­­

Communications Mining Dispatcher Framework

Queue Feeder for RPA Processes that use a Communications Mining Stream

[Overview 3](#_Toc126333520)

[High level UiPath + Communications Mining Architecture 4](#_Toc126333521)

[Fetch and Advance Loop 5](#_Toc126333522)

[UiPath Dispatcher Framework- Main Features 5](#_Toc126333523)

[High Level Diagram 6](#_Toc126333524)

[Settings 6](#_Toc126333525)

[Exception Handling 8](#_Toc126333526)

[Architecture 8](#_Toc126333527)

[Dependencies 8](#_Toc126333528)

[States 9](#_Toc126333529)

[Shared Variables 9](#_Toc126333530)

[Communications Mining specific Workflows 10](#_Toc126333531)

[Using the Framework 14](#_Toc126333532)

[Changes to the Framework Files 14](#_Toc126333533)

# Overview

The Communications Mining Dispatcher Framework is a UiPath Studio template that can be used when integrating Communications Mining with RPA implementations. Its objective is to take the comments from a **Communications Mining Stream** and add them one by one, into one or many UiPath Orchestrator **Unique References Queues (when creating your queues, select the” Enforce unique references” checkbox).** You should define one queue per each of your downstream automation processes that might need the data from the Stream as input. By default, in the configuration file, we’ve set up two process queues (**for Process A and B**), and one generic one, but you can configure as many as you need.

The template is based on the Robotic Enterprise Framework (REFramework), so it covers all the essential Best Practices in RPA Projects (flexible configuration through the Config.xlsx file, exception handling, retry mechanisms, meaningful logging***).***

***Please consult the official REFramework documentation if you are not familiar with it, prior to working with the Communications Mining Dispatcher***:



Starting from a REFramework project, we have encapsulated the two key operations that need to be performed when consuming data from a Communications Mining Stream: the fetching and advancing of the Stream, where **fetching** is the process of getting data out of a Communications Mining Stream and **advancing** is essentially marking the comments as read in the Stream so that we do not return the same ones next time we fetch from a Stream. We’ll discuss them in more detail in the next chapter.

By default, the framework will stop the fetching and advancing cycle when it passes the end of the Stream (the stream has no more “unread” comments). However, if you would prefer it, you can configure it to run continuously, even when it reaches the end of the Stream: simply mark the **ExitOnEmptyStream setting as FALSE in the Cofig.xlsx file.** In this case it will cycle infinitely, and whenever new data becomes available in the stream, it will process it instantly, with no need to wait for the time when the framework’s process was scheduled to run.

The end goal is to have the comments available in a usable Orchestrator Queue, one queue item per comment, containing their corresponding data and predicted Labels and Entities. This way, the consumer downstream automations will have access to the predicted information.

The data is not added to its corresponding queue without passing the **Validation Rules**. There are some basic Validation rules already defined in the framework (mainly for understanding to which process every item pertains) but you can add your own validation algorithms in the code. Also, simply as an example, in the Config.xlsx file, we have **separate validation settings sheets for each downstream automation process (ProcessAValidations, ProcessBValidations)**. Since they were configured just as examples for theoretical processes, feel free to add your own sheets and settings.

***Note: Make sure you don’t have multiple settings with the same name in the Config file, as they will all be added in the same Config dictionary, and they will override each other. In the file, we offered some examples of naming conventions for the validation settings that might be useful. The logic in the workflow that checks the validation rules follows the same conventions, so be careful when you’re implementing your own, in case you want to change them!***

If the validation fails, the information is added to a Human in the Loop queue (with unique references as well), to be validated by a human in Action Center. ***You can add the name of your HitL queue in the Config file.***

