2021

University of North Carolina Wilmington

Master of Science in

Computer Science and Information Systems

Proceedings

https://csbapp.uncw.edu/mscsis

A DATA PIPELINE FOR AMAZON REVIEW COLLECTION AND PREPARATION

Ryan Woodall

A Capstone Project Submitted to the
University of North Carolina Wilmington in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Department of Computer Science
Department of Information Systems and Operations Management

University of North Carolina Wilmington

2021

Advisory Committee

Dr. Ron Vetter, Committee Member 1 Dr. Minoo Modaresnezhad, Committee Member 2

Dr. Douglas Kline, Chair

Table of Contents

1 INTRODUCTION

Modern companies struggle with big data collection and processing, and it has become best practice to accomplish this with data pipelines in the cloud. Knowledge of the tools for building a data pipeline that is maintainable, adaptable, repeatable, and scalable, would be an incredible asset to any aspiring data engineer. By using Azure tools, I can accomplish this task, while utilizing data from Amazon Reviews as an example. Azure Data Factory provides an intuitive environment to integrate data visually, constructing ETL and ELT processes. I wish to learn and explore the use of these technologies, and this is the motivation for this project. By using the problem of amazon reviews, I can demonstrate the design and development of a data pipeline while supplying a data science effort using large amounts of data. The advantages of this approach are that it is perfectly repeatable, easily modifiable, and scalable, which is not the case in typical manual processes. Additionally, I feel that it will create datasets that many people will want to use in future research endeavors.

2 Problem Statement

The primary objective of this project is to design and implement a big data solution that will provide text analytics researchers with datasets that are ready for analysis. More precisely, a maintainable, adaptable, repeatable, and scalable data engineering system that creates as many datasets as desired, using Amazon Reviews as an example. The data pipeline would take a list of URLs (15 amazon products), scrape, normalize, and output data in a consistent, repeatable manner. Output data would be minimally affected and ready for research, where tags and brackets around data, as well as other artifacts produced by the webscraper may be removed, although stop-words would remain. Three primary forms of a single dataset would be produced. To illustrate, if viewing the data pipeline as a big black box, it would grab data and output the raw datasets, a database with no loss of data, and a JSON file of the normalized data. This is an example of a complex data pipeline for producing outputs that would be of great interest to researchers and businesses. By designing the pipeline within Azure's Data Factory version 2 IDE, data engineers can utilize tools and templates for quick development, effectively reduced the time spent creating hard coded solutions. This along with Azure Data Lake, the system would scale without the need for manually deploying more resources. Azure's pay-as-you-go serverless consumption plan eliminates the unnecessary overhead associated

with the technology and infrastructure needed to handle big data solutions. This coupled with the reduction in man-hours is a huge savings for businesses and researchers, making this a worthwhile project.

3 BACKGROUND

This section covers the preliminary research of the following topics: Azure, Visual Studio Code, Python, Amazon reviews, webscraper.io, and Kaggle.

3.1 Azure

Azure is a cloud platform that offers services for building modern cloud-based solutions to an increasingly data-driven world.

3.1.1 Azure Resource Manager (ARM)

ARM is the brain for an Azure account, controlling deployment and management of all resources. Once a resource has been deployed, access control, locks, and tags can all be managed oversee security and organization. The resource manager lies between clients and resources. For example, if an HTTP request comes in from a website or access is requested from the Azure Portal, ARM handles authentication and then sends the request to the Azure service. There are four levels of scope in Azure, Management Groups, Subscriptions, Resource groups, and Resources. If a management policy is applied at the subscription level, all lower levels inherit that policy.

All manageable items in Azure are considered resources and these resources can be managed into groups. Grouping resources can be more than just an organizational strategy. For instance, resources that all share the same life cycle can be contained in a group so that when the cycle ends, all can be deleted by simply deleting the resource group. Resources are managed using a declarative syntax in the form of a JSON file which is known as the Resource Manager template. A particular advantage to using the Resource Manager is that dependencies between resources can be defined. Often, when deploying resources, order matters, as one resource may depend on other resources. Using the RM ensures that resources are deployed in the correct order. Another advantage to RM is ability to apply tags. Tags can provide organization and aid in

locating resources that have some commonality. One such tag may be an Environment tag, where some resources may only be needed for Development. Once development is complete, the resources can be easily located for deactivation (FitzMacken, Rabeler, Harvey, & Lyon, 2020).

3.1.2 Resources

In this section, common Azure resources that are of interest to this project are described. The resources used in this project were all contained in a single resource group named "AmazonReviewsRG."

Data Lake Gen2 storage is a convergence of Azure Blob Storage and Azure Data Lake Storage Gen1. The primary modification to Blob storage is the capability of hierarchical namespace. Where the Gen1 storage was a stand-alone resource, Azure Data Lake Storage Gen2 exists in a standard Azure Storage Account. With the data lake, data can be managed within a single data store that offers the ability to capture data of any size, speed, or type. When integrated with Azure Data Factory, pipelines can be built that can be easily maintained and are scalable (Data Lake Storage for Big Data Analytics: Microsoft Azure 2020). For this project, I used a single data lake for storing all files including those utilized for controlling the operations within the data factory. The data lake container existed in a storage account named "amazonreviewssa."

Azure offers services that allows developers to work locally or in the cloud. **Azure DevOps** Server is used for working locally while Azure DevOps Services is used for development in the cloud. Git repositories are available through Azure Repos and deployment pipelines can be created using Azure Pipelines. Like GitHub, files can be checked in and out, organized into folders, and branches can be created. I initially planned on using GitHub through Azure DevOps for version control, but when working out the project timeline and predicting manhours, I had to trim some features, and this did not make it to the proposed plan.

"Cloud-based ETL and data integration service that allows you to create data-driven workflows for orchestrating data movement and transforming data at scale." (Microsoft Azure) A **Data Factory** is where the main processing of the data in this project will occur. It allows activities to be orchestrated for an automated process that is within a single Pipeline through data-driven workflows. The pipeline would consist of many Copy Activities. Each copy activity consists of two parts. One part would contain a Dataset, Linked Service, and a Source. The other would contain a Dataset, Linked Service, and a Target. Basically, one part is for extracting and the other for loading. It is important to note that although there are two Linked Services for each

Copy Activity, these services can be shared across many Activities and can sometimes mean you only need two. In fact, only three are necessary in the implementation of my system. The greatest advantage of linked services is that it eliminates the need for writing custom data movement components (Azure Data Factory Documentation - Azure Data Factory 2020).

Data Pipelines are a way to manage a group of activities that need to work in some logical order. This simplifies the process by needing only to trigger the pipeline, rather than trigger each activity individually. There can be many pipelines within a Data Factory and therefore, I will only need one Data Factory to accomplish most, if not all, tasks that this project encompasses. As mentioned before, a pipeline is a group of logical activities. There are three main types of activities: data movement, data transformation, and control.

Control activities can do things such as looping, branching, executing other pipelines, HTTP or Web activities which will be useful in replacing the need for individual functions that would need to be managed. These are just a few of the activities, but these are the ones I expect to utilize the most for this project. I controlled all systems activities within a data pipeline with exception to one python function. This includes controls within the database, as all stored procedures were invoked from within a pipeline.

A data movement activity also called a **Copy Activity**. Data is copied from a source data store to a sink data store and Data Factory supports a wide variety of data stores. A data transformation activity is where the data will be cleaned by using data flows. One interesting aspect of Data Factory is the ability to create data flows through visual representations of the data transformation logic. This allows you to create logical transformation activities without having to write code. One such code free application is wrangling data flows. This works by integrating Power Query Online making Power Query M functions available to Data Factory.

Wrangling data flows are the heart of this project, so I need to take a moment to talk about how it works. Power Query is basically an engine that can be used for data transformation/preparation. Data flows can be designed within the Power Query Mashup Editor. This is an easy to use graphical user interface. Behind the scenes, Power Query generates M code and Wrangling Data Flow translates the M code into spark code for cloud scale execution. Wrangling Data Flows supports the following sources: Azure Blob Storage, Azure Data Lake Storage Gen1, Azure Data Lake Storage Gen2, Azure SQL Database, and Azure Synapse Analytics. This means that for the purposes of this project, it will be sufficient since we

will generally be sourcing Azure Data Lake Gen2 and Azure SQL Database. There is a folder named ADFResource and this is where the dataset queries are kept. The mashup editor does have some limitations as some Power Query M functions are not yet supported (Perlovsky, Campise, Coulter, & Ghanayem, 2019).

The primary reason that I researched the wrangling and mapping data flows was to use them to perform all transformations of the data, but later I speak on why this method ultimately had to be aborted.

Mapping data flows can be used as an alternative to wrangling data flows. Transforming data can be done without the need for writing code by dragging and dropping features in a user interface. There is a Data Flow Debug option that, when enabled, a preview of the is visible as it moves through the stages of transformation.

In comparison to wrangling data flows, mapping data flows supports the same data sources and many of the same transformations can be done. There are a few advantages to using mapping data flows when dealing with databases. Unlike wrangling data flows, MDF can handle inserts, updates, deletes, and upserts. Wilhelsmen notes that wrangling data flows works well for data preparation and mapping data flows works well for data transformation (Wilhelmsen, 2020). This is because MDF are best for known data and schemas, whereas WDF works best with known or unknown datasets. Wilhelsman states that the choice depends on the task, desired destination, and familiarity. If data is going to an Azure SQL data warehouse and there is a familiarity to SQL Server Integration Services, then MDF is the way to go. While, if just dealing with a Data Lake storage, both types of Data Flows will work. If there is a familiarity to Power Query Mashup Editor, then WDF is most likely the best option. This project may require the use of both, WDF for data preparation and SQL Database operations, and MDF for data preparation.

Datasets basically refer to data within different data stores. For this project I implement quite a variety of dataset types including JSON, CSV, REST, and SQL. An important note is whenever a copy activity occurs within a Data Pipeline, there are two datasets involved. One is the source dataset and the other is the target dataset. These two datasets are utilized by the copy activity by way of linked services. To clarify, the copy activity is not the only activity that relies on datasets - they are necessary for any type of activity in my data factory. Each activity needs two linked services, although they can share the same linked service.

A **linked service** must be created before creating a dataset, since it must have a connection to an actual data store outside of the data factory. The linked service connects a storage account or other resources utilized in the data factory. This project only needs three of these, since I only need to connect to the data lake, database, and the external web scraper.

There are a few options for ensuring security. An access policy for a Managed Identity in Azure Active Directory can be used to allow Data Factory access to a complete group of resources. However, full access to the Azure account is necessary. The service principal is one of many service principals under the UNCW tenant. This affects the permissions to configure Azure Active Directory, which is required, if using a managed identity in an access policy. Another option is to use a key vault to store keys and secrets where secrets can be the connection strings to the resources within the resource group. The key vault can also contain API tokens for external web services, such as cloud.webscraper.io. However, to use the key vault with data factory, a managed application identity is required, and this must be configured in Active Directory.

The **control flow** utilizes several types of Activities. One type, chaining activities, allows many activities to be chained together within a pipeline with the output of one activity becoming the input of the next activity in the chain. Another type is branching activities allowing multiple activities to be run in parallel. Parameters can be defined and set for the duration of a pipeline runtime, allowing the passing of arguments when activities are triggered. For-each type iterators can be created with custom-state passing and looping containers.

Functions provide the power of serverless computing. Small pieces of code can be written that will run whenever the function is triggered. There is a cost advantage to using functions because it costs only when the function is triggered. There are a wide variety of triggers that can be used, such as, http trigger, blob trigger, queue storage trigger to name a few. Having a choice of languages is another great aspect of Azure Functions. Code can be written in C#, Java, JavaScript, Python, or PowerShell. C# and PowerShell code can be edited right in the Azure Portal which can be helpful, but using Python is not difficult. To write functions using python, Visual Studio Code is necessary. There are several tools/extensions that when installed, make it easy to write functions with most of the boiler plate starter code being included. Functions can be tested locally for then deployed to Azure. All functions are controlled by a single function app which contains all the necessities for connections and security. Bindings are created in the function app, and given a name, direction (in, out, both), and a type. These bindings can be

used by multiple functions or just one. Since Azure requires the use of connection strings or keys and secrets to access different resources, a key vault can be utilized to hold keys and secrets for secure information like connection strings, API tokens, and passwords (Gailey et al., 2020).

Functions can be grouped together to orchestrate serverless applications with Durable Functions extension. The extension is built on top of the Durable Task Framework. **Logic apps** can be created to execute various activities that are triggered by a single initial event. Logic functions can be incorporated into Data pipelines and the azure functions model can be used to write orchestration functions to define stateful workflows. The benefit of using durable functions is that state, checkpoints, and restarts are all managed behind the scenes by the Durable Task Framework. Durable functions can be written in Python using Visual Studio Code, with Azure extensions installed. Code can be tested locally before deployment.

The main reason for using durable functions is simplification. With serverless applications, requirements involved in coordinating stateful functions are automatically handled. One serverless application pattern that is of interest to this project is function chaining. Boilerplate code for function chaining can be found in **Appendix D, section XIV**. Logic apps work in a manner that the output of one function becomes the input of another function. Typical programming constructs, such as conditionals and loops can be implemented where code is read top-down, as most programs function. In function chaining, to invoke other functions by name, the context parameter can be used to pass in parameters. Preface this with the yield command and durable functions framework checkpoints the progress, where if there is an issue, the function instance resumes from the previous yield.

3.1.3 Power Query

Power Query M formula language M formula language (M) is used in the Power Query Mashup Editor which is used to create Wrangling Data Flows in Data Factory (Power Query M formula language reference - PowerQuery M 2020). The mashup editor allows filtering and combining of data from a variety of data sources. Expressions can be written to manipulate or transform data by using functions. M contains a set of operators defining the kinds of expressions that can be used when creating a function. Complex expressions can be built by combining smaller steps, called **let** expressions (See **Appendix D, section XV** for the basic structure of a let

expression) and **if** expressions (Chu et al., 2020). If expressions allow for conditional evaluation. When evaluating expressions, M follows the same model that is used for spreadsheets.

M includes a library of definitions that can be used within expressions. When using definitions, primitive type precedes the name of definition, such as Text.PositionOf("Hello", "ll"). The let expression allows for the creation of complex expressions. Multiple expressions can be assigned to variables. The values can then be combined in a final expression.

The if expression can be used to select between two expressions based on a logical condition (see **Appendix D, section XV** for basic structure of an if expression). If expressions can be used inside of let expression. Recursion can be accomplished within an expression by using the @ symbol when calling recursively. For example, if the expression name is multiplyThis. To call multiplyThis within the expression, use @mulitipyThis.

A function represents a single value that is mapped from a set of argument values. Functions can be written using the form FunctionName = (parameter1, parameter2) => expression. They can then be called with FunctionName(parameter1, parameter2). Optional parameters can be defined in a function using the keyword optional before the parameter.

Expressions that are related can be grouped into sections that are contained within a section document, where section members can be declared. Section member expressions can then be referenced within other sections using a section access expression. Section names must be unique in the global environment and within each section, member names must be unique.

3.2 Visual Studio Code

Azure extensions for Visual Studio Code allow developers to connect directly to azure. Code can be created and tested locally in VSC before deploying to Azure. Inside VSC, searching for extensions is easy with the VS Code Extensions view. Typing Azure in the search box produces a list of Azure extensions that, once installed, offer a variety of shortcuts and autocomplete options making it easy to load boiler plate code to get projects started. Most Azure Services can be accessed with extensions that can be found in the Command Palette. To view the Command Palette, simply press F1.

The Azure Tools extension offers many Azure Services that are commonly used, such as App service, serverless functions, Azure storage, VM management, Azure Resource Groups, Pipelines, Azure CLI and PowerShell. Once signed into Azure from VSC, all current Azure Resources become visible through the Azure Explorer (Azure Tools - Visual Studio Marketplace 2019). Many Azure resources do not allow python code to be edited inside the Azure Portal. For applications using python, Visual Studio Code is a necessary tool.

3.3 Python

To simplify development, python can be written locally using Visual Studio Code. The Python extension as well as the Azure extension for VS code must be installed to begin setting up the local development environment. To configure authentication, a service principle with environment variables must be created. Configuration is stored in local-sp.json.

The Azure SDK for Python contains an extensive collection of libraries for writing python code in serverless functions. The Microsoft Azure Data Factory Management Client Library contains the modules that will be of most importance for this project. (Azure SDK for Python: Azure SDK for Python) The Microsoft Azure SQL Management Client Library is used managing SQL databases. There are five libraries for configuring and interacting with Azure Key Vault. Python can be used in functions to query an Azure SQL Database by importing the pyodbc python library (Brockschmidt et al., 2020).

3.4 Amazon Reviews

The **reviewer_location** field contains the city, state, and country listed on the reviewer's profile. The data is not uniform - entries can vary in format. One entry may be "CA, USA", another may be "Jerusalem, Israel", and yet another may be "Walgreens Function Lead | Arizona". The **reviewer_idea_list_count** is the number of ideas a reviewer has on their Idea List. Whenever a reviewer has created an Idea List, other amazon users can like the list by clicking on the heart and this information is contained in the **reviewer_heart_count.** The number of stars given to product by a reviewer is saved in the **review_stars** field. The entries are uniform, following a "2.0 out of 5 stars" format. Another example of a field that is currently not uniform is the **reviewer_top_1000** field. Reviewers ranking in the within the top 1000 are reported in this field, however VINE VOICE sometimes is reported in this field. There are various fields that are associated with pagination and may contain "next page->" or a html href.

3.5 Other Amazon Reviews Datasets

3.5.1 Julian McAuley Dataset

In 2014, Julian McAuley released a dataset containing product reviews and metadata from Amazon. The dataset included 142.8 million reviews from May 1996 through July 2014 and is available in several formats. The raw review data is 20gb and contains all 142.8 million reviews, although the data can be downloaded in smaller samples. The data fields are quite comprehensive containing the review data, metadata, and image features.

Review data is comprised of the reviewer ID, product asin, reviewer name, helpfulness rating, review text, overall rating of product, summary of the review, time of the review as unix time, and the raw time of the review. The metadata includes product asin, name of product, price of product, url of product image, related products (including also bought, also viewed, bought together, buy after viewing), sales rank information, brand name, and list of categories the product belongs to. In addition to review data and metadata, a dataset for image features is also included in binary format.

Visual features of each product image were extracted using a deep convolutional neural network generating relationships from product recommendations based on two notions of compatibility, substitute and complement products. The patterns in recommendations followed four main constructs: users that viewed this product, also viewed this other product; users that viewed this product, eventually bought that product; users that bought this product, also bought that product; and users bought this product and that product simultaneously. The first two indicate substitutability, while the latter indicate complementary. "Image-based recommendations on Styles and substitutes" (McAuley, Targett, Shi, Hengel, 2015)

3.5.2 Julian McAuley Updated Dataset - Jianmo Ni

The Julian McAuley dataset was updated by Jianmo Ni in 2018 which increased the total number of reviews to 233.1 million. In addition to containing more current data, transaction metadata, more detailed metadata for the product landing page, and five new product categories are included.

The data regarding visual features is where the McAuley dataset and the dataset produced in this project differ in that the primary goal of this project was to produce an automated means

of collecting data into a useable dataset of which contained minimal changes to data and zero analysis. The advantages are a customizable and relatively current dataset that can be tailored to include specific products of current interest.

3.5.3 Multilingual Amazon Reviews Corpus

One other collection of amazon reviews is the Multilingual Amazon Reviews Corpus which was designed to aid in multilingual text classification. Data contained in the dataset were collected from November 1, 2015 and November 1, 2019 and the dataset is available at https://registry.opendata.aws/amazon-reviews. Data includes the review text, review title, star rating, reviewer ID, product ID, and product category. While significantly smaller than that of Ni et al, it contains reviews in English, Japanese, German, French, Chinese, and Spanish. Unlike this project and Ni's dataset, reviews are truncated to 2000 characters and limited to a sample of 20 reviews per reviewer and product. Additionally, reviews were selected based on star ratings so that each rating consisted of 20% of the corpus.

3.6 webscraper.io

Web Scraper offers an intuitive user-friendly interface, in the form of a browser extension, for scraping and downloading data from web pages. This section covers the two platforms, web browser and cloud API, as well as the details involved in building scrape job schema.

3.6.1 Building a schema (web browser)

The web browser version at https://www.webscraper.io is free and contains a user-friendly point and click GUI for creating a complete schema for scrape job. The webscraper is an extension that must be installed on the browser. Once installed, it can be accessed through the browser's developer tools.

3.6.2 Selectors

Webscraper.io offers a variety of selectors for different types of data extraction as well as website interactions. Selectors fall within three categories, data extraction, link selectors, element selectors (Documentation 2020).

Data selectors extract and return data within selected elements. There are eight types of data selectors: Text, Link, Link popup, Image, Table, Element attribute, HTML, and Grouped. Each

type of selector comes with a variety of configuration options. Text selectors extract text data with HTML tags stripped. Configuration options for the text selector include selector, multiple, and regex. Selector is the CSS selector from which the data is being extracted. Multiple option can be set for extracting multiple records within one page and regex allows regulars expressions to be set if substrings are to be extracted. When data is to be extracted from a popup window within the same URL, a link popup selector can be used. The table selector has a couple of unique configuration options to allow header rows and data rows to be selected. Element attribute selectors can be used to extract attribute values from within an HTML element. This is done by setting providing the attribute to be selected in the attribute name configuration option. Grouped selector allows multiple elements to be combined into one record.

Link selectors are used to navigate websites. These selectors can also be used for data extraction in no child is assigned to selector. This selector is used for <a> tags containing the href attribute. Configuration options for link selectors include selector and multiple.

