ExxonMobil

Data Lake Agile Factory (DLAF) – PI Data Collection Service
Technical Design Document

Owner: **UDA Program – Data Lake Team**
Maintained by: **Guanjie Shen, Dexter Obsio**
Effective Date: **January 23, 2020**

Change Summary

Version	Date	Change/ Updated by	Summary
1.0	01/28/2020	Guanjie Shen	Adding Azure Service Design/Implementation
1.1	01/29/2020	Dexter Obiso	Architecture Diagram, Data Model Diagram
1.1	01/30/2020	Guanjie Shen	PI Service Design
1.2	02/24/2020	Guanjie Shen	PI Streaming Design

Pending Issues

Date	Summary

Referenced Document(s)/Position Paper(s)

Version	Name	Link to NGS Document Repository (e.g. SharePoint)

Review/Approve Summary

Date	Role (Reviewer/ Approver)	By	Comments

Contents

Service Background and Overview ***4***

Service Architecture Diagram ***5***

Logical Architecture **5**

Physical Architecture (DEV Environment) **7**

Physical Architecture (PROD Environment) **8**

Data Model Diagram ***9***

PI Collection Service ***10***

Installation and Setup **10**

Create Installer 10
Configuration Settings 10
Install Instructions 11

System Design Considerations **11**

Software Design Pattern 11
Multithreading 12
Error Handling & Logging 13

Service Layer **13**

Program.cs 13

Logic Layer **13**

PICollectionLogic.cs 13
TaskHelper.cs 13
ScheduleHelper.cs 13
PIDataHelper.cs 13
ParseHelper.cs 13
CSVHelper.cs 14
AzureStorageHelper.cs 14
AzureLoggingHelper.cs 14
LocalLoggingHelper.cs 14

Data Layer **14**

Domain Layer **14**

Framework Layer **14**

PIAFService.cs 14

PI Metadata Collection Service ***15***

PI Streaming Service (Design Only) ***17***

Data Factory Pipelines for Data Ingestion ***18***

PI Hourly Pipeline **18**

Service Background and Overview

The purpose PI Collection Service is to extract PI Tag (and metadata) data from the on premise historian and push the data points in Azure. The service should be able to perform various types of interpolation calculations, and supports hourly scheduled collection along with adhoc collection. Collection can also be configured to occur on a tag-by-tag basis.

This service should satisfy the following system design requirements for PI data collection:

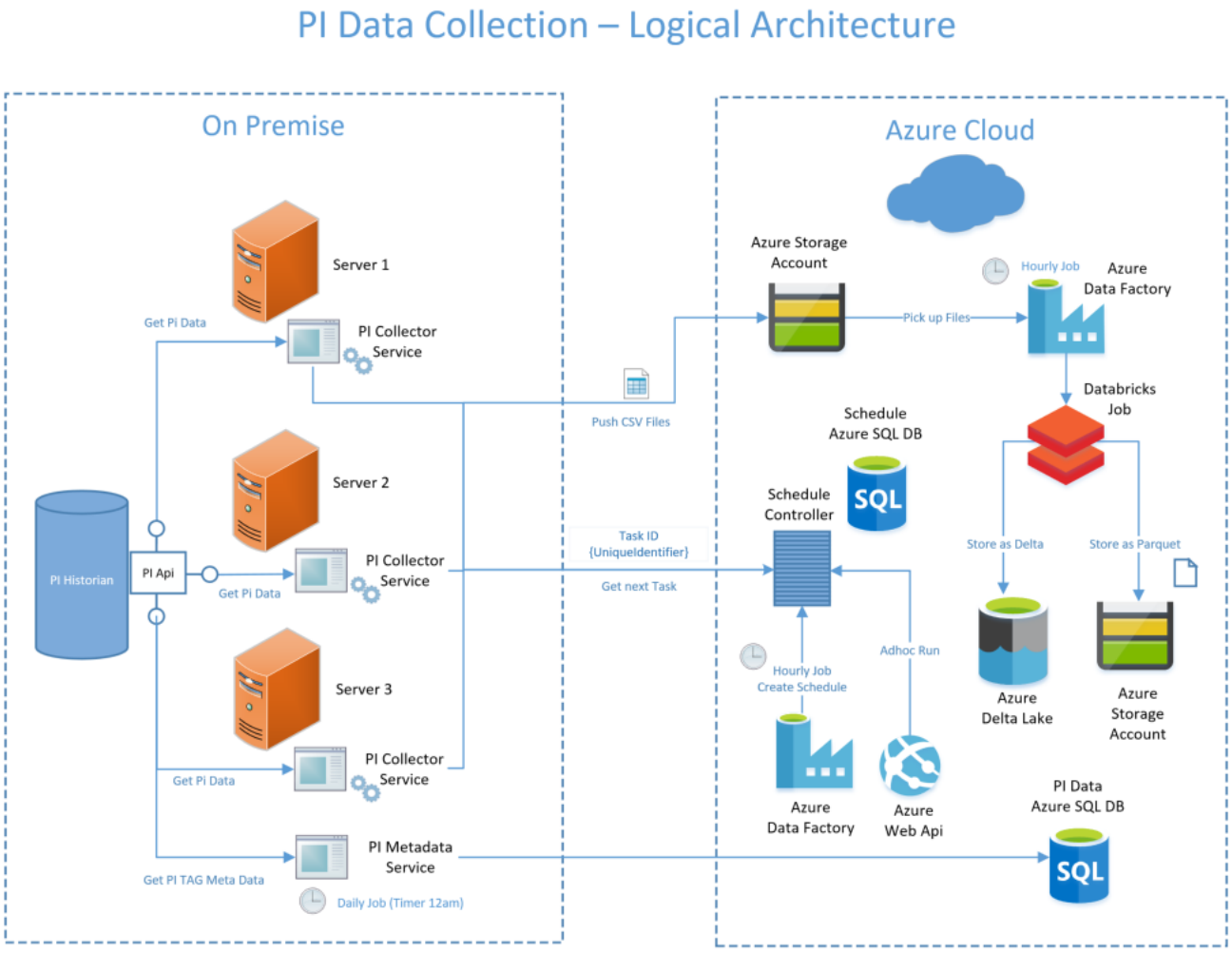
- Focus on Scalability and Ease of Deployment
- High Degree of Concurrency
- Fault-Tolerant and Verbose Logging/Traceability
- Resiliency and Recoverability
- Atomic Transactions
- Decoupled Architecture