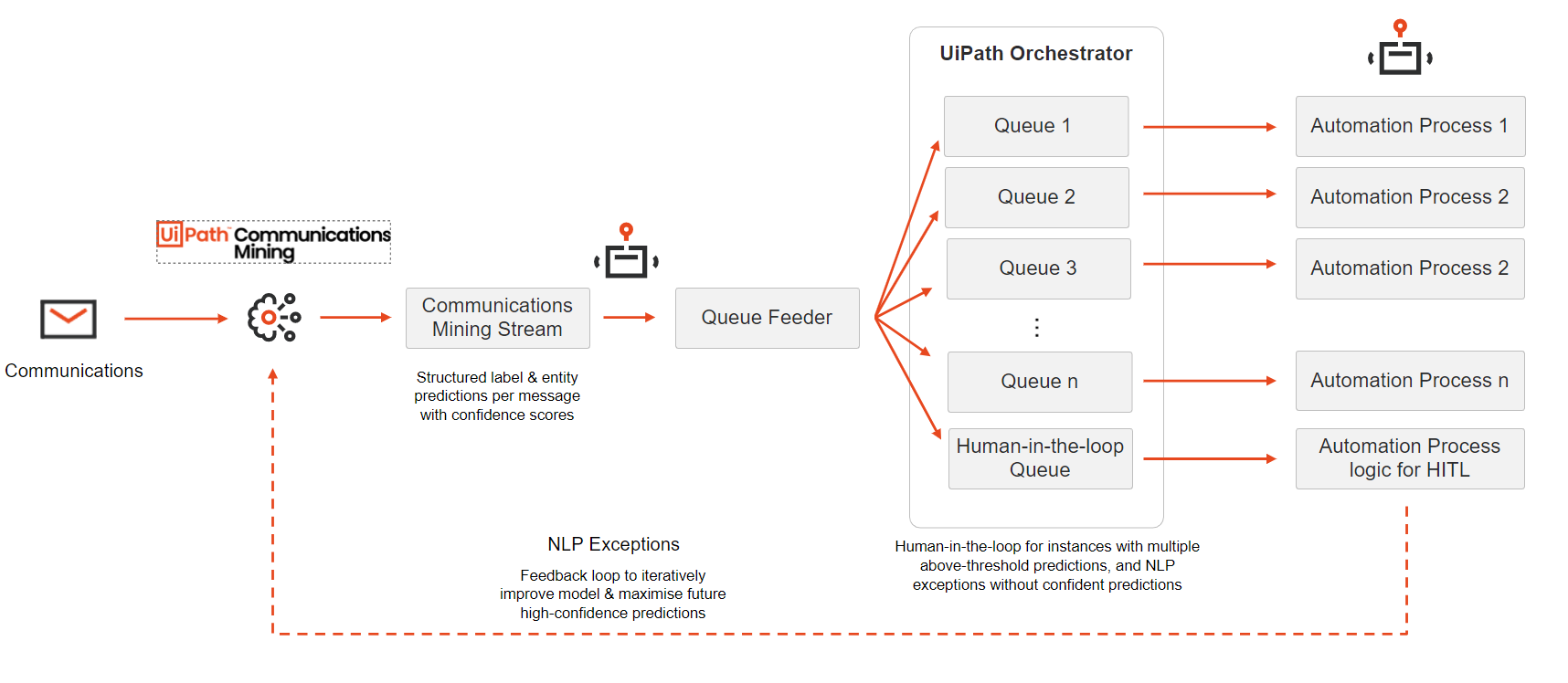
***Note: We recommend that you define a trigger on the Human in the Loop queue, that starts a new Orchestration Process for each new item added in the queue, which would create a task in Action Center. The task would contain the data retrieved from Communication Mining for the current item, and the human could validate it before sending it into its corresponding downstream automation process queue.***

# High level UiPath + Communications Mining Architecture

After successfully training a Machine Learning Model in Communications Mining, we can create a new Stream and configure the thresholds for each of the concepts that we have trained. **A Stream defines a collection of comments from a dataset**. It enables persistent, stateful iteration through the comments, with predicted labels and entities computed using the pinned model version.