When multiple data elements exist within a selected element, **element selectors** can be used. Selected elements within the parent will be returned to child elements and only data that has been selected within the element selector will be extracted. Configuration options for element selectors include selector, multiple, delay, and parent selectors. The delay option allows a delay time to be set before the selector is used. Selector tree can be defined using the parent selector option.

Once a sitemap has been created on the webbrowser, it can be exported as a JSON file which can be imported when needed.

3.6.3 Downside of Web Version

A drawback of using the basic webscraper is the inability to automate the process. One option could be to set up a virtual machine in azure and write scripts to automate processes that would normally be carried out manually. However, virtual machines are an expensive cloud resource. Another option would be to use the cloud version of the webscraper, which would enable access to the scraper via an API.

3.6.4 Cloud webscraper

The cloud.webscraper.io requires a paid subscription. There are four paid plans available, ranging from $50 a month up to $300 a month. The $50 Project Subscription only offers 5,000 page credits per month and denies API access. For API access, the $100 Professional Subscription is required which comes with 20,000 credits. While this is a good amount for development testing, the final runs in production would require the $300 Scale Subscription, where unlimited page credits are allowed. There is a $200 Business Subscription which has a limit of 50,000 credits and a Scale Subscription for $300 that offers unlimited page credits. The next few sections provide an overview of the features the cloud version of the web scraper offers that are relevant to this project.

Creating Jobs

The API allows HTTP requests to be sent or received by automated pipelines. HTTP POST requests are sent to create a sitemap and a scraping job, and this will automatically begin the scrape job (Template shown in **Appendix D, Section II and III**). When a scrape job has finished, an HTTP GET request will download the scraped data, and an HTTP DELETE will delete the scraping job. There are two ways to determine when a scrape job has completed. The first would be to poll the API for the status of the job, however it is requested in the documentation that this method be avoided. Instead, a URL endpoint can be set up to receive a POST form when a scraping job has completed. An API token is used to access the API. The next section describes this method in more detail.

Job Complete Notification

Rather than pinging the API, configuration can be set to receive a push notification whenever a scraping job has completed. The JSON body of this notification will contain, among other things, the scrapingjob_id and custom_id that can then be used in an HTTP GET request for downloading the scraped data. An example for the body of a job completed notification can be viewed in **Appendix D, Section IV**. Next, the Retrieving Data section provides more detail on downloading data.

Retrieving Data

Data can be downloaded in JSON, CSV, MS Excel, or PHP format. Cloud.webscraper.io recommends that data be downloaded in JSON form. API documentation suggests that JSON formatting is consistent, whereas others may not be consistent when using across different platforms. For example, MS Excel does not always handle certain escape sequences in CSV files, and some have reported in the web scraper forum that the PHP export has an incorrect default implementation. Preliminary tests using JSON exports resulted in an error (unexpected characters at the end of records) during import to Azure. CSV imports produce no errors while importing using a logic app. Later in the Discussion section, I explain how this was not the case when importing to Data factory. This is also noted in my journal entry in **Appendix B, section I**. Scraped data is downloaded by way of an HTTP GET containing the scrapjob_id. A template for this GET request is in **Appendix D, section V.**

Deleting Jobs

After data has been imported from a scrape job, an HTTP DELETE can be sent to cloud.webscraper.io API with the scrapingjob_id contained in the URL along with API token. Upon successful deletion, a response will be sent with "success" value set to "true." When implementing the system, it was discovered that deleting the sitemap would delete the job as well, consequentially only one DELETE request would need to be send. A template for DELETE request is in **Appendix D, section VI**.

Scheduler

The **scheduler** allows scraping jobs to be scheduled to run at a set date and time. Configuration options for the scheduler include Scheduler Time Zone, Scheduler Type, Driver, Request interval, Page load delay, Proxy. Types can be set to daily, interval, or custom. Custom is set using a custom Cron expression. The Driver option affects the speed by enabling or disabling the Javascript driver. Full enables Javascript driver. The page request interval allows a certain amount of delay (in milliseconds) between page requests. Similarly, Page load delay allows a delay time to be set to load page before data extraction begins. Finally, the Proxy option is set to enable the use of a proxy while scraping a website. Enabling proxy helps disguise the webscraper from bot detectors.

Parser

The **parser** feature allows elements of the data to be post processed. The parser runs when data download is requested. There are eight parser types offered in the cloud webscraper. Append and prepend text, convert UNIX timestamp, regex match, replace text, remove whitespaces, strip HTML, remove column, and virtual column. The append or prepend text parser includes three configuration options. Append text and Prepend text is for the text to be added to the beginning or end of the text and the Text to place option is for the actual text that is being added. The Convert UNIX timestamp parser includes Time zone and Format configuration options. The Replace text parser's configuration options include Text to replace, Text to place, and Use Regex. Sometimes unwanted columns are created by the scraper. For example, a Link selector generates a column that is not useful in analysis. These columns can be removed using the Remove column parser. Lastly, the virtual column parser can be used to create a column with data from a scraped column. Configuration options include Name, Source columns, and a Separator if multiple source columns are being used. Virtual columns are quite useful when combined with Regex match such as when data within a scraped column needs to be separated into individual columns. One drawback of using the parser for preliminary cleaning is that currently, configuration exists solely on cloud webscraper's UI. This means that there are no visible changes to the sitemap's JSON file describing what is happening to the data. When transparency is needed, this is not ideal.

4 SYSTEMS ANALYSIS

The systems analysis section will begin with the problem statement and project scope. Next, a brief discussion of the organization of system, as well as its inputs, and outputs is provided before going into a detailed outline of the pipeline design. Sections 3.2 and 3.3 each contain organization of a file structure, naming conventions, system inputs, system outputs, and design. Section 3.2 is the originally planned to design. However, during implementation, several design features were changed. The implemented design is described in section 3.3. Finally, comments on the relational model, sitemap, and maintenance are provided.

4.1 SCOPE

Regarding the data, no changes will be made to the text - stop-words and punctuation will not be removed. Although, any artifact created by the webscraper will be removed. Scraping and

transforming resulting data into clean usable form that is ready for scientific analysis is the extent of this project. No scientific analysis is to be performed on the datasets . Transformations will not go beyond the creation of a relational representation. A final JSON file that encompasses the entire collection of datasets will be produced. The range of cloud services will not go beyond that of cloud.webscraper.io and Azure.

4.2 Organization of Planned System

This section outlines the systems file structure and naming conventions. This is the original design, however when building the pipeline, issues arose which required some expansions to the plan. First, I will go over the original plan and then I will present the implemented system. Later in the discussions section, I talk about some of the differences and the factors that lead to these changes in the plan.

4.2.1 File Structure

A Data Lake container named AmazonReviews will store all scraped data, product URLs, and final datasets. Inside the AmazonReveiws folder, the Products folder will contain the list of product URLs, the Reviews folder will contain the raw scraped data retrieved from the web scraper. Reviews will break down further to Rawdata and Normalized folders. These will hold the original and the processed scraped data. Finally, there will be a folder named CSV to hold the final prepared CSV dataset. Outside of the data lake, a database will contain the prepared scraped data.

4.2.2 Naming conventions

File names will follow the above path with filename being a concatenation of (<product name>, "- ", <scrape date>).

4.2.3 System inputs

System will take 15 URLs of amazon products that will be manually placed in a container in an Azure storage account.

4.2.4 System outputs

The final system will output one dataset with multiple forms of that data set.

One form would be a CSV file that will be a non-relational and highly repetitive dataset comprising prepared data from all scraped products. Another form will be a relational view of the data, and the original raw datasets will be preserved as the final form of output.

4.2.5 Design

In this section, I provide a general outline of the five main features in the pipeline. The five features are: Product Loader, Data Scrape, Data Retrieval, Normalization, and CSV.

Product Loader (triggered by scheduler)

The product loader is where the data factory pipeline will check to see if the product URLs list contains any URLs. If they exist, then it will grab some product information and update the database with the product. This sub-pipeline will require three connections. One will be to the Azure Key Vault, another to the product urls list in the Products folder, and the final connection will be to the Amazon Reviews Database.

A single logic app will be employed to control the pipeline and access the key vault and data stores. The app will iterate over the URL list, creating products in the database for each URL. The information, such as product name and ASIN will be parsed from the URL itself. Figure 1 below outlines the Product Loader's activities.

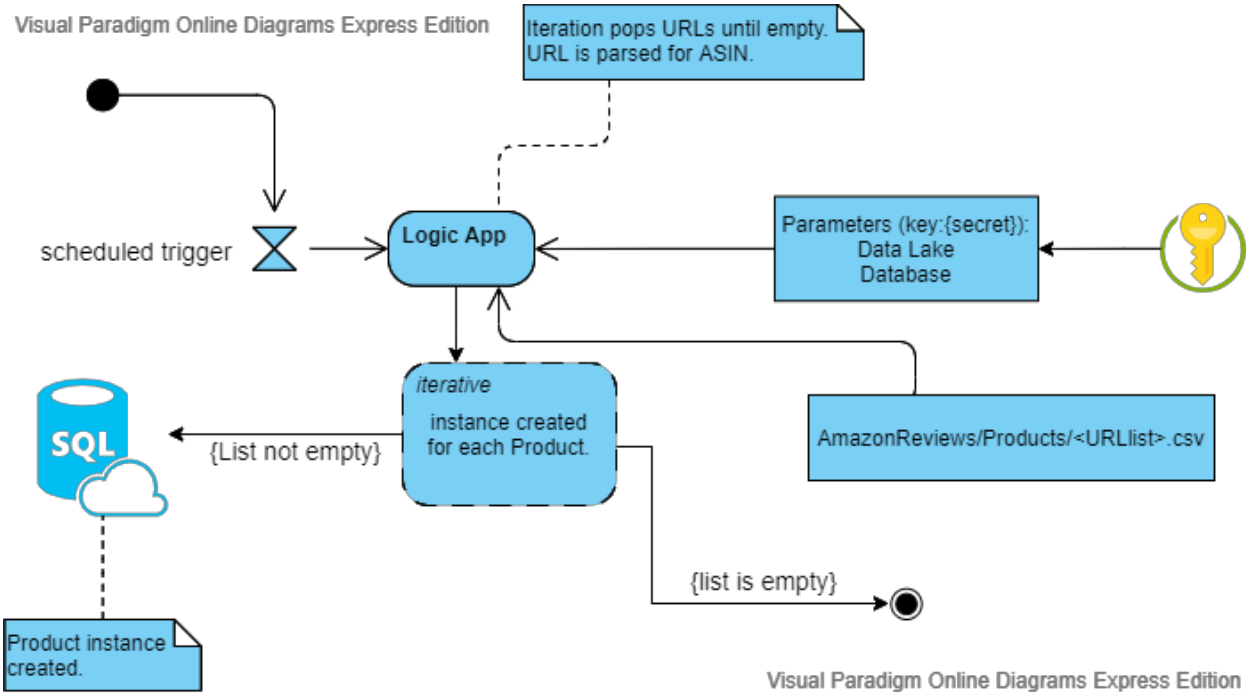


Figure 1. Product Loader Activity Diagram for Original Plan

Data Scrape (triggered by scheduler)

The data scrape sub-pipeline will be responsible for creating the scrape jobs in the cloud web scraper's API. The pipeline will require the same connections that the Product Loader pipeline utilized (Key Vault, Data Lake and the database).

A single logic app will be employed which will check the Product table for new products and then for each product, send an HTTP POST request to the web scraper's API to create a scrape job. This will involve adding the product name to the JSON formatted sitemap for all scraping jobs (To view a copy of the sitemap, see Appendix D.I.). The task will be achieved using a python function described below. A diagram of the Data Scrape pipeline is shown in Figure 2 after the function's description.

Python function

>Select productID, productName and productURL from Product where scrapStatus is NULL,
>If no records are reported, then do nothing.
>else, iterate through records calling the following function
>HTTP trigger function to create scrape job
>Three arguments (URL= <Product.productURL>, custom_id= {concatenation(<Product.productName>, "scrape-job")}, id= <Product.productID>)
>Send POST request to cloud.webscraper.io adding API token from Key Vault, URL, and custom_id
>Update Product.scrapStatus where productID = <productID> to 1
>Create a ScrapeJob instance with scrapeJobID=<custom_id>, web_scraper_start_url = Product.productURL>, and scrapeStartDateTime = <CURRENT_TIMESTAMP>

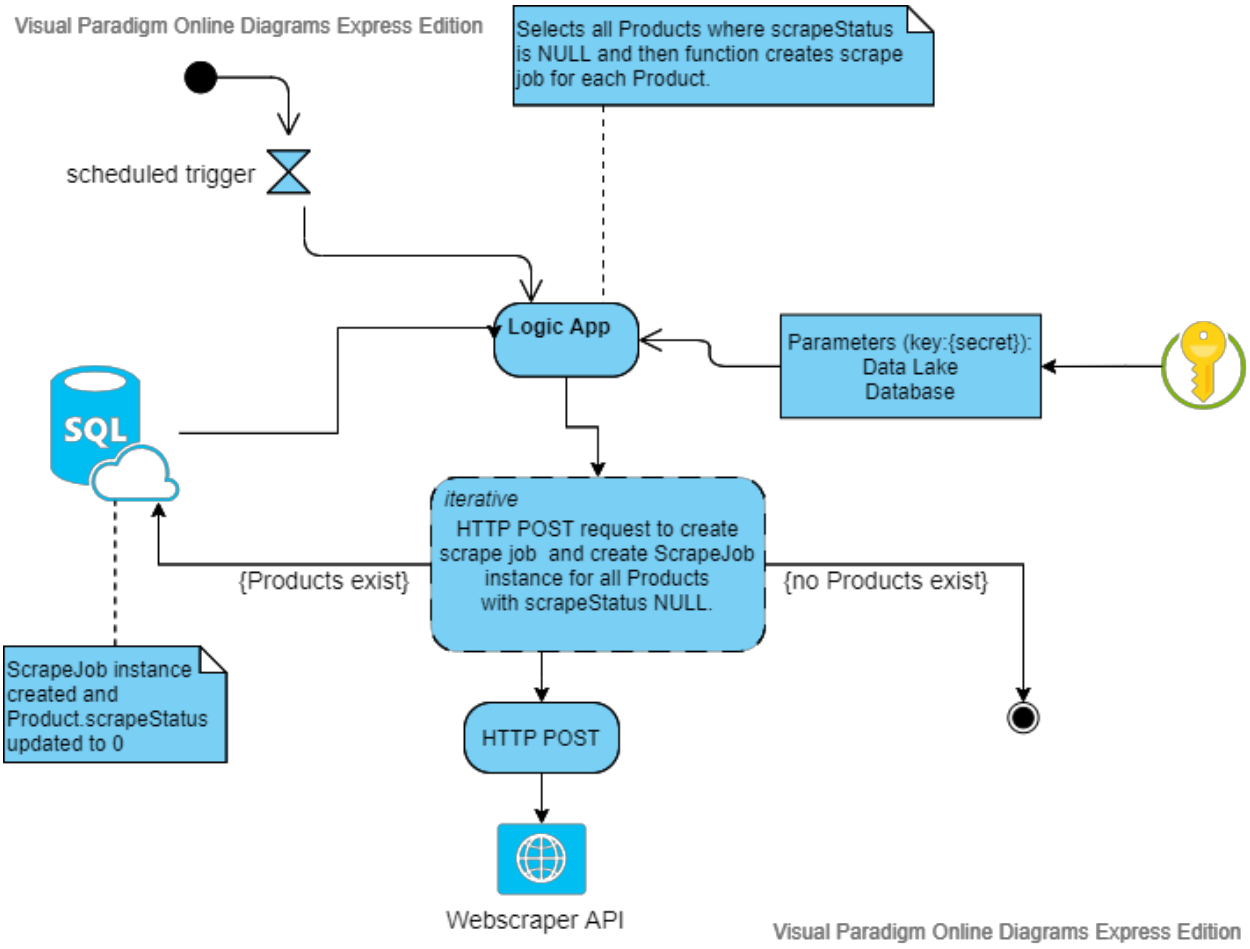


Figure 2. Data Scrape Activity Diagram for Original Plan

Data Retrieval

Figure 3 below shows the activities of the Data Retrieval component. Once a scape job has completed, a notification will be sent from the web scraper's API containing the scraping job id of the completed job. The API requires an endpoint URL to accomplish this task and for this purpose the URL of an HTTP trigger function is provided. The function will wait for a POST request containing the scraping job id and a custom id (which will be the product name) in the request body. A second HTTP function sends a GET request to retrieve the scraped data and load it in the Reviews/RawData folder using the product name as the filename. The ScrapeJob table is then updated in the database before a third HTTP function sends a POST request to the API to delete the scrape job. A description of the functions is provided below.

- An HTTP trigger function waits for job completed POST notification. Parses JSON body to get scrapingjob_id, and custom_id and sets these parameters.
- Next, an HTTP function sends a GET request to retrieve the scraped data. Three arguments are required, scrapingjob_id, productName=<custom_id>, and API token which is retrieved from Key Vault. The scraped data is downloaded to the Azure Data Lake/<productName> and the ScrapeJob table is updated where ScrapeJob.scrapeJobID=custom_id to 1.
- Finally, an HTTP function sends a POST request to delete scrape job. This requires two parameters, the scrapingjob_id, and the API token retrieved from Key Vault.

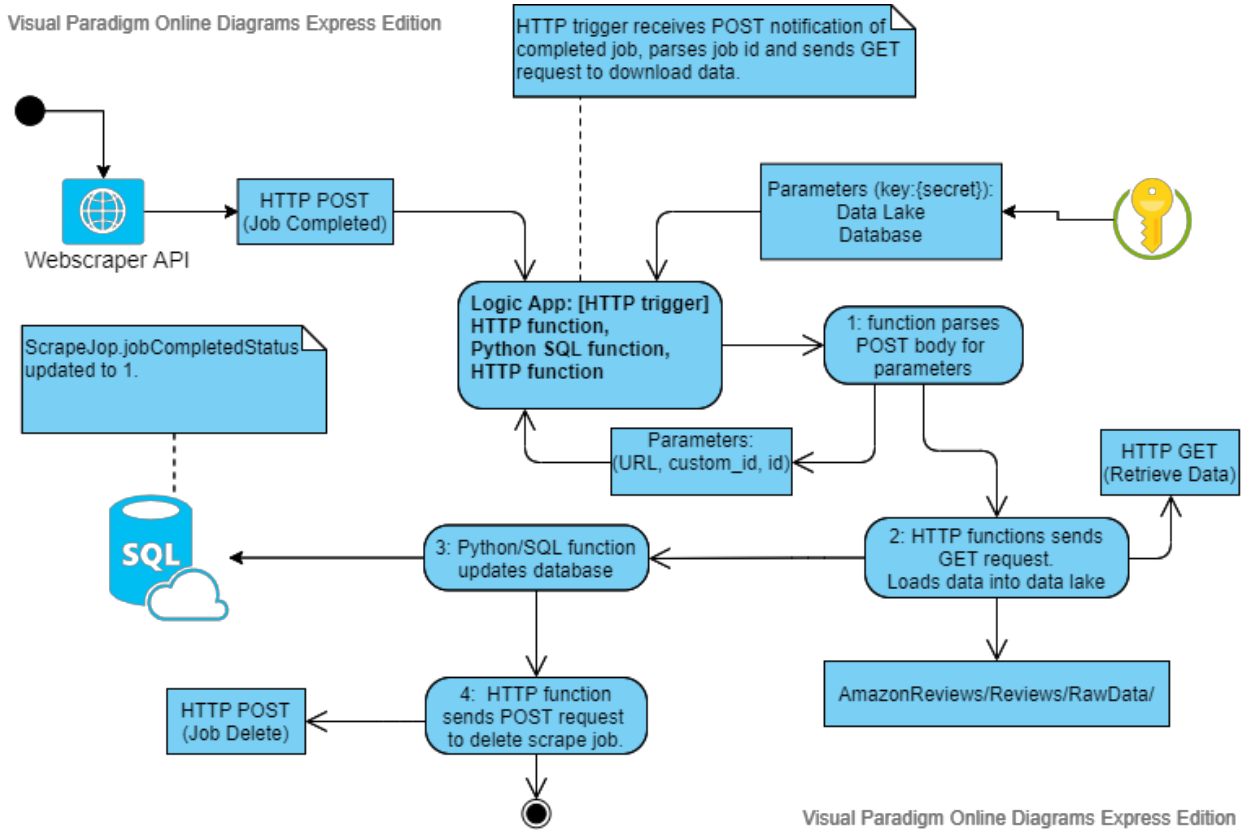


Figure 3. Data Retrieval Activity Diagram for Original Plan

Normalization

The Normalization sub-pipeline is where all preparations of the scraped data will be made and uploaded to the database.

To prepare the data, a mapping data flow will be employed within the Azure Data Factory. This process will convert data for product review stars, reviewer ranking, and review helpful string to integers. Next, new columns of type bool will be added, where True means the reviewer is in the top 1000 overall ranked reviewers, and has a Facebook, Twitter, Youtube, or Instagram account linked to their profile. Finally, columns to hold the product name an ASIN will be added. After the mapping data flow has completed, the data will be saved to the Reviews/Normalized folder. Finally, the data will be loaded to the database.

CSV

The CSV sub-pipeline will be manually triggered to create a final dataset comprised the normalized data from all scraped products. This will be accomplished by using a python function that pulls the datasets from the Normalized folder and appends them to a CSV file.

4.3 Organization of Implemented System

As previously mentioned, things did not always go as planned once I began to build the data pipeline. The next sections describe the finished system's implementation. Later, I will explain in detail the reasons for altering the original plan during implementation and lessons learned in the discussion section.