The PI Collection Service will be composed of the following systems:

- Windows Service deployed on On-premise VMs
- Web API deployed on Azure
- SQL Server Database deployed on Azure
- Azure Storage Accounts
- Azure Data Factory pipelines
- Azure Databricks notebooks
- Azure Delta Lake

Service Architecture Diagram

Logical Architecture



ADF Hourly Job Create Schedule – an hourly job creates a schedule record in the Azure SQL Database for the previous hour. It then generates a Task per Pi Tag for the hour. The Tag Header table contains a list of all Pi Tags.

Azure Web Api ADHOC Run – an api to trigger an adhoc schedule record in the Azure SQL Database for any given time frame. It has two methods:

- Create Schedule – creates an adhoc schedule based on the timeframe
- Generate Tasks - generates a Task per Pi Tag for the timeframe

Azure SQL Schedule Controller – a group of stored procedures that act as the Business Layer to manage the schedules and tasks. The following stored procedures are:

- sp_pi_createschedule_hourly – creates a schedule for the previous hour. This is called by the ADF Hourly Job.

- sp_pi_generate_tasks – generates all tasks for all schedules with a status of READY for all pi tags in the PI_HEADER table.
- sp_pi_generate_tasks_foraschedule – creates the tasks for a schedule.
- sp_pi_getnexttask – assigns a task for a caller (server/thread) based on a priority: Hourly before Adhoc then by status in the order of READY, FAILED with less than 3x of retries, IN PROGRESS/HOLD longer than 5 mins, ASSIGNED but not started over 5 mins. It handles concurrency by ensuring that no task is assigned to multiple callers. It assigns a unique sequence number for that task and locks it for the calling server/thread.

Pi Collector Service – a windows service that has multiple functions:

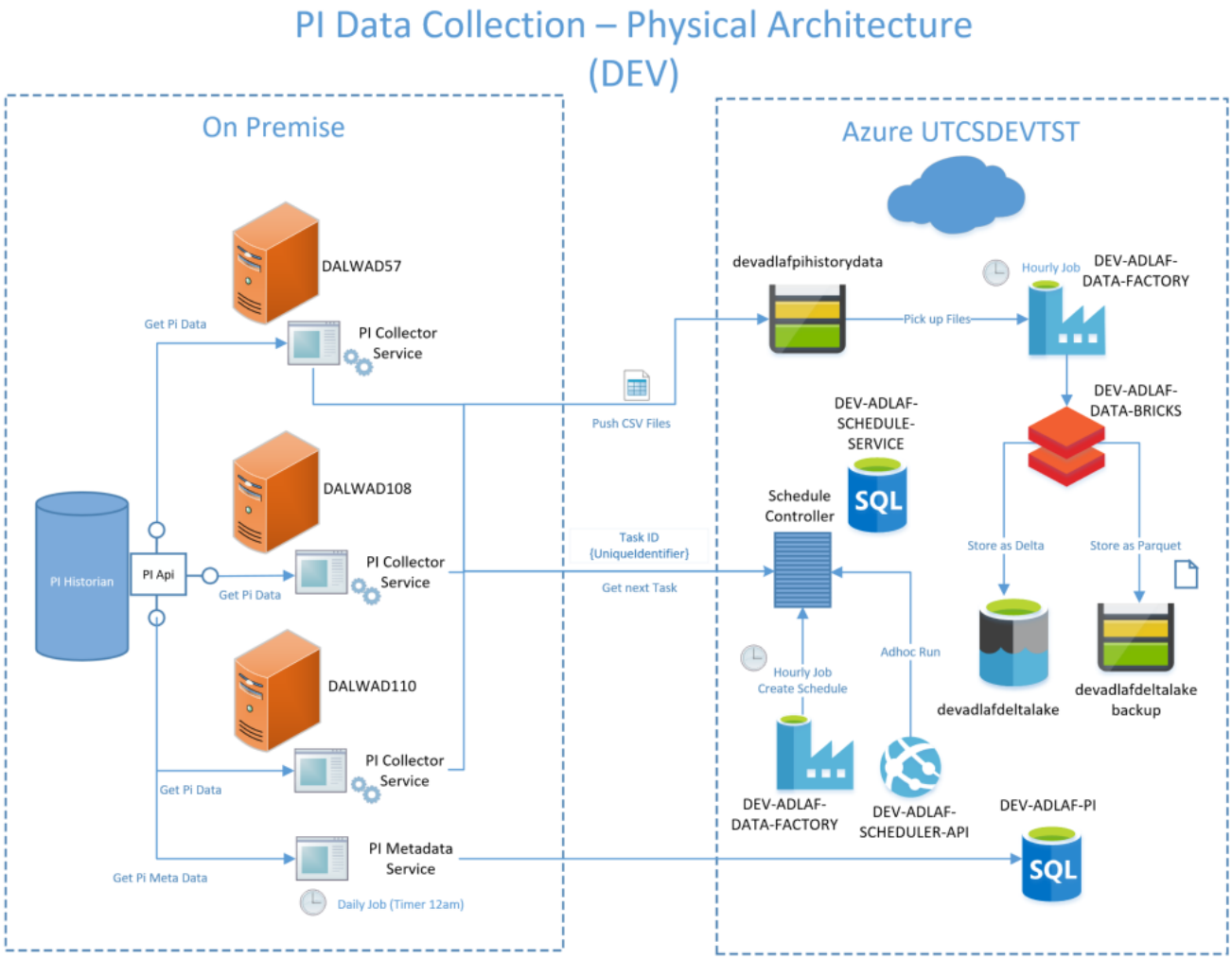
- Gets new tasks from Azure SQL Server
- Calls PI API to get the data for a single tag for a given time frame
- Pushes the PI Data into Azure Storage as a CSV file
- Can scale up multiple threads (upto 20) to perform each task concurrently
- Can scale out to as many servers as needed (currently 3 in DEV, 4 in PROD)
- Logs the task and schedule surveillance data

Pi Metadata Collection Service – a windows service that retrieves PI Tag metadata:

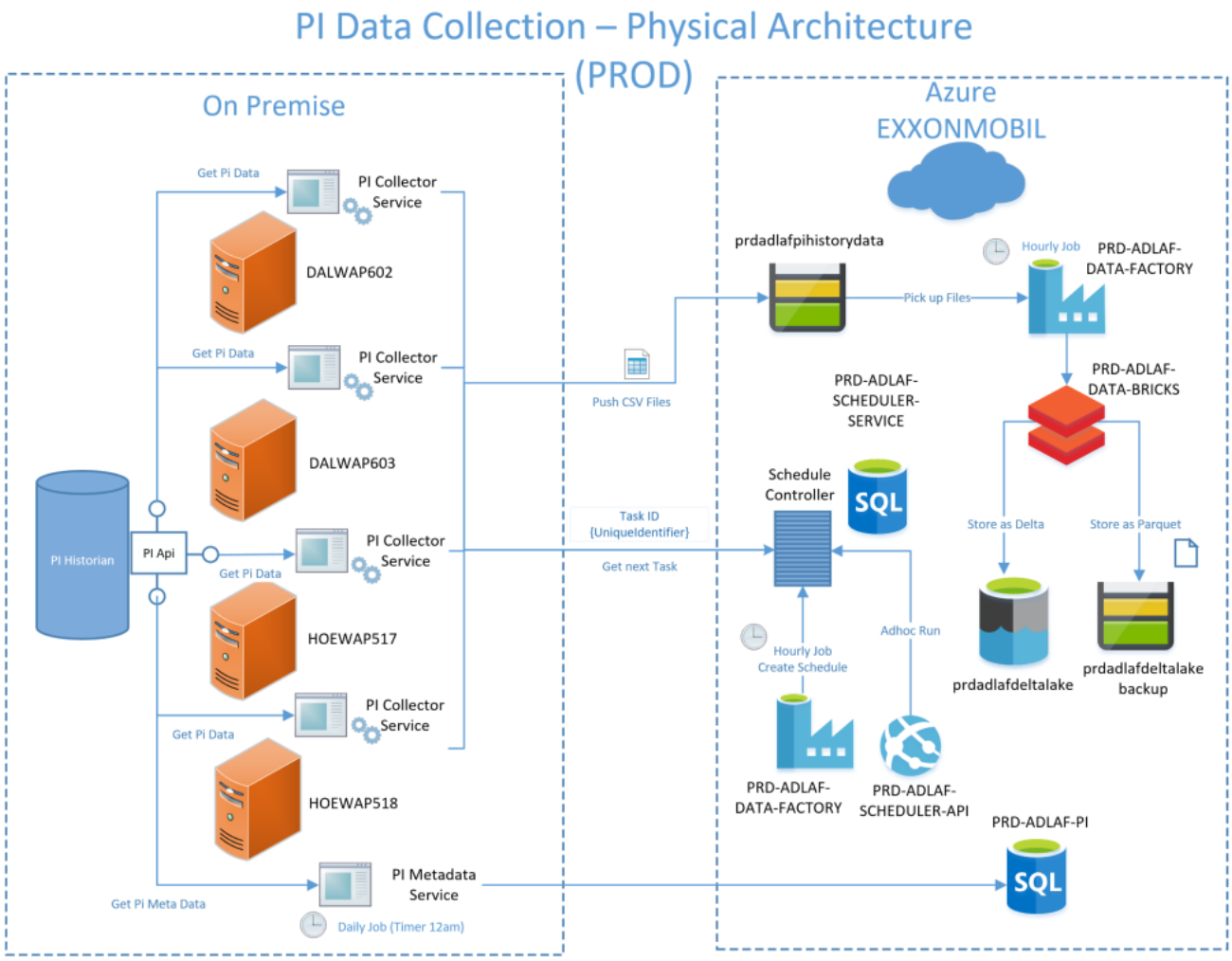
- Gets PI Tag details for Azure SQL Server (PI Database)
- Calls PI API to get metadata for a single tag
- Pushed PI metadata back into Azure SQL Server (PI Database)
- Designed to run single threaded on a single VM (on-premise)
- Logging is pushed into Azure and stored locally

ADF Hourly Job Pick up CSV Files – an hourly job that runs to pickup any CSV files from the storage account. It then calls Azure Databricks notebook to process the CSV and save it into an Azure Delta Lake (delta table) and Parquet file format in a storage account for backup.

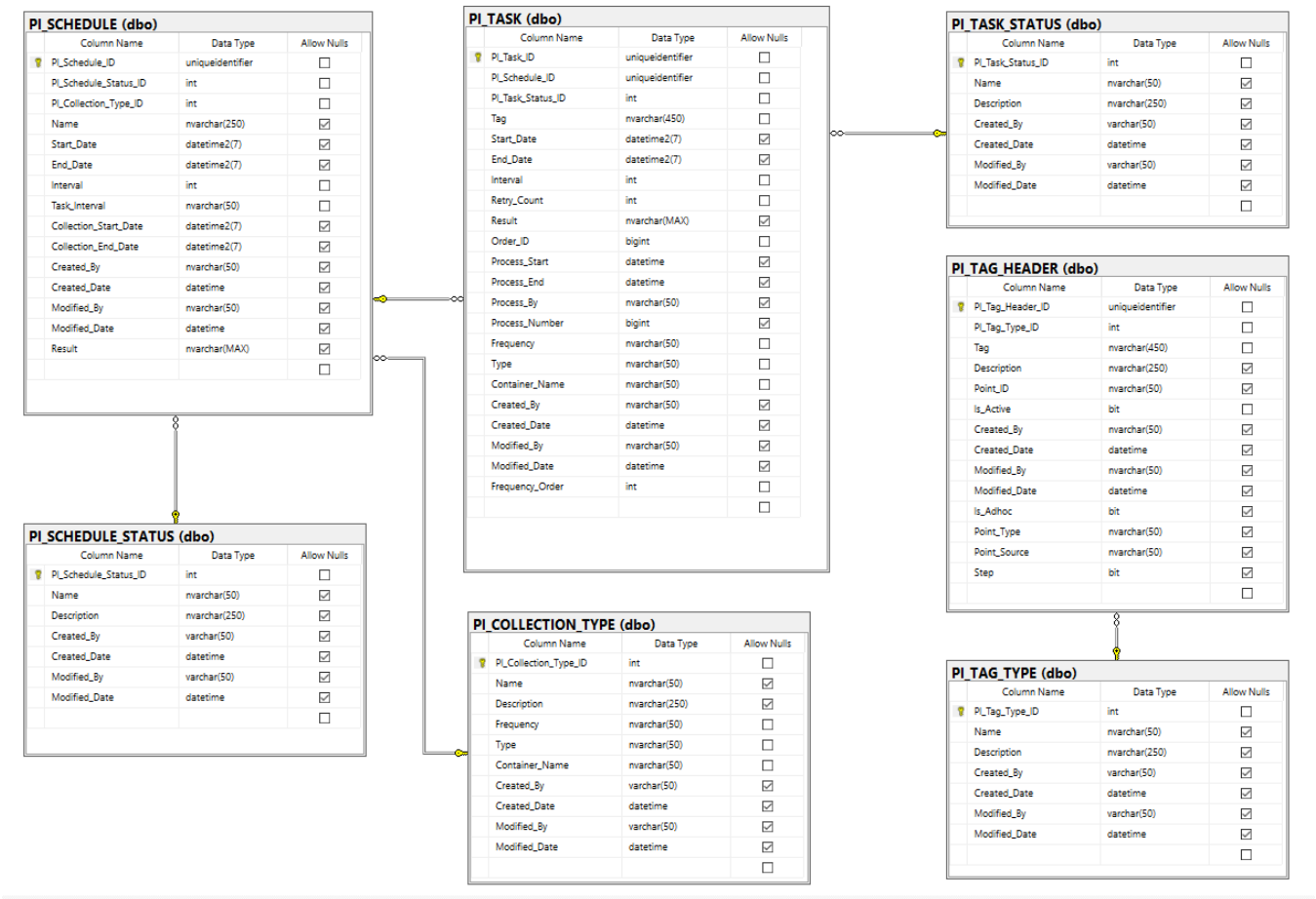
Physical Architecture (DEV Environment)