***We recommend you follow the official Communications Mining Documentation and Academy for the model training steps and details on all the concepts involved.***

The integration of UiPath Studio with Communications Mining basically consists in the **consumption of each of the comments** from the Communications Mining Stream, and the usage of their data and predicted labels and entities in **one or multiple downstream processes**, automated using UiPath. In the diagram bellow, the generic UiPath+ Communications Mining integration is described:



The official approach recommends that for n automations, n + 1 processes are configured in UiPath: **n RPA Processes and one Queue Feeder**. A single feeder process is introduced which is responsible for reading the structured communications out of Communications Mining's Stream and distributing them to the relevant RPA processes via Orchestrator Queues. Any exceptions that may occur due to Communications Mining's extraction can then be marked for manual Human Validation. The processes that get items out of the queues will be standard UiPath automations which read their input data from the queue item’s data.

## Fetch and Advance Loop

For parsing the Stream to consume each of its comments, we need to implement the Fetch and Advance Loop. as described below:

|  |  |
| --- | --- |
| 1. Every Stream has a current comment | 1. You can fetch comments starting at this current comment. Bellow we're fetching 2 |
| 1. Every comment returned from a Stream will have a **sequence\_id**   Text  Description automatically generated | 1. We can use this **sequence\_id** to advance the current comment to the next in the queue. Now, when we fetch 2, we will return comments 2 and 3 |

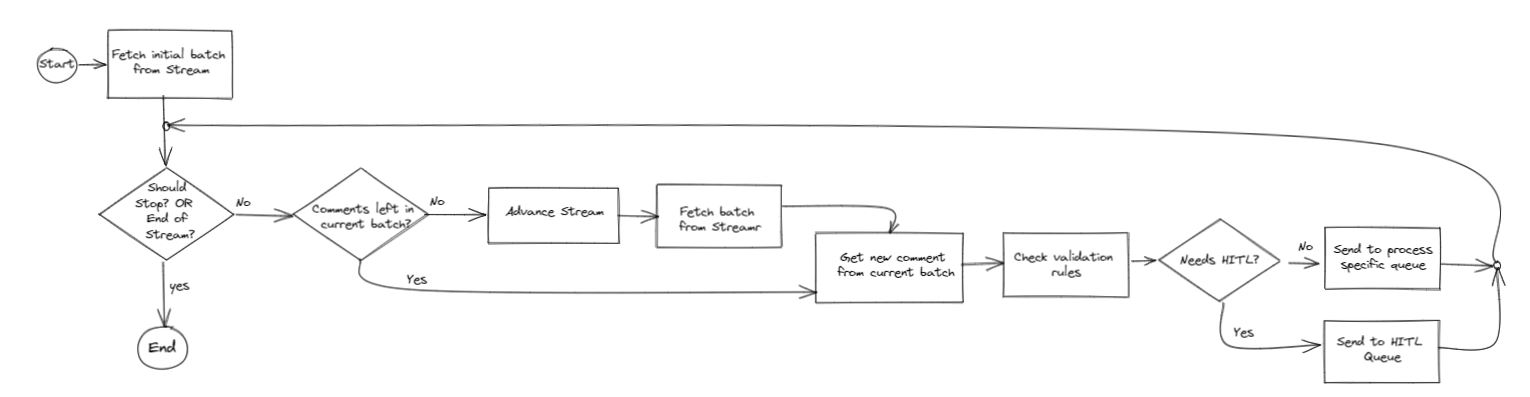
# UiPath Dispatcher Framework- Main Features

For populating the Input Queues of your Consumer RPA Processes with the data retrieved from a Communications Mining Stream, you will need to follow every time, the Fetch and Advance Loop approach described above. So, since the main steps remain the same, regardless of your specific use-case, we have created the current Communications Mining Dispatcher Framework, to be used as a template for that one Queue Feeder Process of your Communications Mining +UiPath integration.