4.3.1 File Structure of Implemented System

A Data Lake container named amazonreviewsdl was created in a storage account named amazonreviewssa. The Products folder holds two csv files. The first file contains a list of product URLs to be scraped and the second file is the same file, but empty. The empty is used to replace the product URLs file after processing. A Reviews folder is used to temporarily hold the raw scraped datasets retrieved from the web scraper. A JSON folder is where the final combined JSON dataset including all prepared data for each product.

If the product review data failed to load into the database, the raw data file will be moved to the FailedReviews folder. Similarly, if the data retrieval from the web scraper fails, the job completed file will be moved to the FailedToRetrieveData folder and the FailedURLs folder will hold product URLs if the pipeline activity for creating a sitemap in the web scraper API fails. When a scraping job completed notification comes from the web scraper's API, the scraping job id and sitemap id are stored in a file in the ScrapingJobIDs folder.

The JobDeleteLogs folder will contain all logs generated when a file is deleted from the data lake.

4.3.2 Inputs and Outputs of Implemented System

The system will initially take 15 URLs of amazon products in the form of a CSV file named productURLslist in the Products folder. URLs can be pasted into the file on a new line. The file contains a single header on the first row named URL.

Once all pipeline processes have completed, the system will output one dataset with multiple form of that data set. One form will be a JSON file containing a non-relational and highly repetitive dataset comprised of the prepared data from all scraped products. Another form will be a relational view of the data, and the final form will be the original raw datasets.

The system will exist solely on the cloud utilizing Azure Cloud Services and cloud.webscraper.io's services. I want to point out that the web scraper is an external service that I did not build, although I had to design the components that integrated with the API in a cloud-based manner to create scrape jobs and retrieve scraped data. Figure 4 below shows an overview of the system.

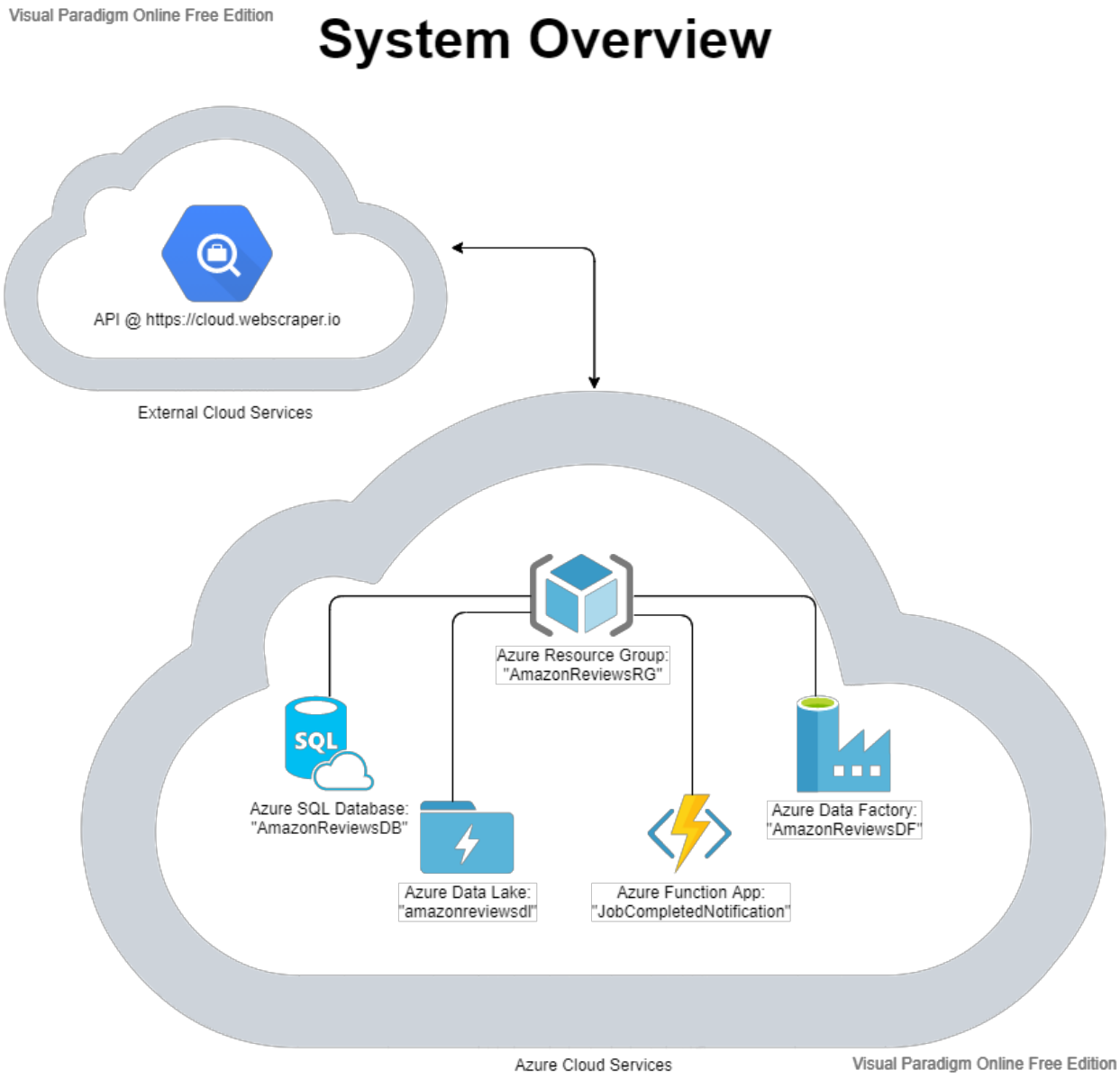


Figure 4. System Overview

4.3.3 Design of Implemented System

In this section, the linked services and datasets are described, and a general outline of the five main features in the pipeline are provided. The five features are: Product Loader, Data Scrape, Data Retrieval, Normalization, and CSV.

Linked Services

All ETL activities in the data factory require a linked service which basically links the activity to or from a storage resource. Since all data in for this project is stored in one of three locations

(Data Lake, Database, or Web Scraper's Cloud), I only needed three linked services to accomplish the necessary tasks. The example in Figure 5 outlines demonstrates the role a Linked service plays in the context of a Copy Activity.

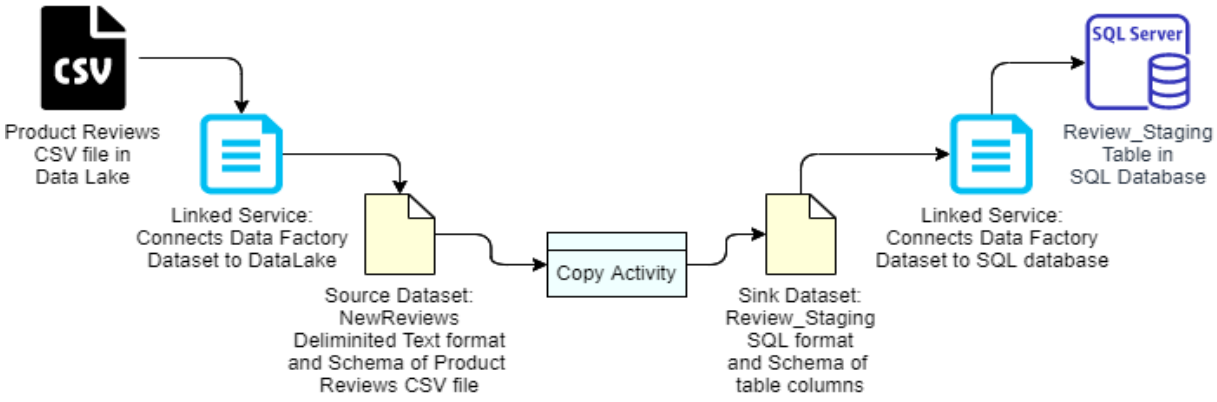


Figure 5. Linked Service role in a Copy Activity

The first linked service, named AmazonReviewsDB_LS, provides access to the database on. SQL Authentication is configured only once, at the time of creating the service. The database linked service is used to connect activities to or from the Review_Staging and/or CSV_Staging datsets. I will go into more detail on datasets in the next section. The following pipelines require a link to the database: Process New Reviews, Process Completed Jobs, Create Jobs, Generate JSON. As with the datasets, an in depth description of each pipeline will be described in the next few sections.

The AmazonReviewsDL_LS linked service provides access to the data lake. It is used by the [Process New Reviews] and [Process Completed Jobs] pipelines and by the following datasets: empty, ProductURLlist, JobsCompletedIDs, Reviews, Processed_URLs, ReviewsFolder, NewReviews, FailedRawReview, ProcessedReviews, DeleteJobs, CSV_dataset, and JSON_dataset.

Unlike the previous two datasets which link the data factory activity to an Azure cloud storage resource, the GETscrapedData linked service provides a link to a cloud storage outside of Azure. This involves configuring a REST service to send a GET request which takes one parameter,

{scrapingjob_id}, that is used in the base URL. This linked service is only used by the RESTgetScrapedData dataset.

Datasets

Datasets are not an actual stored dataset containing data, such as the final JSON dataset of the prepared review data output by this system. They are more like a set of instructions outlining the type of formatting the data files are in and the schema of the data. The datasets can be used for extracting data and or loading data to or from storage. In some cases, I found that I needed more than one dataset to represent the same data. One example is when I needed to look at the Reviews folder. In one case, I was only interested in the folders metadata so that I could generate an array of filenames. The other case was when I needed to look at the actual files in the folder, which involved a different schema. In total, the system uses 15 datasets.

- The ProductURLlist is a dataset used for extracting the ProductURLs.csv file in the Products folder. It is used by [Check csv for URLs] and [Create Jobs] pipelines.
- Empty is a dataset which, like its name implies, of an empty.csv file in the Products folder containing only the "URL" column header. This dataset is used to overwrite the productURLslist.csv file once all product URLs have been processed in the [Create Jobs] pipeline.
- Once scrape jobs have been created for each product URL, the URLs are appended to the processedURLs.scv file in the ProcessedFiles/URLs folder. To accomplish this, the Processed_URLs dataset is used in the [Create Jobs] pipeline.
- The NewReviews dataset represents a single file of which contains the schema for scraped amazon reviews datasets and is used for extracting files from the Reviews folder in the [Check for new Reviews] and [Process New Reviews] pipelines.
- Completed scrape jobs are downloaded to the Reviews folder. To do so, a Reviews dataset formatted for JSON is utilized within the [Process Completed Jobs] pipeline. It takes one parameter {scrapingjob_id} which is used as the filename.
- A third dataset is used to represent the same Reviews folder in the data lake. Unlike the NewReviews dataset which points is for a single file in the folder, the ReviewsFolder dataset does not contain amazon data schema. The dataset is used by a Get Metadata activity in the [Process New Reviews] pipeline.
- In the previous section, I mentioned the GETscrapedData linked service. The RESTgetScrapedData dataset relies on that linked service to retrieve scraped data from

the webscraper and load it to Reviews folder in the data lake. The dataset takes one parameter {scrapingjob_id} and is used in the [Process Completed Jobs] pipeline.

- Occasionally, data failed to load to the database, especially early on, and this required me to implement some method for containing this event. The scrapedFailedRawReview dataset was created for loading a file containing raw amazon reviews to the FailedReviews/RawReviews folder in the event of this type of failure. It is used in the [Process New Reviews] pipeline.
- Completed job notifications trigger a python function that saves the scraping job ids to a blob in the ScrapingJobIDs folder. The JobsCompletedIDs is a CSV formatted dataset containing the schema of the files in the ScrapingJobIDs folder. The dataset is used in the [Check for Completed Jobs] and [Process Completed Jobs] pipelines.
- Once reviews data for a product has been loaded to the database, the data is deleted from the Reviews folder. Before deletion, the data is copied to the Processed/Reviews folder using the ProcessedReviews dataset. The JSON formatted dataset is used in the [Process New Reviews] pipeline.
- The DeleteJobs dataset is formatted as CSV and contains the schema of blobs in the ScrapingJobIDs folder. The dataset is used to delete the scraping job blob once the scraped data has been downloaded to the Reviews folder. It takes one parameter {filename} and is used in the [Process Completed Jobs] pipeline.
- After the scraped data has be loaded to the Reviews folder, it is time to load it in the database. To aid in this , a Review_Staging table in the database is used. The Review_Staging dataset contains the schema of the the table and is used in the [Process New Reviews] pipeline.
- The final output of the system is a the JSON file containing all of product data. To accomplish this, the database contains a CSV_Staging table. The CSV_Staging dataset contains the schema for the table in the database used in the [Generate JSON] and [Generate CSV] pipelines. I should note for clarification, the final output is a JSON file, so JSON_Staging table and dataset would seem to be a more appropriate name. I will later discuss how I initially planned for the final output to be in the form of a CSV file, and why it became necessary to change it to JSON format.
- The CSV_dataset CSV formatted dataset contains the schema for the final output file containing all Product, Review, and Reviewer dat. It is used in the [Generate CSV] pipeline. Similarly, the JSON_dataset JSON formatted dataset contains the schema for the final output and is used in the [Generate JSON] pipeline.

JobCompleteNotification python function

To prevent the need for continuously polling the cloud.webscraper.io API to see if a scraping job has completed, the API sends a job completed notification via a POST request to an endpoint URL. This endpoint URL is the URL that calls an Azure HTTP trigger function named JobCompleteNotification. The python function was written and deployed using Visual Studio Code and can be seen in **Appendix D, Section XVI**. When the function is triggered, the request is parsed to extract the scrapingjob_id and sitemap_id. A logfile is created containing the scrapingjob_id and an outputblob binding is used to create a blob in the ScrapingJobIDs folder in the data lake. The blob contains the scrapingjob_id and sitemap_id. The function returns and HTTP Response to the API containing status_code=200.

Bindings for functions serve a similar purpose as Linked Services in the data factory. They provide the necessary connections used in the function. This function utilizes three types of bindings. The first is an httptrigger type binding named req. It allows GET and POST request methods. The second is a blob type binding named outputblob and with the direction configured to out. The binding saves a blob to the ScrapingJobIDs folder with a timestamp as the filename. The final binding is an http type with the direction configured to out. This binding is used to send a response to the HTTP request and is aptly named $return.

Product URLs Setup

Before getting into the pipelines, I will describe how product URLs can be loaded as input to the system. The Products folder in the azure data lake contains a file named ProductURLs.csv. URLs for the reviews page of an amazon product can be added to the ProductURLslist.csv file here. One URL per line, the URL should not be enclosed in quotations, and the line should not contain a comma after the URL. The file is periodically checked by a pipeline inside the azure data factory. The data factory operations are divided into three independent groups of pipelines, each performing a certain task. The first folder named CheckForURLs deals with new product URLs and creates scrape jobs. The second named CompletedJobs deals with completed scrape jobs and includes the downloading of the scraped data. The third named ProcessReviews deals with loading the scraped data into the database.

Pipelines

- Check csv for URLs - The pipeline is triggered by a schedule trigger which is currently set to 15 minute intervals. Figure 6 shows a diagram of the pipeline activities and a screenshot of the data factory is shown in Figure 7. The intervals can be set to any amount of time. When triggered, a Lookup activity named productURLlist looks at the first line in the ProductURLs.csv file contained in the Products folder. Control flows to an If Condition activity next where if the first line contains a record, an Execute Pipeline activity executes the [Create Jobs] pipeline. If no record exists, the pipeline terminates until triggered again. The pipeline utilizes one linked service, AmazonReviewsDL_LS, and one dataset, ProductURLlist.

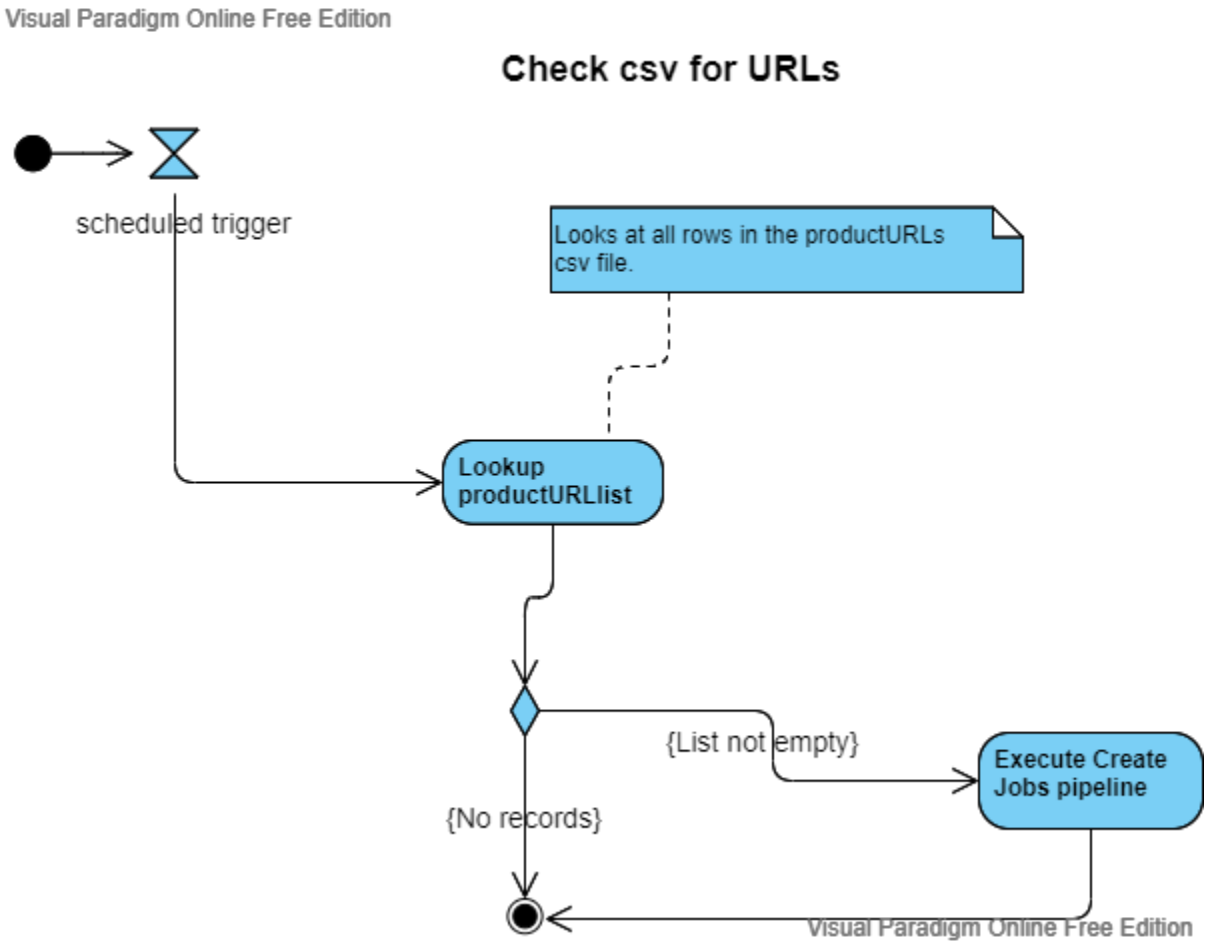


Figure 6. Check for URLs pipeline Activity Diagram

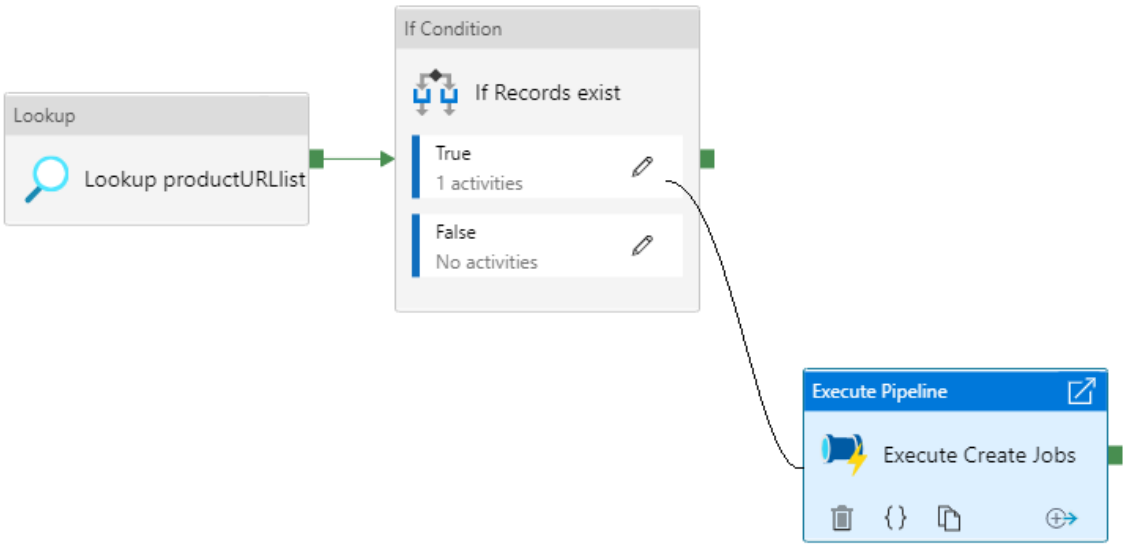


Figure 7. Check for URLs pipeline Data Factory Design

- Create Jobs - The pipeline contains six variables: splitURLarray [Array] for the substrings separated on "/", URL [String], Name [String], sitemap_id [String], scrapingjob_id [String], and failedURL [String] which is a dummy variable to handle exceptions. As shown in Figure 8, the pipeline begins with the same type of Lookup activity as previous, except this type it looks at all lines in the file. Control is passed to a ForEach activity that iterates over the records in the URL list. For each record, the URL is split on '/' sections are set to the splitURLarray variable. The two necessary component in the splitURLarray is the product name which is set to the Name variable. Next the entire URL is set the URL variable. Control is passed to a Web activity which sends a POST request to the cloud.webscraper.io API to create a new scraping sitemap. The POST request contains the json sitemap with the Name and URL variable passed into the value fields for "_id" and "startURL." A post response indicating the successful creation of the sitemap is received. The response contains a new sitemap_id which is set to the sitemap_id variable. Next a second Web activity sends another POST request to the API to create a new scraping job for the sitemap. The sitemap_id is passed into the POST body and a response is received upon successful creation of the scraping job. This response contains a new scrapingjob_id which is set to the scrapingjob_id variable. Finally, a stored procedure named ScrapeJobCreated is called that creates a new scraping job

with the new scrapingjob_id in the ScrapeJob table. The flow then proceeds to the next URL in the file. After all URLs have been processed, the URLs are copied to a ProcessedURLs folder and removed from the ProductURLs.csv file. The pipeline utilizes the following linked services: AmazonReviewsDL_LS, AmazonReviewsDB_LS. The ProductURLlist dataset and ScrapeJobCreated stored procedure are also utilized in the pipeline.

