Loka Demo Documentation

Created by: Jasminko Sofo

Date: Sarajevo 12/12/2022

Contents

[Introduction 3](#_Toc121871923)

[Technology Stack 4](#_Toc121871924)

[Architectural Overview 5](#_Toc121871925)

[Infrastructure 5](#_Toc121871926)

[Implementation Details 6](#_Toc121871927)

[Data Lake 6](#_Toc121871928)

[Azure Data Factory 6](#_Toc121871929)

[Pipelines 6](#_Toc121871930)

[Data Flows 8](#_Toc121871931)

[Data Warehouse 8](#_Toc121871932)

[Other Notes 9](#_Toc121871933)

# Introduction

This document contains documentation of the entire infrastructure developed for Loka demo project.

# Technology Stack

Below is given list of most important tech stack proposed (not everything was used for this project but would be proposed for implementation on a real-life system)

Programming:

1. T-SQL
2. Bicep (IaC for deploying Azure resources) – not used

DevOps:

1. Azure DevOps pipelines (for automating CI/CD) – not used

Orchestration and ETL tools:

1. Azure Data Factory

Windows programs used:

1. Visual Studio 2022 Professional
2. Visual Studio Code
3. Microsoft SQL Server Management Studio
4. Microsoft Azure Storage Explorer

Azure:

1. Data Lake storage
2. SQL Server (Serverless)
3. Azure Data Factory
4. Key Vault (for storing secrets) – not used
5. Azure Data Explorer

# Architectural Overview

## Infrastructure

Cloud ETL solution was implemented on Azure. ADF was used as Orchestration and primary ETL tool. Final transformations were completed using stored procedures on the SQL DB. This can be seen on .

Diagram

Description automatically generated

Figure High level architectural overview

# Implementation Details

## Data Lake

Structure of the data lake can be seen on the Figure 2.

On the Data Lake there are two containers:

1. **src** – contains source data
2. **raw** – used for data processing and archive

Raw container contains two folders:

1. **landing** – folder where delta of the documents which were not previously loaded is moved over
2. **archive** – location where loaded files are moved over after processing but in the original form in which they were dropped off without any transformations

Graphical user interface, text, application, Word

Description automatically generated

Figure : Azure Data Lake

## Azure Data Factory

## Pipelines

There are three pipelines for this process:

1. Master pipeline – this is master pipeline which orchestrates the entire ELT process. It is scheduled to run Daily at 7:00 UTC.

Diagram

Description automatically generated

Figure : Master data pipeline

1. Src to Landing – This pipeline loads data from the src to the landing folder. The load process was not developed in an optimal way. Simple approach was used to loop through all files on the archive and determine the last modified stamp on the loaded files. This is inconvenient with huge data sets and unnecessarily wastes resources. Instead, the process should be modified to use some external storage to store the latest loaded timestamp after loading all the files to avoid looping through ever increasing number of files on the archive. I’ve added sample table to the etl schema on the DW for demo purpose, but I did not implement actual flow using it. Also, in the implemented approach I am not relying on the date and time from the file name but instead on the file’s timestamp. However, in real scenario I would assess if this would be actually feasible, or it would be needed to parse date and time from the file itself for whatever reason. For example, I do not have any guarantees from the source system that daily data will be dropped off in order or that original file creation time would be preserved if they were moved from another location. My aim was to build a working PoC that can potentially open more fruitful discussion. This implementation was rather fast and avoided wasting resources for PoC.

Graphical user interface, application, Teams

Description automatically generated

Figure : Src to Landing pipeline

1. Load to DW – This pipeline loads data from the landing folder to the DW and archives loaded data to the archive space. The process gets list of all files from the landing folder and if any file exists it truncates import tables on the DW, performs load of the data to the import tables and then invokes stored procedures which apply final touches before moving over data to the dbo schema.

Graphical user interface, application

Description automatically generated

Figure : Load to DW pipeline

## Data Flows

There is only one data flow which loads data into the DW and archives it after the load is completed. Its implementation is relatively simple. It first expands all json nested parameters, then it separates VehicleEvent and OperatingPeriod data into two separate flows. Finally, only required attributes are selected and stored onto the corresponding tables on the import schema.

Graphical user interface, application, Word

Description automatically generated

## Data Warehouse

Storage on the Data Warehouse was also kept quite simple. I’ve separated Vehicle Event and Operating Period data. For each of them there is one table on the import schema and one on the dbo schema. Dbo schema is data’s final destination in this PoC.

Transformations for Vehicle Event on the SQL side were done using stored procedure. On the import schema this table has very limited restrictions. All fields allow nulls and only Lat and Long are set as decimal format. Transformations made while moving to the dbo schema include renaming Id, changing data type and creating new attribute CreatedDate.

Table

Description automatically generated

Figure : Vehicle Event Data

Transformations for Operating Period were done in a similar way. While transitioning from import to the dbo I created new date column for each of the DateTime attributes, applied proper data types and restricted all columns to have non null values.

Graphical user interface, application

Description automatically generated

Figure : Operating Period Data

Note, I did not implement checks on the SP to make sure no null values were stored. I was merely trying to demonstrate that not all transformations have to be done in one place and using only one tool.

# Other Notes

The below are my concerns, assumptions and potentially discussion points which I did not list anywhere else but wanted to point out:

1. The sample data set is per second and hence there will be 86 400 files per day (if system is up and running 24/7). It is unclear how big these files can be, but I assume they will be much bigger than the sample data given here. Depending on how long the files are stored in the source location this can become quite an issue if it is not purged frequently enough. Hypothetically speaking, the source system could already be considered as landing area, and it might be beneficial to immediately remove loaded files and not rely on another process to clean the storage at unknown intervals. For this exercise, I assumed that we are not allowed to touch the source system for whatever reason (e.g., lack of permissions, reliance of other parts of the system on the data being there…).
2. The requirement “fetching process should only get data from a certain day on each run and should run every day” is something I would find ambiguous. Is it only 1 day before the run regardless of if more days were not loaded, any number of days before the run date that were not loaded previously, only 1 day but with arbitrary delay (e.g., 2 days ago) … Despite the huge number of files, I opted out for the approach where I would use more resources but have a smaller chance to miss any data.
3. I understand that the initial load of my process is not safeguarded against an unlimited number of files. In a case like this I would normally try to understand how the source system works and for how long it preserves the data. Only then I would implement safeguard measures to avoid potentially processing millions of files in one run (e.g., if the source folder contained data for 1 year).
4. I would normally ask lots of questions on what the system does to understand how the data might behave. Here I lack that knowledge and I had to make some assumptions to make fast progress. For example, It is not clear how vehicles relate to the operating periods. This would be apparent from some system demo for the product that generates this data.
5. I did not build any mechanism to handle dirty data as it was not part of the requirements. However, I never worked with a perfect system which did not generate any type of data inconsistencies.
6. For the lack of understanding how this data would further be consumed other than being quarriable I did not optimize DW for loading large amounts of data. For example, partitioning on the tables could be done or some sort of aggregation.