By simply modifying the necessary settings to connect it to your own Stream, you save implementation time, and you can be sure that the framework follows all the required RPA Best Practices.

Just like REFramework, it can be used exactly as it is, as a **Plug And Play Framework,** since without any modification to the code (just by adding your settings in the Configuration Excel file), it gets each of the comments from the defined Stream (as objects of type CommunicationsMining.Result, defined in the **CommunicationsMining** package(*See Dependencies section*), and adds their data in a corresponding Queue. Alternatively, it can be completely customized, and you can add your own logic of the steps to be taken in the Feeder Process (the rules for validating the Communications Mining predictions, for example)

## High Level Diagram



The Communications Mining Dispatcher Framework uses the Fetch and Advanced Loop Approach described above, and we can advance one comment at a time, or one batch of comments at a time (we can set up the size of the batch in the Config file). Remember that the Dispatcher is used as **feeder to one or multiple downstream processes,** so in the Config file we also define the corresponding queue for each of these processes and the rules to add the item to the queue.

The overall steps are as follows:

1.We **Fetch** the initial batch of comments. This will return a **batch of comments** from the Communications Mining Stream, the **Sequence\_Id** of the last item in the batch and the number of items **Filtered Out** of the current batch (if there can be filters applied in Communications Mining for the current Stream).

2. **If no exception occurred** (we successfully connected to the Stream) and **we’re not at the end of the Stream**, we check if there are **items left to process in the current batch** (we might even Fetch batches with no comments, if there are filters applied in Communications Mining for the current Stream, and none of the comments in the current batch applies).

* 1. If yes, we process them one by one: Depending on the Consumer RPA Process to which the current item belongs, we check the validation rules of the item’s data.
  + If it passes, we **add a new Queue Item**, with all its data and predicted Labels and Entities to its corresponding Queue (as set in the Config excel file).The UID of the item will be the set as the reference of the QueueItem. **Your queue needs to have the “Enforce unique references” checkbox selected** to avoid creating duplicates in the queue!
  + If it doesn’t pass the validation rules, or if there were Multiple Labels predicted for the current item, we add its data and predicted Labels and Entities to the **Human In The Loop Queue to be Manually Validated**
  1. If there are no more items in the current batch, we first **advance** the Stream (using the **Sequence\_Id** of the last item of the batch, to move past it) and then we **fetch** a new batch from the Stream (which will start with the next comment after the last one in the previous batch)

1. If we’re at the end of the Stream (If the last Fetch retrieved **no comments in the batch** and there are **no filtered out comments**, we know we advanced past the end of the Stream) the processing **ends.**

## Settings

All the settings that we need for configuring the Dispatcher are found in the **Data/Config.xlsx file**

* **The Settings Sheet –** make sure you add the Communications MiningAPITokenCredential Asset in Orchestrator

|  |  |
| --- | --- |
| **Name** | **Description** |
| Orchestrator**ProcessA**QueueName | This is the Queue where our Dispatcher will push VALID Comments to be processed by the **RPA Consumer Process A** |
| Orchestrator**ProcessB**QueueName | This is the Queue where our Dispatcher will push VALID Comments to be processed by the **RPA Consumer Process B** |
| Orchestrator**HITL**QueueName | This is the Queue where our Dispatcher will push Comments that did not pass the Validation Rules defined for their corresponding Process. The HITLQueue will be processed by the Human In The Loop, Orchestration Process that creates Validation Actions for each of the queueitems added. |
| Orchestrator**General**QueueName | This is the Queue where our Dispatcher will push Comments that were not categorised for a specific RPA Consumer Process. |
| Communications MiningApiTokenCredential | The Communications Mining API Token needed for Fetching from the Stream and advancing within the Stream, stored in a Credential Asset |
| ExitOnEmptyStream | If this setting is False, the framework will run continuously, even when we reach the end of the Stream. |