Physical Architecture (PROD Environment)



Data Model Diagram



PI_TAG_HEADER – contains a full list of Pi Tags. An Is_Active flag indicates if it will be included in the hourly job. A Is_Adhoc flag indicates if it will included with the adhoc job run.

PI_SCHEDULE – contains all the schedules and it's run details. Run details contain information on how the tasks should be broken out (HOUR/DAY), PI parameters and schedule surveillance data (status, process_start, process_end).

PI_TASK – contains all the tasks for a schedule along with PI parameters and surveillance data (status, process_start, process_end, retry_count). Any errors or info regarding execution is also logged in the Result field.

PI_SCHEDULE_STATUS – a look up table of status for schedules

- DRAFT – Not ready to be tasked
- READY – Ready to be tasked
- TASKED – schedule has been tasked (tasks created)
- IN PROGRESS – one or more task has started processing
- COMPLETED –all the tasks finished successfully
- FAILED – one or more task failed

PI_TASK_STATUS – a look up table of status for tasks

- READY – Task is ready to be picked up for processing
- ASSIGNED – Task has been picked up by a thread to be processed
- IN PROGRESS – Task has started processing
- COMPLETED – Task finished successfully
- FAILED – Task finished with exception

PI_COLLECTION_TYPE – a look up table of types for a schedule

- HOURLY-AVG - hourly job for average interpolation
- HOURLY-RAW – hourly job for raw data extraction
- ADHOC-AVG - adhoc run for average interpolation
- ADHOC-RAW - adhoc run for raw data extraction

PI Collection Service

The PI Collection Service is a windows service developed using .NET Framework 4.6.1 to execute PI data collection tasks created by the Scheduling Service.

Installation and Setup

The following section goes into detail on how to make changes to the solution, deploy, and install on the required servers.

Create Installer

1. Build the solution using the "Release" configuration
2. Copy the contents of **… \DLAF_SERVICE_PI\DLAF.SERVICE.PI.ADHOC\bin\Release** into a new directory
3. Copy the C:\Windows\Microsoft.NET\Framework64\vx.xxxx\InstallUtil.exe into the same directory
4. These are all the required files to perform the installation on the server

Configuration Settings

These are settings that can be configured for each particular installation (*DLAF.SERVICE.PI.ADHOC.exe.config*) All config settings are required

```
numberOfThreads: Number of concurrent executions per process (see multi-threading section for more details)

localExportDirectory: Directory for where local PI data files should be written (ensure that the account executing the service has write access to that location)

localLogDirectory: Directory for where local log files should be written (ensure that the account executing the service has write access to that location)

apimSubcriptionKey: Subscription key to APIM

tenantEndpoint: OAuth 2.0 token endpoint (Get from Azure AD)
```

```
clientID: Client ID regisgration for this service in Azure

scope: Azure AD scope for APIM

clientSecret: Client Secret for the Azure AD App registration for this service

loggingUrl: URL for Azure APIM logging service

storageAccountName: Azure Storage account for exported PI Data

storageAccountKey: Azure  Storage account key for exported PI Data

databaseServer: Database server containing the scheduling DB

databaseName: Database Name containing the scheduling DB

databaseId: Database ID (Read/Write Permissions)

databasePassword: Database Password
```