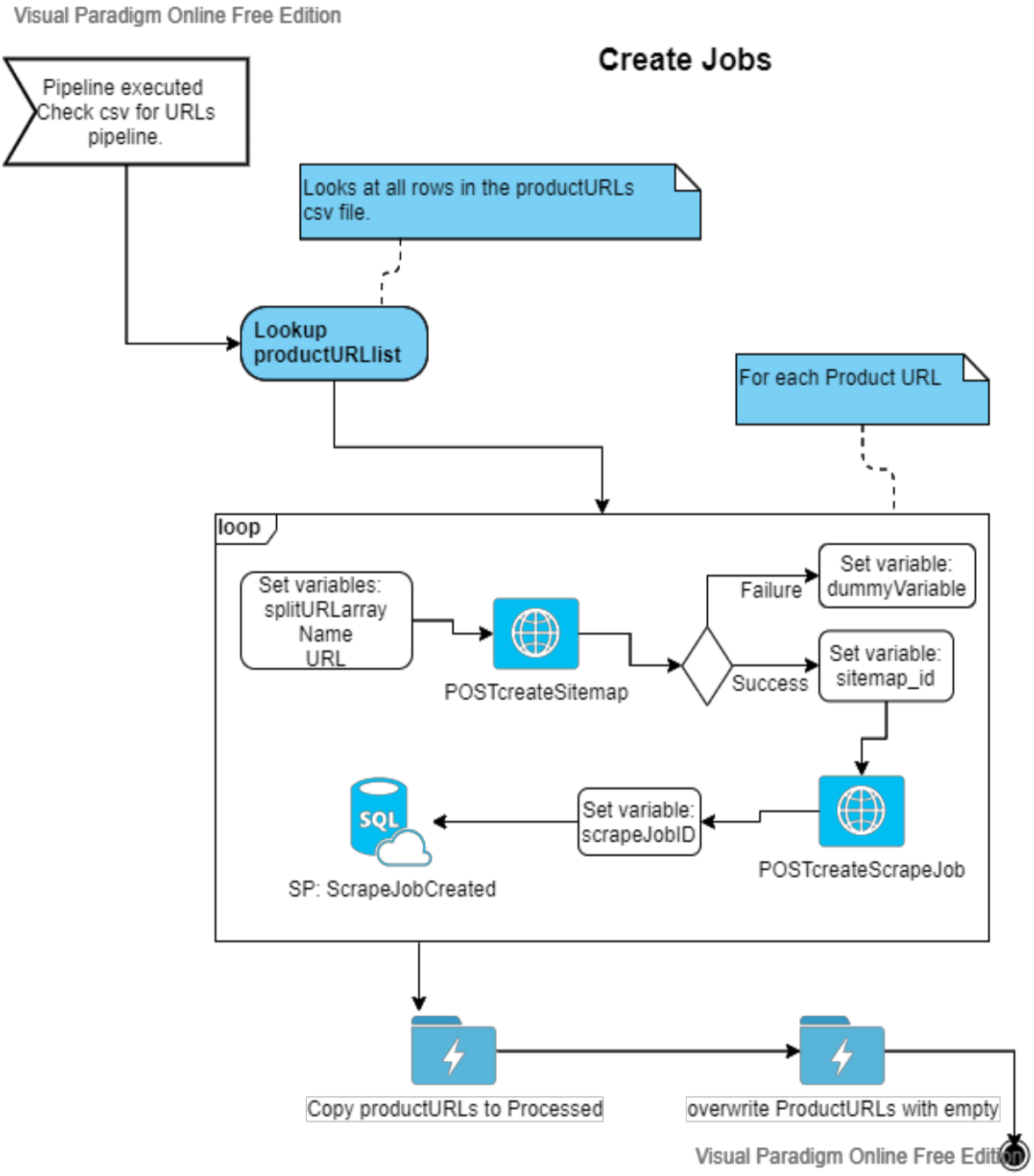


Figure 8. Create Jobs pipeline Activity Diagram

- Check for Completed Jobs - The pipeline is triggered by a schedule trigger which is currently set to 15 minute intervals. See Figure 9 for a diagram of the pipelines activities. Like the Check csv for URLs pipeline, the Check for Completed Jobs pipeline just checks the ScrapingJobIDs folder for new files by way of a Lookup activity. Then an If

Condition activity is used to check if the Lookup contained a file. If so, an Execute pipeline activity is used to execute the Process Completed Jobs pipeline. If no file exists, the pipeline terminates until triggered again. The pipeline utilizes the AmazonReviewsDL_LS linked service and the JobsCompletedIDs dataset.

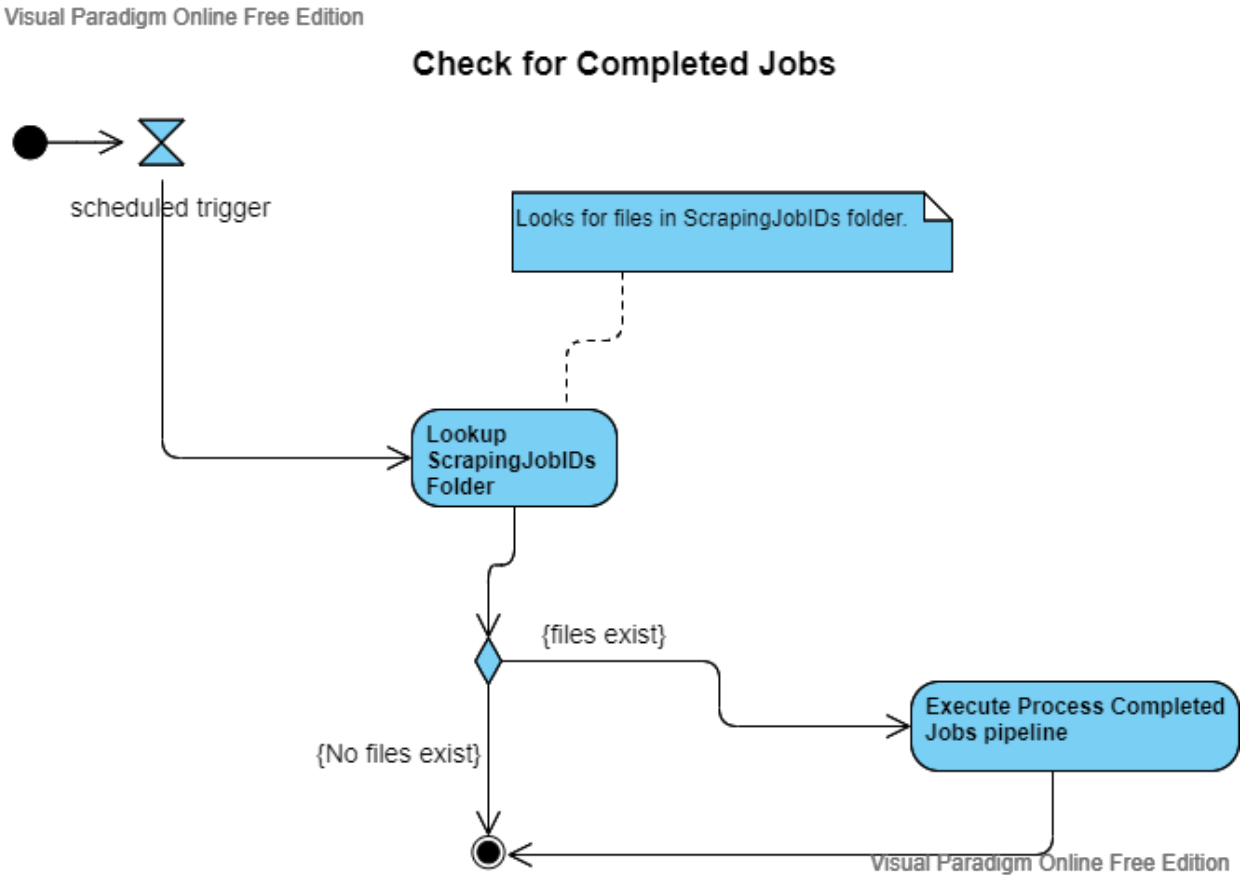


Figure 9. Check for Completed Jobs pipeline Activity Diagram

- Process Completed Jobs - The pipeline contains four variables: ScrapingJobIDs_filename_array [Array] which will hold all filenames in ScrapingJobIDs folder, scrapingjob_id [String], sitemap_id_array [Array] which will accumulate all sitemap ids of completed jobs, and SuccessfulJobs_filename_array [Array] which will hold all filenames for jobs that are successfully downloaded (See Activity Diagram in Figure 10). The pipeline begins with a Get Metadata activity that gets the Item name and child items in the JobCompletedIDs folder.

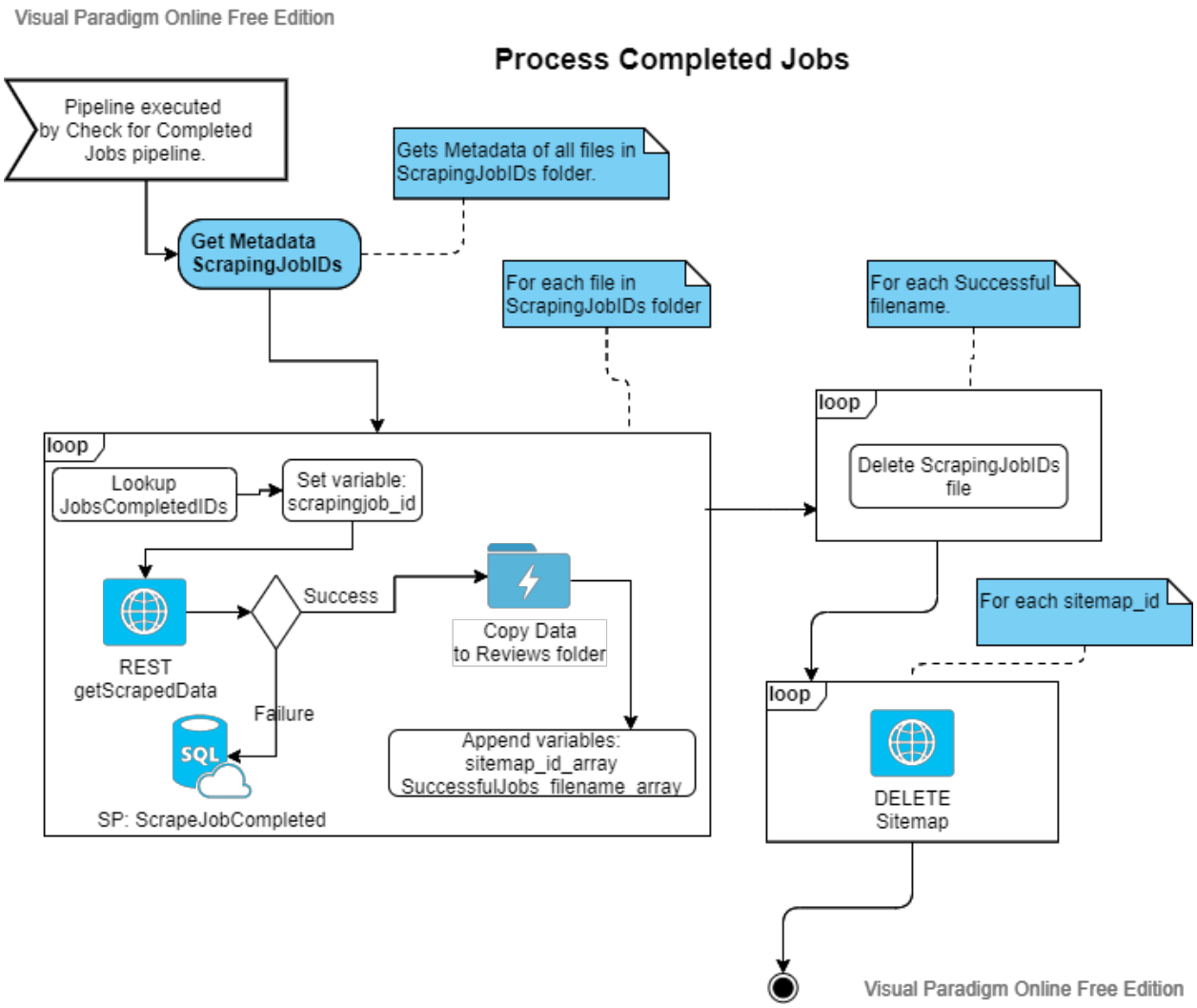


Figure 10. Process Completed Jobs pipeline Activity Diagram

Control flows to a ForEach activity where each file (child item) in the folder is processed. The iteration begins with a Lookup activity to look up the scrapingjob_id and sitemap_id in the file (a diagram of the Data Factory layout is shown in Figure 11). The scrapingjob_id is set to the scrapingjob_id variable control moves to a Copy data activity. To retrieve the data from the webscraper's API, a REST dataset containing a REST Linked Service must be utilized. The Linked Service contains the URL to send a GET request to the API. The URL contains the scrapingjob_id and an API token to identify the dataset to download. With the REST dataset as the copy source and the Reviews folder dataset as the sink, the scraped data is downloaded to the Reviews folder with a timestamp as the filename. Once the download has completed successfully, control flow

splits into three branches with the first send appending the sitemap_id from the Lookup activity to the sitemap_id_array. The second branch appends the filename (Name from Lookup activity) to the SuccessfulJobs_filename_array. The third branch calls a stored procedure named ScrapeJobCompleted that sets the job_status to 1 and the dataRetrieval_status to 1 in the ScrapeJob_Status table and sets the scrapejobComplete_date to current date/time in the ScrapeJob table. Figure 11 shows the Data Factory Design View which may provide some clarity of the processes involved.

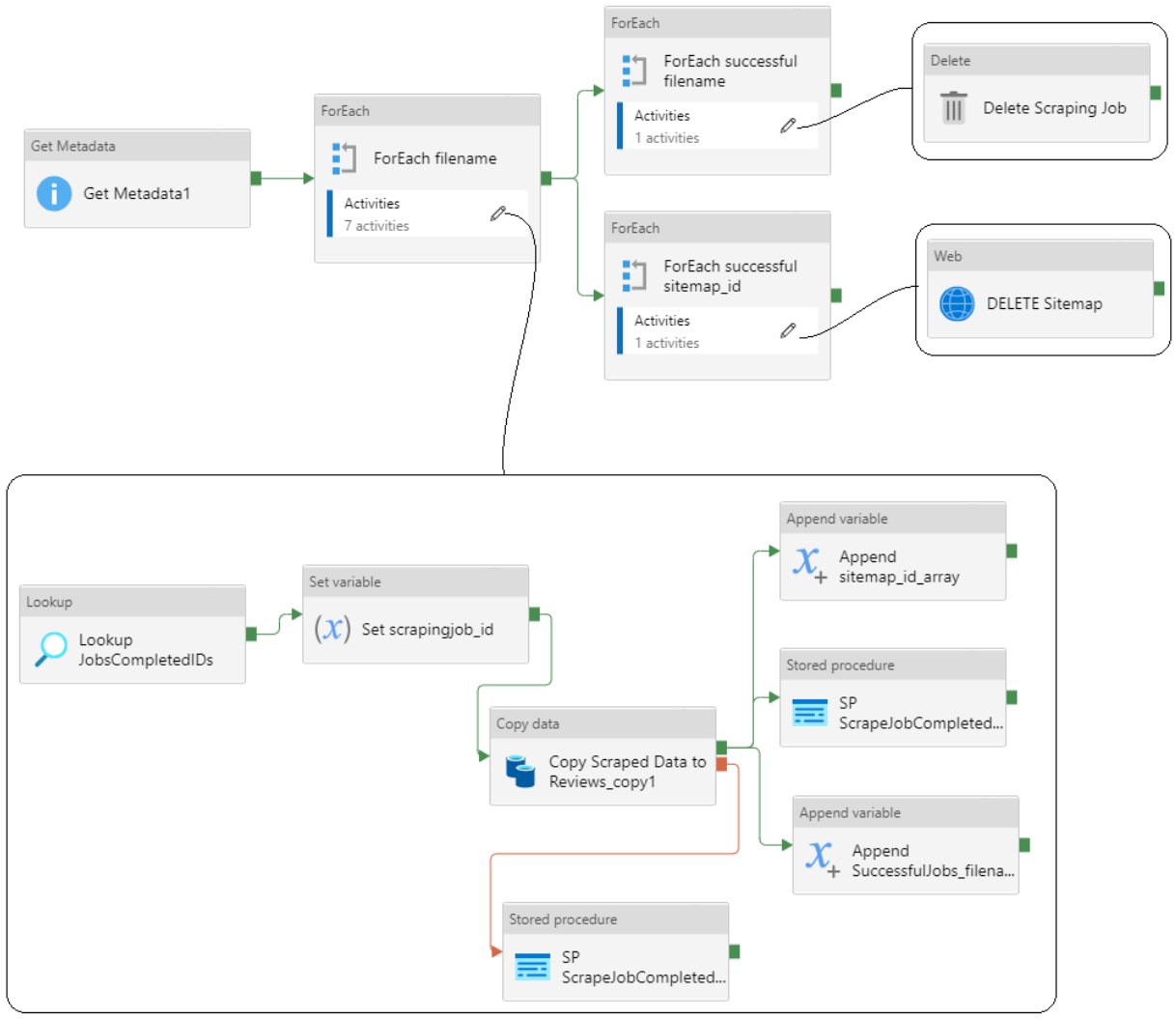


Figure 11. Process Completed Jobs Data Factory Design

The process continues until all scrape job data has been downloaded. After the ForEach activity has completed, control flow branches into two more ForEach activities. One iterates over the SuccessfulJobs_filenames_array and for each filename, a Delete activity is used to delete the file from the ScrapingJobIDs folder. The other one iterates over the sitemap_id array and uses a Web Activity to send an HTTP DELETE request to the web scraper's API requesting to delete the sitemap. Deleting the sitemap deletes the scrape job and scraped data as well. The pipeline utilizes the following linked services: AmazonReviewsDL_LS, GETscrapedData, and AmazonReviewsDB_LS. The RESTgetScrapedData dataset and ScrapeJobCompleted stored procedure are also utilized in the pipeline.

- Check for new Reviews - The pipeline is triggered by a schedule trigger which is currently set to 15 minute intervals. Like the Check csv for URLs and the Check for Completed Jobs pipelines, the Check for new Reviews pipeline checks the Reviews folder for new files by way of a Lookup activity. Next, an IF Condition activity executes the Process New Reviews pipeline if files exist. The pipeline utilizes the AmazonReviewsDL_LS linked service and NewReviews dataset.
- Process New Reviews - The pipeline contains two variables (filenames [Array], filename [String]). The figures below contain an activity diagram and data factory layout, respectively. As shown in Figure 12, a Get Metadata activity first gets all filenames in the Reviews folder and passes control to a ForEach activity that iterates over the metadata item names and appends them to the filenames array.