* **The Assets Sheet –** make sure you add the corresponding assets in Orchestrator

|  |  |
| --- | --- |
| **Name** | **Description** |
| StreamProject | This is the name of the project that you created your dataset in. This is the text before the / in datasets drop down at the top of Communications Mining. For example, in bayes-inc/integrations-tutorial, the dataset owner is bayes-inc. |
| StreamDatasetName | This is the name of your dataset. This is the text after the / in datasets drop down at the top of Communications Mining. For example, in bayes-inc/integrations-tutorial, the dataset name is integrations-tutorial. |
| StreamName | This is the API name of the Stream that you created. You can find this by viewing the Streams page. |
| StreamSize | This is the number of comments that you want to read from the Stream |
| Communications Mining BaseUrl | This is the ENDPOINT URL OF THE Communications Mining API |

* **Each of the Process Validation Sheets**

|  |  |
| --- | --- |
| **Name** | **Description** |
| <<ProcessName>>\_Label | The naming convention of the setting is : The label that marks a comment as being designated to be handled by current process+ "\_"+"Label" keyword.  The value of it is : the name of the downstream process  Example: Name: Policy\_Label; Value: ProcessA |

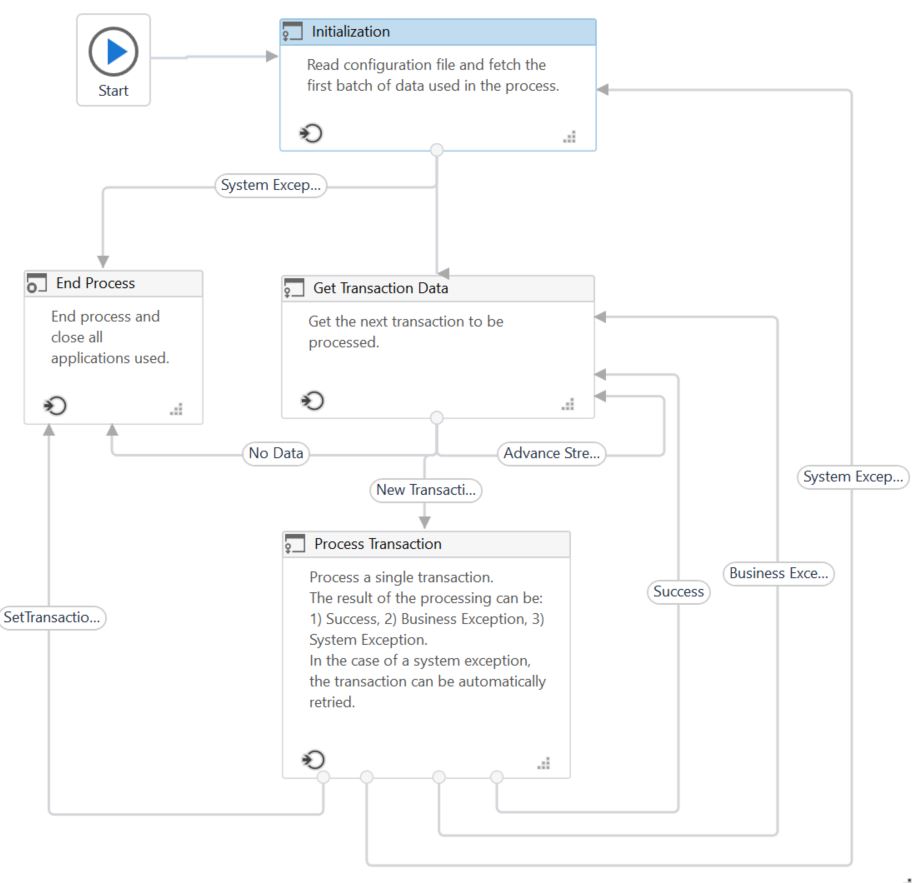
Since the Dispatcher can populate input Queues for one or more Downstream Processes, we propose that you create a new Sheet in the Config file for each of the Processes, in which you will define the validation rules for that Process. The naming convention of the Sheet should be: “<<ProcessName>>Validations”. By Default, the Config file contains 2 Validations Sheets for Process A and Process B

## Exception Handling

The Framework collects every Exception that occurs during the processing of each of the Transaction Items (Communications Mining Comments) into two DataTables: one for System Exceptions and another for BusinessRuleExceptions.