Install Instructions

1. Copy the installer folder to the server you would like to install in
2. Inside the copied folder launch CMD
3. Run the follwing: installutil DLAF.SERVICE.PI.ADHOC.exe
4. Supply the user credentials for the service account with access to the PI server (saUIT-UDA-DLAF01-DEV@NA.xom.com / Password in ITPA)
5. Launch Windows Services and start the service
6. Set the windows service for delayed start & restart the service after 10 minutes in the event of failure

System Design Considerations

The following section goes over the thought process and design considerations for the choices of features that have been included or excluded from the system.

Software Design Pattern

This service has been developed using the repository pattern with nested projects with in a single solution:

- *DLAF-SERVICE-PI-ADHOC (Service Layer)*
  - Implements multithreading on business logic methods
- *DLAF-SERVICE-PI-ADHOC.Business (Logic Layer)*
  - Contains core business logic
  - Utilizes data access layer to extract data objects

- *DLAF-SERVICE-PI-ADHOC.Data (Data Access Layer)*
  - Connects to SQL server and acts as a CRUD repository for data models

- *DLAF-SERVICE-PI-ADHOC.Domain (Domain Layer)*
  - Contains a collection of entity and data transfer objects

- *DLAF-SERVICE-PI-ADHOC.Framework (Framework Layer)*
  - Abstracts methods required to invoke the OSIsoft PI SDK

The choice to use .NET 4.6.1 was due to the OSIsoft PI SDK only supporting that version of .NET Framework. The data access *DLAF-SERVICE-PI-ADHOC.Data* uses .NET Standard 2.0 so that it's compatible with the Entity Framework Core (EFCore) library and .NET 4.6.1 projects.

Language version is set to C# 7.2 in order to support asynchronous methods on the Main method in Program.cs

The solution has been set up to execute as a console app when running locally, and a windows service when deployed. This is due to the the lack of troubleshooting capabilities when executing windows services using Visual Studio.

Multithreading

From initial benchmarking of collection times, we found that the average time to execute a single collection (single tag for one hour) averaged to be between 1-3 seconds. For a given hour, the minimum requirement would be to able process up to 14000 collection tasks (for ~7000 tags) within 60 minutes.

Using a single threaded design, this would mean that if this service was executing on a single VM it may take up to :

(14,000 tasks x 3 seconds)/60 = **Upwards of 700 minutes**

If executed across three VMs, this would go down to 233 minutes; however, with the limitations of an on-premise system we are limited in the degree to which we can horizontally scale.

When considering a multithreaded system design, there were a few things to take into consideration:

- Collection speeds and parallelism is further bottlenecked by the PI Historian itself (only hosted on two servers for Kearl)
- At what degree of task parallelism do we experience no further benefit in collection time, either limited by the PI Historian or the hosted server
- How do we ensure tasks are executed independently and execeptions will not affect other threads and the service overall

After testing, we discovered that 20 theads per process allowed us deliver sufficient performance while maintaining acceptable CPU and memory usage on the VMs.

With three VMs, this would mean 60 concurrently executing threads:

(14,000 tasks x 3 seconds)/60/60 threads = **Upwards of 11.6 minutes**

Error Handling & Logging

Error handling within the service is primarily contained in the Service and Logic layers through a series of Try-Catches.

In the event of an exception, the error message is sent to the logging API in Azure, logged in the database on Task table, and also recorded in a log file on the server (in the event of no internet connectivity).

Log files are on the server are stored in two different ways:

- Service level exceptions are stored in the log file prefixed with the server name
- Task level exceptions are stored in the log folders prefixed with the Schedule GUID and the files are separated based on tag name

When an expection is caught the thread should immediately terminate, restart and start executing the next available task.

In the event the entire service crashes, the windows service is scheduled to continuously retry to restart after a 10 minute delay.

Service Layer

Program.cs

Start()

This method executes the collection and is determines how many task/threads to kick off based off the app configuration. It also contains the delay duration before the service checks to see if there is another active task.

Logic Layer

PICollectionLogic.cs

CollectData()

Main logic method that executes collection. Is invoked by the service layer to initialize collection for a given task.

