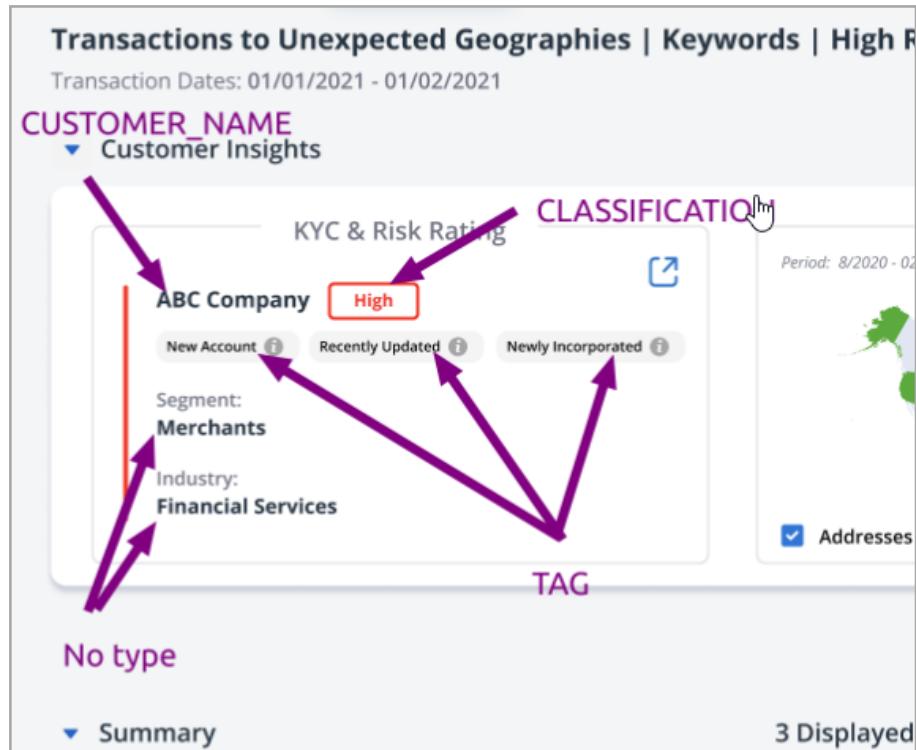


Figure 16: Example Indication of how for KYC, No type Data is Displayed as Key: Value Pairs

## 20.2.6. For Geographical Activity Widget (InsightType.GEOGRAPHICAL\_ACTIVITY)

- **HIGH\_RISK\_COUNTRIES**
  - **Description:** Contains a list of high-risk country codes.
  - **Validations:** REQUIRED, UNIQUE.
  - **Data Format:**
    - **Type:** Array of country codes stored as string
    - **Example:** ["US", "IR", "KP"].
- **MEDIUM\_RISK\_COUNTRIES**
  - **Description:** Contains a list of medium-risk country codes.
  - **Validations:** REQUIRED, UNIQUE.
  - **Data Format:**
    - **Type:** List of country codes.
    - **Example:** ["MX", "BR", "IN"]
- **LOW\_RISK\_COUNTRIES**
  - **Description:** Contains a list of low-risk country codes.
  - **Validations:** REQUIRED, UNIQUE.
  - **Data Format:**

- Type: List of country codes.
- Example: ["CA", "AU", "DE"].
- ADDRESS
  - **Description:** Defines an address for geographical activity insights.
  - **Validations:** None (optional field).
  - **Data Format:**
    - Type: JSON string.
    - Structure:

```
1 | { "AD": "address", "CC": "country_code", "CL": "classification" }
```

#### Fields:

- AD: Address.
- CC: Country Code.
- CL: Classification ("High", "Medium", "Low")

### 20.2.6.1. For Historical Alerts Widget (InsightType.HISTORICAL\_ALERTS)

- CATEGORY - Each category field defines single pie chart of drop down. Name for section of dropdown is defined in field display name. Total alerts is sum of values in stored JSON string
  - **Description:** Contains data to build a historical alerts pie chart.
  - **Validations:** REQUIRED.
  - **Data Format:**

Type: JSON string representing alert states and their counts.

Example:

```
1 | { "Opened": 10, "Closed": 5, "False Positives": 0 }
```

### 20.2.6.2. For Customer Activity Widget (InsightType.CUSTOMER\_ACTIVITY)

- TRX\_IN
  - **Description:** Sum of incoming transactions over a period.
  - **Validations:** REQUIRED, UNIQUE.
  - **Data Format:**

- **Type:** Numeric value.
- **Example:** 15000.00
- TRX\_OUT
  - **Description:** Sum of outgoing transactions over a period.
  - **Validations:** REQUIRED, UNIQUE.
  - **Data Format:**
    - Type: Numeric value.
    - Example: 12000.00
- TRX\_IN\_DETAIL
  - **Description:** Details to display on chart hover for incoming transactions. Will be displayed as key-value pair - display\_name: value
  - **Validations:** None (optional field).
  - **Data Format:** Can include any additional details as required.
- TRX\_OUT\_DETAIL
  - **Description:** Details to display on chart hover for outgoing transactions. Will be displayed as key-value pair - display\_name: value
  - **Validations:** None (optional field).
  - **Data Format:** Can include any additional details as required.
- TRX\_POPULATION\_IN
  - **Description:** Incoming transactions of a population segment.
  - **Validations:** UNIQUE (optional field).
  - **Data Format:** Numeric value.
- TRX\_POPULATION\_OUT
  - **Description:** Outgoing transactions of a population segment.
  - **Validations:** UNIQUE (optional field).
  - **Data Format:** Numeric value.
- TRX\_POPULATION\_IN\_DETAIL
  - **Description:** Details of population incoming transactions for chart hover. Will be displayed as key-value pair - display\_name: value
  - **Validations:** None (optional field).
  - **Data Format:** Can include any additional details as required.
- TRX\_POPULATION\_OUT\_DETAIL
  - **Description:** Details of population outgoing transactions for chart hover. Will be displayed as key-value pair - display\_name: value
  - **Validations:** None (optional field).

### 20.2.7. InsightPeriod Class

The InsightPeriod class represents a period for an insight widget and is used to define the time range over which the data is relevant. It has the following attributes:

- `from_ref`: Reference to the field in the dataset that contains the start date of the period.
- `to_ref`: Reference to the field in the dataset that contains the end date of the period.

Period can be defined for all widgets beside KYC.

Fields can be re-used as period for different widgets (I.E. you can define single from to and use it as reference duplicate reference for each widget) same as each widget can reference to relevant period columns.

## 20.3. Dataset Metadata

Dataset metadata should be build according to configuration defined in evaluation flow. All fields referenced should be calculated and stored to dataset.

Few considerations:

1. Dataset should be in UPDATE mode.
2. Primary key set should be same as input dataset of evaluation flow.
3. DPV should be same as DPV of input dataset.
4. To display currency at Customer Activity widget for sum of transactions `business_type=BusinessType.CURRENCY` and relevant units value should be assigned. For example:

```
1 Field(  
2     business_type=BusinessType.CURRENCY,  
3     units="USD",  
4     identifier="tr_out",  
5     display_name="Transaction Out",  
6     data_type=DataType.DOUBLE,  
7 ),
```

## 20.4. ETL Process for Customer Insights Dataset

To enable the Customer Insights widgets in the UI, you need to prepare a separate dataset that provides all the necessary data. This dataset should be calculated independently from the main analysis flow and must include all relevant fields referenced in your customer insights configuration.

## 20.4.1. Calculating Relevant Fields

Ensure that all fields used in the Customer Insights widgets are calculated and populated correctly. This includes any fields referenced in the InsightField configurations within your EvaluationFlow metadata.

## 20.4.2. Accessing Historical Alerts Data

For widgets that display historical alerts data (e.g., the **Historical Alerts** widget), you need to access the alerts data stored in the system. To retrieve this data, you must connect to the database using specific environment variables for authentication:

- POSTGRES\_REPORT\_USER\_USERNAME
- POSTGRES\_REPORT\_USER\_PASSWORD

---

**Note:** These credentials are available only if pgBouncer is enabled in your environment. If pgBouncer is not enabled, please contact support to grant the necessary permissions to the user specified by CDD\_POSTGRES\_USERNAME.

---

### 20.4.2.1. Querying the Alerts Data

To access the alerts data:

1. **Connect to the database** using the provided credentials.
2. **Query the view *rp\_alert\_fields***, filtering by the relevant mapper identifier to retrieve alerts specific to your mapper configuration.

The *rp\_alert\_fields* view contains a log of changes for each alert. To obtain the most recent data for each alert, you should:

- Join the records using the primary keys.
- Select the latest record available for each specific alert.

### 20.4.2.2. Processing Resolution Codes

The *rp\_alert\_fields* view includes a column named *resolution\_code*, which indicates the resolution status of each alert. To correctly display data in the pie chart (e.g., in the Historical Alerts widget), you need to map these resolution codes to alert states.

Here is an example of how to map resolution codes to alert states:

```
1 | if not resolution_code:
2 |     state = "Open"
3 | elif resolution_code in non_issue_resolution_codes:
```

```
4     state = "False Positives"
5 else:
6     state = "Closed"
```

- **Open:** Alerts with no resolution\_code.
- **False Positives:** Alerts with a resolution\_code that is in your predefined list of non-issue resolution codes (non\_issue\_resolution\_codes).
- **Closed:** All other alerts.

**Note:** Examine the alert resolution codes configured in your Incident Case Management (IC) system to create the non\_issue\_resolution\_codes list that defines which codes correspond to false positives.

#### 20.4.2.3. Additional Considerations

- The dataset is queried by the tr\_timestamp column. Assign a relevant value to this field, such as the execution date of your ETL process, to ensure accurate querying.
- When the IC retrieves records, it returns the record that is closest to the alert's occurred\_on value. We select the first entry where tr\_timestamp is less than occurred\_on (tr\_timestamp < occurred\_on). So ensure your data accommodates this logic to display the correct records in the UI.

### 20.5. Analysis Method in Risk

An additional filter on alerts called Analysis Type is now supported. This value comes from the risk configuration and must be set during migration to the enhanced UI for all risks used in your solution.

**Note:** The analysis method determines the layout of the alert (either rule or AI). If analysis method is not configured, the alert will not render the redesigned view properly, so analysis method must be configured for all risks when moving to the redesigned view.

To support this, a new attribute `analysis_method` has been added to the risk YAML configuration. It is an enum with the following supported values:

- AI: Alert was triggered based on model output.
- RULE: Alert was triggered by strictly defined rules.
- RT\_RULE: Alert was triggered by strictly defined rules using real-time capabilities.

## 20.6. Feature Explainability Configuration

### 20.6.1. Overview

To enhance feature explainability within your solution, new metadata attributes have been introduced to the `Field` class in your dataset configuration.

The widgets described in this section will be displayed at any alerted activity that was triggered by risk (rule) defined with AI `analysis_method`.

The `is_explainability_column` attribute allows you to mark certain fields in your input dataset as explainability columns. These addition allow you to include helper data that is not directly used in detection or decisioning but is essential for providing detailed explanations in the UI.

### 20.6.2. Example of such Field metadata

```

1  Field(
2      data_type=DataType.STRING,
3      identifier="keyword_matches_explainability",
4      display_name="Keyword Matches Explainability",
5      is_explainability_column=True,
6  ),
7  Field(
8      display_name="name",
9      data_type=DataType.STRING,
10     identifier="name",
11     encrypted=True,
12     is_explainability_column=True,
13 )

```

To efficiently handle the potentially large data associated with explainability fields (e.g., reasonably sized JSON strings used for populating categorical widgets like pie charts), these fields are stored in a separate PostgreSQL table. This table is automatically created and follows a naming pattern based on the evaluation flow identifier:

```
1 | {evaluation_flow_identifier}_expl (e.g., tr_analysis_expl)
```

During the *publish\_evaluated\_activities* API call, values for these explainability columns are written to this separate table instead of the main evaluation flow table. This separation ensures better performance and scalability when dealing with large explainability data.

After determining which widgets for which features are required based on project needs, you can proceed to configure the explainability metadata using additional attributes provided in the Field class:

1. **business\_type and units** - These attributes are used together to enhance a feature with additional contextual meaning, providing more information to the end-user.
  - a. **business\_type**:
    - i. An enum currently supporting a single value: CURRENCY.
    - ii. Indicates the type of business data the feature represents.
  - b. **units**:
    - i. Specifies the unit associated with the feature, such as the currency code (USD, EUR, etc.) when **business\_type** is CURRENCY.
2. **explainabilities** - a list of classes that define the explainability (widget) configuration for the feature.
3. **dynamic\_description** - A Jinja template string that enables the configuration of dynamic descriptions for the feature. Similar to risk dynamic tagging but offers extended capabilities.

### 20.6.3. Explainabilities

In this chapter, we will dive into the configuration of explainability widgets, which enhance the user interface by providing detailed explanations for features used during detection. You can configure multiple widgets per feature by populating the **explainabilities** attribute of the Field class. This attribute accepts a list of Explainability class instances, each defining a specific widget and its properties.

#### 20.6.3.1. Explainability Class

The Explainability class represents a single explainability widget and has the following attributes:

- **identifier**: A unique identifier for the explainability widget.

- **type**: Defines the type of the explainability widget. The allowed values are specified in the ExplainabilityType enum.
- **json\_column\_reference (optional)**: An optional reference to another field (marked with is\_explainability\_column=True) from which the widget will retrieve its data. Each key that is intended to be used from this JSON should be described using the values attribute.
- **time\_range\_value (required for certain widget types)**: The numerical value representing the time range for the widget (e.g., 6 for 6 months).
- **time\_range\_unit (required for certain widget types)**: The unit of time for time\_range\_value (e.g., TimeRangeUnit.DAY/WEEK/MONTH/YEAR).
- **category\_lbl (required for certain widget types)**: The key that defines the category label in the data.
- **category\_var (required for certain widget types)**: The key that defines the variable used for the widget (e.g., the value used to build a pie chart).
- **priority (optional)**: Determines the display priority of the widget when multiple widgets are defined for a feature. The widget with the highest priority will be displayed in the alert page, while others will appear on the feature's drill-down page.
- **values**: A list of ExplainabilityValueProperties instances that define how the widget's data is constructed and displayed.

### 20.6.3.2. ExplainabilityType Enum

The ExplainabilityType enum defines the types of explainability widgets available:

- **CATEGORICAL**: Represents a categorical widget, such as a pie chart.
- **HISTORICAL**: Represents a historical widget, such as a trend chart over time.
- **BINARY**: Represents a binary widget, typically displaying yes /no or true /false information.
- **BEHAVIORAL**: Represents a behavioral widget, combining aspects of historical and categorical data.

### 20.6.3.3. Attribute Validations Based on ExplainabilityType

Depending on the type of the explainability widget, certain attributes are required or must not be set. The validation rules are as follows:

- For HISTORICAL Widgets:
  - Required Attributes: time\_range\_value, time\_range\_unit
- For BINARY Widgets:
  - Attributes Not Allowed: time\_range\_value, time\_range\_unit, category\_lbl, category\_var
- For CATEGORICAL Widgets:

- Required Attributes: category\_lbl, category\_var
- Attributes Not Allowed: time\_range\_value, time\_range\_unit
- For BEHAVIORAL Widgets:
  - Required Attributes: time\_range\_value, time\_range\_unit, category\_lbl, category\_var

The validation ensures that the necessary attributes for each widget type are provided and that prohibited attributes are not set. If these rules are not followed, a validation error will be raised.

## 20.6.4. ExplainabilityValueProperties Class

The ExplainabilityValueProperties class defines the properties of each value used in the widget. It has the following attributes:

- **key**: A unique key (up to 4 characters) that identifies the value within the widget. This key is used to reference the value introduced in order to have shortened conventions for json keys stored in explainability columns and also used in REST API JSON. In order to understand how this key should be used FE reads other attrs of this class and acts accordingly.
- **name**: The display name of the value, used in the UI.
- **dynamic\_value** (optional) A reference to a field in the input dataset or explainability column. The reference is specified using dot notation (e.g., 'activity.amount' or 'activity.customer\_name').
- **static\_value** (optional): A static value to display, commonly used for thresholds or constant values (e.g., 50000 or "High Risk").
- **type** (optional): Defines how the FE will interpret and display the value. It is specified using the ExplainabilityValueType enum.
- **filter** (optional): Allows filtering of values based on simple conditions (e.g., displaying only countries classified as "High Risk").