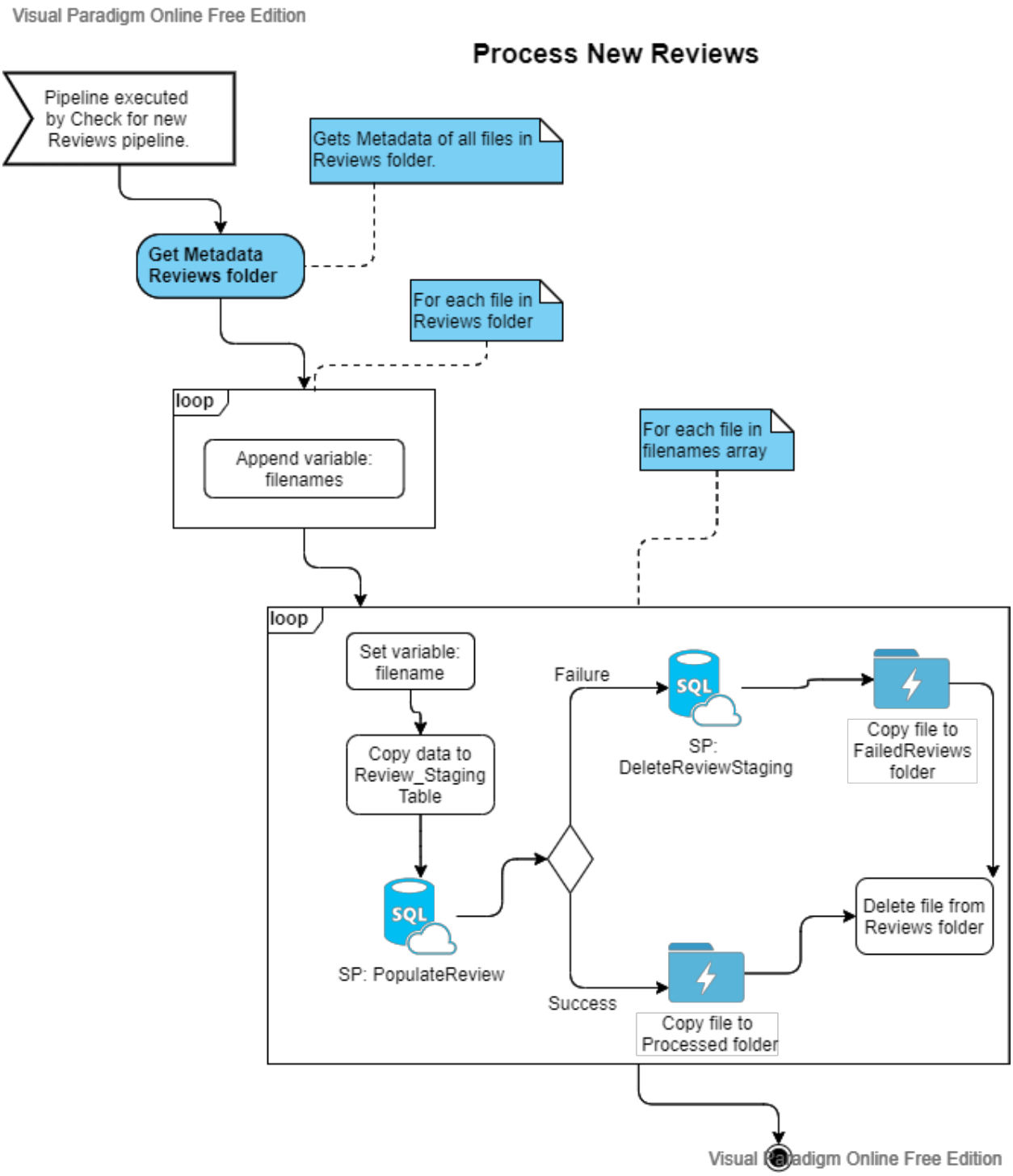


Figure 12. Process New Reviews pipeline Activity Diagram

Next each filename in the array is processed in a second ForEach activity (Figure 13 best shows the Data Factory design for this iteration). First, the filename is set to the

filename variable. A Copy activity is then used to copy the scraped data in the file to the Reviews_Staging table in the database. A stored procedure is then called that inserts the data into the Product, Reviewer, and Review tables for each review before a Copy data activity copies the raw scraped data to the Processed folder. Finally, the file is deleted from the Reviews folder with a Delete activity. The AmazonReviewsDL_LS and AmazonReviewsDB_LS linked services are used, and the PopulateReview and DeleteFromStaging stored procedures are utilized in the pipeline. The following datasets are utilized: ReviewsFolder, NewReviews, ReviewStaging, ProcessedReviews, and FailedRawReview.

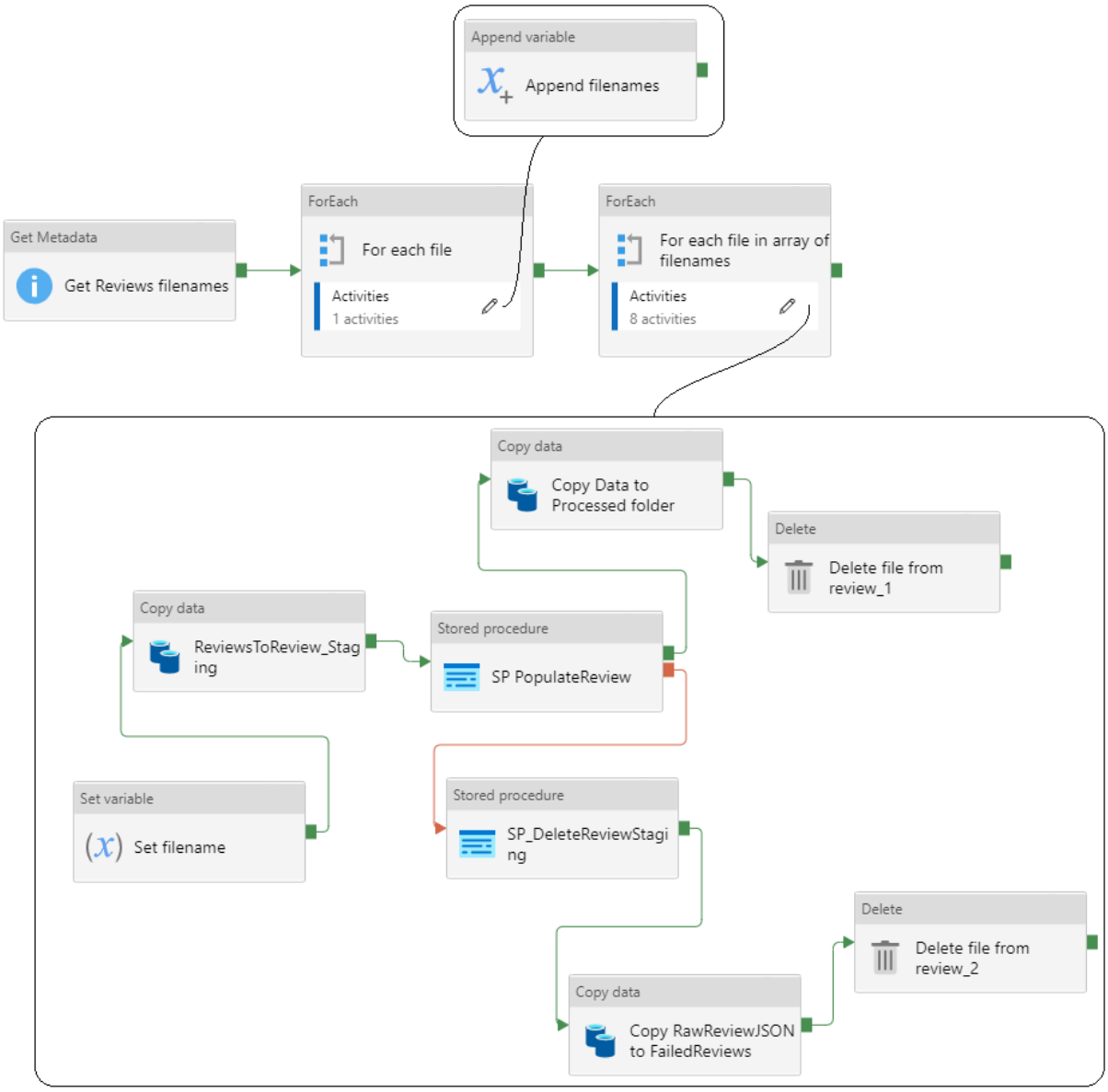


Figure 13. Process New Reviews Data Factory Design

- Generate JSON - The pipeline is manually triggered and is responsible for producing a comprehensive, highly repeatable, JSON formatted dataset containing data from the Product, Review, and Reviewer tables. The activity diagram and data factory layout of the pipeline can be seen in Figure 14. First, a stored procedure is called to load all data into the CSV_Staging table which is then copied to the JSON folder in the Azure data lake. The copy activity uses a JSON dataset which saves the JSON data to a file named amazon_reviews_dataset-<UTC-Timestamp>.json. Finally, a second stored procedure is called to delete all records on the CSV_Staging table. The pipeline utilizes the

AmazonReviewsDL_LS and AmazonReviewsDB_LS linked services, the CSV_Staging and JSON_datset datasets, and the CSV_Staging and Delete_CSV_Staging stored procedures.

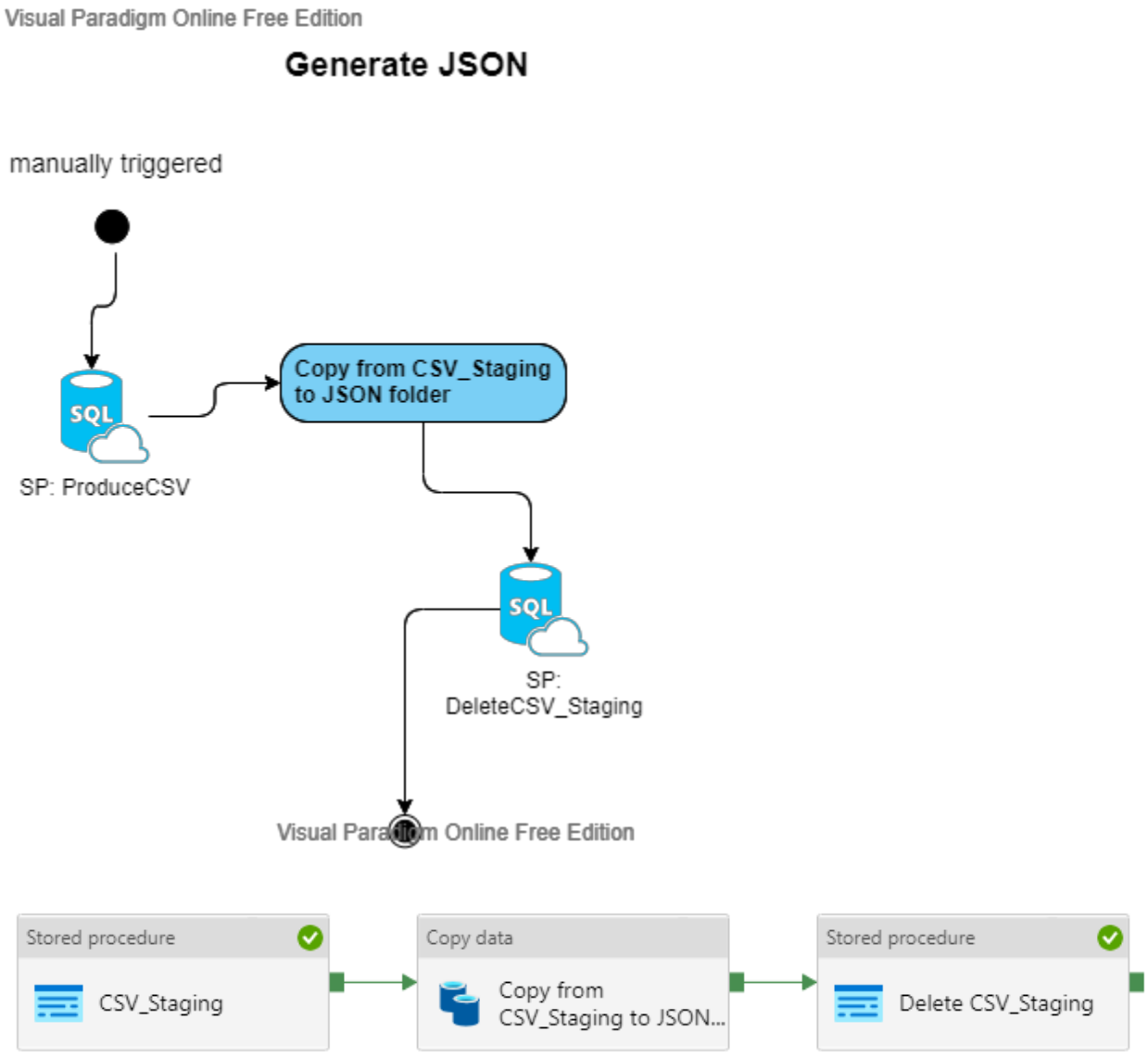


Figure 14. Generate JSON pipeline Activity Diagram and Data Factory Design

- Generate CSV - An alternate data output pipeline that produces a comprehensive, highly repeatable dataset. The CSV dataset contains data from the Product, Review, and Reviewer tables. First, a stored procedure is called to lead all data into the CSV_Staging table which is then copied to the CSV folder in the Azure data lake. The copy activity uses a CSV dataset which saves the CSV data to a file named amazon_reviews_dataset-<UTC-Timestamp>.csv. Finally, a second stored procedure is called to delete all records

on the CSV_Staging table. The pipeline utilizes the AmazonReviewsDL_LS and AmazonReviewsDB_LS linked services, the CSV_Staging and CSV_dataset datasets, and the CSV_Staging and Delete_CSV_Staging stored procedures

4.4 Relational Model

The relational model consists of five tables: Product, ScrapeJob, ScrapeJob_Status, Review, and Reviewer.

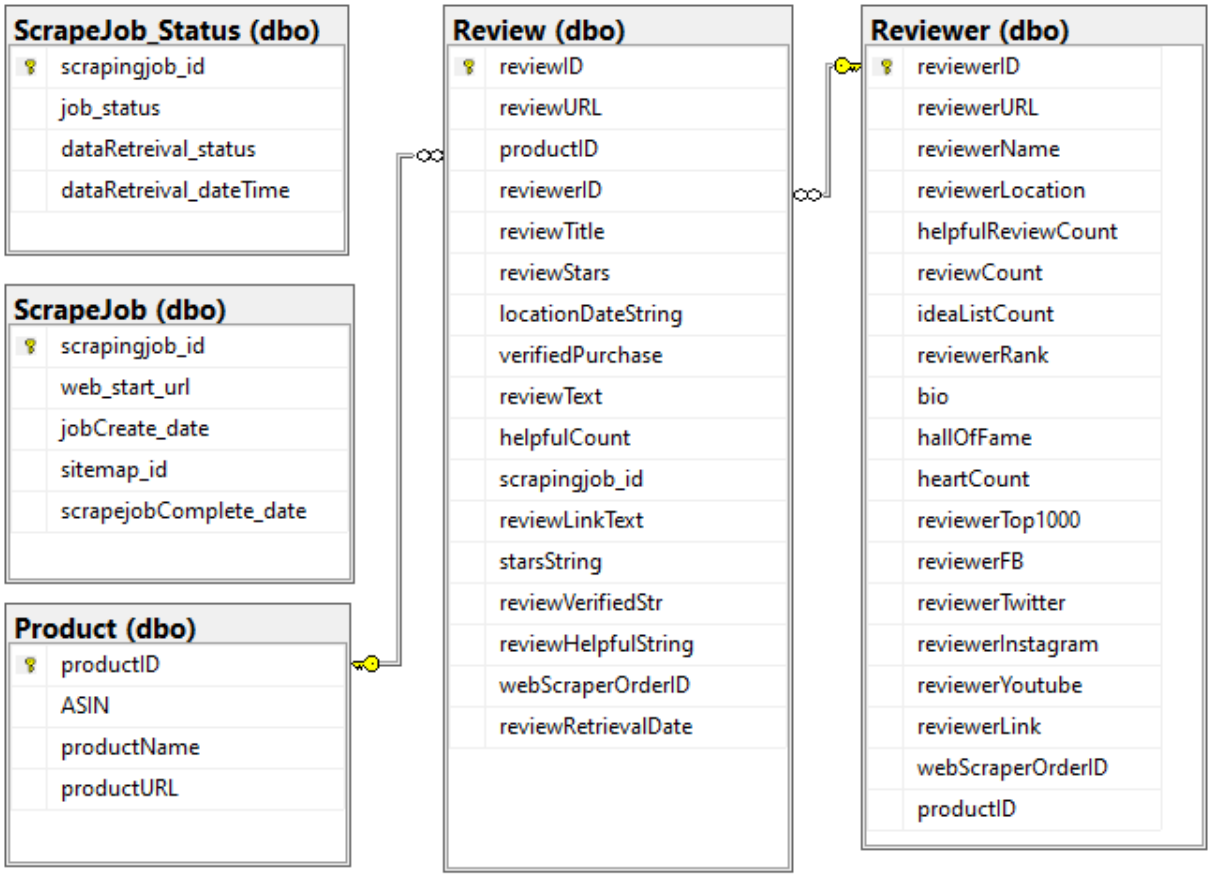


Figure 15. Relational Model Diagram

4.4.1 Product Table

Relationships will begin with the Product table when product instances are created by PopulateReviews stored procedure (refer to Figure 15 for relational model). The product table has a one to many relationship with the Review table. There are four non-Null attributes in the

Product table: productID, ASIN, productName, productURL. Two attributes are for recording status and will allow Nulls. One is the scrapeStatus in which starts with Null, and then is set to 1 whenever a scrape job for the product has been created.

4.4.2 ScrapeJob Table

The ScrapeJob table has three attributes that will not allow Nulls: scrapeJobID, web_scraper_start_url, and jobCreate_date. sitemap_id stores the scrape jobs sitemap id and scrapejobComplete_date stores a timestamp when data is retrieved. ScrapeJob shares a value on ScrapeJob.web_scraper_start_url and Product.productURL and ScrapeJob.scrapingjob_id and Review.scrapingjob_id.

4.4.3 ScrapeJob_Status Table

The ScrapeJob_Status table has a one to one relationship with ScrapeJob on scrapingjob_id. The second column, job_status does not allow nulls and is of type bit. When a job is created status is 0, and when completed, status changes to 1. The dataRetrieval_status of type bit is null until scraped data has been downloaded to the Reviews folder and changed to 1. The final column is dataRetrieval_dateTime and holds a timestamp of the time that data is downloaded.

4.4.4 Review Table

The Review table has attributes for the data from the review along with reviewID, productID, reviewerID, scrapeJobID, and scrapeDateTime and web-scraper-order. The scrapeDateTime is acquired from the ScrapeJob table. The review table shares a relationship with Product on productID and with Reviewer on reviewerID.

4.4.5 Reviewer Table

The Reviewer table has a one to many relationship with Review on ReviewerID and all other attributes are fields in the scraped data. The idea is that when a review instance is created, a reviewerID is generated and used to create the Review instance.

4.5 Sitemap

A sitemap is to be created to be used in the webscraper. The sitemap will include pagination references to show where multiple pages were visited recursively within an individual review.

Each pagination will include a link for that page. The fields of interest in sitemap will fall inside of review_link. The review link will contain the following items: title, stars, location date, verification, text, helpful string, and reviewer link. The reviewer link will contain name, location, helpful review count, review count, heart count, idea list count, ranking, image, bio, hall of fame, top 1000, and linked facebook, twitter, Instagram and youtube accounts. Note: Social media fields were removed due to inconsistent records. A diagram of the sitemap tree is in **Appendix C, section V**.

4.6 Maintenance

Once the system is in production, the only maintenance that should be necessary is rotating secrets and API tokens for security. This system design will not include automation for this process. Azure offers a service that collects information about the performance of deployed solutions in real time. With Azure Monitor, developers and DevOps can utilize Azure Insights, which offers recommendations based on performance. For example, it may pick up resources that are no longer being used and notify developers that they should be taken offline to save money.

5 PROJECT PLAN

The project plan includes the specifications of deliverables, cost in terms of man-hours and resources, which includes financial projections for the completion of the project.

5.1 SPECIFICATIONS OF DELIVERABLES

There are three primary deliverables of which define the completion of this project.

Document containing:

Outline and description of motivation and background, the project design and results/findings, and thoughts on potential future work.

System with these capabilities:

Azure Data Pipeline containing sub-pipelines for loading products, creating scrape jobs, retrieving scraped data, processing and saving data in a relational model, and producing a CSV or JSON dataset of the entire collection of reviews.

Quantify 15 products:

Generate a non-relational and highly repetitive CSV or JSON file of the entire collection of reviews for 15 products. Present a relational view of the data for products, reviewers, and reviews while preserving the original raw data retrieved from the web scraper.

5.2 Planned Cost in Terms of Man-Hours

The table in **Appendix C, section I** shows a detailed estimation of time requirements for backlog items for planning, building, testing, live production, analysis, writing, and preparation of defense phases. Items that fall early in the build have been given more time than others. The thinking behind this is that similar builds should go more quickly after completing some of the earlier builds.

The table in **Appendix C, section II** takes time estimates from the item backlog and applies them to a timeline based on predicted hours available per week. There are couple clarifications that should be made. The first is regarding week 2 (technically week 5) where the number of available hours for the entire week 6. The thinking behind this is that since there will have be some complex milestones reached at this time, it would be a good point to take a step away. This would allow some likely needed rest while offering some time to reflect on how things have been going thus far. The other clarification is regarding week 8. Fifty hours have been logged as estimated time required, however this is the amount of time predicted for the full-scale production run. In calculating total hours, only 23 man-hours are considered.

5.3 RESOURCES

As per the scope of this project, cloud.webscraper.io and Azure will be the extent of cloud service resources used for this project.

5.3.1 Cloud.webscraper.io

The need for API access to the webscraper began in the middle of October, so I have paid for a Project Subscription at $100 per month thus far. I foresee needing access until mid-March. During the research, building, and testing phases of the project, the Project Subscription will be fine. However, preliminary tests on the webscraper produced around 1100 records for one

product, took nearly 2500 page credits. If I plan to use the scraper for 50 products, I will likely need more than 100,000 credits. This means that from mid-February through mid-March, when I run the pipeline at full scale, the $300 Scale Subscription will be required. This puts the total cost of the webscraper subscription at $800 dollars.

5.3.2 Azure

The Azure subscription is free with my UNCW account. However, not all of services which I will be using are free. These paid services normally include a monthly upfront fee, but I will not be using them enough to incur any fees upfront. Additional fees are charged using a consumption plan where fees are charge per use of service. Looking at the financial projection (**Appendix C, section III**) created using the Azure Pricing Calculator, I have calculating a potential cost for the month of February or March (depending on when I run the production phase). The assumption is that earlier months will be only a small percentage of this estimate. Using an estimate just under 20% for calculating the months of November through February, it comes to $80. Adding this to the projected cost for March, comes to $186. In summation, that is $800 dollars for the webscraper and $186 for Azure Cloud Services, which comes to a total estimate of **$986**.