TaskHelper.cs

Helper class that contains methods to perform business logic associated with PI Scheduled Tasks

ScheduleHelper.cs

Helper class that contains methods to perform business logic associated with PI Schedules

PIDataHelper.cs

Helper class that contains methods to perform data cleansing on PI data retrieved from the PI SDK

ParseHelper.cs

Helper class that contains methods to perform generic data cleansing

CSVHelper.cs

Helper class that contains methods to write PI data to CSV (if required)

AzureStorageHelper.cs

Helper class that contains methods to write PI data to Azure blob storage

AzureLoggingHelper.cs

Helper class that contains methods to write logging data into the logging microservice in Azure

LocalLoggingHelper.cs

Helper class that contains methods to write logging data into the local on-premise file share

Data Layer

Implements Entity Framework Core through .NET Standard 2.0 (scaffolded off existing database)

Domain Layer

Contains classes for entities and data transfer objects.

Framework Layer

Contains classes associated with working with the PI SDK

PIAFService.cs

This class initializes all methods required to use the PI SDK for data collection.

Constructor

The constructor calls the Init() method

Init()

This method initializes the connection to the PI Server, using the default server connections specified on the VM. This can be viewed using the PI SDK tool installed.

CollectDataAvg(string PITagName, DateTime StartTime, DateTime EndTime, double Interval)

This method collects PI data by taking the average for a given time range and interpolatation interval.

CollectData(string PITagName, DateTime StartTime, DateTime EndTime)

This method collects raw PI data for a given time range.

PI Metadata Collection Service

The PI Metadata Collection Service is a windows service developed using .NET Framework 4.6.1 to execute PI data metadata collection. This service has been built based upon the PI Collection Service.

Installation and Setup

The following section goes into detail on how to make changes to the solution, deploy, and install on the required servers.

Create Installer

5. Build the solution using the "Release" configuration
6. Copy the contents of **… \DLAF_SERVICE_PI\DLAF.SERVICE.PI.ADHOC\bin\Release** into a new directory
7. Copy the C:\Windows\Microsoft.NET\Framework64\vx.xxxx\InstallUtil.exe into the same directory
8. These are all the required files to perform the installation on the server

Configuration Settings

These are settings that can be configured for each particular installation (*DLAF.SERVICE.PI.ADHOC.exe.config*) All config settings are required

`numberOfThreads:` Number of concurrent executions per process (see multi-threading section for more details)

`localExportDirectory:` Directory for where local PI data files should be written (ensure that the account executing the service has write access to that location)

`localLogDirectory:` Directory for where local log files should be written (ensure that the account executing the service has write access to that location)

`apimSubcriptionKey:` Subscription key to APIM

`tenantEndpoint:` OAuth 2.0 token endpoint (Get from Azure AD)

`clientID:` Client ID regisgration for this service in Azure

`scope:` Azure AD scope for APIM

`clientSecret:` Client Secret for the Azure AD App registration for this service

`loggingUrl:` URL for Azure APIM logging service

`storageAccountName:` Azure Storage account for exported PI Data

`storageAccountKey:` Azure Storage account key for exported PI Data

`databaseServer:` Database server containing the scheduling DB

```
databaseName: Database Name containing the scheduling DB

databaseId: Database ID (Read/Write Permissions)

databasePassword: Database Password
```

Install Instructions

7. Copy the installer folder to the server you would like to install in
8. Inside the copied folder launch CMD
9. Run the follwing: installutil DLAF.SERVICE.PI.ADHOC.exe
10. Supply the user credentials for the service account with access to the PI server (saUIT-UDA-DLAF01-DEV@NA.xom.com / Password in ITPA)
11. Launch Windows Services and start the service
12. Set the windows service for delayed start & restart the service after 10 minutes in the event of failure

System Design Considerations

The following section goes over the thought process and design considerations for the choices of features that have been included or excluded from the system.

Software Design Pattern

This service has been developed using the repository pattern with nested projects with in a single solution:

- *DLAF-SERVICE-PI-ADHOC (Service Layer)*
  - Implements multithreading on business logic methods
- *DLAF-SERVICE-PI-ADHOC.Business (Logic Layer)*
  - Contains core business logic
  - Utilizes data access layer to extract data objects
- *DLAF-SERVICE-PI-ADHOC.Data (Data Access Layer)*
  - Connects to SQL server and acts as a CRUD repository for data models
- *DLAF-SERVICE-PI-ADHOC.Domain (Domain Layer)*
  - Contains a collection of entity and data transfer objects
- *DLAF-SERVICE-PI-ADHOC.Framework (Framework Layer)*
  - Abstracts methods required to invoke the OSIsoft PI SDK

The choice to use .NET 4.6.1 was due to the OSIsoft PI SDK only supporting that version of .NET Framework. The data access *DLAF-SERVICE-PI-ADHOC.Data* uses .NET Standard 2.0 so that it's compatible with the Entity Framework Core (EFCore) library and .NET 4.6.1 projects.