In the End Process State you can use the Tables for Handling the Exceptions as per your Business Logic ( Create Excel Files with them, send them attached to a reporting email, etc)

# Architecture



## Dependencies

We have created Custom Activities that handle the main operations that could be performed from UiPath to integrate with Communications Mining: **Fetch Stream, Advance Stream.**Also, the Framework’s Transactions are of type **CommunicationsMining.Result**, a data type defined in the package that will hold all the pieces of information defined for each comment and its corresponding predicted Labels and Entities

You need to have the **CommunicationsMining** package in one of your feeds, in order for the Dispatcher Framework to load correctly.

At the moment, you can download it from Marketplace: [https://marketplace.uipath.com/listings/communication-mining-Streams](https://marketplace.uipath.com/listings/communication-mining-triggers) .It will be added in the official UiPath feed in the future

## States

Since the Framework is basically a REFramework, it’s a State Machine with the exact same States, so we recommend ***you consult the official REFramework documentation*** for more details on each State.

The only modification is the addition of the **Advance Stream Transition** between the Get Transaction Data State and the Process Transaction State. In case there are no items to process in the current batch fetched, the execution returns to the GetTransactionData State for further advancement in the Stream.

## Shared Variables

The variables bellow are declared in the Main.xaml file and shared as arguments to the workflows invoked in the Framework or they simply decide the execution flow through the framework:

|  |  |
| --- | --- |
| **Variable Name** | **Description** |
| **ShouldStop (Boolean)** | True when the job is forcefully stopped (from Orchestrator) |
| **TransactionItem(CommunicationsMining.Result);** | The Transaction Item is represented by one comment from the Communication Mining Stream. We’re processing one item at a time and adding its data to the corresponding queue |
| **SystemException(Exception);** | Used during transitions between states to represent exceptions other than business exceptions. |
| **BusinessException(BusinessRuleException);** | Used during transitions between states and represents a situation that does not conform to the rules of the process being automated. |
| **TransactionNumber(Int32);** | Sequential counter of transaction items. |
| **Config(Dictionary(String,Object));** | Dictionary structure to store configuration data of the process (settings, constants , assets and Validation properties). |
| **RetryNumber(Int32);** | Used to control the number of attempts of retrying the transaction processing in case of system exceptions. |
| **TransactionData(IList(CommunicationsMining.Result));** | The batch of comments currently retrieved from the Stream, by the latest Fetch |
| **ConsecutiveSystemExceptions(Int32);** | Used to control the number of consecutive system exceptions. |
| **BusinessExceptionsDT(DataTable);** | Table with details on the BusinessRulesExceptions occurred during the processing of the Transactions. One row contains info about one faulty transaction |
| **ApplicationExceptionsDT(DataTable);** | Table with details on the System Exceptions occurred during the processing of the Transactions. One row contains info about one faulty transaction |
| **GlobalRetryInterval(TimeSpan);** | The global retry interval set by default for every Retry Scope in the Framework. |
| **GlobalMaxAttempts(Int32);** | The global nr of Max Attempts set by default for every Retry Scope in the Framework. |
| **CurrentSequenceId(String);** | The Sequence ID retrieved by the latest Fetch of a Stream Batch. It’s the Sequence Id of the last item in the Current Stream Batch |
| **CurrentBatchFilteredResults(Int32);** | The number of items that don’t fit the Filter that was defined for the Stream in Communication Mining and were filtered out by the latest Fetch (Filtered out of the Current Fetched Batch ) |
| **CommunicationsMiningApiToken(SecureString);** | The API Token defined in Communication Mining. Its value should be stored in a Credential Asset in Orchestrator |
| **CurrentBatchNumber(Int32);** | It’s a good practice to split your Stream into multiple batches (to help with the performance time of retrieving the data) This will tell us what’s the current batch that’s being processed |
| **ShouldAdvanceTheStream(Boolean);** | In case there are no items to process in the current batch fetched, the execution returns to the GetTransactionData State for further advancement in the Stream |