6 Discussion

6.1 Design Changes

6.1.1 File Structure

One of the deviations from the original plan was the addition of several new folders. The ScrapingJobIDs folder had to be created due to an issue that resulted from a limitation of the Azure Function in Python. My original plan was to update the database from the function when it was triggered by a job completed notification from the web scraper. As it turns out, if writing a function in Python, the only type of database that azure supports a binding for is CosmosDB. This meant that I would need another way to handle completed jobs. The work around that I decided to use was to write a blob to a folder named ScrapingJobIDs. The blob would contain the job's scraping job id and sitemap id. Then I could have the [Check for Completed Jobs] pipeline periodically check for new blobs. The other folders that were added were for error handling. Most of the implementation for handling errors was not included in the implementation of this system, however I wanted to know if data for a particular product did not download to the data lake or properly load to the database. For these two exceptions, I created a copy activity that moved the data to a folder for failed processes. This way, there

would be an easy way to see that something went wrong. The JobDeleteLogs folder was added to hold a complete log of all deletes that occur in the data factory and a JSON folder was added to hold a final JSON output dataset of all collected data.

6.1.2 Mapping Data Flow

The initial design of the data pipeline included the use of mapping data flow in data factory to process the reviews and make the necessary data type conversions in preparation for the database. For example, the product name and ASIN is included in the product reviews needed to be extracted from the product reviews URL (more on the necessary conversions is discussed in the data conversions section). To accomplish this task, a mapping data flow named parse reviews was creating in the data factory. When a product reviews dataset was downloaded, this data flow was called passing in the filename of the dataset as a parameter. The result was a new dataset containing additional columns for parsed and converted values. Reading documentation and trying to figure out how to do the necessary conversions was quite time-consuming and was responsible for close to a month of build time. The mapping data flow component of the data factory is far from intuitive and documentation is incomplete, covering one or two broad examples rather than drilling down into each function. To further complicate matters, version two of the data factory is still in development and it was not uncommon to leave for the night and come back in the morning and find that screen layouts and menu choices had been changed, and something that was once available in one place had been moved to another. Much of my learning process from through trial and error involving a lot of debugging. One frustration was the result of a single, vague error message, “Error: Bad Request,” with no elaboration as to where the bad request was occurring. Eventually, I was able to successfully process the data, but the mapping data flow took longer than expected. This raised questions of efficiency, so the process was abandoned.

Instead of using the mapping data flow, I began to read up on how to use regular expressions at the time of setting values to variables to convert the data and create new columns. Then I could use a stored procedure in SQL server passing in the variables for the columns in the Reviewer and Review tables in the database. Even though this accomplished the task, data factory used a new runtime for each batch of 50 reviews. The cost of using data factory is calculated by the multiplying the time used for each runtime by $0.025. This became a real factor when I noticed that it was taking way too long to load the reviews into the database. This issue involved the many runtimes and the amount of acceptable time that a runtime could

remain inactive, which is 4 minutes. This meant that a task that should take a second could potentially take up to 4 minutes to complete. For this reason, the use of a stored procedure to load the data was also abandoned. Instead, a copy activity within the data pipeline was used to copy the product reviews json file, unchanged, directly into a table in the database named Reviews_Staging. Then a stored procedure could do all necessary parsing and converting at the time of copying the data into the Reviewer, Review, and Product Tables at a fraction of the time and cost. This was efficient, but I was finished with debugging yet. The next section covers some of the issues related to using Azure SQL database and the problems that surfaced once I approached the final days of the build.

6.1.3 SQL Server

While I knew quite a bit of SQL going into the project, I was not as familiar with some of the methods necessary for updating the database in large batches. Several weeks were spent working on writing and testing the stored procedure to load the data into the database. Cleaning and converting the data needed to happen within the same procedure. The conversions are described in the Data Conversions section coming up. The final script to create the stored procedure is shown below:

```
CREATE PROCEDURE [dbo].[PopulateReview]
(
   @productID INT OUTPUT
)
AS
BEGIN
    DECLARE @URL                        VARCHAR(500)
       DECLARE @lengthProductName INT
       DECLARE @locationASIN            INT
       DECLARE @lengthASIN        INT
       DECLARE @productName             VARCHAR(100)
       DECLARE @ASIN              CHAR(10)

       SET @URL = (SELECT TOP 1 [web-scraper-start-url] FROM Review_Staging)
       SET @lengthProductName = CHARINDEX('/',@URL,24)-24
       SET @locationASIN = (CHARINDEX('/',@URL,@lengthProductName + 25))+1
       SET @lengthASIN = (CHARINDEX('/',@URL,@locationASIN))-@locationASIN
       SET @productName = SUBSTRING(@URL,24,@lengthProductName)
```

```
   SET @ASIN = SUBSTRING(@URL,@locationASIN,@lengthASIN)

INSERT INTO Product
   (
          [ASIN],
          productName,
          productURL
   )
VALUES
   (
          @ASIN,
          @productName,
          @URL
   )
SET @productID = @@IDENTITY;

INSERT INTO Reviewer
   (
          reviewerURL,
          reviewerName,
          reviewerLocation,
          helpfulReviewCount,
          reviewCount,
          ideaListCount,
          reviewerRank,
          bio,
          hallOfFame,
          heartCount,
          reviewerTop1000,
          reviewerFB,
          reviewerTwitter,
          reviewerInstagram,
          reviewerYoutube,
          reviewerLink,
          webScraperOrderID,
          productID
```

```
   )
SELECT
          CONCAT('https://amazon.com',
                 Review_Staging.[reviewer_link-href]),
          Review_Staging.reviewer_name,
          Review_Staging.reviewer_location,
          (SELECT (CASE WHEN Review_Staging.reviewer_helpful_review_count
                 LIKE '' THEN NULL
                      ELSE CONVERT(INT, REPLACE(SUBSTRING(Review_Staging.
                          reviewer_helpful_review_count,1,
                          LEN(Review_Staging.
                          reviewer_helpful_review_count)),',','',''))END)),
          (SELECT (CASE WHEN Review_Staging.reviewer_review_count
                 LIKE '' THEN NULL
                      ELSE CONVERT(INT, REPLACE(SUBSTRING(Review_Staging.
                          reviewer_review_count,1,
                          LEN(Review_Staging.reviewer_review_count)),
                          ',','',''))END)),
          (SELECT (CASE WHEN Review_Staging.reviewer_idea_list_count
                 LIKE '' THEN NULL
                      ELSE CONVERT(INT, REPLACE(SUBSTRING(Review_Staging.
                          reviewer_idea_list_count,1,LEN(Review_Staging.
                          reviewer_idea_list_count)),',','',''))END)),
          (SELECT (CASE WHEN Review_Staging.reviewer_ranking
                 LIKE '' THEN NULL
                      ELSE CONVERT(INT, REPLACE(SUBSTRING(Review_Staging.
                          reviewer_ranking,2,LEN(Review_Staging.
                          reviewer_ranking) - 1),',','',''))END)),
          Review_Staging.reviewer_bio,
          Review_Staging.reviewer_hall_of_fame,
          (SELECT (CASE WHEN Review_Staging.reviewer_heart_count
                 LIKE '' THEN NULL
                ELSE CONVERT(INT, REPLACE(SUBSTRING(Review_Staging.
                          reviewer_heart_count,1,LEN(Review_Staging.
                          reviewer_heart_count)),',','',''))END)),
          Review_Staging.reviewer_top_1000,
```

```
        Review_Staging.reviewer_fb,
        Review_Staging.reviewer_twitter,
        Review_Staging.reviewer_instagram,
        Review_Staging.reviewer_youtube,
        (SELECT SUBSTRING(Review_Staging.reviewer_link,11,CHARINDEX('>',
                          Review_Staging.reviewer_link, 1)-13)),
        Review_Staging.[web-scraper-order],
        @productID
FROM    Review_Staging

INSERT INTO Review
   (
        productID,
        reviewerID,
        reviewURL,
        reviewTitle,
        reviewStars,
        locationDateString,
        verifiedPurchase,
        reviewText,
        helpfulCount,
        scrapingjob_id,
        reviewLinkText,
        starsString,
        reviewVerifiedStr,
        reviewHelpfulString,
        webScraperOrderID,
        reviewRetrievalDate
   )
SELECT
        @productID,
        Reviewer.reviewerID,
        CONCAT('https://amazon.com',Review_Staging.[review_link-href]),
        Review_Staging.review_title,
        CONVERT(INT, SUBSTRING(review_stars,1,1)),
        Review_Staging.review_location_date,
```

```
            (SELECT (CASE WHEN review_verified LIKE 'Verified Purchase'
                          THEN 1
                          ELSE 0 END)),
             Review_Staging.review_text,
             (SELECT (CASE WHEN review_helpful_string NOT LIKE ''
                          THEN (CASE WHEN (SUBSTRING(review_helpful_string,1,
                                      CHARINDEX(' ',review_helpful_string,
                                       1))) LIKE 'One' THEN 1
                                       ELSE CONVERT(INT, REPLACE(SUBSTRING(
                                      (SUBSTRING(review_helpful_string,1,
                                             CHARINDEX(' ',
                                             review_helpful_string,1)-1)),
                                             1,LEN((SUBSTRING(
                                             review_helpful_string,1,
                                             CHARINDEX('',
                                             review_helpful_string,
                                             1)-1)))),',','',''))END)
                          ELSE 0 END)),
              ScrapeJob.scrapingjob_id,
              Review_Staging.review_link,
              Review_Staging.review_stars,
              Review_Staging.review_verified,
              Review_Staging.review_helpful_string,
              Review_Staging.[web-scraper-order],
              ScrapeJob.scrapejobComplete_date
    FROM    Review_Staging
       LEFT JOIN Reviewer  ON Reviewer.webScraperOrderID =
                          Review_Staging.[web-scraper-order]
            JOIN Product   ON Product.productURL =
                               Review_Staging.[web-scraper-start-url]
            JOIN ScrapeJob ON ScrapeJob.web_start_url =
                               Review_Staging.[web-scraper-start-url];


    DELETE
    FROM Review_Staging
END
```

Once the stored procedure working as expected, I began to test the entire system and quickly realized that data for some products were not always loading. The error message led me to realize that size constraints for some of the columns was not large enough. I spent several days trying to pinpoint which columns were creating the issue and finally I raised all string columns to hold at least 1000 characters. This solved the issue, and the system began work properly. The last thing that needed to be implemented was a way to create the CSV file contained all collected data. For this I used a Copy Activity to copy everything in the database to a single dataset. However, when looking at the final dataset, the data was scattered as if data within a record, column was being split at commas as well and consequently spreading the data across multiple columns. This is the reason that the final output was changed to a JSON datatype which kept values paired with the correct key.

6.1.4 Cloud Web Scraper

The web scraper performed as expected, however cost does become a factor. During development and testing, a Professional subscription costing $100 per month was used. This subscription came with 20,000 page credits, which was not enough to build a significant dataset, so the plan was to upgrade to the Scale subscription once in the production phase, which offered unlimited page credits. However, the web scraper's documentation noted the subscription did not include the built-in proxy feature of which all other subscriptions offered. Since the intention was to scrape Amazon reviews, rotating the proxy was necessary to avoid bot detection. What that meant, was that to use the scale version, I would have needed to pay for a customized proxy from a 3rd party vendor which would have increased the cost tremendously. Consequently, for the purpose of this paper, the business subscription for $200 a month was used, which offered 50,000 page credits and this was enough to complete scrape jobs for 15 product URLs.

6.1.5 Azure Key Vault and Security

The original plan was to use an Azure Key Vault to store credentials, API token, and connection strings. I encountered a problem in attempting to implement this feature due to the nature of my Azure subscription where UNCW is the tenant of my Microsoft account. To implement Azure Key Vault, a Managed Identity Application ID must be created in Azure Active Directory, of which I do not have access. Consequently, I was unable to use Azure Key Vault and instead used connection strings through my subscription to authenticate resources. My credentials to the Azure SQL database were stored in the linked service, and my API token string to

cloud.webscraper.io was stored as a global parameter in the data factory. Thus, implementing the key vault is recommended when building the data pipeline in a live setting. I should note that although the key vault was not implemented in this build, resources in the Azure Cloud were protected by a firewall which only contained policies for Dr. Kline's and myself's IP addresses.

6.2 Reviews Data

A sample of 15 product URLs were selected across 9 subcategories in the Audio Books & Originals category. Subcategories included Bios & Memoires, Self Development, Literature & Fiction, Business & Careers, Science Fiction & Fantasy, Teen & Young Adult, Health & Wellness, Computers & Technology, and Kids. A total of 17, 962 reviews were collected during scraping and prepared in the data factory pipelines. The webscraper selectors for social media links failed to capture data and were not included in the final dataset. Raw scraped data were maintained in original format and saved in the Processed/Reviews folder after being loaded to the database. All preparations made to data were performed within the database rather than mapping dataflow in the data factory (point to detours here) because this was found to be more efficient.

6.3 Data Conversions

For reference, the following is an example of actual scraped data for a single amazon review.


```
{"web-scraper-order":"1617174470-1718",
"web-scraper-start-url":"https://www.amazon.com/Hitchhikers-Guide-Galaxy/
      product-reviews/B0009JKV9W/ref=cm_cr_dp_d_show_all_btm?ie=UTF8&
      reviewerType=all_reviews",
"review_pagination":"Next page→",
"review_pagination-href":"https://www.amazon.com/Hitchhikers-Guide-Galaxy/
      product-reviews/B0009JKV9W/ref=cm_cr_arp_d_paging_btm_174?ie=UTF8&
      pageNumber=174&reviewerType=all_reviews",
"review_link":"Absolutely Hilariously Wonderfully Awsome",
"review_link-href":"https://www.amazon.com/gp/customer-reviews/
      R1Z61UWB53FRX6/ref=cm_cr_arp_d_rvw_ttl?ie=UTF8&ASIN=B0009JKV9W",
```

"reviewer_link":"C. Randall",
"reviewer_link-href":"https://www.amazon.com/gp/profile/amzn1.account.AGJ6L2DAF3S2B6PM5ZQDN5RMYJ4A/ref=cm_cr_srp_d_gw_btm?ie=UTF8",
"reviewer_name":"C. Randall",
"reviewer_location":"Grand Rapids, MI United States",
"reviewer_helpful_review_count":"93",
"reviewer_review_count":"22",
"reviewer_heart_count":"0",
"reviewer_idea_list_count":"1",
"reviewer_ranking":"#49,760,856",
"review_title":"Absolutely Hilariously Wonderfully Awsome",
"review_stars":"5.0 out of 5 stars",
"review_location_date":"Reviewed in the United States on May 19, 2003",
"review_verified":"",
"review_text":"This is definately one of my most favortie books. Like a good movie, you can read it again and again and find something new to love about it every time. Be careful reading this in public. People might think you're a little strange as you keep laughing out loud every minute or two. If you've never read this book, make sure you give it a try. I'm sure you'll love it too.",
"review_helpful_string":"One person found this helpful",
"reviewer_image":"<div class=\"a-section a-spacing-none circular-avatar-image\"style=\"background-image: url("//d1k8kvpjaf8geh.cloudfront.net/gp/profile/assets/search_avatar-8059b2ed8a963eda51ee0b024a379bc98b88e8b72ba77c7c37204308ce09b47b.png");background-size:contain;\"><img alt=\"\"src=\"/avatar/default/amzn1.account.AEB5C5VEGK7WTBS6KHRJ7NTGA7DA?square=true&max_width=460\"id=\"avatar-image\"> ",
"reviewer_bio":"I love movies so much it borders on obsession. If I wasn't trying to make a career out of it, people might...Read more",
"reviewer_hall_of_fame":"",

"reviewer_top_1000":"",
"reviewer_fb":"",
"reviewer_twitter":"",
"reviewer_instagram":"",
"reviewer_youtube":""}

The next three sections describe the transformations to the data that occurs at the time of loading to Product, Reviewer, and Review tables. For reference, below is a look at the same review once transformed and updated to the tables. For presentation purposes, the data is displayed as JSON for clarity.

{"reviewID":90189,
"reviewURL":"https://amazon.comhttps://www.amazon.com/gp/customer-reviews/
R1Z61UWB53FRX6/ref=cm_cr_arp_d_rvw_ttl?ie=UTF8&ASIN=B0009JKV9W",
"productID":48,
"reviewerID":60844,
"reviewTitle":"Absolutely Hilariously Wonderfully Awsome",
"reviewStars":5,
"locationDateString":"Reviewed in the United States on May 19, 2003",
"verifiedPurchase":0,
"reviewText":"This is definately one of my most favortie books. Like a
good movie, you can read it again and again and find something new
to love about it every time. Be careful reading this in public.
People might think you're a little strange as you keep laughing
out loud every minute or two. If you've never read this book,
make sure you give it a try. I'm sure you'll love it too.",
"helpfulCount":1,
"scrapingjob_id":3960619,
"reviewLinkText":"Absolutely Hilariously Wonderfully Awsome",
"starsString":"5.0 out of 5 stars",
"reviewVerifiedStr":"",
"reviewHelpfulString":"One person found this helpful",

"webScraperOrderID":"1617174470-1718",
"reviewRetrievalDate":2021-04-01 02:11:07.2900000,
"ASIN":"B0009JKV9W",
"productName":"Hitchhikers-Guide-Galaxy",
"productURL":"https://www.amazon.com/Hitchhikers-Guide-Galaxy/product-
reviews/B0009JKV9W/ref=cm_cr_dp_d_show_all_btm?ie=
UTF8&reviewerType=all_reviews",
"reviewerURL":"https://amazon.comhttps://www.amazon.com/gp/profile/amzn1.
account.AGJ6L2DAF3S2B6PM5ZQDN5RMYJ4A/ref=cm_cr_srp_d_gw_btm?ie=UTF8",
"reviewerName":"C. Randall",
"reviewerLocation":"Grand Rapids, MI United States",
"helpfulReviewCount":93,
"reviewCount":22,
"ideaListCount":1,
"reviewerRank":49760856,
"bio":"I love movies so much it borders on obsession. If I wasn't trying to
make a career out of it, people might...Read more",
"hallOfFame":"",
"heartCount":0,
"reviewerTop1000":"",
"reviewerFB":"",
"reviewerTwitter":"",
"reviewerInstagram":"",
"reviewerYoutube":"",
"reviewerLink":"https://images-na.ssl-images-amazon.com/images/I/51N3ifveuuL.
_CR37,0,425,425_SX48_.jpg"}

6.3.1 Product Details Data Fields

The product name and ASIN was extracted from the product reviews URL and was otherwise unchanged.

6.3.2 Reviewer Details Data Fields

Reviewer URL was acquired from the reviewer_link-href which required no further parsing.

Many of the fields, such as Helpful review counts for reviewers, were converted from strings to integers and this typically meant that only commas needed removing. For example: 1,223 was converted to 1223. However, sometimes more was required, as was the case in Reviewer ranking. Data for reviwer_ranking came in the form of "#1,465,332 and therefore required the leading "#" to be removed as well. The reviewer idea list count, heart count, and review count were also converted to integers. The final conversion required for reviewer fields involved extracting the link to the reviewer's image from the reviewer_link. This simply required removing the following leading and trailing characters: username.

6.3.3 Review Details Data Fields

Four review fields in the final dataset were the result of conversions, extractions and or additions to the data. The review URL field was a concatenation of "https://amazon.com" and the review_link-href field. The original data for the review stars field were in the form of "1.0 out of 5 stars," and this was maintained in the Review Stars String field, however the first character in the string was converted to an integer and saved in the Review Stars field. The Review Verified field was added and consisted of a 0 or 1, where a 1 was applied if the review verified string field contained "Verified Purchase." The final additional field related to review involved a more complex conversion. Data for the review helpful string came in the form of either "One person found this review helpful" or in the case of greater than one person, "2 people found this review helpful." To generate an integer value for the Helpful Count field in the final dataset, the substring up to the first space in the phrase was extracted and if it contained "One", then the field was set to the integer 1, otherwise the substring was simply converted to an integer. Note that when converting to integers, NULL data fields were set to 0.

6.3.4 JSON Dataset

As mentioned in the SQL Server section under Design Changes, the final output format was changed to JSON. The following shows the same review example used above at the final stage of output.


```
{"Product Name":"Hitchhikers-Guide-Galaxy",
"ASIN":"B0009JKV9W",
"Product URL":"https://www.amazon.com/Hitchhikers-Guide-Galaxy/product-
      reviews/B0009JKV9W/ref=cm_cr_dp_d_show_all_btm?ie=UTF8
      &reviewerType=all_reviews",
"Reviewer Name":"C. Randall",
"Reviewer Location":"Grand Rapids, MI United States",
"Reviewer URL":"https://amazon.comhttps://www.amazon.com/gp/profile/
      amzn1.account.AGJ6L2DAF3S2B6PM5ZQDN5RMYJ4A/ref=cm_cr_srp_d_
      gw_btm?ie=UTF8",
"Reviewer Helpful Review Count":93,
"Reviewer Review Count":22,
"Reviewer Idea List Count":1,
"Reviewer Rank":49760856,
"Reviewer Bio":"I love movies so much it borders on obsession.  If
      I wasn't trying to make a career out of it, people might...
      Read more",
"Reviewer Hall Of Fame":"",
"Reviewer Heart Count":0,
"Reviewer Top 1000":"",
"Reviewer Link":"https://images-na.ssl-images-amazon.com/images/I/
      51N3ifveuuL._CR37,0,425,425_SX48_.jpg",
"Review URL":"https://amazon.comhttps://www.amazon.com/gp/customer-
      reviews/R1Z61UWB53FRX6/ref=cm_cr_arp_d_rvw_ttl?ie=UTF8&ASIN=
      B0009JKV9W",
"Review Title":"Absolutely Hilariously Wonderfully Awsome",
"Review Stars":5,
```

"Review Location Date String":"Reviewed in the United States on May
19, 2003",
"Review Verified Purchase":false,
"Review Text":"This is definately one of my most favortie books.
Like a good movie, you can read it again and again and find
something new to love about it every time. Be careful reading
this in public. People might think you're a little strange as
you keep laughing out loud every minute or two. If you've
never read this book, make sure you give it a try. I'm sure
you'll love it too.",
"Review Helpful Count":1,
"Review Link Text":"Absolutely Hilariously Wonderfully Awsome",
"Review Stars String":"5.0 out of 5 stars",
"Review Verified String":"",
"Review Helpful String":"One person found this helpful",
"Review Retrieval Date":"2021-03-31T18:00:45.83"}

6.4 Metrics

Data were collected on scrape job creation date and time and data retrieved date and time. Note that cloud.webscraper.io only permitted three parallel jobs to run, therefore the time of job creation did not always equate to job start times. For this reason, for the purposes of experimentation, Product URLs were added to the product URLs list in azure in smaller batches. Additionally, job completion times did not always equate to data retrieval time since the retrieval pipeline operated on a scheduled trigger. The trigger was set to 15 minutes so collected data for scrape times were rounded to the nearest quarter hour. (See Scrape Time Metrics in appendix) The number of scraped reviews per product ranged from 293 to 2690 and 17,962 total reviews were collected. Scrape times per product ranged from 1.5 to 9.75 hours per review with a combined scraping time 67.5 hours. In looking at individual jobs, the number of records scraped per hour ranged from approximately 122 to 326 with a combined approximate average of records scraped per hour of 266.