### Validations:

- Only one of dynamic\_value or static\_value should be defined. Defining both will raise a validation error.
- The length of the key must not exceed 4 characters.

### 20.6.4.1. ExplainabilityValueType Enum

The ExplainabilityValueType enum specifies the type of the value, informing the frontend how to display it. The allowed values and their associated data types are:

- **COUNTRY**: Expected data type str. The value represents a country code, and the frontend may display it with a flag.

- **CRITICALITY:** Expected data type str. Indicates the criticality level (We expect some specific values to be set for this value - "Very High", "High", "Medium", "Low").
- **TITLE:** Expected data type str. Used for title texts in the binary widget.
- **DESCRIPTION:** Expected data type str. Used for description texts in the binary widget.
- **TREND:** Expected data type float or int. Used for trend values in historical and behavioral charts.
- **THRESHOLD:** Expected data type float or int. Represents a threshold value in historical and behavioral charts.
- **POPULATION:** Expected data type float or int. Used for population/segment metrics in historical and behavioral charts.
- **COUNT:** Expected data type float or int. Represents a count of items.
- **SUM:** Expected data type float or int. Represents a sum of values.

For each ExplainabilityType, there are specific requirements and restrictions on the ExplainabilityValueType values used in the values attribute:

- For HISTORICAL Widgets:
  - Required Value Types: TREND
  - Value Types Not Allowed: TITLE, DESCRIPTION
  - Other Value Types: Optional
- For BINARY Widgets:
  - Required Value Types: TITLE, DESCRIPTION
  - Value Types Not Allowed: TREND, THRESHOLD, POPULATION
  - Other Value Types: Optional
- For CATEGORICAL Widgets:
  - Value Types Not Allowed: TREND, THRESHOLD, POPULATION, TITLE, DESCRIPTION
  - Other Value Types: Optional
- For BEHAVIORAL Widgets:
  - Required Value Types: TREND
  - Value Types Not Allowed: TITLE, DESCRIPTION
  - Other Value Types: Optional

The system validates that the required ExplainabilityValueType values are present and that prohibited types are not included in the values list for the widget. If these rules are violated, a validation error will occur.

The ExplainabilityValuesFilter class allows you to filter the data displayed in the widget based on simple conditions. This is particularly useful when you want to display only certain values from a json\_colum\_reference (e.g., countries classified as "High").

- values (List[Any]): A list of values to filter on.
- type (ExplainabilityValuesFilterType): The type of comparison to perform. It can be either EQ (equal) or NEQ (not equal) for now.

## 20.6.5. Examples of Widget Configurations

### 20.6.5.1. BINARY Widget (Customer is PEP)

```
1  Field(
2      identifier="is_pep",
3      display_name="Customer is PEP",
4      data_type=DataType.BOOLEAN,
5      # ... other attributes ...
6      business_type=BusinessType.CURRENCY,
7      units="USD",
8      explainabilities=[
9          Explainability(
10             identifier="binary_expl",
11             type=ExplainabilityType.BINARY,
12             values=[
13                 ExplainabilityValueProperties(
14                     key="cn",
15                     name="Customer Name",
16                     dynamic_value="name",
17                     type=ExplainabilityValueType.TITLE,
18                 ),
19                 ExplainabilityValueProperties(
20                     key="ca",
21                     name="Customer Address",
22                     dynamic_value="address",
23                     type=ExplainabilityValueType.DESCRIPTION,
24                 ),
25                 ExplainabilityValueProperties(
26                     key="cip",
27                     name="Customer is PEP",
28                     dynamic_value="is_pep",
29                 ),
30                 ExplainabilityValueProperties(
31                     key="ts",
32                     name="Transactions Sum",
33                     dynamic_value="amount",
34                     type=ExplainabilityValueType.SUM,
35                 ),
36             ],
37         )
```

```
38     ],
39 ),
40
```

**Notes:**

- The key "cn" is a generic value used to optimize the response on the web service side.
- The name attribute provides the display name for each key.
- The dynamic\_value references the column from the activity dataset.
- Setting type=ExplainabilityValueType.TITLE informs the frontend that this value should be used as a title of binary widget and ExplainabilityValueType.DESCRIPTION is a description.
- The ExplainabilityValueProperties instance with key="cip" has no type specified, so it will be displayed as a simple key-value pair like Customer is PEP: true.

### 20.6.5.2. CATEGORICAL Widget (High risk countries)

```
1 Field(
2     data_type=DataType.BOOLEAN,
3     identifier="high_risk_countries",
4     display_name="High Risk Countries",
5     description="Accumulation of transactions sent to high-risk countries.",
6     # ... other attributes ...
7     business_type=BusinessType.CURRENCY,
8     units="EUR",
9     explainabilities=[
10         Explainability(
11             identifier="hrc_categorical",
12             type=ExplainabilityType.CATEGORICAL,
13             category_lbl="ct",
14             category_var="s",
15             json_column_reference="high_risk_countries_explainability",
16             values=[
17                 ExplainabilityValueProperties(
18                     key="ct",
19                     name="Receiver Country",
20                     type=ExplainabilityValueType.COUNTRY,
21                 ),
22                 ExplainabilityValueProperties(
23                     key="s",
24                     name="Transactions Sum",
25                     type=ExplainabilityValueType.SUM,
26                 ),
27                 ExplainabilityValueProperties(
```

```
28         key="c",
29         name="Transactions Count",
30         type=ExplainabilityValueType.COUNT,
31     ),
32     ExplainabilityValueProperties(
33         key="cr",
34         name="Criticality",
35         type=ExplainabilityValueType.CRITICALITY,
36         filter=ExplainabilityValuesFilter(
37             type=ExplainabilityValuesFilterType.EQ,
38             values=["Very High", "High"],
39         ),
40     ),
41 ],
42 ],
43 ],
44 )
```

#### Notes:

- category\_lbl="ct" specifies that the key "ct" defines the category label.
- category\_var="s" specifies that the key "s" defines the variable used to build the pie chart.
- json\_column\_reference="high\_risk\_countries\_explainability" indicates that the data for this widget was pre-computed and stored in the high\_risk\_countries\_explainability column as a JSON string.
- For key="ct", type=ExplainabilityValueType.COUNTRY means that country codes are stored here, and the frontend will render them as flags. It can also be defined as a simple string without any type.
- The filter applied to key="cr" specifies that only countries with the classifications "Very High" or "High" should be displayed.

#### 20.6.5.3. Simpler CATEGORICAL Widget Example (Keyword Matches)

```
1 explainabilities=[
2     Explainability(
3         identifier="km_categorical",
4         type=ExplainabilityType.CATEGORICAL,
5         category_lbl="kw",
6         category_var="s",
7         json_column_reference="keyword_matches_explainability",
8         values=[
9             ExplainabilityValueProperties(
10                 key="kw",
```

```
11         name="Keyword Group",
12     ),
13     ExplainabilityValueProperties(
14         key="s",
15         name="Sum of Transactions",
16         type=ExplainabilityValueType.SUM,
17     ),
18     ExplainabilityValueProperties(
19         key="c",
20         name="Transactions Count",
21         type=ExplainabilityValueType.COUNT,
22     ),
23 ],
24 )
25 ]
```

**Notes:**

- This example represents a categorical widget for keyword matches.
- category\_lbl="kw" indicates that the key "kw" defines the category label (the keyword group).
- category\_var="s" indicates that the key "s" defines the variable used to build the chart (sum of transactions).
- json\_column\_reference="keyword\_matches\_explainability" means the data is stored in the keyword\_matches\_explainability column.

#### 20.6.5.4. HISTORICAL Widget (Amount of money sent by customer for last 6 months)

```
1 Field(
2     display_name="Amount",
3     data_type=DataType.LONG,
4     identifier="amount",
5     description="This is the aggregated amount of transactions for the account within
6     the analysis period.",
7     # ... other attributes ...
8     units="EUR",
9     business_type=BusinessType.CURRENCY,
10    explainabilities=[
11        Explainability(
12            identifier="hist_expl",
13            type=ExplainabilityType.HISTORICAL,
14            time_range_value=6,
```

```
14     time_range_unit=TimeRangeUnit.MONTH,
15     values=[
16         ExplainabilityValueProperties(
17             key="a",
18             name="Transactions Trend",
19             dynamic_value="amount",
20             type=ExplainabilityValueType.TREND,
21         ),
22         ExplainabilityValueProperties(
23             key="st",
24             name="Segment Trend",
25             dynamic_value="population_amount",
26             type=ExplainabilityValueType.POPULATION,
27         ),
28         ExplainabilityValueProperties(
29             key="tr",
30             name="Threshold",
31             static_value=50_000,
32             type=ExplainabilityValueType.THRESHOLD,
33         ),
34     ],
35 ],
36 ],
37 ),
38 ]
```

## Notes:

- This widget displays a trend of the aggregated transaction amounts sent by a customer over the last 6 months.
- Assuming the analysis period is 1 month, we expect to see 6 data points for 6 months if 6 cycles were run.
- time\_range\_value=6 and time\_range\_unit=TimeRangeUnit.MONTH specify the time range for the trend.
- The ExplainabilityValueProperties include:
  - key="a" for the customer's transactions trend.
  - key="st" for the segment (population) trend.
  - key="tr" for a static threshold value of 50,000 EUR.
- Data is queried based on tr\_partition column.

## 20.6.5.5. BEHAVIORAL Widget (Amount of money categorized by country sent by customer during last 6 month)

```
1  Field(
2      display_name="Amount",
3      data_type=DataType.LONG,
4      identifier="amount",
5      description="This is the aggregated amount of transactions for the account within
6      the analysis period.",
7      # ... other attributes ...
8      units="EUR",
9      business_type=BusinessType.CURRENCY,
10     explainabilities=[
11         Explainability(
12             identifier="hrc_behavioral",
13             type=ExplainabilityType.BEHAVIORAL,
14             time_range_value=6,
15             time_range_unit=TimeRangeUnit.MONTH,
16             category_lbl="ct",
17             category_var="s",
18             json_column_reference="high_risk_countries_explainability",
19             values=[
20                 ExplainabilityValueProperties(
21                     key="ct",
22                     name="Country of Receiver",
23                     type=ExplainabilityValueType.COUNTRY,
24                 ),
25                 ExplainabilityValueProperties(
26                     key="s",
27                     name="Sum of Transactions",
28                     type=ExplainabilityValueType.TREND,
29                 ),
30                 ExplainabilityValueProperties(
31                     key="cr",
32                     name="Criticality",
33                     type=ExplainabilityValueType.CRITICALITY,
34                 ),
35                 ExplainabilityValueProperties(
36                     key="c",
37                     name="Count of Transactions",
38                     type=ExplainabilityValueType.COUNT,
39                 ),
40             ],
41         ],
42     ),
```

### Notes:

- This widget is a combination of categorical and historical widgets.
- It shows numerical values split across categories (e.g., countries) over time.
- An example use case is displaying how much money, categorized by countries, was sent by a customer during the analysis period.
- `time_range_value=6` and `time_range_unit=TimeRangeUnit.MONTH` specify that data over the last 6 months should be included.
- `category_lbl="ct"` and `category_var="s"` are used to define the categories and variables for the widget.
- `json_column_reference="high_risk_countries_explainability"` indicates where the data is sourced from.

## 20.7. Dynamic Description

### 20.7.1. Overview

Dynamic descriptions are Jinja templates rendered during API calls to the platform's API service. They allow you to create rich, contextual descriptions for features by incorporating dynamic data from your evaluation input and result together with metadata.

In a dynamic description template for a feature, you can use the following dynamic values:

- `activity.{input_dataset_column_name}`: Access any column value from the input dataset.
- `metadata.{feature_name}.{metadata_attribute}`: Access metadata of any feature. For example, when writing a dynamic description for the `customer_is_pep` feature, you can access metadata of the `amount` feature. Available metadata attributes include:
  - `display_name`
  - `category`
  - `description`
  - `business_type`
  - `units`
  - `explainabilities.{explainability_id}.{any_explainability_attribute}`
- `explainability.{feature_name}.{explainability_id}`: Access processed data from any explainability.

You are allowed to use Markdown in dynamic descriptions. For example, **\*\*Text\*\*** will render as bold.

### 20.7.1.1. Example 1: Basic Dynamic Description

```
1 Customer **{{ activity.name }}** born on **{{ activity.birth_number }}**  
2 {% if activity.is_pep is True %}  
3     has status of Politically Exposed Person.  
4 {% else %}  
5     is not Politically Exposed Person.  
6 {% endif %}  
7 This customer spent **{{ activity.amount | prettyf_number }}{{ metadata.amount.units |  
8 currency_symbol }}**  
** for this analysis period.
```

This is a very basic example of a dynamic description. Here, we are using activity columns; some of them are features, and some can be columns marked with `is_explainability_column`.

We also see the usage of the amount feature's metadata, specifically the currency units.

Additionally, we use some custom functions exposed in the template—`prettyf_number` and `currency_symbol`. Descriptions of these functions are provided below.

When rendered, this template might look like:

Customer **Name Surname** born on **01/01/1970** has status of Politically Exposed Person.

This customer spend **1000\$** for this analysis period.

### 20.7.1.2. Example 2: Dynamic Description for Categorical Widget

```
1 {% set expl_data = explainability.keyword_matches.km_categorical %}  
2 The sum of customer's transactions with matching keywords is  
3 {{ expl_data | sum(attribute='s') | prettyf_number }}  
4 {{ metadata.keyword_matches.units | currency_symbol }}.  
5 Keywords are: **{{ expl_data | map(attribute='kw') | join(',') }}**.
```

This is an example of a dynamic description for a categorical widget. Here, we are accessing explainability data and calculating the sum of transactions that matched specific keywords using Jinja syntax. Then, we collect a list of keywords and display them as comma-separated values.

The usage of `{% set %}` demonstrates how to optimize dynamic description code by declaring variables.

When rendered, this template might look like:

The sum of customer's transactions with matching keywords is 123,123\$.

Keywords are: **Crypto, AML, Fraud**.

### 20.7.1.3. Example 3: Complex Dynamic Description for High-Risk Countries Widget

```
1  {% set expl_data = explainability.high_risk_countries.hrc_categorical %}
2  {% set high_risk_countries = expl_data | selectattr('cr', 'in', ['High', 'Very High'])
3  | list %}
4  {% set sum_high_risk = high_risk_countries | sum(attribute='s') %}
5  {% set total_transaction_sum = activity.amount %}
6  {% set percentage = (sum_high_risk / total_transaction_sum * 100) | round(2) %}
7  {% set country_count = high_risk_countries | length %}
8  {% set highest_amount_country = high_risk_countries | sort(attribute='s',
9  reverse=True) | first %}
10 The sum of customer's transactions to countries with risk defined
11 as High or Very High is **{{ sum_high_risk | prettify_number }}{{ metadata.amount.units
12 | currency_symbol }}**.
13 It is {{ percentage }}% of the total transaction sum for the analysis period.
14 In total, there are {{ country_count }} countries like this.
15 The highest amount was sent to
16 **{{ highest_amount_country.ct | country_full }} -
17 {{ highest_amount_country.s | prettify_number }}{{ metadata.amount.units | currency_
18 symbol }}**.
```

In this example, more complex calculations and data formatting are performed.

- Access data for high-risk countries is explained.
- Filter countries classified as "High" or "Very High" risk.
- Calculate the sum of transactions to these countries.
- The percentage of the total transaction sum is determined.
- The number of such countries is counted.
- The country with the highest transaction amount is identified.
- custom functions like prettify\_number, currency\_symbol, and country\_full for formatting are used.

When rendered, this template might look like:

The sum of customer's transactions to countries with risk defined as High or Very High is **50,000\$**.

It is 25% of the total transaction sum for the analysis period.

In total, there are 3 such countries.

The highest amount was sent to **Italy - 20,000\$**.

#### 20.7.1.4. Example 4: Dynamic Description for Historical Widget

```
1  {% set total_6m = explainability.amount.hist_expl | sum(attribute='a') %}
2  {% set md = metadata.amount %}
3  {% set expl_md = md.explainabilities.hist_expl %}
4  {% set avg_monthly = total_6m / expl_md.time_range_value %}
5  Customer {{ activity.name }} spent {{ activity.amount | pretty_number }}
6  {{ md.units |
7  currency_symbol }}** for analysis
8  period and {{ total_6m | round(2) | pretty_number }}{{ md.units |
9  currency_symbol }}
10 ** in the last {{ expl_md.time_range_value }}
11 {{ expl_md.time_range_unit | lower }}s.\n
12 This is {{ avg_monthly | pretty_number }}{{ md.units | currency_symbol }}
13 on average per {{ expl_md.time_range_unit | lower }}**.\n
14 {% set percentage_difference = ((activity.amount - avg_monthly) / avg_monthly * 100) |
15 round(2) %}
16 Which is {{ percentage_difference }}% different than the average monthly spending.
17
```

This template is similar to the previous ones but tailored for a historical widget. Here,

- Transaction amounts over the last 6 months, are summed.
- Average monthly spending is calculated.
- Metadata for the amount feature and its explainabilities are accessed.
- The percentage difference from the average is calculated.
- Custom functions for formatting are used.