Language version is set to C# 7.2 in order to support asynchronous methods on the Main method in Program.cs

The solution has been set up to execute as a console app when running locally, and a windows service when deployed. This is due to the the lack of troubleshooting capabilities when executing windows services using Visual Studio.

PI Streaming Service (Design Only)

The PI Streaming Service is a windows service developed using .NET Framework 4.6.1 to execute high frequency PI data collection utilizing PI Data Pipes and Azure Event Hub.

Data Factory Pipelines for Data Ingestion

Data generated by the PI Collector are processed by Azure Data Factory (ADF) pipelines.

PI Hourly Pipeline

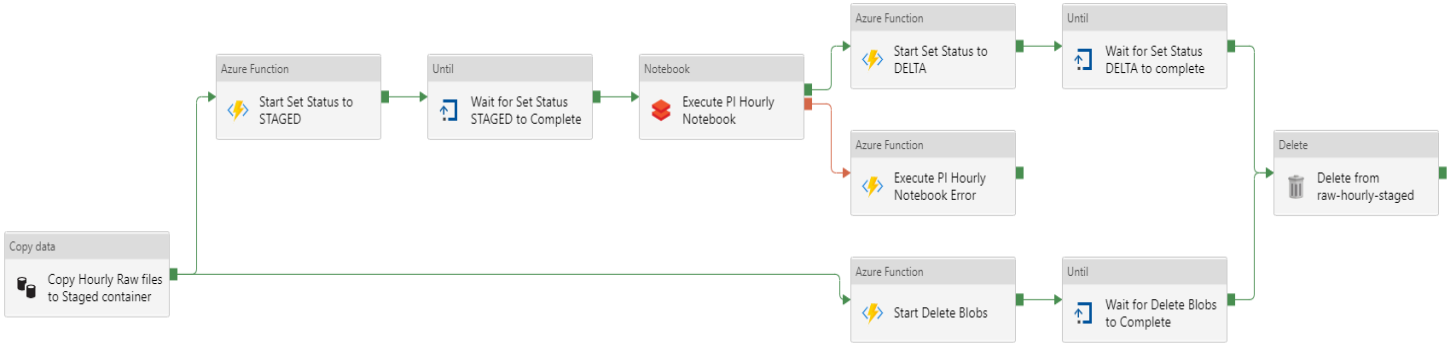


Fig 1: ADLAF_PI_HOURLY_DATA ADF pipeline diagram

This pipeline ingests data generated by the hourly PI Collector process. Here are the steps in the above pipeline:

1. Copy Hourly Raw files to Staged container
   - Copy all csv files from raw-hourly container into the raw-hourly-staged container.
   - Enable hourly collector schedules to be run at the same time this ingestion process is running. All newly created files in the raw-hourly container will be process the next time when this ADF pipeline is run.

2. Start Set Status to STAGED
   - Update the task status in Azure SQL table PI_TASK in the SCHEDULE-SERVICE database to staged status (PI_Task_Status_ID=10).
   - This is done using the Azure Function App ADLAF_ADF_HELPER. This function app is written using Durable Pattern to allow the step to run longer than the maximum timeout of 230 seconds set by Azure Data Factory.
   - This step invokes the SetStatusHttpStart function, which gets the function parameters and kicks off the orchestrator process.
   - The orchestrator process returns a list of URIs. One of these URI is used by the next step to check the status of the 'Set Status' operation.
   - This POST function takes three parameters:
     - container: name of the container with all the hourly csv files
     - status: task status id
     - tasks: number of tasks to be updated at a time. This is optional with default set to 500.
   - Sample body content:
     {"container":"raw-hourly-staged","status":"10"}

3. Wait for Set Status STAGED to Complete
   - Check the status every 30 seconds
   - Use the URI returns in the "statusQueryGetUri" value from the previous step to get status

- Loop until status is not “Pending” or “Running”. The status would be “Completed” if the previous process finishes successfully.

4. Execute PI Hourly Notebook
   - Append records to the PI raw data delta file.

5. Start Delete Blobs
   - Delete csv files from the original source container, raw-hourly.
   - This is another durable function from the Azure Function App ADLAF_ADF_HELPER.
   - This step invokes the DeleteBlobsHttpStart function, which get the function parameters and kicks off the orchestrator process.
   - This POST function takes two parameters:
     - rawContainer: name of the container with all the hourly csv files
     - stagedContainer: name of the staged container

     Sample body content:

     {"rawContainer":"raw-hourly","stagedContainer":"raw-hourly-staged"}

6. Wait for delete blobs to Complete
   - Check the status every 30 seconds
   - Use the URI returns in the “statusQueryGetUri” value from the previous step to get status
   - Loop until status is not “Pending” or “Running”. The status would be “Completed” if the previous process finishes successfully.

7. Delete from raw-hourly-staged