6.5 Lessons Learned

If I had to come up with three things that I would do differently if I could go back to the beginning of the build, I would have taken the time to learn C#, represented all data using a JSON schema, and focused on doing transformations using SQL rather than spending so much time working with mapping data flows in data factory.

While I eventually reduced the number of Azure serverless functions to only one for receiving job completed notifications, a great deal of my time was spent trying to learn how to write functions in Python. There is a good bit of documentation for writing in Python, but Microsoft does not yet offer the ability write and edit functions from within the Azure Portal. Additionally, one of the major roadblocks that I encountered early on and ultimately led me to change the design of how the system handled job completed notifications from the web scraper was the inability to link the function to the Azure SQL Server Database. I found it odd that a binding was supported for CosmosDB but not Azure SQL database. In fact, serverless functions written in C# on the .NET platform does support a direct link to the SQL database. This was frustrating and the design changes took a lot of time that I had not planned for. The reason I decided to use Python in the beginning was because I thought it would save me time since I was more familiar with the language. Unfortunately, this decision ultimately cost me more in man hours once it was all said and done, and this is why I would go with C# and .NET if I did it again.

Throughout the project, I found myself flipping back and forth between the JSON and CSV file types resulting in a mix of the two in the final implementation. A lot of my time was spent learning how to write expression to allow for dynamic input in data factory activities. Many times, I needed to extract part of a record from the scraped data and when working with a CSV file, escape commands would be included in the string rather that escaping, which generated strings of strings or led to difficulties using separators to isolate data. Working with JSON data did not present these issues, but it took me quite a while to figure out to extract the data because many of the expression functions would not work with objects and produced the dreaded “Bad Request” error. Sifting through the documentation was not always helpful because the general examples never seemed to relate to what I was trying to do. Eventually I figured out how to work with JSON and CSV, for example, the following expression is for a look up activity to check if files exist in a dataset: @contains(activity('Lookup Reviews folder').output, 'firstRow'). Eventually, I discovered an easier way work with the data regardless of the file type using Copy activities, such as when scraped review are copied to the

Review_Staging table. This method simply required a dataset for the source and a dataset for the sink, and mapping to explicitly identified. Mapping JSON dataset proved to be much simpler, since I could import a sample file to set the schema for the dataset. Consequentially, I would choose to work with only JSON files if rebuilding the system.

Trying to implement a mapping data flow to parse, convert, and prepare data in new columns took several weeks of reading, trying, reading, trying, until I eventually figured things out. Unfortunately, as I approached the third week, I was already noticing the data factory was costing more than I anticipated. At one point, I calculated that to run a debug session was costing around $2.50 per hour and I was spending a lot of time debugging. One of the reasons that I was spending so much time debugging was when I was trying to transform data with a mapping data flow, it was taking forever. I worked with small datasets during debugging (around 40 reviews) and it was sometimes taking 30 seconds to a minute to run. To run in live production, I would be working with thousands of reviews, so cost was quickly becoming a concern. When I eventually tried to run a larger file (around 2000 reviews), preparing and loading seemed to go on forever. In fact, I only let if finish once and it took 7 hours. This prompted me to check my account and found that instead of being around $25.00 for the month, I was already at $150.00 for just 2.5 weeks. This required me to change the system's design once again and I decided to do all transformations of the data at the same time the Product, Reviewer, and Review tables were be updated. Using a stored procedure to accomplish this took care of the 2,000 plus records in seconds. For this reason, if I were to do the project again, I would avoid using a mapping data flow. I do feel that a mapping data flow would be a powerful tool if a database were not available, but it does come at a high cost.

7 Future Work

After looking at the prepared data, I realized that more preparations could have been implemented, however time constraints would not permit these transformations to be implemented. For example, the locationDateString could have been separated into Country, Year, Month, and Day. Another area that I feel could be addressed is related to keeping track of scrape job durations. The metrics for this purpose relied on when the pipeline that checked for completed jobs ran, which was every 15 minutes. I feel that this can tightened up quite a bit if there were implementation that polled the web scraper's API for the status of scrape jobs and log the time when they change from a status of scheduled to in progress. Then the time of the

job completed notification could be recorded, resulting in an accuracy within seconds rather than minutes.

While building and testing the system, it became abundantly clear to me that the cost of the cloud-based webscraper would be the largest factor in determining the overall feasibility of the design. The measure of success of the system was overshadowed by the fact that out of the approximate $930.00 to build and run the system, $600.00 was contributed to the webscraper, and the result was a little over 17,000 records from 15 products. The question is, does this return justify the investment. I feel that a worthwhile future project would involve building a similar system that incorporates another means to scrape the data. There are two alternative plans that come to mind, both utilizing a virtual compute environment within Azure.

The web browser version of webscraper.io is a free browser extension where sitemaps can be manually uploaded to scrape review data in the same manner as the cloud version. The scraped data can then be manually downloaded and stored for preparation. Scripts would automate the manual processes, resulting in a fully automated system existing solely on the cloud. While the above system would solve the problem of webscraper costs, the complexities involved in writing the scripts needed for automation could overshadow the benefits of these savings.

A second solution could be to bypass webscraper.io altogether and writing a custom webscraper to accomplish the same tasks. For example, Scrapy is a Python library that offers and entire web scraping framework that can be utilized with an API. Selenium and Beautiful Soup are two other popular python libraries for web scraping that could be employed. As with the previous design, a virtual environment could be utilized to run the Python code, so the system would exist solely on the cloud.

While accomplishing the same goals as the current system, both alternative solutions would remove the cost of using the cloud-based version of webscraper.io. I feel that a project comparing the current build with that of one or both alternative builds would be of great interest, consequently making it a worthwhile effort for future work.

REFERENCES

1) Azure Data Factory Documentation - Azure Data Factory. (2020). Retrieved October 15, 2020, from https://docs.microsoft.com/en-us/azure/data-factory/
2) Azure SDK for Python: Azure SDK for Python. (n.d.). Retrieved November 15, 2020, from https://azure.github.io/azure-sdk-for-python/
3) Azure Tools - Visual Studio Marketplace. (2019). Retrieved November 23, 2020, from https://marketplace.visualstudio.com/items?itemName=ms-vscode.vscode-node-azure-pack
4) Brockschmidt, K., M., Wells, J., Mabee, D., Stein, S., Toliver, K., . . . Hatakeyama, D. (2020, May 29). Use Python to query a database - Azure SQL Database & SQL Managed Instance. Retrieved November 23, 2020, from https://docs.microsoft.com/en-us/azure/azure-sql/database/connect-query-python?tabs=windows
5) Chu, A., Gailey, G., Gillum, C., King, J., Beliaev, A., Valavala, P., . . . Knox, S. (2020, March 12). Durable Functions Overview - Azure. Retrieved November 10, 2020, from https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview
6) Data Lake Storage for Big Data Analytics: Microsoft Azure. (2020). Retrieved November 23, 2020, from https://azure.microsoft.com/en-us/services/storage/data-lake-storage/
7) Documentation. (2020). Retrieved September 5, 2020, from https://webscraper.io/documentation
8) FitzMacken, T., Rabeler, C., Harvey, B., & Lyon, R. (2020, September 01). Azure Resource Manager overview - Azure Resource Manager. Retrieved October 20, 2020, from https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/overview
9) Gailey, G., Shoemaker, C., Dykstra, T., Fowler, C., Lin, C., Mabee, D., . . . Brockschmidt, K. (2020, September 20). Azure Functions Overview. Retrieved November 21, 2020, from https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview
10) Perlovsky, D., Campise, K., J., Coulter, D., & Ghanayem, M. (2019, November 01). Wrangling data flows in Azure Data Factory - Azure Data Factory. Retrieved October 20, 2020, from https://docs.microsoft.com/en-us/azure/data-factory/wrangling-data-flow-overview
11) Power Query M formula language reference - PowerQuery M. (2020). Retrieved November 10, 2020, from https://docs.microsoft.com/en-us/powerquery-m/
12) Wilhelmsen, C. (2020, September 13). Comparing Mapping and Wrangling Data Flows in Azure Data Factory. Retrieved October 30, 2020, from https://www.cathrinewilhelmsen.net/2019/05/13/comparing-mapping-wrangling-data-flows-azure-data-factory/
13) Your Machine Learning and Data Science Community. (2019). Retrieved November 23, 2020, from https://www.kaggle.com/
14) McAuley, J., Targett, C., Shi, Q., and Hengel, A., 2015. Image-based recommendations on styles and substitutes. http://cseweb.ucsd.edu/~jmcauley/pdfs/sigir15.pdf
15) AWS, 2019. Multilingual Amazon Reviews Corpus https://registry.opendata.aws/amazon-reviews

APPENDIX A

Comprehensive Description of Elements Contained in ARM Template

A declarative syntax is used in the ARM template (**Appendix D, section VIII**). The simplest structure of an arm template contains eight elements: $schema, contentVersion, apiProfile, parameters, variables, functions, resources, and outputs. These elements are broken down to further describe the elements contained in each. The following has been organized in a manner that separates the required and optional elements.

I. Required Elements for ARM template

The **$schema** element is required and declares the location of the file that describes the version of the template language. For deploying groups using the Azure Resource Manager tools extension in Visual Studio Code, use https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#.

The **contentVersion** element is required, but any number in the form of 1.0.0.0 can be used. It is simply an aid in assuring the correct template is being used when deploying resources.

The **resources** element contains the resources to be created. It has its own template (See **Appendix D, section XI**) containing the following elements: condition, type, apiVersion, name, comments, location, dependsOn, tags, sku, kind, copy, plan, properties, resources.

Required Elements for Resources

The **type** element is used to define which resource is being deployed. The format for defining a resource is [<namespace of resource provider>.<resource type>]. The **apiVersion** element is used to designate the version of the REST API being used to create the resource. Note that this can also be set at a higher level by using apiProfile to set the API version globally. The **name** element is used to provide a name to the resource. The **location** element is required for most resources. The value defines the region in which the resource will reside. It is best practice to use a parameter to specify location. When creating the parameter, the default should be set to resourceGroup().location. This allows the template user to set a location in which they have to permission to deploy to.

Optional Elements for Resources

The **condition** element is of type bool and can be set to "true" if resource is to be included in current deployment. The **comments** element can be used to provide documentation for the resource. It is best practice to specify the purpose for a resource. While the **dependsON** element is not required, it is necessary to use when certain resources must be deployed before the current resource. Multiple resources can be listed separated by a comma. An important note is that all resources that the current resource depends on must exist within the same ARM template or must already be deployed. The **tags** element can be used to list tags associated with the resource. The **sku** element can be used to define the resource SKU, if one is provided. The **kind** element can be used when a resource type is defined by a value, which is not the case for most resources. The **copy** element can be used if more than one instance of the resource is to be deployed. The **plan** element is used when a resource provides values for the plan to deploy. The **properties** element allows resource-specific configurations to be set. The **resources** element can be used to define resources that depend on the resource. An important note is using this element does not imply that a child resource depends on this resource. That must be implicitly set within the child resource.

II. Optional Elements for ARM template

The **apiProfile** element can be helpful when adding resources. API version is a requirement when adding resources so declaring with the apiProfile prevents the need for including it in each resource being deployed.

Resource deployment can be customized by setting **parameters** to be used during execution. It is recommended to use parameters for settings like SKU, size, or capacity, as these may vary based on environment. It is also suggested to use parameters for resource names. This allow easy identification for the resource throughout the template. Parameters should always be used for names and passwords or secrets. For passwords or secrets, the type should be set to securestring, and if passing sensitive data, the type should be set to secureObject. The parameters element contains its own template containing the following elements: parameter-name, type, defaultValue, allowedValues, minValue, maxValue, minLength, maxLength, and metadata. A template for parameters is in **Appendix D, section VIII**

Required Elements for Parameters template

- The **parameter-name** is required and is simply the name of the parameter. One constraint is that it must be a valid JavaScript identifier. It is considered best practice to use camel case for parameter names
- The **type** element is required, and it defines the data type of the element value. Acceptable data types include string, securestring, int, bool, object, secureObject, and array.

Optional Elements for Parameters template

- The **defaultValue** element can be set to always pass a preset value unless otherwise explicitly provided. Best practices advise using default values for all non-sensitive parameters.
- The **allowedValues** element is not required, but it can be used to ensure that only the correct values will be allowed. It is recommended that allowedValues be used sparingly.
- The **minValue** element is not required and can be used to set a minimum allowed value (inclusive). This element only applies to parameters of int type.
- The **maxValue** element is not required and can be used to set a maximum allowed value (inclusive). This element only applies to parameters of int type.
- The **minLength** element can be used to set a minimum length (inclusive). This element applies to string, secure string, and array type parameters.
- This **maxLength** element can be used to set a maximum length (inclusive). This element applies to string, secure string, and array type parameters.
- A **metadata** object can be placed anywhere in a template for adding extra details such as comments or aurthorship. However, when a metadata object is used in parameters, comments should be placed in a "description" element. The Resource Manager ignores metadata objects at the time of deployment. Best practices suggest that a description be provided for each parameter.

The variables element can be used to simplify the template by setting commonly used lines or expressions to variables. It is advised that a variable be used for any value that will be used more than once. Further, using variables for complex expressions makes the code easier to read. The element contains its own template (See Appendix D, section IX) that contains the following elements: variable-name, variable-object-name, and copy.

- The required **name** element is the name the variable is to be called. Its value can be set statically, or it can be a complex-variable-type for dynamic use. One example of a complex-variable-type could be a concatenation of a string with parameters. Camel case is considered best practice when naming variables
- The optional **copy** element can be used when it is necessary to set several values to a variable. For instance, this can be used to concatenate an incremented value to a string during iteration. Copy can also be used in resources, properties in a resource and output.

The functions element can be used to define expressions for use throughout the ARM template, rather than rewriting every time it is needed. This element contains its own template (See Appendix D, section XII). There are some limitations when using functions. Access to global parameters and variables defined in the ARM template is denied. Additionally, a function cannot call another function. When calling a function, parameters must be set, as default parameters are not allowed. The following elements are contained in the functions template: namespace, function-name, parameter-name, parameter-value, output-type, and output-value.

Required Elements for Functions template

- The **namespace** element is used to avoid naming conflicts with template functions
- The **function-name** element should be unique. To call the function, the function name is preceded by namespace. [<namespace>.<function-name>()].
- The **output-type** element is used to set the type of the output values.
- The **output-value** element value is the expression to be evaluated and returned.

Optional Elements for Functions template

- The **parameter-name** element can be used to define parameters to be passed when calling the function.
- The **parameter-value** element can be used to set the type of the parament value that can be passed.

Outputs template

The outputs element can be used to specify return values in deployment. It has its own template (See appendix D, section XIII) and that contains the following elements: output-name, condition, type, value, copy.

Required Elements for Outputs template

- The **output-name** element is used to set the name of the output value.

- The **type** element is used to set the output data type.

Optional Elements for Outputs template

- The **condition** element of type bool can be used to specify if the output value is returned. Default if true.
- The optional **value** element contains the expression to be evaluated and returned.
- The **copy** element can be used if more than one output value is necessary.

APPENDIX B

Journal

I. Data Import

One of the decisions that needs to made is regarding the file type of which data is imported from scrape jobs. Cloud.webscraper.io offers four main choices: JSON, CSV, XLMS, and PHP. Documentation for the webscraper recommends using JSON. The reason being, JSON is cleaner, particularly in differentiating escape sequences and end of record sequences. The documentation warns that problems have been mentioned that PHP exports with incorrect default implementation. CSV format is discouraged, as it is dependent on the type of CSV reader, where unusual side-effects can occur when differentiating end of file and end of record. Although, initial attempts at importing JSON data via HTTP GET reports an error stating it is unable to import the JSON file, found unexpected characters at the end of the record. When building the data factory, importing as JSON directly mapping to a JSON dataset did not produce the same error as when attempting to import using a logic app. Therefore, it was decided that all data imports would be formatted as JSON using data factory.

II. Job Completed Notification

To receive job completed POST notifications from the web scraper API, it is necessary to configure an HTTP Endpoint. A HTTP trigger function using Python in Visual Studio Code serves this purpose. Initial tests using this function results in a status 500 response (internal server error). The expectation is to receive the POST form, parse the JSON body, and create a message containing the scrapingjob_id in a queue storage container. A subsequent queue trigger function then pops the queue for the scrapingjob_id. The function then sends a GET request to cloud.webscraper.io to download data. It seems that a logic app solves this problem, in that two separate functions are required to accomplish this task. An alternative to using a queue to hold the scrapingjob_id is to store it in a database. Using a logic app within a pipeline in data factory allows parameters and variables to be set. This makes applying unique naming conventions more convenient. Additionally, it allows the separation of the job completed notifications, and the retrieval of data. This option seems to fit a big data solution more appropriately. [NOTE: When building the pipeline, this plan was altered so that everything would be done in the data factory. The need for logic apps and queue triggers would no longer

be necessary and pipelines for data retrieval were set up as scheduled triggers instead. This simplified the design by having a single function app that created a blob containing the scraping job and sitemap ids whenever a job completed notification occurred. When the data retrieval pipelines would trigger, it would simply check for new blobs in the ScrapingJobIDs folder.]

III. Creating Scrape Jobs

To create new scrape jobs, changes to the sitemap must be made. In particular, the web start URL for different products must be updated. The process is to send a POST request to delete the current sitemap the API, a second POST request to upload the new sitemap, a third POST request to delete the scrape job, and a fourth POST request to create the new scrape job. The web scraper's API offers another option in which a POST request is sent to update a scrape job which overwrites the current web start URL. The API automatically overwrites the URL in the sitemap as well, and consequentially, four API calls is reduced to just one API call. There is a limit on the number of API calls that can be made within a certain amount of time which makes the latter option more optimal. [NOTE: While building the data factory, this plan was somewhat simplified. Rather than working with one job at a time, the sitemap was altered before sending the request to the API. The new product URL and name was inserted into the sitemap and then the request created an entirely new sitemap rather than overwriting the same one in the API. The sitemap id was captured, and another request was made to create a scrape job using the id. This returned a scraping job id which was subsequently stored in the ScrapeJob and ScrapeJob_Status tables on the database. Once the scraped data was received, sending one request to delete the sitemap effectively deleted the scrape job as well.]

IV. Proxy Utility

A major factor in the effectiveness of the web scraper is in avoiding bot detectors. Manually running the web scraper produces irregular results. One scrape job may produce 400 records with additional runs of the same sitemap producing 1000 records for one, and 70 records for another. This anomaly is likely due to interruptions to the scrape job by bot detectors. The paid version offers an option to utilize a proxy feature. This feature rotates through different ip addresses of which disguise the scraper from detection. Tests, when using the proxy feature produces consistent, repeatable results. This option appears to be a necessity which is also another good reason for using a paid subscription for the web scraper. However, the paid subscriptions come with a limited amount of page credits - 20,000 for Profession and 50,000 for

Business - unless the Scale subscription is purchased. [NOTE: The scale subscription seemed perfect for the final tests of the pipeline however it does not come with the proxy feature. Instead, a feature to add a customized proxy is provided. This meant that to rotate proxies, an additional purchase of a third party proxy service would be required adding substantial costs to the overall project. For this reason, the Business subscription was used to scrape 15 products for the final run.]