This will be rendered into:

Customer **John Doe** spent **10,000\$** during the analysis period and **60,000\$** in the last 6 months.

This represents an average of **10,000\$ per month**.

Which is 0% diffCustom Jinja functions

In addition to standard Jinja functions, the following custom functions are available:

- `pretty_number`: Formats numbers with commas for better readability.
  - Example: 12345 becomes 12,345.
- `currency_symbol`: Converts a currency ISO 4217 code into its symbol.
  - Example: USD becomes \$.
- `country_full`: Converts an ISO 3166-1 country code into the full country name.
  - Example: IT becomes Italy..

#### Best practices

To ensure optimal performance and maintainability of your dynamic descriptions, consider the following best practices:

- Keep Templates Simple.** Avoid complex calculations or data processing within the template. Perform heavy data manipulations during the ETL process instead and use values with activity. dot notation.
- Use Variables.** Utilize `{% set %}` statements to define variables and avoid redundant code as this enhances readability and reduces processing time.
- Access only the data you need.** Avoid loading large datasets into the template context.

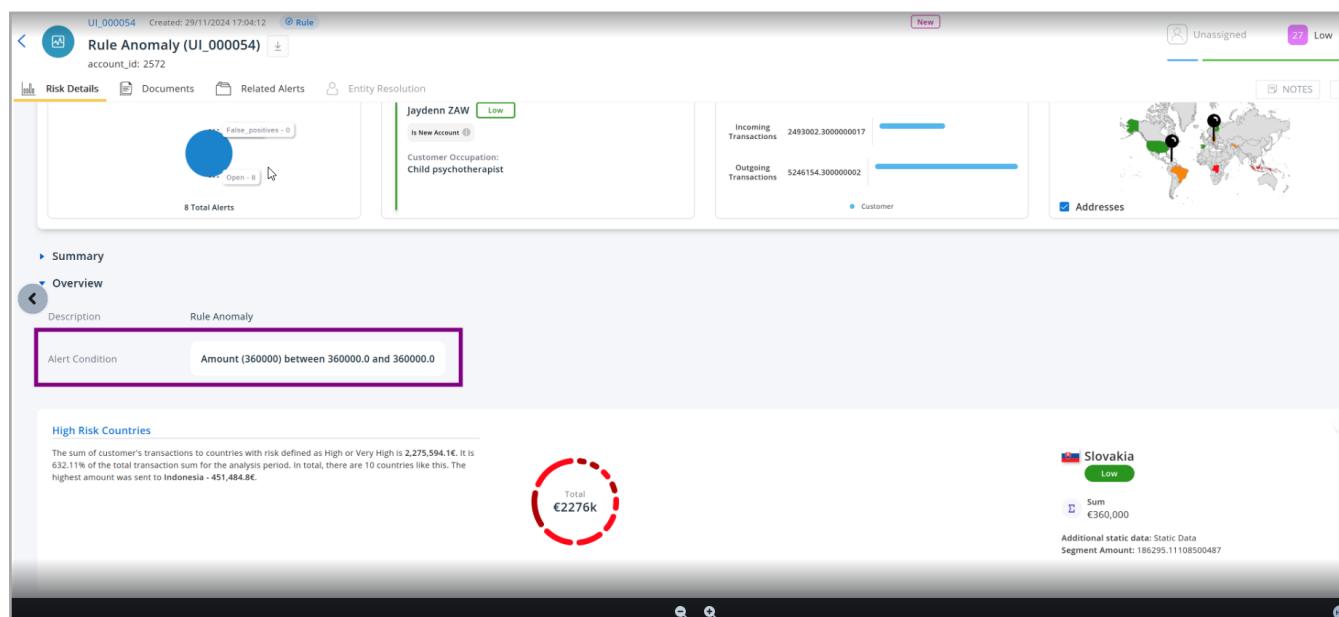
## 20.8. Explainability Configuration for Risk Based on RULE

In this sub section of the chapter, how to configure explainability for alerts generated by risks that are not based on AI model is covered, specifically concerning those with the analysis methods RULE or RT\_RULE. To provide this, the risk YAML has been enhanced with new metadata attributes.

This section is only relevant for risks that have RULE or RT\_RULE set as their analysis\_method. If attempts to use these configurations with risks of other analysis methods, are made, the settings will be ignored, or an exception may be raised.

## 20.9. Display Condition and Business Units

New attributes have been added to the risk YAML to allow the display of the condition based on which an alert was triggered in the **Alert Condition** section of the UI.



1. **condition\_display** - configures the alert condition as simple plain text.
2. **dynamic\_condition\_display** - configures the alert condition using a Jinja template, similar to other attributes that support dynamic tagging.

Example:

```
1 dynamic_conditions_display: "Amount ({{ activity.amount }})  
2 between {{ enrichments.min_amount }} and {{ enrichments.max_amount }}"
```

## 20.10. Risk Explainabilities

Another section introduced in the risk YAML is risk\_explainabilities. Under the hood it represents a list of RiskExplainability class instances described with YAML, allowing you to configure explainability widgets for the risk.

Each risk explainability configuration includes the following attributes:

- name: Unique name for the risk explainability within the risk.
- display\_name: Name that will be displayed in the UI for this explainability.
- description: Static description that will be displayed in case if dynamic\_description is not provided.
- dynamic\_description (optional) - A Jinja template string used to create a dynamic description for the risk explainability. Allows inclusion of dynamic values from activity, enrichments, and variables using dot notation.
- feature (optional): An optional reference to a feature in the input dataset of evaluation flow risk is related to.
- explainabilities (optional): A list of Explainability instances that define explainability widgets specific to the risk. For risk explainabilities, a list with more than one explainability is not supported. Same as for dynamic\_description reference to variables and enrichments can be used for dynamic values.

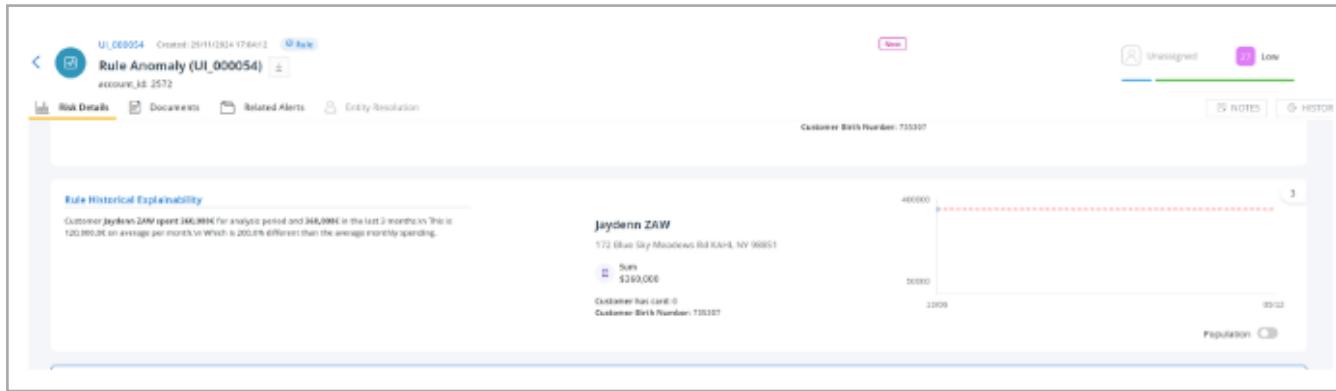
The feature attribute is represented by FeatureReference class under the hood and have 2 attributes:

- identifier: The identifier of the feature you are referencing that must correspond to an existing feature in your dataset.
- explainability (optional): The identifier of a specific explainability within the feature to reference. If provided, must correspond to an existing explainability within the referenced feature.

Using feature allows the risk explainability to inherit properties from a feature, such as its dynamic description or explainabilities, to avoid from risk to risk code duplications.

Risk explainability can be configured in few ways:

1. Configure feature.identifier only without referencing to explainability. In this case risk explainability will inherit dynamic description from feature
2. Define feature.explainability and leave explainabilities list empty and in this case the UI will display only the widget inherited from the feature.
3. Define Both feature.explainability and explainabilities and UI will create a combined widget where the feature's widget is displayed on the left, and the risk's widget is displayed on the right same as displayed at the screenshot below.



**Note:** There is a limitation when referencing a feature explainability - you cannot refer to explainabilities of type HISTORICAL or BEHAVIORAL.

## 20.11. Reference Implementation

### 20.11.0.1. Files with Examples

Reference implementation can be located in the Reference branch in JupyterHub, same as before.

The following is located in the default domain:

- Feature explainability configuration in the wrangling.py dataset. You can find configurations for each type of widget using the new attributes.
- 3 risk YAML examples can be found for each available analysis method:
  - risk\_rule.yaml - RULE
  - risk\_ai.yaml - AI
  - risk\_rt\_rule.yaml - RT\_RULE
  - in risk with RULE and RT\_RULE analysis methods configured risk\_explainabilities can be found together with dynamic\_condition\_display.
- Customer insights and OpenAI configuration examples can be found at evaluation-flows/evaluation\_flow\_tr.py.
- The customer insights dataset can be found at datasets/customer\_insights.py.

- In notebooks/wrangling.ipynb, an example can be found of how to create JSON strings for the explainability column purpose and also usage of validate\_explainabilities API which performs validations according to metadata configured.

## 20.11.1. Implementation Utils

### 20.11.1.1. Validation Util

The validate\_explainabilities function is part of the public API and is used to validate the data associated with explainability configurations in your dataset, specifically the data referenced by the json\_column\_reference in your explainability configurations. This function helps data engineers ensure that the explainability data adheres to the expected structure and content defined in the metadata.

The explainability data referenced by json\_column\_reference should be built in the following manner:

```
1  {
2      "data": [
3          {
4              // Objects containing keys as defined in the explainability configuration
5              // These keys should match those specified in the `values` metadata
6          }
7      ]
8  }
```

Each object within the "data" array should contain key-value pairs where the keys correspond to those defineFunction signatured in your explainability's values configuration.

### 20.11.1.2. Function Signature

```
1  def validate_explainabilities(
2      context: JobExecutionContext,
3      df: DataFrame,
4      evaluation_flow_id: str,
5      *,
6      show_all_columns: bool = False,
7  ) -> Optional[DataFrame]:
```

#### Arguments

- context (JobExecutionContext): The execution context provided by the job environment. This includes access to solution metadata and Spark sessions.

- df (DataFrame): The Spark DataFrame that contains the explainability columns you want to validate. This DataFrame should include the columns referenced by json\_column\_reference in your explainability configurations.
- evaluation\_flow\_id (str): The identifier of the relevant evaluation flow. This is used to retrieve the correct explainability configurations from the solution metadata.
- show\_all\_columns (bool, optional): A flag to determine the output of the function.
  - If True, the function returns the full DataFrame, including all original columns and the validation results.
  - If False (default), the function returns only the explainability columns and the validation result columns.

### 20.11.1.3. Validations Performed

The function performs the following validations on each explainability column in the DataFrame:

1. **Structure Validation:**
  - a. **Valid JSON:** Checks if the explainability column contains valid JSON.
  - b. **Contains "data" Key:** Ensures that the root level of the JSON object contains the "data" key.
  - c. **"data" is a List of Objects:** Verifies that the "data" key maps to a list of dictionaries (objects).
2. **Data Validation:**
  - a. **Expected Keys:** Checks that each object in the "data" array contains the expected keys as defined in the explainability's values metadata.
  - b. **Data Types:** Validates that the values associated with each key are of the expected data types, as specified by ExplainabilityValueType.
3. **Error Aggregation:**
  - a. Records any discrepancies or errors found during the validation steps.
  - b. Adds validation error messages to new columns in the DataFrame for each explainability column.

For each explainability column, the function adds two new columns to the DataFrame:

- {column\_name}\_structure\_validation\_errors: Contains error messages related to the structure of the JSON data.
- {column\_name}\_data\_validation\_errors: Contains error messages related to the content of the data (missing keys, incorrect data types).

#### 20.11.1.4. Validation Errors

- Explainability json record is not a valid JSON.: The data in the explainability column is not valid JSON.
- Explainability record json must have 'data' root key.: The JSON object is missing the "data" key at the root level.
- Explainability data object should be a list of dictionaries.: The "data" key does not map to a list of objects.
- Missing/Extra Keys: Indicates that the expected keys are missing or there are unexpected keys.
  - Example: "found missing/extra keys in data object: {'ct', 's'} -> expected keys are {'ct', 's', 'c'}"
- Incorrect Data Types: Specifies that a value is not of the expected data type.
  - Example: "key 's' should be <class 'int'> but got <class 'str'>"

---

**Important note:** Validations run, only if execution environment is JupyterHub. If it is airflow function do nothing to not affect performance of runs. So the main purpose is to test implementation before moving it to BAU flow.

---

#### 20.12. OpenAI Summarization

In order to enable OpenAI summarization on the alert UI, it is required to enable relevant configuration at evaluation flow metadata.

It will only work correctly in combination if there is an active updated license on the customer environment.

```
1  from thetaray.api.solution.evaluation import (
2      ...
3      AlertConfiguration
4  )
5  def evaluation_flow() -> EvaluationFlow:
6      return EvaluationFlow(
7          ...,
8          alert_conf=AlertConfiguration(llm_summarization=True)
9      )
```

## 21. Enabling Monitoring Reports

The ThetaRay Platform autogenerated three monitoring reports at key points in the life-cycle of the evaluation process and distributes them to specified targets.

- The Data Upload monitoring report is autogenerated whenever a data upload task is completed.
- The Alert volume monitoring report is autogenerated whenever the data distribution task is completed.
- The Model update monitoring report is autogenerated whenever a model update or version change is merged with the operational branch in the git repository.

**To enable the monitoring reports generation and distribution, perform the following:**

1. In the JupyterLab Files side panel, navigate to and open:

```
/settings/tr_settings.py
```

---

**Note:** If the environment is of the upgraded type (and not a new deployment), the user should add the folder and the tr\_settings.py, by his/ her self.

---

2. The reports configuration settings are specified:

```
data_report_enabled = False
alert_report_enabled = False
model_report_enabled = False
```

3. Set to true the setting of the report you want to enable.
4. There is a new attribute in dataset metadata named 'record\_identifier', pointing to one of the dataset fields, it is used in upload report in order to identity the corrupted record.
5. There is a new attribute in dataset metadata named 'emit\_corrupted\_record', in order to protect PII. (default value -> False)

```
def card_dataset():
    return DataSet(
        identifier="card",
        display_name="card",
        field_list=[
            Field(identifier="card_id", display_name="card_id", data_type=DataType.LONG,
array_indicator=False),
            Field(identifier="disp_id", display_name="disp_id", data_type=DataType.LONG,
```

```
array_indicator=False),
    Field(identifier="type", display_name="type", data_type=DataType.STRING, array_
indicator=False),
    Field(identifier="issued", display_name="issued", data_type=DataType.STRING,
array_indicator=False),
    Field(identifier="codes", display_name="codes", data_type=DataType.STRING,
array_indicator=False)
],
ingestion_mode=IngestionMode.OVERWRITE,
publish=True,
primary_key=[ "card_id" ],
record_identifier="disp_id",
emit_corrupted_record = False,
num_of_partitions=4,
num_of_buckets=7,
data_permission="dpv:public"
)

def entities():
    return [card_dataset()]
```

**Note:** For more information on Monitoring reports, refer to the current *SaaS Admin Guide* (for SaaS deployments), and the *Application Integration Interface* guide (for developer API information).