## Communications Mining specific Workflows

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Workflow Name** | **Description** | **Input Arguments** | **Output**  **Arguments** | **Invoked Workflows** |
| **GetNextStreamBatch** | We're trying to get the next Stream Batch of Communications Mining comments.  The Fetch Stream activity will connect to Communications Mining and will populate the Fetch output object with:  -the collection of results (of the size we requested)  -the Sequence Id of the current batch (the Sequence ID of the last comment in the retrieved batch)  - the number of Filtered out comments (in case we applied filters on our Communications Mining Stream, the comments that don't fit the filter will be skipped)  The Fetch activity performs an HTTP request to Communications Mining | in\_Config(Dictionary(String,Object));  in\_CommunicationsMiningApiToken (Secure String); | out\_CurrentSequenceId(String);  out\_CurrentTriggerBatch(IList(CommunicationsMining.Result));  out\_FilteredResults(Int32); | # |
| **AdvanceStreamBatch** | We're trying to Advance the Stream of CommunicationsMining comments.  The Advance Stream activity will connect to CommunicationsMining and using as an input the Sequence Id of one of the Comments in the Stream, it will mark the comments (before and including the one with the given Sequence ID) as read in the Stream so that we do not return the same ones next time we fetch from a Stream. If you fetch multiple times in a row without advancing a Stream, you will get the same emails every time.  The Advance activity performs an HTTP request to CommunicationsMining | in\_CurrentSequenceId  (String);  in\_Config  (Dictionary(String,Object));  in\_CommunicationsMiningApiToken (SecureString); | # | # |
| **GetTransactionData** | Get a transaction item from an array comments.  Since there are multiple transactions, we use the argument in\_TransactionNumber as an index to retrieve the correct transaction to be processed. If there are no more transactions left in the current batch, we need to advance the Stream and fetch the next batch. If we fetch multiple times in a row without advancing a Stream, you will get the same results every time.  IF There are items in the batch and there are still some left to process we take the instance of the next Transaction Item in the batch.  Otherwise, we flag the fact that there are no more items left to process in the current batch and that we need to advance the Stream . We don't set io\_TransactionItem to nothing here as that would stop the processing of the whole framework, and maybe there are still items in next batches .  The STOP condition is set in the Get Transaction Data STATE | in\_Config(Dictionary(String,Object));  in\_CommunicationsMiningApiToken(SecureString);  io\_TransactionData(IList(CommunicationsMining.Result));  io\_TransactionItem(CommunicationsMining.Result);  io\_CurrentSequenceID(String);  io\_TransactionNumber(Int32);  io\_FilteredResults(Int32);  io\_CurrentBatchNumber(Int32); | out\_ShouldAdvanceTheStream(Boolean);  io\_TransactionData(IList(c:Result));  io\_TransactionItem(CommunicationsMining.Result);  io\_CurrentSequenceID(String);  io\_TransactionNumber(Int32);  io\_FilteredResults(Int32);  io\_CurrentBatchNumber(Int32); | AdvanceStreamBatch.xaml;  GetNextStreamBatch.xaml; |
| **CheckValidationRules** | This is a basic validation algorithm example that decides whether the predictions are valid solely on the number of labels predicted for the current item. If we have one Label => we have a successful validation and we just need to get the name of the downstream process from the config file. If we have multiple labels, the automatic validation is set as unsuccessful.  Add your own logic for deciding the name of the Consumer process and whether the predictions on the items are Valid OR need human validation  If we have only one label predicted for the current item, we have to get the name of its corresponding process.  