## 21.1. Enabling Monitoring Reports Notifications

### 21.1.1. Define New External Notification Target

Under domains/[DOMAIN\_NAME]/targets/

```
from thetaray.api.solution.external_notification_target import ExternalNotificationTarget,
TargetAuth, AuthenticationMode
def external_notification_target():
    return ExternalNotificationTarget(
        identifier="reports_notifications",
        name="reports_notifications",
        url="[ANY_URL]",
        verify=False,
        auth=TargetAuth(mode=AuthenticationMode.BASIC_AUTH, "configuration": {
            "username": <username>,
            "password": <password>
        })
    )
```

```
)  
def entities():  
    return [external_notification_target()]
```

## 22. Calendar SLA

### 22.1. Overview

An SLA is a time frame for resolving each step of a workflow, such as the time an analyst has to close an alert. Our new feature, which excludes weekends and local holidays from the SLA time, provides more accurate and realistic deadlines for our analysts, benefiting their productivity and efficiency.

This documentation provides step-by-step instructions for data engineers to set up automatic updates for national calendar information using the Calendarific API and Airflow. With this setup, data engineers can create a Jupyter notebook containing the default countries and dates, which will be automatically updated every quarter according to our purchased license. The updated calendar dates will then be posted to the IC API endpoint for use in Service Level Agreement (SLA) calculations.

### 22.2. Getting Started

This section provides detailed instructions for each step required to configure Calendars for users to be able to choose from in IC settings on the **Investigation Center**.

To implement the Calendar SLA functionality, please refer to the relevant code files within the project repository. Navigate to the following file paths to access the code:

- **Notebook location path** - reference/domains/default/notebooks/update\_non\_working\_days.ipynb
- **DAG Location path** - reference/dags/update\_non\_working\_days.py

#### 22.2.1. Steps to Configure

1. Create a Jupyter notebook with a pre-set list of countries, including their start and end dates.
  - Users do not need to configure start and end dates, as they are already set
  - The notebook will automatically retrieve holidays for all countries on the list below, unless otherwise specified
  - Additional countries can be added or removed as required
2. The api token is stored under the *CALENDARIFIC\_API\_TOKEN* name in env variables / secrets.
3. The final line contains the function `calendar.update_non_working_days` that will perform the POST to the IC API.

```
from datetime import date as Date, datetime as DateTime
from typing import List, Optional

from dateutil.relativedelta import relativedelta as RelativeDelta

from thetaray.api.calendar_sla import (
    Calendar as BaseCalendar,
    get_first_day_of_the_quarter,
)
from thetaray.api.context import init_context
from thetaray.common import Settings

# Specify countries (in iso-3166-1-alpha-2 format) for which non-working days should be
# updated
countries: List[str] = [
    "AR",
    "AU",
    "BR",
    "BG",
    "CA",
    "CL",
    "CN",
    "HR",
    "CY",
    "CZ",
    "DK",
    "EE",
    "FI",
    "FR",
    "DE",
    "GR",
    "HK",
    "HU",
    "IN",
    "IE",
    "IL",
    "IT",
    "LV",
    "LT",
    "LU",
    "MT",
    "MX",
    "NL",
    "PA",
    "PL",
    "PT",
    "QA",
```

```
"RO",
"SA",
"SG",
"SK",
"SI",
"KP",
"ES",
"SE",
"TW",
"AE",
"GB",
"US",
"UA",
"NG",
"AU",
"GH",
"CM",
]
# Specify start and end of the period in which non-working days should be updated
start: Date = get_first_day_of_the_quarter() # Default - first day of the current quarter
end: Date = start + RelativeDelta(years=1) # Default - in a year

class Calendar(BaseCalendar):
    # Specify your own token here if needed
    api_token: str = Settings.CALENDARIFIC_API_TOKEN

    def get_holidays(self, year: int, country: str, region: Optional[str] = None) -> List[Date]:
        """
        Fetches holidays' dates for a given year, country and region

        :param year: Year
        :param country: Country code ISO 3166-1 alpha-2
        :param region: Region code ISO 3166-2

        :return: List of holiday dates
        """
        # Comment the line below and write your own integration, if needed
        return super().get_holidays(year, country, region)

context = init_context(execution_date=DateTime(1970, 1, 1))

calendar = Calendar(context, distribution_target_identifier="ic")
calendar.update_non_working_days(start, end, countries)
```

## 22.2.2. Scheduling the DAG

The reference implementation already includes default scheduling dates, which are set to run quarterly on the first minute of the day in the server's timezone. If necessary, these defaults can be modified. It's recommended to schedule the DAG during off-hours for customers.

```
import datetime

from airflow.models import DAG
from thetaray.api.airflow import RunNotebookOperator, ExecutionDateMode


default_args = {
    "owner": "Airflow",
    "depends_on_past": False,
    "start_date": datetime.datetime(1970, 1, 1),
    "retries": 1,
    "retry_delay": datetime.timedelta(minutes=5),
    "execution_date_mode": ExecutionDateMode.PERIOD_END,
}

dag = DAG(
    dag_id="update_non_working_days",
    catchup=False,
    default_args=default_args,
    schedule_interval="0 0 1 */3 *" # Quarterly at 00:00 on day-of-month 1
)

update_non_working_days = RunNotebookOperator(
    task_id="update_non_working_days",
    domain="default",
    notebook_name="update_non_working_days",
    dag=dag,
)

update_non_working_days

if __name__ == "__main__":
    dag.cli()
```

## 22.3. On Prem Specific Instructions

For on-prem customers who have not yet integrated the Calendarific API on their end, we offer two options:

1. Whitelisting the Calendarific API and deploying a custom or default configuration. This option is easy to implement and offers full access to our comprehensive holiday database without any additional steps required on your end.
2. Alternatively, you can send a POST to our IC API. However, this option requires more effort and may result in additional configuration steps on your end. We recommend choosing the first option for a hassle-free experience.

### 22.3.1. Option #1 - Whitelisting the Calendarific API

Three steps:

1. Whitelist the Calendarific IP addresses in your firewall to allow incoming traffic from Calendarific.
2. Retrieve our API key.

---

**Note:** To use our available API key, please contact *Customer Support*

---

3. Please see section 17.2 and configure the notebook if you require any changes. The reference implementation allows for flexibility and custom configuration. You may configure the notebook by commenting line 48 and writing your own integration.

---

**Note:** In the case that you need to modify the airflow DAG, be aware that you have the option to modify the start and end dates for which you are retrieving holidays from the Calendarific API.

---

### 22.3.2. Option #2 - POST to our IC API

In the case that you cannot whitelist Calendarific and wish to use your own third party provider or custom calendars, we provide the option to POST to our API. In order to integrate your calendars, please follow the instructions below.

1. To send your calendar information to our IC API, please use the following endpoint and method:
  - Endpoint URL: api/configuration/sla-country-calendar
  - HTTP Method: POST
2. Include the following parameters in your POST request:
  - "location": (string, required) the ISO 3166 -1 code of the country for which you are sending calendar information
  - "region": (optional) the name of the subdivision for which you are sending calendar information

- "nonWorkingDays": (array of strings, required) a list of dates that represent non-working days for the country/subdivision specified

Example:

```
{  
  "location": "US",  
  "region": "California",  
  "nonWorkingDays": [  
    "2023-07-04",  
    "2023-09-04",  
    "2023-11-23",  
    "2023-12-25"  
  ]  
}
```

3. Send a POST request to the following endpoint, passing the JSON object as the request body:

```
POST /api/configuration/sla-country-calendar
```

4. Include the appropriate authentication headers, accessible in the calendar object, in your request. Please contact **Customer Support** if you need assistance with authentication.

#### Example - Set SLA Calendar Parameters

```
POST /api/configuration/sla-country-calendar
```

```
{  
  "location": "UA", //Required, must be two character ISO 3166-1 code  
  "region": "Kyivska oblast", // Optional, we are expecting  
  to receive subdivision name instead of 3166-2 code  
  "nonWorkingDays": [  
    "2023-03-03" //expected date format: yyyy-mm-dd  
  ]  
}
```

**Note:** After sending the POST request in the specified format, the uploaded calendar will be available for use in the Investigation Center IC Settings dropdown. The name of the calendar for example, could be 'UA (Kyiv oblast)' and it would take into account the non-working days you declared in "nonWorkingDays" when calculating the remaining time left in your analysts SLA.

## 23. Drift Management

### 23.1. Drift Introduction and Overview

The information included in this section provides the user with details on how drift is managed and deployed in ThetaRay.

Monitoring drift in every ThetaRay SONAR deployment is a key element in the provision and maintenance of a stable and reliable application.

Although drift monitoring is applied as part of BAU in general in all deployments, it should be noted that configurations made in the pre deployment staging phase are activated only in the production phase.

Drift monitoring tests for drift over two key system stages:

1. **Logging info** - where tests results do not stop run and are only logged in MLflow.
2. **Statistical info** - where test results do stop alert creation run if drift detected, (compared to the configuration of the threshold values) .

#### 23.1.1. Logging Info Stage

Tested elements include:

- a. Statistics and distributions of the raw data, aggregated data and algorithms output (PSI, Z-test, etc).
- b. Top 3 trigger features distribution vs history.
- c. Algo derived alerts creation.

#### 23.1.2. Statistics Stage

Tested elements include:

- a. Anomaly ratio statistics.
- b. Fusion score distribution statistics
- c. Threshold statistics exceeding a threshold in pre-defined statistical tests

#### Visualizing Test Results

As part of the testing procedure, and as an aid to understanding test results, graph data is gathered and used to build informative data graphs.

Two examples of Feature CSI Results are shown below.

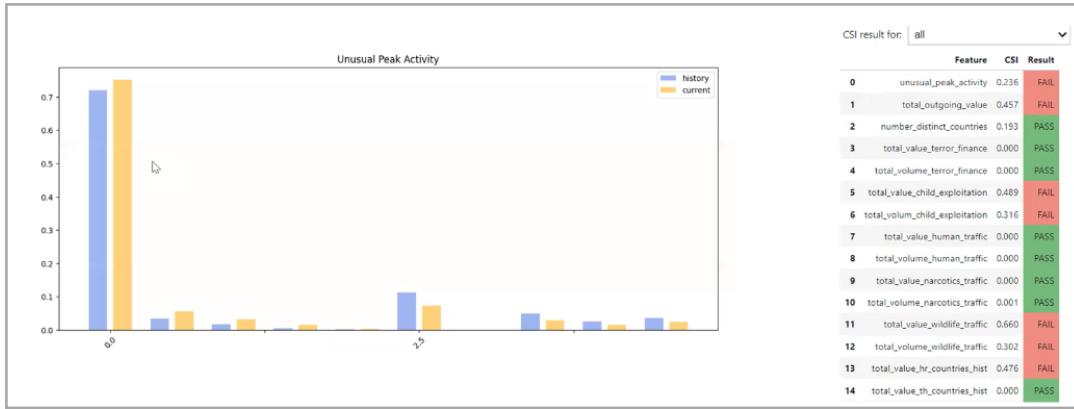


Figure 17: Example Drift Monitoring Results that Highlights an Unusual Activity Peak

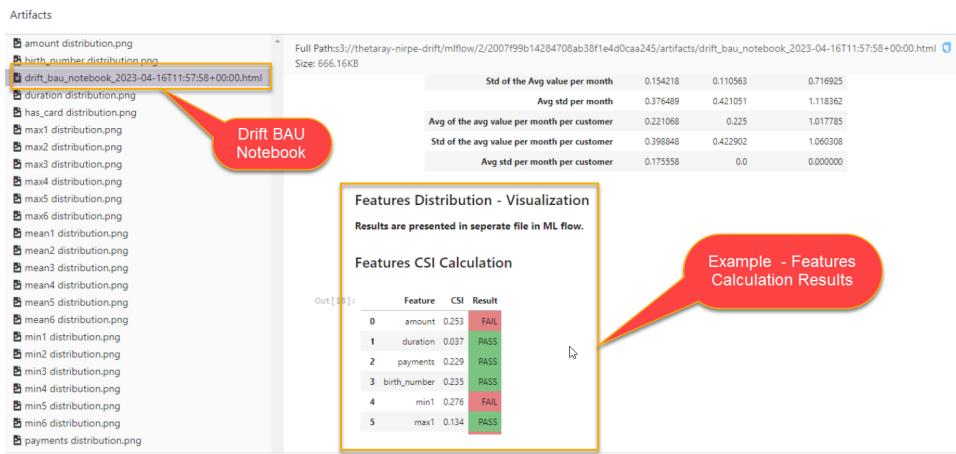


Figure 18: Example Feature Distribution - Visualization - Features CSI Calculation Results

## 23.2. How does Drift Detection Work in your Deployment?

In this section, to help understand how drift is detected in ThetaRay, we will look at the overall process by reviewing the following topics:

- Drift Notebook and how it is applied in the drift management process
- Understanding the importance of setting insightful metric variables
- Addition of drift detection into the BAU flow
- What happens when a test run fails, indicating drift, and what should the user do next?

### 23.2.1. Drift Notebook, its Application in the Drift Management Process

The drift notebook integrates the drift test code and its output into a single document that combining visualizations, narrative text, formulas, explanations, and metric comparison charts.

Lets take a deep dive into the drift notebook and associated files.

The drift notebook associated settings are:

- **drift\_bau\_notebook.jpnb**- the main drift notebook that runs in BAU
  - general access path : -> domains -> default -> notebooks ->
- **drift\_settings.py**: Drift settings file
  - general access path: -> settings ->
- **drift.lib. py**: Drift analysis repository file - contains the functions used in the main notebook
  - general access path: -> lib -> drift\_analysis->

### 23.2.1.1. drift\_bau\_notebook.jpnb

The drift notebook is divided into several subsections:

**Section** - Including initial general required packages, display configurations, spark configurations, and summary findings report of last notebook execution.

```

[1]: import mlflow
import numpy as np
from scipy.stats import norm
from thetaray.api.context import init_context
from thetaray.api.dataset import dataset_functions
from thetaray.api.drift import send_drift_notification
from thetaray.api.evaluation import load_evaluated_activities
from thetaray.api.evaluation import evaluate_step
from thetaray.api.mlflow.adapter import get_run_id
from thetaray.api.metric import DriftMetricPublisher
import pyspark.sql.functions as f
from thetaray.common.data_environment import DataEnvironment
from pyspark.sql.functions import udf
from pyspark.sql.types import *
import pandas as pd
import datetime
import matplotlib.pyplot as plt; plt.rcParams()
from matplotlib.ticker import MultipleLocator, MaxNLocator
from collections import OrderedDict
from pyspark.ml.feature import Bucketizer, QuantileDiscretizer

pd.set_option('display.max_columns', 500)
pd.set_option('display.max_rows', 200)

spark_conf = {'spark.authenticate': False,
              'spark.network.crypto.enabled': False,
              'spark.sql.repl.eagerEval.enabled': True,
              'spark.sql.autoBroadcastJoinThreshold': -1,
              'spark.executor.memory': "8g",
              'spark.driver.memory': "4g",
              'spark.executor.cores': "2",
              'spark.executor.instances': "1",
              'spark.sql.adaptive.enabled': "true",
              'spark.dynamicAllocation.enabled': "true",
              'spark.dynamicAllocation.initialExecutors': "1",
              'spark.dynamicAllocation.maxExecutors': "10",
              'spark.dynamicAllocation.minExecutors': "0",
              'spark.dynamicAllocation.shuffleTracking.enabled': "true"
            }
# 2023_04_16_11_57_58
context = init_context(execution_date=datetime.datetime(2023, 4, 16, 11, 57, 58), spark_conf=spark_conf, allow_type_changes=True)
params = context.parameters
spark = context.get_spark_session()

Last executed at 2023-04-18 10:27:17 +00:00
2023-04-18 13:25:01,730:INFO:thetaray.common.logging:load_risks took: 0.04416060447692871
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/opt/spark-3.2.0/jars/spark-unsafe_2.12-3.2.0.jar) to constructor java.nio.DirectByteBuffer(long,int)
WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).

```

Notebook - Imported required packages

Notebook General params setup

Example Spark log

## Section - Key reference file required imports

```
[2]: from settings.drift_settings import *
from lib.drift_analysis.drift_lib import *
Last executed at 2023-04-17 08:39:54 in 20ms
```

Imported packages

**settings\_drift\_settings** - This settings script is used by the notebook as reference source during the drift checking run executions.

An example segment of the drift settings python script is shown below.

```
1 from thetaray.common.data_environment import DataEnvironment
2 import datetime
3 ### General parameters
4 # **Define the following parameters:**
5
6 # 0. <u>General</u>: data environment + execution date. Using for reading the data with the provided execution dates.
7 # 1. <u>Datasets</u>: parameters for data reading. Fill the relevant dataset names of: Transactions_DF, Final_DF (before analysis), Anomalies_DF.
8 # 2. <u>Relevant Columns</u>: parameters for columns definitions.
9 # 3. <u>Static parameters</u>: define different parameters that will be used in different functions.
10 # 4. <u>Analysis settings</u>: define the historical period and the current period.
11
12
13 # 0. General Settings
14 # -----
15 data_env = DataEnvironment.PRIVATE
16 from_job_ts = datetime.datetime(1970,1,2)
17 to_job_ts = datetime.datetime(2023,4,16,11,57,58)
18
19 # 1. Datasets
20 # -----
21 EVALUATION_FLOW = "tr_analysis"
22 LAST_DF_NAME = 'wrangling'
23 TRANSACTIONS_DF_NAME = 'transaction'
24
25 # 2. Relevant Columns
26 # -----
27 DATE_ANALYSIS_COL = "date_loan" # analysis_dataframe
28 DATE_COL = "date" # transaction_dataframe
29 ENTITIES_COLS = ["account_id"]
30 ALGO_STEP = "tr_evaluation"
31 DATE_AGG_LEVEL = 'date_agg_level'
32 AMOUNT_COL = "amount"
33
34 # 3. Static parameters
35 #
```

Required packages

Example General Setting, Dataset and Feature variable definitions

Figure 19: Example Section of the Drift Python Settings Script

**lib.drift\_analysis.drift\_lib**- This is the file that holds all the drift notebook functions, charts, statistical calculations that are required by the notebook.

An example segment of the drift analysis file is shown below.

```

import mlflow
import numpy as np
from scipy.stats import norm
from thetaray.api.dataset import dataset_functions
from thetaray.api.drift import send_drift_notification
from thetaray.api.evaluation import load_evaluated_activities
from thetaray.api.evaluation import evaluate_step
from thetaray.api.mlflow import get_run_id
from thetaray.api.metric import DriftMetricPublisher
import pyspark.sql.functions as F
from thetaray.common.data_environment import DataEnvironment
from pyspark.sql.functions import udf
from pyspark.sql.types import *
import pandas as pd
import datetime
import matplotlib.pyplot as plt; plt.rcParams['rcParams']['font.size'] = 10
from matplotlib.ticker import MultipleLocator, MaxNLocator
from collections import OrderedDict
from pyspark.ml.feature import Bucketizer, QuantileDiscretizer
from settings.drift_settings import *

# failing tests
FAILING_TESTS = tests_thresholds.keys()
FAILING_TESTS_RESULTS = {}
metrics = {'Test': [], 'Result': [], 'Value': [], 'Threshold': []}
metrics_output = {}

# Permutation Functions
def split_train_current(context, dataset_name, date_col, AGG_TYPE):
    reference_data = dataset_functions.read(context, dataset_name, data_environment.data_envn, from_job_ts=from_job_ts)
    reference_data = create_agg_level(reference_data, date_col, AGG_TYPE)
    reference_data = reference_data.where(f.col(date_col).start_period_train) & (f.col(date_col).end_period_train)
    reference_data = reference_data.sort([date_col] + INIT_COLS)

    new_batch_data = dataset_functions.read(context, dataset_name, data_environment.data_envn)
    new_batch_data = create_agg_level(new_batch_data, date_col, AGG_TYPE)

    return reference_data, new_batch_data

def cast_bulk_list(df, col_list, type):
    for column_name in col_list:
        if column_name in df.columns:
            df = df.withColumn(column_name, f.col(column_name).cast(type))
    return df

def date_agg(df, date_col, agg_level):
    '''Format date aggregation column to datafram from existing date column.
    input: spark Dataframe,
    name of column to convert to date column
    agg_level - 'month', 'week' or 'day' for monthly, weekly or daily aggregation
    return: New spark Dataframe
    '''
    if agg_level == "month":
        if 'year_month' not in df.columns:
            df = df.withColumn('year_month', f.concat(f.year(f.col(date_col)), f.lit('_'), f.lpad(f.month(f.col(date_col)), 2, '0')))
    return df
    elif agg_level == "week":
        if 'week_date' not in df.columns:
            df = df.withColumn('week', f.date_trunc('week', f.col(date_col))).withColumn('week_date', f.concat(f.year(f.col('week')), f.lit('_'), f.lpad(f.month(f.col('week')), 2, '0'), f.lit('_'), f.lpad(f.dayofmonth(f.col('week')), 2, '0')))
    return df
    elif agg_level == "day":
        if 'date' not in df.columns:
            df = df.withColumn('date', f.concat(f.year(f.col('date')), f.lit('_'), f.lpad(f.month(f.col('date')), 2, '0'), f.lit('_'), f.lpad(f.dayofmonth(f.col('date')), 2, '0')))
    return df

```

Example of required packages

I

Examples of test results and various function definitions

Figure 20: Example Section of Drift Analysis File - drift\_lib.py

Returning to the Drift Notebook, let's continue our journey through the various Notebook sections.....

### Section - Notebook Test Thresholds

Threshold metrics can be modified if required. Increasing or decreasing these values will impact on drift check results.

```

Test Thresholds: ↵
[4]: tests_thresholds
Last executed at 2023-04-16 19:24:07 in 8ms
[4]: {'ANOMALY_PERCENTAGE': 5,
      'IQR_ALGO_SCORE': 0.1,
      'CSI_FEATURES_TEST': 0.25,
      'MAX_CSI_FEATURE_CHANGE_TEST': 0.5,
      'TRIGGER_FEATURES': 0.25,
      'CSI_ANOMALY_PERCENT': None}

```

### Section - Data Read & Data Preparation

In this Data Preparation section, various variables are set. This includes evaluated activities, features transaction references and filtering the data to take the drift tests.

## Example:

### ▼ Data Read & Data Preparation

```
[5]: reference_evaluated_activities = load_evaluated_activities(context, EVALUATION_FLOW, data_environment=data_evn, start_job_ts = from_job_ts,end_job_ts = to_job_ts)
reference_evaluated_activities = create_agg_level(reference_evaluated_activities,DATE_ANALYSIS_COL,AGG_TYPE)
reference_evaluated_activities = reference_evaluated_activities.where((f.col(DATE_ANALYSIS_COL)>=start_period_train) & (f.col(DATE_ANALYSIS_COL)<=end_period_train))
reference_evaluated_activities = reference_evaluated_activities.sort([DATE_ANALYSIS_COL] + ENTITIES_COLS)

new_batch_evaluated_activities = load_evaluated_activities(context, EVALUATION_FLOW, data_environment=data_evn)
new_batch_evaluated_activities = create_agg_level(new_batch_evaluated_activities,DATE_ANALYSIS_COL,AGG_TYPE)
Last executed at 2023-04-16 19:24:08 in 918ms

[6]: features_df_ref, features_df_new = split_train_current(context, LAST_DF_NAME, DATE_ANALYSIS_COL,AGG_TYPE)
transactions_ref, transactions_new = split_train_current(context, TRANSACTIONS_DF_NAME, DATE_COL,AGG_TYPE)
Last executed at 2023-04-16 19:24:09 in 980ms

[7]: # keep only last period as current period
max_date = new_batch_evaluated_activities.select(f.max('date_agg_level')).collect()[0][0]
new_batch_evaluated_activities = new_batch_evaluated_activities.filter(f.col('date_agg_level')>=max_date)
features_df_new = features_df_new.filter(f.col('date_agg_level')>=max_date)
transactions_new = transactions_new.filter(f.col('date_agg_level')>=max_date)
Last executed at 2023-04-16 19:24:17 in 7.91s

[8]: analysis_features = []
for col in reference_evaluated_activities.columns:
    if "rank" in col:
        notnulls = reference_evaluated_activities.select(col).dropna().count()
        if notnulls>0:
            col = col.replace(ALGO_STEPS, ".").replace("rank","")
            analysis_features.append(col)

# Casting both dataframes
cast_list = [DATE_AGG_LEVEL, ALGO_STEP+"_score"] + LIST_ADDITIONAL_FEATURE + analysis_features

features_df_ref = cast_bulk_list(df = features_df_ref, col_list = cast_list, type=DoubleType())
features_df_new = cast_bulk_list(df = features_df_new, col_list = cast_list, type=DoubleType())
reference_evaluated_activities = cast_bulk_list(df = reference_evaluated_activities, col_list = cast_list, type=DoubleType())
new_batch_evaluated_activities = cast_bulk_list(df = new_batch_evaluated_activities, col_list = cast_list, type=DoubleType())

reference_evaluated_activities, agg_col = date_agg(reference_evaluated_activities, DATE_ANALYSIS_COL,AGG_TYPE)
new_batch_evaluated_activities, agg_col = date_agg(new_batch_evaluated_activities, DATE_ANALYSIS_COL,AGG_TYPE)
features_df_ref, agg_col = date_agg(features_df_ref, DATE_ANALYSIS_COL,AGG_TYPE)
features_df_new, agg_col = date_agg(features_df_new, DATE_ANALYSIS_COL,AGG_TYPE)
transactions_ref, agg_col = date_agg(transactions_ref, DATE_COL,AGG_TYPE)
transactions_new, agg_col = date_agg(transactions_new, DATE_COL,AGG_TYPE)

if 'week' in agg_col:
    time = 'week'
elif 'date' in agg_col:
    time = 'day'
else:
    time = 'month'
```

## Tests:

### 1. Raw Data Distribution

#### Example Test Code

```
if RAW_DATA_DISTRIBUTION:
    ls = ['Train Period', 'Current Period']
    dfs = [transactions_ref, transactions_new]
    data = []
    for i in range(2):
        df = dfs[i]
        df = df[1]
        stats = []
        stats['index'] = ls[i]
        stats['Avg number of transactions per [time]'] = df.groupby(DATE_AGG_LEVEL).select(f.avg("count")).collect()[0][0]
        avg_std = df.groupby(DATE_AGG_LEVEL).agg(f.avg(AMOUNT_COL).alias("Avg"), f.stddev(AMOUNT_COL).alias("Std"), f.countDistinct(ENTITIES_COLS).alias("Cnt"), select(f.avg("Avg"), f.stddev("Avg"), f.avg("Std"), f.avg("Cnt")))
        stats['Avg of avg transaction amount per [time]'] = avg_std[0]
        stats['Std of avg transaction amount per [time]'] = avg_std[1]
        stats['Std of transaction amount per [time]'] = avg_std[2]
        stats['Number of customers per [time]'] = avg_std[3]
        avg_std = df.groupby(DATE_AGG_LEVEL).agg(f.sum(AMOUNT_COL).alias("Sum"), f.stddev(AMOUNT_COL).alias("Std"), select(f.avg("Sum"), f.avg("Std"))).collect()
        stats['Avg number of transactions per [time] per customer'] = avg_std[0]
        stats['Avg sum transactions per [time] per customer'] = avg_std[1]
        stats['Std transactions per [time] per customer'] = avg_std[2]
        data.append(stats)

    NEW_DF = pd.DataFrame.from_dict(data).set_index("index")
    NEW_DF['Proportion of change'] = NEW_DF['Current Period']/NEW_DF['Train Period']
    NEW_DF
else:
    NEW_DF = None
```

Raw Data Distribution-  
Test code

index	Train Period	Current Period	Proportion of change
Avg of number of transactions per month	14546.014085	23072.000000	1.586139
Avg of avg transaction amount per month	6107.138274	6690.573546	1.095533
Std of avg transaction amount per month	969.663582	1639.372377	1.690661
Std of transactions amount per month	9601.684412	10110.318799	1.052998
Number of customers per month	2544.126761	4459.500000	1.752861
Avg number of transactions per month per customer	5.717488	5.173674	0.904886
Avg sum transactions per month per customer	33693.159426	33280.521202	0.987753
Std sum transactions per month per customer	6721.757005	6876.188516	1.022975

```
if RAW_DATA_AMOUNT_DISTRIBUTION:
    amount_dist = agg_dist(transactions_ref, AMOUNT_COL, [agg_col, DATE_AGG_LEVEL]).sort(DATE_AGG_LEVEL).toPandas()
    amount_dist_new = agg_dist(transactions_new, AMOUNT_COL, [agg_col, DATE_AGG_LEVEL]).sort(DATE_AGG_LEVEL).toPandas()
else:
    amount_dist = None
    amount_dist_new = None
```

Last executed at 2023-04-16 18:41:38 in 52.10s

## Amount Distribution Across History & Current Period - Example Segment

feature_name	year_month	date_agg_level	count_positive	count_all	positive_prop	min	perc_10	perc_20	perc_25	perc_30	perc_40	perc_50	perc_60	perc_70	perc_75	perc_80	perc_90	perc_95	perc_99	max	avg	stddev	IQR	range	
0	amount	1993_01	276.0	177	177	1.0	0.8	20.1	107.1	200.0	300.0	500.0	700.0	900.0	1100.0	1100.0	5123.0	13967.0	19961.0	46552.0	49752.0	3966.99	8637.84	900.0	49751.2
1	amount	1993_02	277.0	395	395	1.0	2.9	25.6	73.8	146.1	300.0	700.0	1000.0	3900.0	6242.0	9000.0	12644.0	23330.0	31332.0	47831.0	62100.0	6903.61	10947.68	8853.9	62097.1
2	amount	1993_03	278.0	676	676	1.0	1.7	37.8	115.5	175.3	300.0	700.0	3000.0	5100.0	7651.0	10100.0	12644.0	20281.0	27700.0	44749.0	51700.0	6997.51	9962.18	9924.7	51698.3
3	amount	1993_04	279.0	913	913	1.0	2.5	42.1	128.7	169.0	357.1	900.0	3469.0	6007.0	9574.0	12138.0	14491.0	23330.0	33600.0	47773.0	60500.0	80814.5	11060.12	11969.0	60497.5
4	amount	1993_05	280.0	1306	1306	1.0	3.1	49.4	141.7	192.6	294.2	1100.0	3700.0	6300.0	10900.0	13200.0	17288.0	24863.0	37064.0	49700.0	60000.0	8943.92	12123.18	13007.4	59995.9
5	amount	1993_06	281.0	1880	1880	1.0	2.2	25.6	124.8	173.7	254.8	1100.0	3350.0	6148.0	11427.0	14800.0	19665.0	30100.0	36700.0	56400.0	76500.0	9729.76	13549.23	14626.3	76497.8
6	amount	1993_07	282.0	2399	2399	1.0	1.0	14.6	111.6	157.6	208.1	997.0	3000.0	4773.0	7634.0	10800.0	14100.0	22500.0	31200.0	47721.0	62000.0	7512.0	10814.44	10642.4	61999.0
7	amount	1993_08	283.0	2938	2938	1.0	1.0	14.6	100.8	145.0	199.0	840.0	2500.0	4500.0	7200.0	9600.0	13204.0	21882.0	30000.0	47024.0	63400.0	7157.53	10541.90	9455.0	63399.0
8	amount	1993_09	284.0	3441	3441	1.0	0.7	14.6	100.9	140.5	190.6	713.0	2300.0	4200.0	6545.0	9226.0	12700.0	20795.0	28082.0	46312.0	62800.0	6811.60	10169.50	9085.5	62799.3
9	amount	1993_10	285.0	3989	3989	1.0	1.0	14.6	103.5	138.4	190.0	900.0	2520.0	4236.0	6623.0	8901.0	12537.0	21300.0	29302.0	46241.0	63700.0	6911.73	10327.17	8762.6	63699.0