We take the name of the Downstream (Consumer) Process from the Config file, based on that ONE label predicted for the current item. In the Config file, the naming convention of the process name setting is : The label of the comment + "\_"+"Label" keyword.  If the current item was predicted to have multiple labels, we need the human to decide how to proceed in the downstream automation. So the automatic validation success should be marked as false so that the current item will be added to the Human in the Loop queue for later manual Validation | in\_Config(Dictionary(String,Object));  in\_TransactionItem(CommunicationsMining.Result); | out\_SuccessfulValidation  (Boolean);  out\_ConsumerProcessName  (String); | # |
| **CreateDictionaryFromCommunications MiningItem** | We'll need to add the information taken from CommunicationsMining for the current Item to a queue. So we're creating a dictionary based on it. We'll use the dictionary for adding the defining properties of the new queue item. | in\_CommunicationsItem(CommunicationsMining.Result); | out\_Dictionary(Dictionary(String,Object)); | # |
| **AddTransactionItemToQueue** | Adding a new item to the queue. All of its properties should be already set up in the in\_QueueItemProperties dictionary  **Make sure that your queue has the Enforce unique references checkbox selected** | in\_MaxAttempts(Int32);  in\_RetryInterval(TimeSpan);  in\_QueueItemProperties(Dictionary(String,Object))>;  in\_QueueItemReference(String)>;  in\_QueueName(String)>;  in\_OrchestratorFolder(String); |  |  |
| **Process** | The purpose of the Dispatcher is to populate corresponding queues with the the information obtained in Communications Mining for each of the items so that they can be processed by Consumer Processes in the Downstream Automation.  In this workflow, we have to add the current item to its corresponding queue.  Steps:  1.We're creating a dictionary based on the TransactionItem. We'll use the dictionary for adding the defining properties of the new queue item  2. Based on the information obtained in Communications Mining for the current item, we're deciding its corresponding Consumer Process and checking the validation rules against the predicted data  3.If the validation is successful, we're adding the item to the queue of the Consumer Process. If not, we're adding it into the Human In The Loop queue, to be validated and potentially processed by a human  For the current Transaction:  If a BusinessRuleException is thrown, the transaction is skipped.  If another kind of exception occurs, the current transaction can be retried. | in\_TransactionItem(CommunicationsMining.Result);  in\_Config(Dictionary(String,Object));  in\_RetryInterval(TimeSpan);  in\_MaxAttempts(Int32); |  | CreateDictionaryFromCommunicationsItem.xaml;  CheckValidationRules.xaml;  AddTransactionItemToQueue.xaml; |
| **ExceptionsHandler** | This workflow should be used as a final Exception Handler in the Framework.  If the input DataTables are populated, they contain details on all the Application and/or Busines Rule Exceptions that occured during the current run of the process | in\_Config(Dictionary(String,Object));  in\_RetryInterval(TimeSpan);  in\_MaxAttempts(Int32);  in\_dt\_BusinessExceptions(DataTable);  in\_dt\_ApplicationExceptions(DataTable);  in\_FatalException(Exception); |  |  |

# Using the Framework

## Changes to the Framework Files

Before using the Framework:

* make sure you configure all the required assets in Orchestrator (see the Settings chapter) and make the necessary modifications in the **Data/Config.xlsx** file.
* make sure that the queue in which you will add the items exists in Orchestrator **and it has the “Enforce unique references” checkbox selected** to avoid adding duplicates to the queue and processing the same item multiple times in the downstream automations.
* Add your own validation rules in **Communications Mining/ CheckValidationRules.xaml .** At the moment we only check if the current item has multiple labels predicted. If yes, the validation fails. If not, we take the Process Name corresponding to the current item, based on its label.