## Test 2.- Features Distribution Test

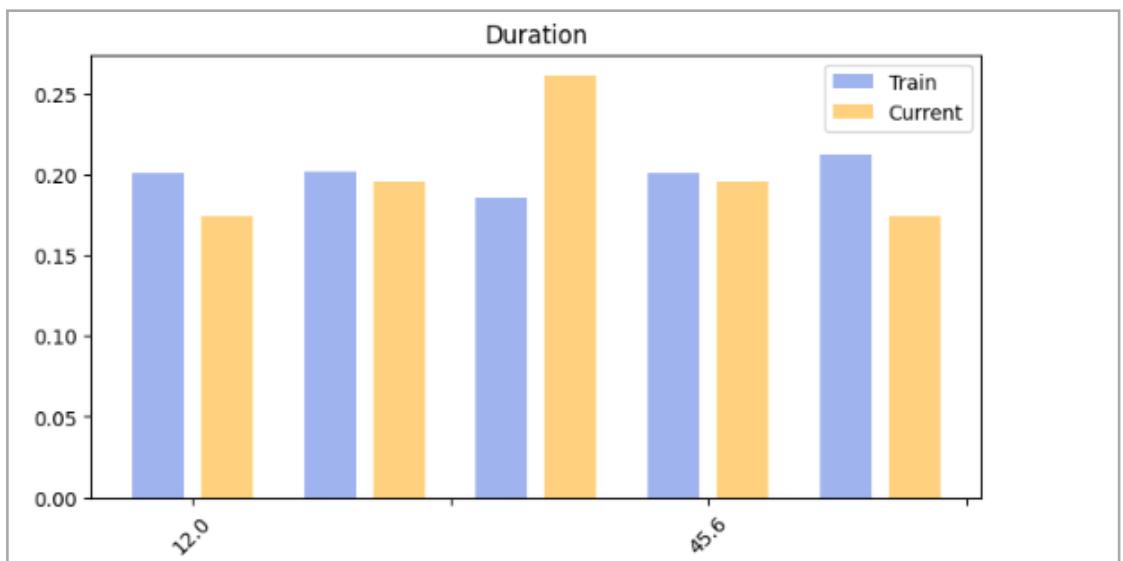
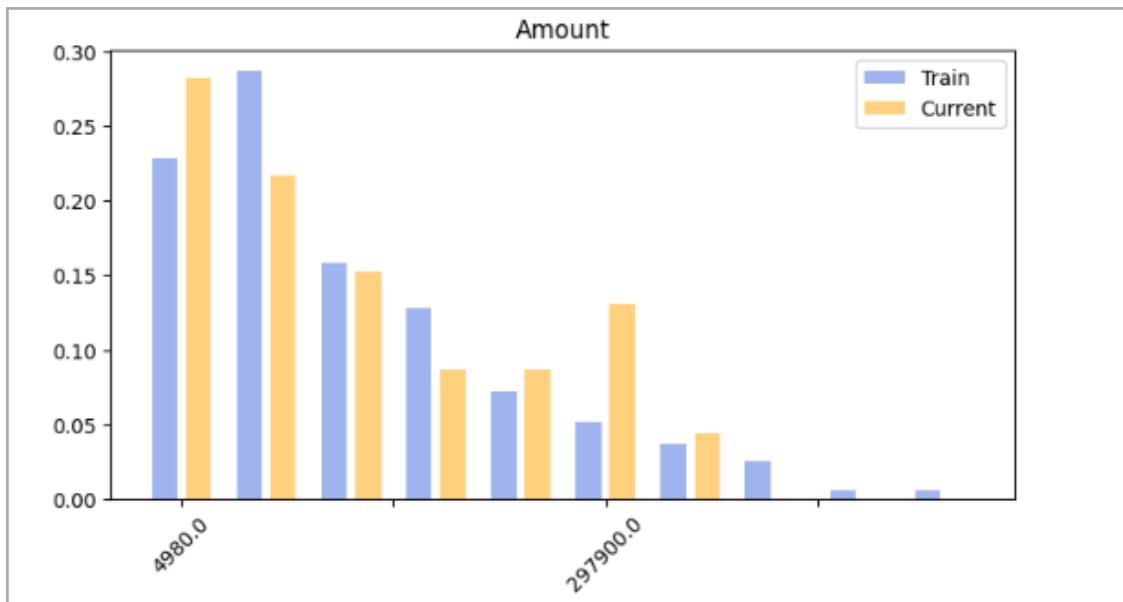
General features Distribution - Features Distribution - Visualization

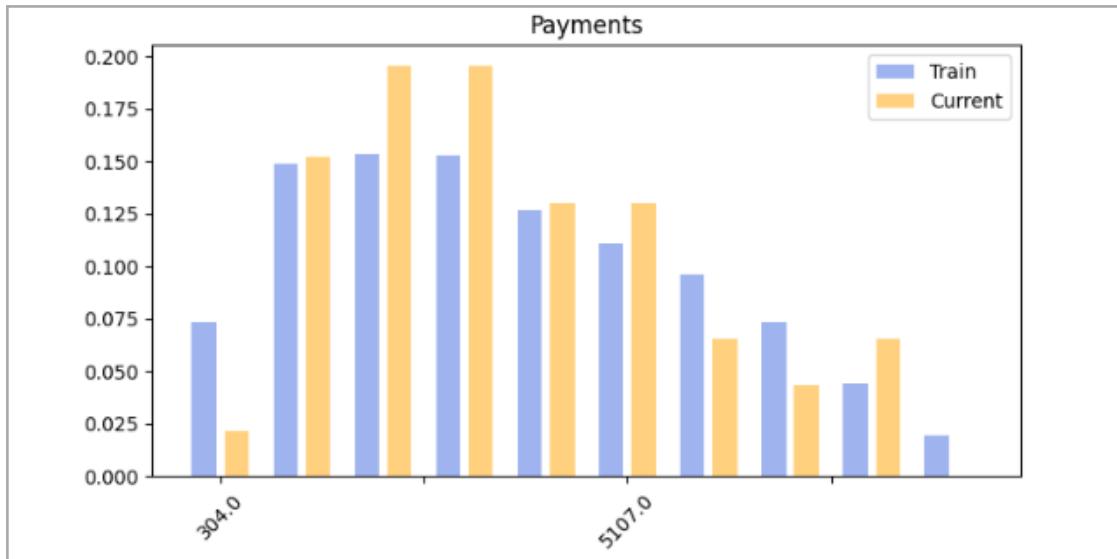
Example Code:

```
[16]: # if FEATURES DISTRIBUTION VISUALIZATION:
  for column in analysis_features:
    fig = barchart_compare(features_df.ref, features_df.new, column, NUM_OF_BINS)
    if fig:
      mlflow.log_figure(fig, f"{column} distribution.png")
  plt.show()
```

Last executed at 2023-04-16 18:57:49 in 5m 19.86s

Example Charts





### Features CSI Calculation - Test Code

Example:

```
## Features CSI Calculation

if CSI_FEATURES_TEST:
    df_res1 = CSI_feature(spark, features_df_ref, features_df_new_analysis_features, CSI_THRESHOLD, NUM_OF_BINS, sample_size = SAMPLE_SIZE).toPandas()
    df_res1s = df_res1.style.apply(lambda s: ['background-color: #E58282' if v else 'background-color: #79C57C' for v in s == 'FAIL'],subset=['Result']).set_precision(3)
    if 'CSI_FEATURES_TEST' in FAILING_TESTS:
        csi_res1 = len(df_res1[df_res1['Result']=='FAIL'])/len(df_res1) > tests_thresholds['CSI_FEATURES_TEST']
        metrics['Test'].append('ABOVE_CSI_THRESHOLD_PERCENTAGE')
        metrics['Result'].append('Failed' if csi_res1 else 'Pass')
        metrics['Value'].append(round(len(df_res1[df_res1['Result']=='FAIL'])/len(df_res1),4))
        metrics['Threshold'].append(tests_thresholds['CSI_FEATURES_TEST'])
        metrics['output']['ABOVE_CSI_THRESHOLD_PERCENTAGE'] = round(len(df_res1[df_res1['Result']=='FAIL'])/len(df_res1),4)
    if 'MAX_CSI_FEATURE_CHANGE_TEST' in FAILING_TESTS:
        csi_max_test = df_res1['CSI'].max() > tests_thresholds['MAX_CSI_FEATURE_CHANGE_TEST']
        metrics['Test'].append('MAX_CSI_FEATURE_CHANGE')
        metrics['Result'].append('Failed' if csi_max_test else 'Pass')
        metrics['Value'].append(round(df_res1['CSI'].max(),4))
        metrics['Threshold'].append(tests_thresholds['MAX_CSI_FEATURE_CHANGE_TEST'])
        metrics['output']['MAX_CSI_FEATURE_CHANGE'] = round(df_res1['CSI'].max(),4)
    else:
        df_res1s = None
```

*df\_results*

Example:

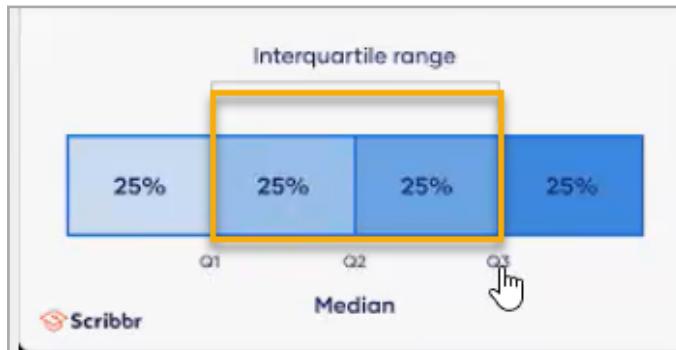
Ref	Features	CSI	Result
0	amount	0.253	FAIL
1	duration	0.037	PASS
2	payments	0.229	PASS
3	birth_number	0.235	PASS
4	min1	0.276	FAIL
5	max1	0.134	PASS
6	mean1	0.321	FAIL
7	min2	0.205	PASS

Ref	Features	CSI	Result
8	max2	0.333	FAIL
9	mean2	0.138	PASS
10	min3	0.245	PASS
11	max3	0.294	FAIL
12	mean3	0.192	PASS
13	min4	0.154	PASS
14	max4	0.139	PASS
15	mean4	0.187	PASS
16	min5	0.171	PASS
17	max5	0.066	PASS
18	mean5	0.220	PASS
19	min6	0.175	PASS
20	max6	0.295	FAIL
21	mean6	0.182	PASS
22	has_card	0.000	PASS

### 3. Anomalies Distribution

Anomaly percentage History & Current Periods (Percentage + IQR)

The interquartile (IQR) range defines the spread of the middle half of your distribution - Q2+Q3.



Example - Interquartile Range Description

IQR Example code

```

if IQR_ALGO_SCORE:
    reference_perc_list = reference_evaluated_activities.agg(f.expr('percentile_approx((ALGO_STEP)_score, array(0.25, 0.75))').alias("perc")).collect()[0].perc
    new_batch_perc_list = new_batch_evaluated_activities.agg(f.expr('percentile_approx_Q1((ALGO_STEP)_score, array(0.25, 0.75))').alias("perc")).collect()[0].perc

    reference_iqr = reference_perc_list[1] - reference_perc_list[0]
    new_batch_iqr = new_batch_perc_list[1] - new_batch_perc_list[0]
    print(f"train iqr: {reference_iqr}, new batch iqr: {new_batch_iqr}")
if 'IQR_ALGO_SCORE' in FAILING_TESTS:
    iqr_metric = float(reference_iqr - new_batch_iqr).abs_()
    iqr_drift = iqr_metric > tests_thresholds['IQR_ALGO_SCORE']
    metrics['Test'].append('IQR_ALGO_SCORE')
    metrics['Result'].append('Failed' if iqr_drift else 'Pass')
    metrics['Value'].append(round(iqr_metric, 4))
    metrics['Threshold'].append(tests_thresholds['IQR_ALGO_SCORE'])
    metrics['Output']['IQR_ALGO_SCORE'] = round(iqr_metric, 4)
else:
    reference_iqr = None
    new_batch_iqr = None

[Stage 624:> (0 + 7) / 7
train iqr: 0.023989592495584466, new batch iqr: 0.00205958419201413

Train IQR:
reference_iqr
0.023989592495584466
Current Period IQR:
new_batch_iqr
0.00205958419201413

The test calculates the zscore of the current period anomaly percentage compared to the training period. The Threshold sets the maximum zscore possible for passing this test.

if ANOMALY_PERCENTAGE:
    reference_anomaly_percent = anomaly_percentage(reference_evaluated_activities)
    new_batch_anomaly_percent = anomaly_percentage(new_batch_evaluated_activities)
    if 'ANOMALY_PERCENTAGE' in FAILING_TESTS:
        curr_anomaly_percent = new_batch_anomaly_percent['ratio'].iloc[0]
        anomaly_percent_zscore = (curr_anomaly_percent - reference_anomaly_percent['ratio'].mean()) / reference_anomaly_percent['ratio'].std()
        anomaly_percent_drift = anomaly_percent_zscore.abs_() > tests_thresholds['ANOMALY_PERCENTAGE']
        metrics['Test'].append('ANOMALY_PERCENTAGE')
        metrics['Result'].append('Failed' if anomaly_percent_drift else 'Pass')
        metrics['Value'].append(round(curr_anomaly_percent, 4))
        metrics['Threshold'].append(tests_thresholds['ANOMALY_PERCENTAGE'])
        metrics['Output']['ANOMALY_PERCENTAGE'] = round(curr_anomaly_percent, 4)
    reference_anomaly_percent = reference_anomaly_percent.T
else:
    reference_anomaly_percent = None
    new_batch_anomaly_percent = None

```

**IQR Calculation**

**IQR Train**

**New batch IQR**

### reference\_anomaly\_percent

Example:

date_loan	1997 10-20	1996 08-24	1996 08-06	1995 12-11	1997 10-08	1995 12-10	1994 06-01	1998 06-18	1997 10-14
ratio	0.5	05	1.0	0.5	0.5	1.0	0.0	1.0	0.5

### new\_batch\_anomaly\_percent

Example:

date_loan	ratio
1998-00-30	0.5
1998-00-10	1.0

### Trigger Features Distribution

Example:

```

if TRIGGER_FEATURES_TEST:
    df_res3 = tf_drift_by_month(reference_evaluated_activities, new_batch_evaluated_activities, agg_col, analysis_features, ALPHA)
    df_res3 = df_res3.toPandas()
    results_string_tf = f"There are no anomalies generated for the following Analysis Feature: {no_anomalies}"
    if 'TRIGGER_FEATURES' in FAILING_TESTS:
        tf_res = len(df_res3[df_res3['Result']=='FAIL'])/len(df_res3) > tests_thresholds['TRIGGER_FEATURES']
        metrics['Test'].append('TRIGGER_FEATURES')
        metrics['Result'].append('Failed' if tf_res else 'Pass')
        metrics['Value'].append(round(len(df_res3[df_res3['Result']=='FAIL'])/len(df_res3),4))
        metrics['Threshold'].append(tests_thresholds['TRIGGER_FEATURES'])
        metrics['output']['ANOMALY_PERCENTAGE'] = round(len(df_res3[df_res3['Result']=='FAIL'])/len(df_res3),4)

    df_res3s = df_res3.style.apply(lambda s: ['background-color: #E58282' if v else 'background-color: #79C57C' for v in s == 'FAIL'], subset=['Result']).set_precision(3)
else:
    results_string_tf = None
    df_res3s = None

```

### results\_string\_tf

Example:

There are no anomalies generated for the following Analysis Feature: ['frequency', 'duration', 'min1', 'max1', 'mean1', 'min2', 'max2', 'mean2', 'min3', 'max3', 'mean3', 'min4', 'max4', 'mean4', 'min5', 'max5', 'mean5', 'min6', 'max6', 'mean6']"

### df\_results

Example:

ref	trigger_feature	result	Zscore
0	amount	FAIL	2.110
1	birth_number	PASS	0.410
2	has_card	PASS	0.000
3	payments	PASS	0.960
4	type_card	PASS	0.410
5	type_display	PASS	1.320

### Anomaly Percentage Distribution

Example:

```

if CSI_ANOMALY_PERCENT_TEST:
    res, psi, df_compare, train_group = calculate_PSI(reference_evaluated_activities, new_batch_evaluated_activities, CSI_THRESHOLD_10, ALGO_STEP+"_score")
    results_string_csi = f'Anomaly score PSI = {psi:.3f}. Test is {res}'
    if 'CSI_ANOMALY_PERCENT' in FAILING_TESTS:
        metrics['Test'].append('CSI_ANOMALY_PERCENT')
        metrics['Result'].append('Failed' if res == 'FAIL' else 'Pass')
        metrics['Value'].append(round(psi,4))
        metrics['Threshold'].append(CSI_THRESHOLD)
        metrics['output']['CSI_ANOMALY_PERCENT'] = round(psi,4)
    else:
        results_string_csi = None
else:
    results_string_csi = None

results_string_csi
'Anomaly score PSI = 2.086. Test is FAIL'

```

### Final Results

Example:

```

drift_detected = 'Failed' in metrics['Result']

final_res = pd.DataFrame(metrics).set_index('Test')
final_res

```

Test	Result	Value	Threshold
ABOVE_CS_THRESHOLD_PERCENTAGE	Failed	0.2609	0.25
MAX_CS_FEATURE_CHANGE	Pass	0.3327	0.50
IQR_ALGO_SCORE	Pass	0.0219	0.10
ANOMALY_PERCENTAGE	Pass	0.0000	5.00
ANOMALY_PERCENTAGE	Pass	0.0000	5.00
TRIGGER_FEATURES	Pass	0.1667	0.25
CS_ANOMALY_PERCENT	Failed	2.0864	0.25

## Metrics Output

### Example

```

metrics_output
{'ABOVE_CS_THRESHOLD_PERCENTAGE': 0.2609,
 'MAX_CS_FEATURE_CHANGE': 0.3327,
 'IQR_ALGO_SCORE': 0.0219,
 'ANOMALY_PERCENTAGE': 0.1667,
 'CSI_ANOMALY_PERCENT': 2.0864}

with mlflow.start_run(nested=True):
    drift_metrics = DriftMetricPublisher(
        execution_date=context.execution_date, publish_to_es=True, publish_to_mlflow=True,
    )
    drift_metrics.log_drift_metrics_and_params(
        evaluation_identifier=EVALUATION_FLOW, evaluation_step_identifier=ALGO_STEP, metrics=metrics_output
    )

send_drift_notification(context, drift_detected, EVALUATION_FLOW, ALGO_STEP)
if drift_detected:
    raise RuntimeError('Drift detected, failing the run')
context.close() ⌂

```

## 23.2.2. The Importance of Setting Insightful Metric Variables

When Running the notebook to check for drift, you are comparing the data from the latest batch of ingressed data with metrics data the model was trained on.

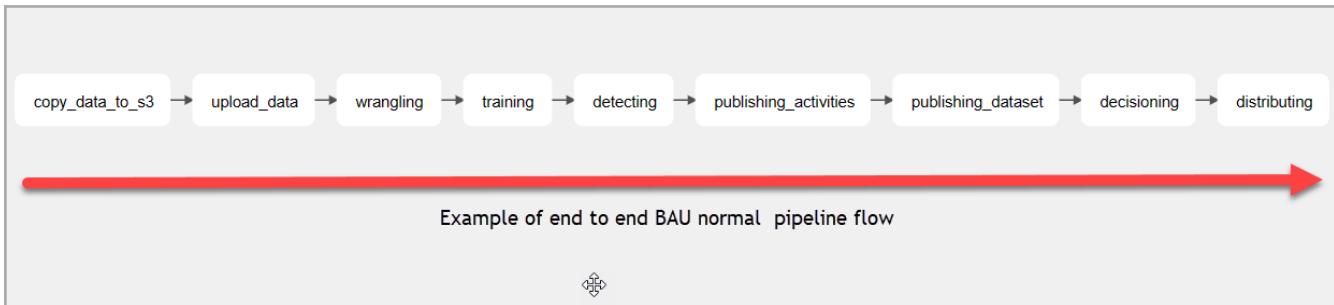
It is important to understand that data trends can and do change over time. To run drift monitoring efficiently as possible, the user is required to be aware of the difference between check results that may indicate serious drift and results that may be simply due to spikes caused by temporary or seasonal event occurrences.

However, continual and excessive drift results may not be transient or temporary in nature and may in fact be due to a fundamental data trend shift. In this case, the user is obliged to check the metric variable values and consider adjusting and testing the results when modifications are made to certain variables.

Such trial adjustments should be applied to the notebook, as an example, refer to [How does Drift Detection Work in your Deployment?](#) image in the previous section.

## 23.2.3. Addition of Drift Detection into the BAU Flow

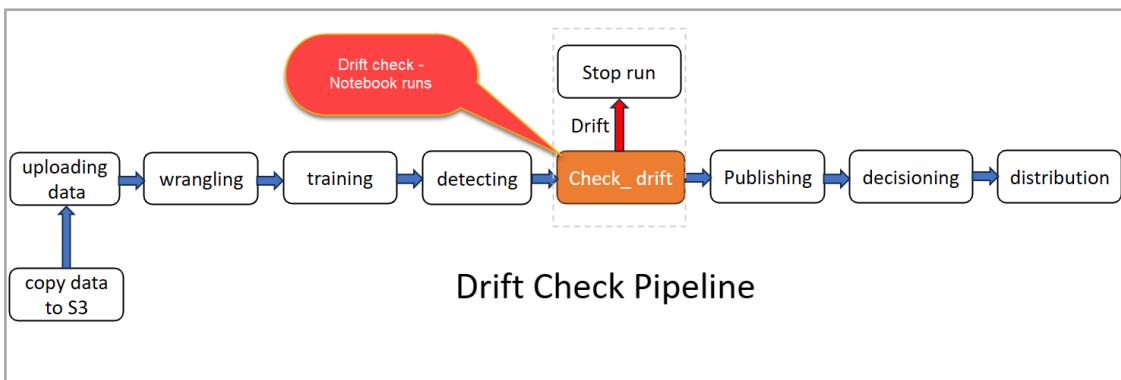
As a starting point, lets take a look at a standard end to end data pipeline as is encountered in the BAU.



**Figure 21:** Example - BAU End to End Flow

As an example, a new data batch is processed by a DAG run that starts with and includes copying data to the data repository, then passes through a series of notebooks as shown in the above flow before (in most deployments) being distributed to an alert case manager.

Drift detection is implemented by adding a scheduled drift notebook run to the BAU flow as shown in the following image.



**Figure 22:** Example Adding the Drift Detection Notebook to the BAU Run to Check for Drift

### What is checked by the drift notebook DAG?

The detailed structure and function of the drift notebook are covered in the following paragraph, but as a brief insight to the core tests performed, the drift notebook script runs and compares data model stored threshold values with percentage statistics results, derived from three key data pipeline segments:

- Raw dataset statistics
- Features distribution statistics
- Algo derived detected alert statistics

Results of the tests are logged in MLflow and should any of the test results indicate exceeded threshold values, the test run is marked as 'failed'. The user is notified of the failure, requiring an investigation into discovering the reason.

The following image shows an example of feature test run indicating passed and failed results.

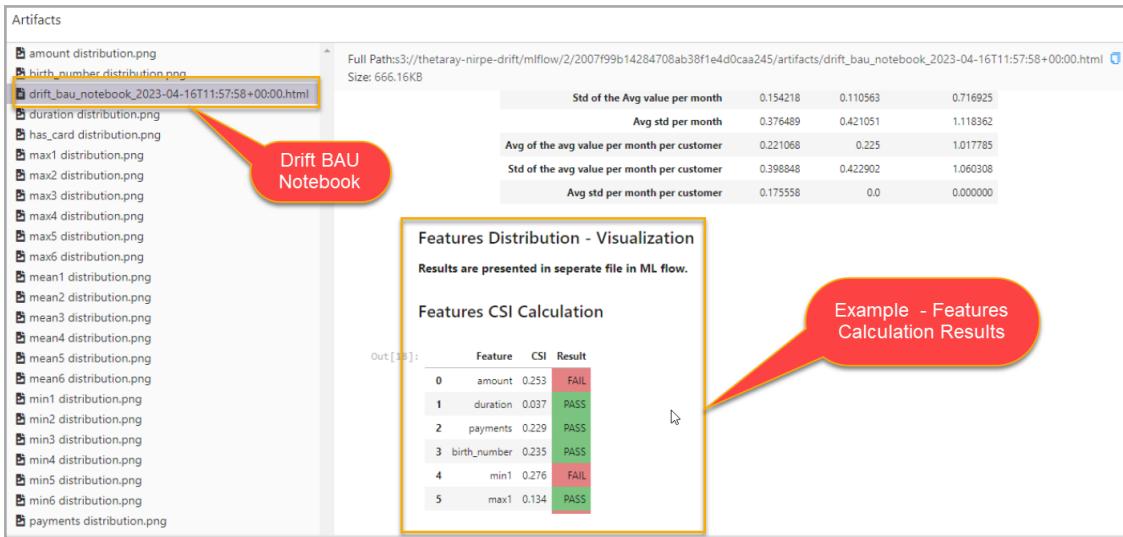


Figure 23: Example Features Distribution Test Results

### 23.2.4. When Drift is Indicated, what should the User do Next?

When drift is detected, the failure status is sent to *MLflow* (as shown in the previous paragraph). The user is required as part of his / her to investigate the reason for the failure. As discussed previously in this section, drift indication may be due to transient or event driven occurrence. There are other possible reasons for the failure indication. For instance, the most recent data batch upload could have been duplicated or perhaps become corrupted.

Specifically regarding drift:

- Drift metrics are logged to both *Opensearch* and *MLflow* (under the run associated with DAG's Task)
- An event with the drift status is sent to *Opensearch*. In case drift is detected an alert will be triggered through *Opensearch* and will be delivered to the configured destinations
- The *Airflow* DAG run will be suspended in case of a drift by default, the notebook can be altered to prevent this if needed. In this case an additional alert will be triggered by *Opensearch* to indicate a failure in execution of a DAG

In summary, the reasons for possible drift should be rationalized by you, the user. Your knowledge of the type of data, seasonal trends and possible temporary data transients, play a major part in the investigation strategy adopted.

You may decide for instance, that although drift appears to be indicated, taking all factors into account, to wait for further drift indications and for the moment at least, to simply 'keep an eye' on the situation. Then, in time if drift is continuously indicated, attempt changes to the drift notebook such as modifying threshold values, or adding or editing existing variables.

## 23.3. Productized Metrics - Information

### 23.3.1. Metrics - Transaction level

- Metrics with comparison to historical data:
  - Avg of number of transactions per month
  - Avg of avg transaction amount per month
  - Std of avg transaction amount per month
  - Std of transactions amount per month
  - Number of customers per month
  - Avg number of transactions per month per customer
  - Avg sum transactions per month per customer
  - Std sum transactions per month per customer
  - Avg of number of rejected rows
  - Avg of avg transaction amount per month

### 23.3.2. Metrics - Feature level

- Metrics with comparison to historical data:
  - count\_positive
  - count\_nulls
  - count\_all
  - positive\_prop
  - min
  - percentiles analysis
  - max
  - avg
  - stddev
  - IQR
  - Characteristics Stability Index of Analysis across all features
- Visualization of each feature's histogram

### 23.3.3. Metrics - Algorithm level

- Metrics with comparison to historical data:

- Anomaly percentage history & current periods
- Change in frequency of Occurrence of Features across Top 3 Trigger Features
- Anomaly Score PSI
- Anomaly Score Distribution

## 23.4. Drift in Machine Learning (CRA)

**Disclaimer:** The content of this sub section is draft and therefore its content should be classified as non GA.

### 23.4.1. Introduction

When a machine learning model is deployed in production, it faces real-world data. As the environment changes, this data might differ from what the model has seen during training. This phenomenon, known as data drift, can reduce the model's accuracy.

Data drift refers to changes in the statistical properties and characteristics of the input data. It occurs when a machine learning model encounters data that deviates from the data it was initially trained on. This shift in input data distribution can lead to a decline in the model's performance. Models are generally expected to perform well on data similar to the training data, but they might struggle with data that significantly changes over time.

Key points in drift identification:

1. Baseline Establishment: Define the baseline metrics based on the training data.
2. Evaluation of New Batches: Compare new data batches against the baseline to detect deviations.
3. Immediate Detection: Drift can be declared immediately if a minimum number of metrics indicate deviation, speeding up the detection process.
4. Drift analysis: investigation of the root cause and further recommendations.

### 23.4.2. Drift Identification

Once a baseline is established based on the training period, it must be evaluated for potential drift in new data batches. Every run, before new alerts are generated, drift monitoring is performed to identify deviations that could impact the quality of the alerts. If a drift is detected (failed predefined statistical tests), analysis is initiated to determine the next steps.

Drift analysis includes various tests such as data frame distribution tests, class distribution tests, and class composition tests. These tests ensure that the quality and distribution of raw data remain consistent with the training data.

Consistent monitoring and timely detection allow for appropriate corrective actions, such as retraining the model or adjusting features, to be taken before significant performance degradation occurs.

Identified drift is visually available through the Jupyter Notebooks "drift\_notebook\_CRA" or in MLflow if the notebook is run through Airflow.

Tests for Drift Identification:

- Class Percentage Z-score: Monitors the deviation of class percentages compared to the training period. An alert is triggered if the Z-score is statistically significant.
- PSI (Population Stability Index) of Anomaly Score: Tracks how probabilities of each classshift over time. An alert is triggered if the PSI is below 0.25.

If any of these tests fail, the execution is stopped, and a thorough investigation is conducted to identify the root cause of the drift. This process ensures that any drift is promptly detected and addressed to maintain model performance and reliability.

As a deviation from the baseline may be an isolated incident, retraining is recommended if X of Y (which is by default three of five) consequent batches are identified as deviating from the baseline.

### 23.4.3. Input Data

The drift monitoring notebook requires as input data the evaluated\_activities table. The following variables should be set up in the notebook:

- List of Class Probability Columns: Columns representing the probabilities for each class.
- Winning Class Column: The column that contains the class with the highest probability.
- Winning Class Probability Column: The column that contains the probability of the winning class.
- Training Period: The timeframe used for training the model.
- Test Period: The timeframe used for evaluating the model.

### 23.4.4. Drift Monitoring and Test Examples

The following tests are examples of ongoing monitoring of feature calculation algorithm performance and their suitability to the ongoing data feed. By default, these tests are for logging purposes but can be configured as "blocker tests" to stop a run in case of failure.

For each test, the process is:

1. Calculation of Metric
2. Assessment of Metric (Pass/Fail)
3. If set as a "blocker test", conduct an investigation into the root cause of failure (e.g., data drift, concept drift, or other underlying reasons)

4. Address identified issues according to their domain (e.g., retraining the model for data drift, redesigning the feature set for concept drift)

### Summary Statistics of Analysis Features across Data Frame

The summary statistics of values for each analysis feature compared to past periods help identify significant changes in feature distributions, indicating data drift.

The input features of the CRA classification AI model could be numerical, categorical, and binary. In the description below you may find the metrics specification for each column type.

#### Metrics monitored:

- Numerical Features:
  - Average and standard deviation per month
  - Number of investigated entities per month
  - Average number and standard deviation per investigated entity
  - Feature distribution over time including min value, max value, percentiles (10%, 25%, 50%, 75%, 90%, 99%), count of positive values, average, standard deviation, skewness, kurtosis, IQR, and range
- Categorical and Binary Features:
  - Count of all categories
  - Count for each category
  - Percentage of each category

#### Tests monitored:

##### CSI of Feature Values (Numerical Features Only)

Monitors shifts in feature values compared to the training period. The formula is similar to PSI but calculated per characteristic:

$$CSI = \sum_{bins} (\%Actual - \%Expected) * \ln \frac{\%Actual}{\%Expected}$$

Where

- “Actual” – Measured Values percentage of samples that fall into bin in test sample
- “Expected” – Projected Values percentage of samples that fall into bin in train sample

Suggested interpretation of results

CSI Range	Interpretation
CSI <= 0.1	Little change (no action required)
0.1 < CSI <= 0.25	Moderate change
0.25 > CSI	Significant change

As a default, If CSI is > 0.25 drift is identified for this particular feature.

**Example of the presented logs:**

	Feature	CSI	Result
0	transaction_total	0.203	FAIL
1	aml_f_1	0.332	FAIL
2	aml_f_2	0.281	FAIL
3	aml_f_3	0.325	FAIL
4	aml_f_4	0.725	FAIL
5	aml_f_5	0.214	FAIL

### Category Percentage Deviation (Categorical and Binary Input Data Features)

Monitors category percentage deviation compared to the training period. An alert is triggered if the change is statistically significant. Steps for calculations:

1. Calculate the percentage of rows for each category each month
2. Calculate the Z-score:

$$Zscore = \frac{\text{current value} - \text{means}}{\text{standard deviation}}$$

3. Define a significance level (e.g.,  $\alpha = 0.025$ ) and calculate the corresponding critical value from the normal distribution (1.96).
4. If the Z-score is above the critical value, the percentage of the class in the current month significantly deviates from the training period.

Example of the presented logs:

	date_agg_level	year_month	client_type	percent	P_value	Z_Score	Result
0	120.000	1999-12	Funds	46.154	0.038	2.073	FAIL
1	120.000	1999-12	Complex Entity / Corporate	15.385	0.304	-1.027	PASS
2	120.000	1999-12	Simple Entity / Corporate	30.769	0.601	0.523	PASS
3	120.000	1999-12	SVP	7.692	0.072	-1.802	PASS

### Class PercentMonitors deviation (AI Model Output Data - Class Probabilities)

Monitors deviation of class percentages compared to the training period using Z-score and critical values. Alerts are triggered if changes are statistically significant.

Example of the presented logs:

	date_agg_level	year_month	scrn_ml_classification	anom_percent	P_value	Z_Score	Result
0	120.000	1999-12	H	38.462	0.502	0.671	PASS
1	120.000	1999-12	M	23.077	0.679	-0.414	PASS
2	120.000	1999-12	VH	30.769	0.898	0.128	PASS
3	120.000	1999-12	L	7.692	0.134	-1.499	PASS

## Deviation in Feature Occurrence in Top 3 Trigger Features

Monitors significant changes compared to the training period. Steps for calculations:

1. Calculate the number of rows where current features were in the top 3 trigger features each month.
2. Calculate the Z-score:
- 3.

$$Zscore = \frac{\text{current value} - \text{means}}{\text{standard deviation}}$$

4. Define a significance level (e.g.,  $\alpha = 0.025$ ) and calculate the corresponding critical value from the normal distribution (1.96).
5. If the Z-score is above the critical value, the occurrence of the current feature in the top 3 trigger features significantly deviates from the training period.

Example of the presented logs:

	trigger_feature	1999-12_Zscore	1999-12_result
0	aml_f_1	-0.070	PASS
1	aml_f_2	-0.470	PASS
2	aml_f_3	0.740	PASS
3	aml_f_4	-0.270	PASS
4	aml_f_5	1.190	PASS
5	transaction_total	-0.850	PASS

Frequency o... M

	TF_1_Count_current	TF_1_Count_history	TF_1_Percent_current	TF_1_Percent_history	TF_2_Count_current	TF_2_Count_history	T
trigger_feature							
aml_f_1	1.000	41.000	33.330	28.470	0.000	23.000	
aml_f_2	0.000	22.000	0.000	15.280	1.000	26.000	
aml_f_3	1.000	18.000	33.330	12.500	1.000	22.000	
aml_f_4	0.000	30.000	0.000	20.830	0.000	24.000	
aml_f_5	1.000	13.000	33.330	9.030	0.000	21.000	
transaction_total	0.000	20.000	0.000	13.890	1.000	28.000	

## PSI of Class Proportion Change

Monitors shift in class proportions over time. PSI is calculated to measure how the distribution of customers in the population has changed. It's calculated over class probabilities of each class observed.

$$PSI = \sum((Actual\%) - Expected\%) * \left( \ln\left(\frac{Actual\%}{Expected\%}\right) \right)$$

- “Actual” – Measured Values: percentage of samples that fall into the bin in the test sample
- “Expected” – Projected Values: percentage of samples that fall into the bin in train sample

Suggested interpretation of results:

PSI Range	Interpretation
PSI <= 0.1	Little change (no action required)
0.1 < PSI <= 0.25	Moderate change
0.25 > PSI	Significant change

Example of the presented logs:

```
For the M class the anomaly score PSI = 1.594. Test is FAIL
For the L class the anomaly score PSI = 2.180. Test is FAIL
For the VH class the anomaly score PSI = 1.669. Test is FAIL
For the H class the anomaly score PSI = 1.064. Test is FAIL
```

## 24. Appendix A - Cross Platform Instance Concurrency Management

### 24.1. Overview

Multiple ThetaRay Platform Instances, each identified by a different Solution name can be deployed within a single environment to support segregating data and compute across multiple business entities (typically these are business units belonging to a single organization).

As each of these instances is scheduling its workflows using a dedicated instance of Airflow to schedule automated activities, some, using a common resource such as Postgres, would like to ensure that the database is not overloaded in the presence of parallel activities originating from different platform instances.

To support this, it is possible to configure the system with a maximum number of work slots which will be consumed by specific activities. If not enough slots are available to run the activity, the action will be suspended until slots are available. The amount of work slots required by an activity is determined by its level of parallelism. Activities consuming work slots include:

- Publishing operations of Datasets / Evaluated Activities to Postgres
- Decisioning (identify risks)
- Alert Distribution

Configuration of the maximum level of concurrency and timeouts is performed by implementation level settings within `tr_settings.py`.

The settings include:

- `total_work_slots` with default value **32**
- `request_timeout` with default value **3600** seconds
- `work_slot_timeout` with default value **300** seconds

#### Detailed Description of Settings:

- `total_work_slots` - the maximum number of slots available in the environment (across all platform instances)
- `work_slot_timeout` - the interval between retries of acquiring slots, if not enough resources are available
- `request_timeout` - the maximum amount of time to wait for slots to be available before failing

For detailed information about 'parallelism' settings controlling the number of work slots consumed by each operation, see the API definitions for the relevant activities detailed earlier in the User Guide.



## 25. Appendix B - Encrypting Existing Data

Data at Rest Encryption enables protection of sensitive data as it's stored at rest, through data encryption. The encryption process is only applied to data elements for which encryption is setup as part of the implementation data, allowing gradual adoption of the data encryption capabilities within the environment.

As sensitive data within a given data element (Dataset, Evaluation Flow, Parquet Index, Graph or Alert Mapper) is either fully encrypted or not encrypted - a conversion process is required in order to encrypt existing non encrypted data to encrypted form prior to enabling encryption on a dataset.

The following steps should be applied to migrate non encrypted data:

1. Enable data at rest encryption at environment level as detailed in the Installation Guide document
2. Configure relevant Datasets / Graph elements as encrypted / audited. It should be noted that at this point no operations should be performed at Platform and Investigation Center (mappers associated with Evaluation Flows originating from these Datasets) until the data migration process completes.
3. Use the Python APIs supporting the data migration process to convert all data elements for which encryption was enabled. The API calls should be performed from within Jupyter notebooks that are orchestrated through DAGs running on Airflow to allow parallelism and resiliency of the overall process. In case ParquetIndexes are converted to encrypted form, these should be encrypted only after all associated Datasets and Evaluation Flows have been converted.

The migration process covers the following data elements:

1. Datasets data in Minio (Parquet files).
2. Published Datasets data in Postgres.
3. Parquet Indexes associated with Dataset stored in Minio (Parquet files + indexes).
4. Evaluated Activities in Minio (Parquet files).
5. Published Evaluated activities in Postgres.
6. Graph data in Postgres.
7. Alert data in Postgres - sensitive alert fields, notes.
8. Alert data in Minio - attachments.

The following ordering should be applied to the migration process ->

1. Dataset and Evaluation Flows / Graphs in Postgres and Minio can be migrated in parallel with dependencies between different migration processes (parallelism should be based on available resources).
2. Parquet Indexes should be migrated by recreating the index from it's source - the publish\_to\_parquet APIs should be used with job\_ts date ranges covering the date ranges within the index.
3. IC migration should be performed once platform migration is concluded.

The following section details the Python APIs supporting the migration of each of the above data elements.

**Note:** A field serving as the identifier across multiple datasets (such as customer id or account id) should remain consistent and encrypted across all datasets to avoid unresolvable references (primarily when accessing the data from the Investigation Center). For example, if the Customer ID column is encrypted as part of an Input Dataset to an Evaluation Flow, the Customer ID column should be encrypted on the Transactions Dataset to enable consistent references between the Dataset, thus allowing the Transactions Tab of the Investigation Center to function correctly.

## 25.1. Postgres Data Encryption

APIs for encrypting the data in Postgres associated with the following entities - Datasets, Evaluation Flows, Graphs.

The migration process requires a dedicated Postgres user for conducting migration activities with permissions for reading / writing local files on the Postgres server. Once the migration is complete, the user should be removed from the database. The following statements should be issued in order to grant or revoke permissions to an from the user:

Use these functions:

```
from thetaray.api.encryption.migration.utils.privileges import Privilege, grant_privileges, revoke_privileges
```

for reference these are the definitions:

```
class Privilege(Enum):  
    READ_FROM_SERVER = 'pg_read_server_files'  
    WRITE_TO_SERVER = 'pg_write_server_files'
```

```
def grant_privileges(
```

```

        superuser_username: str,
        superuser_password: str,
        target_username: str,
        privileges: List[Privilege]
    ) -> None:
        """
        function for granting privileges to a user
        :param superuser_username: str username of user that can grant privileges to other
        users
        :param superuser_password: str corresponding password of the username
        :param target_username: str username of the requested user to grant the privileges to
        :param privileges: List[Privilege] list of all the privileges to grant to the user
        :return None:
        """
    """
```

```

def revoke_privileges(
        superuser_username: str,
        superuser_password: str,
        target_username: str,
        privileges: List[Privilege]
    ) -> None:
    """
    function for revoking privileges from a user
    :param superuser_username: str username of user that can revoke privileges from other
    users
    :param superuser_password: str corresponding password of the username
    :param target_username: str username of the requested user to revoke the privileges
    from
    :param privileges: List[Privilege] list of all the privileges to revoke from the user
    :return None:
    """

```

As the credentials for the user are embedded as a parameter to the Postgres related migration APIs, it is recommended to provide the credentials in encrypted form, using the following process:

```
from thetaray.api.encryption.migration.utils.aes import encrypt_string
print(encrypt_string(<superuser_password>))
```

Detail of the API's related to migrating data in Postgres:

encrypt\_published\_datasets:

API Elements	Detail
Method	encrpyt_published_datasets - encrypts all of the postgres tables specified by

API Elements	Detail
	<b>datasets_identifiers</b> according to the fields marked with <i>encrypted=True</i> on the Dataset field_list
Import	from thetaray.api.encryption.migration import encrypt_published_datasets
Parameters required	<p><b>context:</b> Thetaray JobExecutionContext</p> <p><b>datasets_identifiers:</b> list of Thetaray datasets identifiers to run the encryption process for</p> <p><b>schema:</b> the requested schema to encrypt datasets in</p> <p><b>pg_username:</b> postgres superuser username</p> <p><b>pg_password:</b> postgres superuser password</p>
Parameters optional	<p><b>data_environment:</b> Thetaray data environment to run the encryption for - sandbox or main schema</p> <p><b>n_threads:</b> determine the number of threads to execute the process with. for partitioned datasets - threading between partitions</p> <p>for non-partitioned datasets - threading between other datasets</p> <p><b>include_migrated:</b> flag to force execution on datasets that already migrated once</p>

Example of usage:

```

1 encrypt_published_datasets(
2     context,
3     ['trx_enriched', 'daily_wrangling', 'monthly_wrangling'],
4     'solution_my_solution'
5     pg_username,
6     pg_password,
7     n_threads=6
8 )

```

encrypt\_published\_evaluated\_activities

API Elements	Detail
Method	encrypt_published_evaluated_activities - encrypts all of the postgres evaluation flow tables specified by <b>evaluation_flows_identifiers</b> according to the fields marked with <i>encrypted=True</i> on the Dataset field_list
Import	from thetaray.api.encryption.migration import encrypt_published_evaluated_activities
Parameters required	<b>context:</b> Thetaray JobExecutionContext

API Elements	Detail
	<p><b>evaluation_flows_idenfiers:</b> list of Thetaray evaluation flows identidiers to run the encryption process for.</p> <p>The process encrypt the corresponding table of the evaluation flow based on the input dataset metadata</p> <p><b>schema:</b> the requested schema to encrypt evaluation flows in</p> <p><b>pg_username:</b> postgres superuser username</p> <p><b>pg_password:</b> postgres superuser password</p>
Parameters optional	<p><b>data_environment:</b> Thetaray data environment to run the encryption for - sandbox or main schema</p> <p><b>n_threads:</b> determine the number of threads to execute the process with.</p> <p>for partitioned datasets - threading between partitions</p> <p>for non-partitioned datasets - threading between other datasets</p> <p><b>include_migrated:</b>flag to force execution on evaluated activities that already migrated once</p>

### Example Usage

```

1  encrypt_graphs(
2    context,
3    ['netviz_graph', 'another_graph'],
4    'solution_my_soltution'
5    pg_username,
6    pg_password,
7    n_threads=6
8  )

```

API Elements	Detail
Method	encrypt_graphs-it encrypts the nodes and edges tables in postgres based on the graph metadata
Import	from thetaray.api.encryption.migration import encrypt_graphs
Parameters required	<p><b>context:</b> Thetaray JobExecutionContext</p> <p><b>graphs_identifiers:</b> list of Thetaray graphs identifiers to run the encryption process for.</p> <p>The process encrypt the corresponding nodes and edges tables of the graph based on the graph metadata</p> <p><b>schema:</b> the requested schema to encrypt graphs in</p>

API Elements	Detail
	<p><b>pg_username:</b> postgres superuser username</p> <p><b>pg_password:</b> postgre superuser password</p>
Parameters optional	<p><b>data_environment:</b> Thetaray data environment to run the encryption for - sandbox or main schema</p> <p><b>n_threads:</b> determine the number of threads to execute the process with.</p> <p>for partitioned datasets - threading between partitions</p> <p>for non-partitioned datasets - threading between other datasets</p> <p><b>include_migrated:</b> flag to force execution on graphs that already migrated once</p>

Example of usage

```

1 encrypt_graphs(
2     context,
3     ['netviz_graph', 'another_graph'],
4     'solution_my_soltution'
5     pg_username,
6     pg_password,
7     n_threads=6
8 )
```

## 25.2. Minio Data Encryption

Detail of the API's related to migrating data in Minio:

### 25.2.1. encrypt\_parquet\_datasets:

API Elements	Detail
Method	encrpyt_parquet_datasets - encrypts all of the Minio tables specified by <b>datasets_identifiers</b> according to the fields marked with <i>encrypted=True</i> on the Dataset field list
Import	from thetaray.api.encryption.migration import encrypt_parquet_datasets
Parameters required	<p><b>context:</b> Thetaray JobExecutionContext</p> <p><b>datasets_identifiers:</b> list of Thetaray datasets identifiers to run the encryption process for</p> <p><b>warehouse_db_name:</b> string (ie.'public','manager')</p>
Parameters	<b>minio_access_key:</b> S3 access key for login to Minio

API Elements	Detail
optional	<b>minio_secret_key</b> : S3 secret key for login to Minio

Example of usage:

```

1 | encrypt_parquet_datasets(
2 |   context,
3 |   datasets_identifiers=['trx'],
4 |   warehouse_db_name='public'
5 | )
```

### 25.2.2. encrypt\_parquet\_evaluated\_activities:

API Elements	Detail
Method	<b>encrypt_parquet_evaluated_activities</b> encrypts all of the Minio evaluation flow tables specified by <b>evaluation_flows_identifiers</b> according to the fields marked with <i>encrypted=True</i> on the Dataset field_list
Import	from thetaray.api.encryption.migration import: encrypt_parquet_evaluated_activities
Parameters required	<b>context</b> : Thetaray JobExecutionContext <b>evaluation_flows_identifiers</b> : list of Thetaray evaluation flows identifiers to run the encryption process for. <b>warehouse_db_name</b> : string (ie. 'public', 'manager')
Parameters optional	<b>minio_access_key</b> : S3 access key for login to Minio <b>minio_secret_key</b> : S3 secret key for login to Minio

Example Usage

```

1 | encrypt_parquet_evaluated_activities(
2 |   context,
3 |   evaluation_flows_identifiers=['ef_monthly'],
4 |   warehouse_db_name='public'
5 | )
```

### 25.2.3. publish\_to\_parquet

**Note:** Parquet Indexes should be repopulated from their respective sources in order to encrypt their data. This process should be done after the source for the index has been encrypted (Minio data for datasets / evaluated activities).

API Elements	Detail
Method	publish_to_parquet
Import	publish_to_parquet
Parameters required	The publish_to_parquet is an existing and general platform API, not relevant specifically for this project and the parameters are similar to every other use case

Example of usage

```

1 for dataset_identifier, index_identifier in zip(datasets_identifiers, indexes_
2     identifiers):
3     publish_to_parquet(
4         context,
5         dataset_identifier='',
6         index_identifier=''
7     )

```

## 25.3. IC Data Encryption

### 25.3.1. encrypt\_ic

API Elements	Detail
Method	encrypt_ic: Triggers encryption of ic data based on ic_namespace, solution_name and evaluation flow specified.
Import	from thetaray.api.encryption.migration import: encrypt_ic
Parameters required	<b>ic_namespace:</b> The relevant Investigation Center namespace <b>solution_name:</b> The relevant solution to run encryption for <b>evaluation_flow_identifier:</b> The evaluation flow to encrypt the its corresponding mapper

Example Usage

```

1 | encrypt_ic(

```

```
2 | 'apps_namespace',  
3 | 'your_solution',  
4 | 'evaluation_flow_in_solution'  
5 | )
```

## 25.4. Reference Notebooks

The reference notebooks are provided via Solutions side-branch under tools/encryption\_migration. Although they're only presented as a reference, the execution of their functionality is required for completing the migration process.