APPENDIX C

Graphs and Diagrams

I. Estimated time by Project Phase

Estimate of Project Time Requirements				
Phase	**Feature**	**Backlog Item**	**Estimated Time in Hours**	**Notes**
Planning		Create and Debug webscraper sitemap	6	
		Complete design of individual components including a list of Azure resources	15	
		Finalize file structure including naming conventions	3	
		Compile a list of 50 products	4	This may require some of Dr. Kline's time as well (1 hour est.)
Build		Deploy necessary Azure resources	5	
		Configure initial Keys and Secrets in Key Vault	2	
		Create a CSV file containing URLs of 50 products	2	
	Product Loader	Product Loader Logic App scheduler function	6	
		python iterator to read URLlist and create Product instances in Database	20	I'm allowing more time for these stages since they will be the first Logic apps that are built. I'm planning on similar items taking less time in the future.
	Data Scrape	Update Key Vault configurations	1	
	Data Retreival	HTTP trigger for job completed notifications	8	
		HTTP function to retrieved scraped data	15	
		HTTP function to Delete Scrap jobs on web scraper	8	
		Update Key Vault configurations	1	
	Normalization	Scheduler function to check if data exists to be	6	
		Wrangling Data Flow	40	This is quite extensive because this is where all the cleaning and preparation of the data will happen. This may require some of Dr. Kline's time (3).
		Python function to input normalized data into database	8	
		Update Key Vault configurations	1	
	CSV	Create manual trigger function to create CSV file	8	
		Update Key Vault configurations	1	
Testing/Debugging	All	Testing using CSV file containing 3 products	30	
Production Run	All	Run the system with complete system using CSV file containing 50 products	50	This is the estimated time requirement for running 50 products. My time should be around 5 hours.
Analysis		Compile notes on findings and issues/challenges along the way	30	
Writing		Complete the Document for my defense	50	
Defense Preparation		Build presentation including talking points and power point slides	40	
		Total estimated time required:	349	
		350 hours/15 weeks is just over 23 hours		

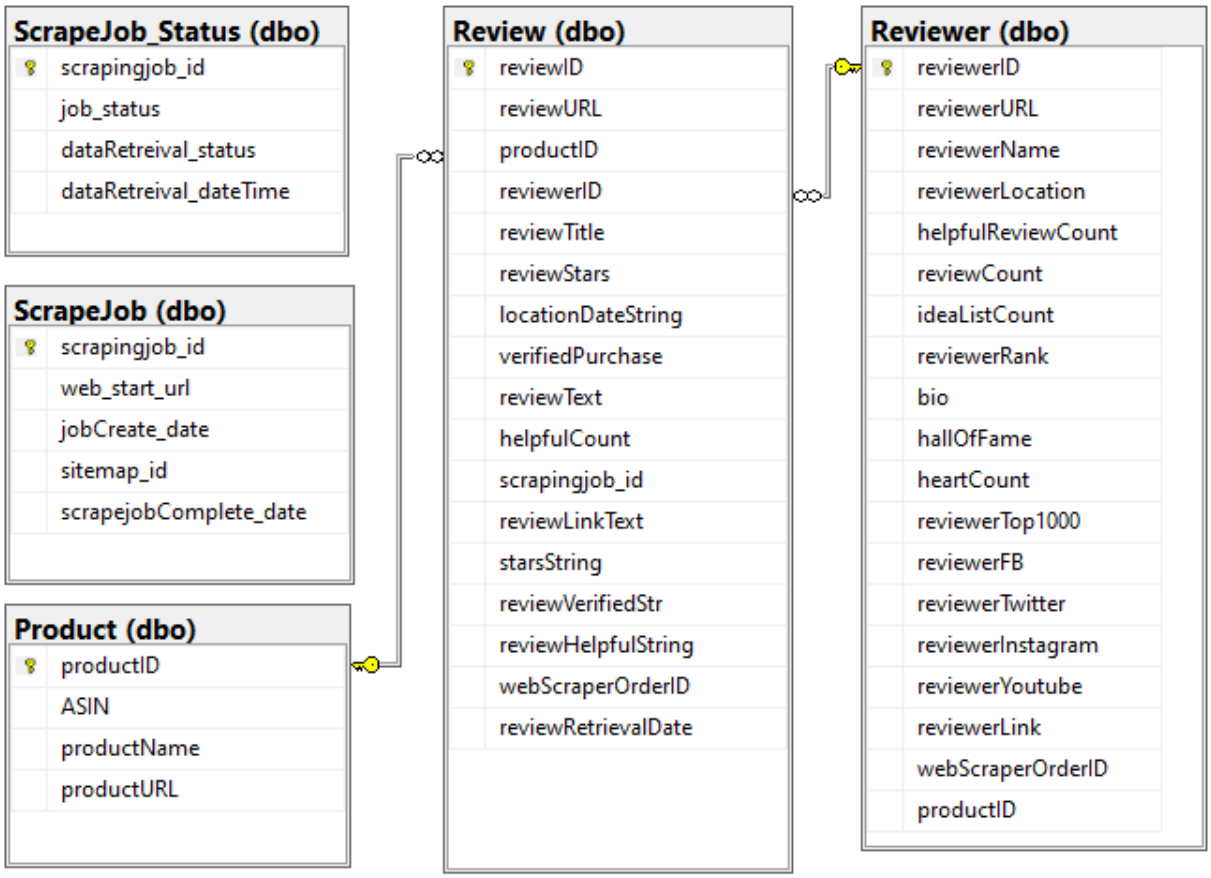
II. Estimated Project Timeline

Weekly Breakdown of Project Timeline					
Week	**Date Range**	**Available Hours**	**Backlog Items**	**Time est.**	**Notes**
i	12/13 - 12/19	16	Create and Debug webscraper sitemap	6	
			Begin design of individual components including a list of Azure resources	10	
ii	12/20 - 12/26	18	Finish design of individual components including a list of Azure resources	5	
			Finalize file structure including naming conventions	3	
			Compile a list of 50 products	4	
			Deploy necessary Azure resources	5	
iii	12/27 - 01/02	23	Configure initial Keys and Secrets in Key Vault	2	
			Create a CSV file containing URLs of 50 products	2	
			Product Loader Logic App scheduler function	6	
			Begin Python iterator to read URLlist and create Product instances in Database	13	
1	01/03 - 01/09	48	Finish Python iterator to read URLlist and create Product instances in Database	7	
			Update Key Vault configurations	1	
			scheduler function to check if new scrape jobs exit	3	
			Logic App - python function to create scrape jobs.	30	
			Update Key Vault configurations	1	
			Begin HTTP trigger for job completed notifications	6	
2	01/10 - 01/16	6	Finish HTTP trigger for job completed notifications	2	
			Begin HTTP function to retrieved scraped data	4	
3	01/17 - 01/23	23	Finish HTTP function to retrieved scraped data	11	
			HTTP function to Delete Scrap jobs on web scraper	8	
			Update Key Vault configurations	1	
			Scheduler function to check if data exists to be normalized	3	
4	01/24 - 01/30	23	Complete Scheduler function to check if data exists to be normalized	3	
			Wrangling Data Flow	20	
5	01/31 - 02/06	23	Complete Wrangling Data Flow	20	
			Begin Python function to input normalized data into database	3	
6	02/07 - 02/13	23	Finish Python function to input normalized data into database	5	
			Update Key Vault configurations	1	
			Create manual trigger function to create CSV file	8	
			Update Key Vault configurations	1	
			Testing using CSV file containing 3 products	8	
7	02/14 - 02/20	23	continue testing	22	
8	02/21 - 02/27	23	Run the system with complete system using CSV file containing 50 products	50	While the production run may take 50 hours, it will not require a lot of my time. It could feasibly be run 3 times in one week. During this time, I can work on paper.
9	02/28 - 03/06	23	Compile notes on findings and issues/challenges along the way	23	
10	03/07 - 03/13	23	Continue compiling notes	7	
			Begin writing the Document for my defense	16	
11	03/14 - 03/20	23	Continue with writing	23	
12	03/21 - 03/27	23	Finish writing document	11	
			Begin work on Presentation	12	
13	03/28 - 04/03	23	Continue work with presentation	23	
14	04/04 - 04/10	23	Finalize presentation	5	This leaves roughly a 40-hour pad for adjustments.
15	04/11 - 04/17	23			
Total Available Hours:		**410**	Estimated total hours required:	**367**	
			NOTE: Only 23 required hours are added for Week 8		

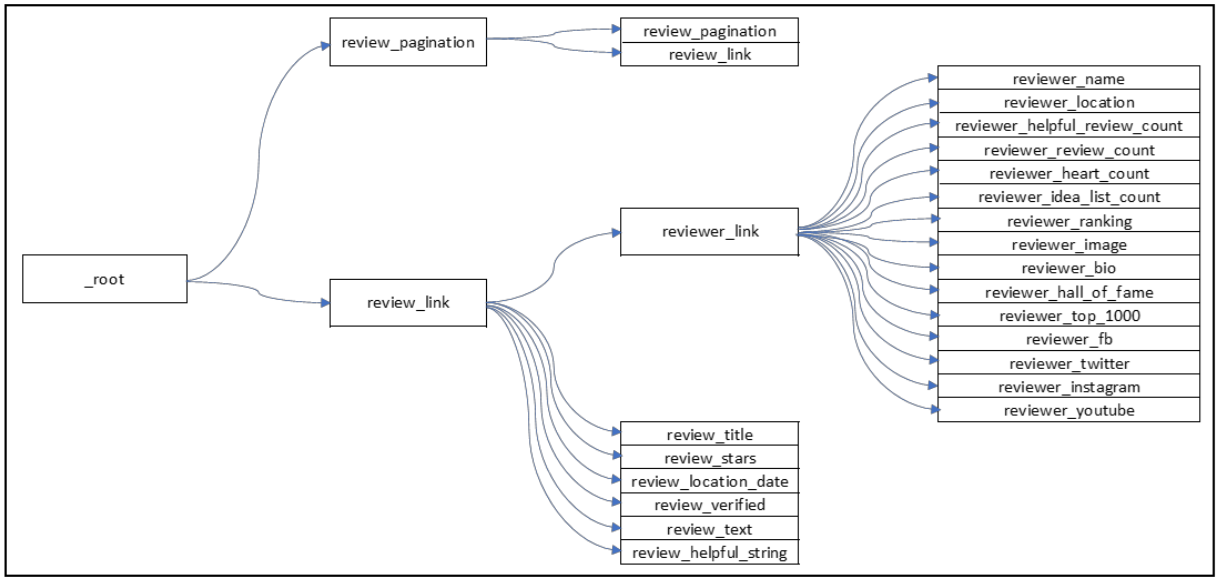
III. Financial Projection for the month of March

Microsoft Azure Estimate				
Your Estimate				
Service type	Region	Description	Estimated monthly cost	Estimated upfront cost
Storage Accounts	East US	Block Blob Storage, General Purpose V2, LRS Redundancy, Hot Access Tier, 10 GB Capacity - Pay as you go, 10,000 Write operations, 10,000 List and Create Container Operations, 10,000 Read operations, 100,000 Archive High Priority Read, 10,000 Other operations. 10 GB Data Retrieval, 1,000 GB Archive High Priority Retrieval, 10 GB Data Write	$0.32	$0.00
Azure SQL Database	East US	Single Database, vCore, RA-GRS Backup Storage, General Purpose, Provisioned, Gen 5, Local Redundancy, 50 8 vCore instance(s) x 1 Hours, 10 GB Storage, 0 GB Backup Storage	$62.04	$0.00
Azure Functions	East US	Consumption tier, 128 MB memory, 100 milliseconds execution time, 10,000 executions/mo	$0.00	$0.00
Data Factory	East US	Azure Data Factory V2 Type, Data Pipeline Service Type, Azure Integration Runtime: 10 Activity Run(s), 50 Data movement unit(s), 200 Pipeline activities, 200 Pipeline activities – External, Data Flow: 2 x 8 Compute Optimized vCores x 1 Hours, 2 x 8 General Purpose vCores x 1 Hours, 0 x 8 Memory Optimized vCores x 730 Hours, Data Factory Operations: 20 x 50,000 Read/Write operation(s), 10 x 50,000 Monitoring operation(s)	$43.50	$0.00
Key Vault	East US	Vault: 1,000 operations, 0 advanced operations, 0 renewals, 0 protected keys, 0 advanced protected keys; Managed HSM Pools: 0 Standard B1 HSM Pool(s) x 730 Hours	$0.03	$0.00
Support		Support	$0.00	$0.00
		Licensing Program	Microsoft Online Services Agreement	
		Total	$105.88	$0.00

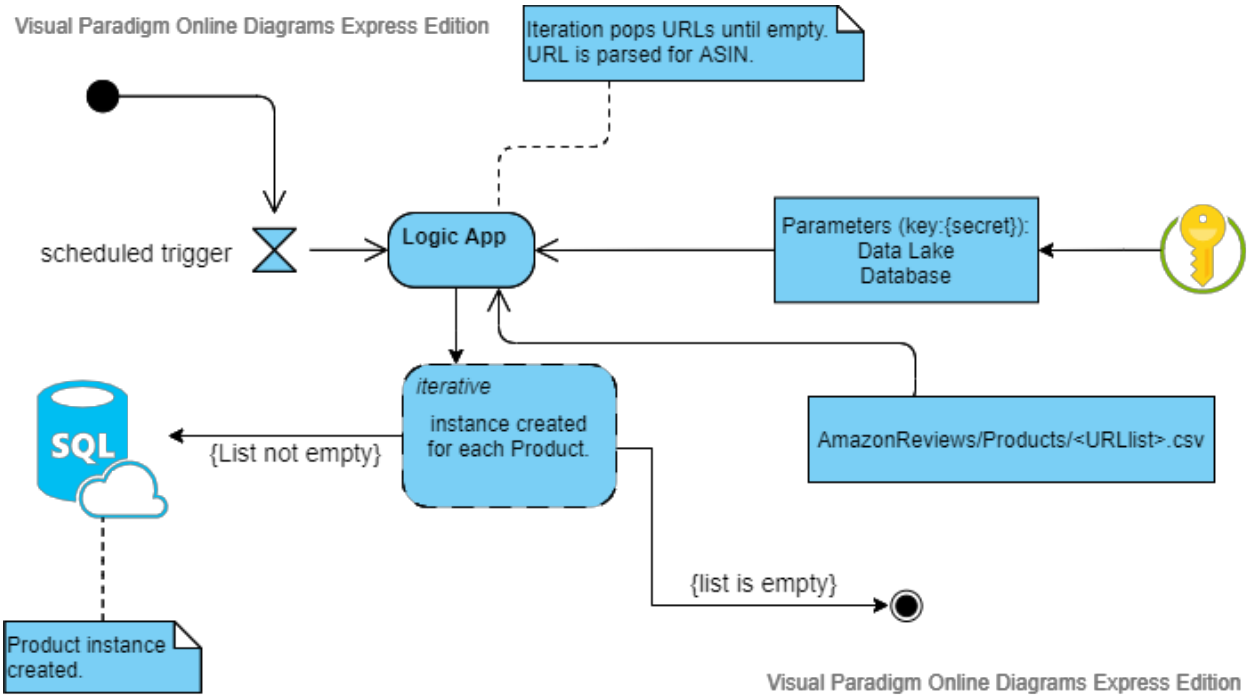
IV. Relational Data Model



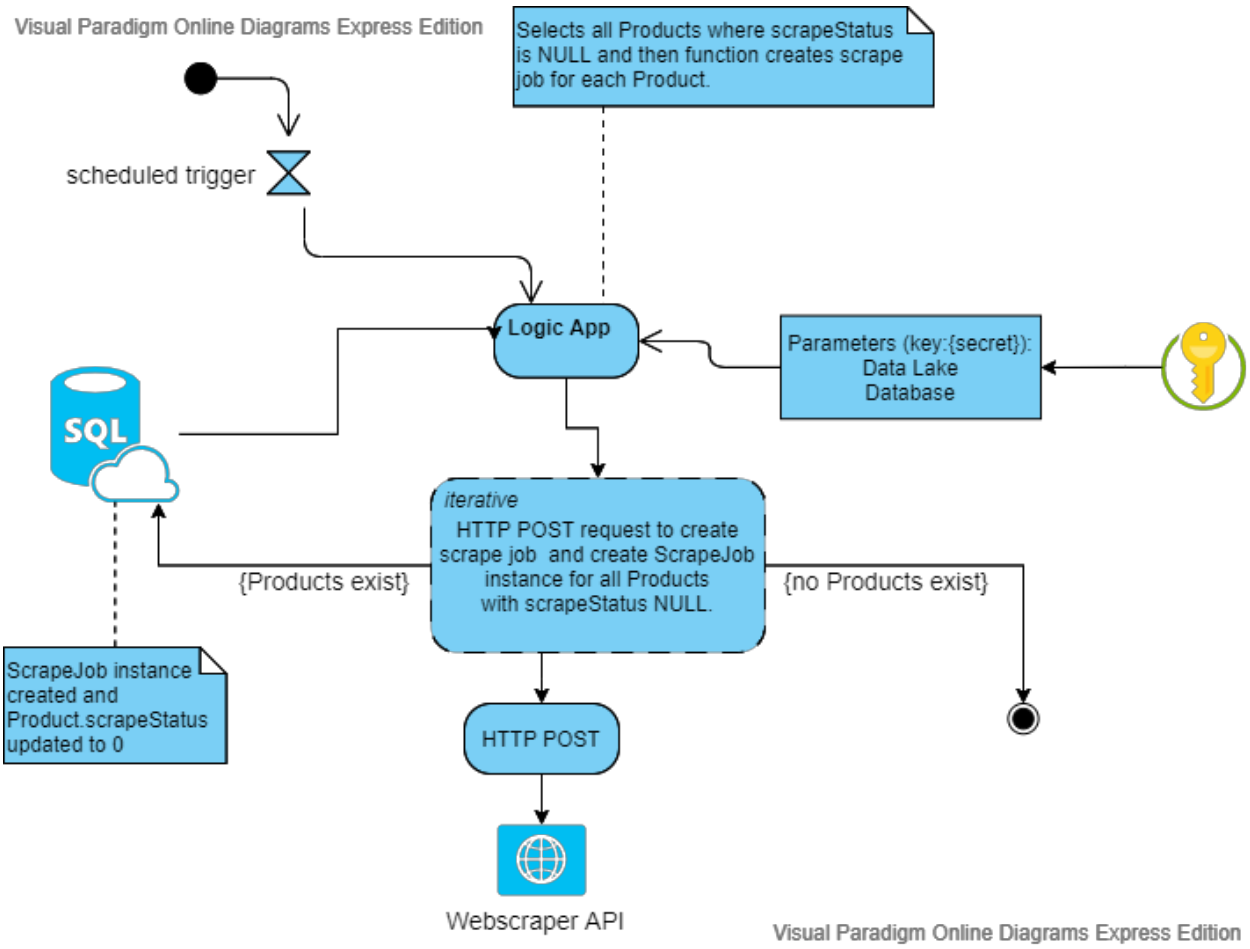
V. Sitemap tree



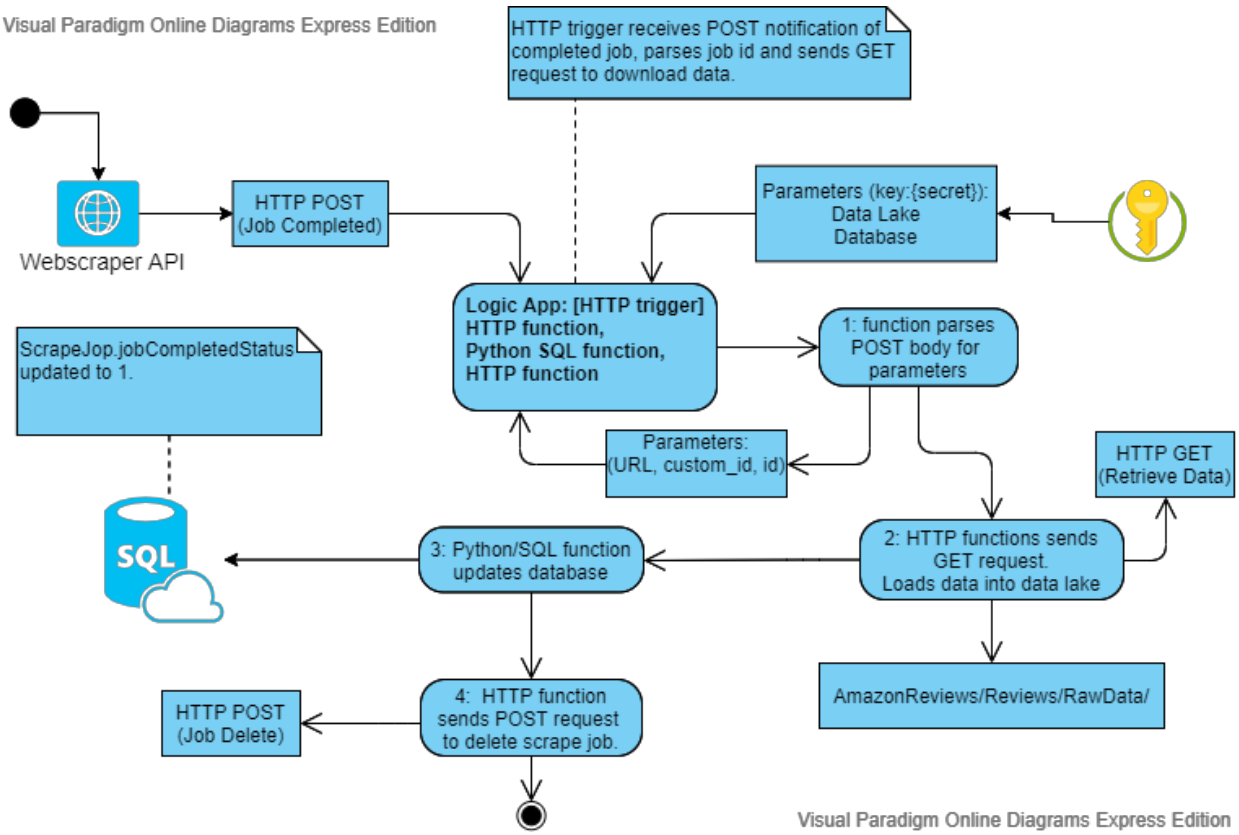
VI. Planned Product Loader Pipeline Activity Diagram



VII. Planned Data Scrape Pipeline Activity Diagram

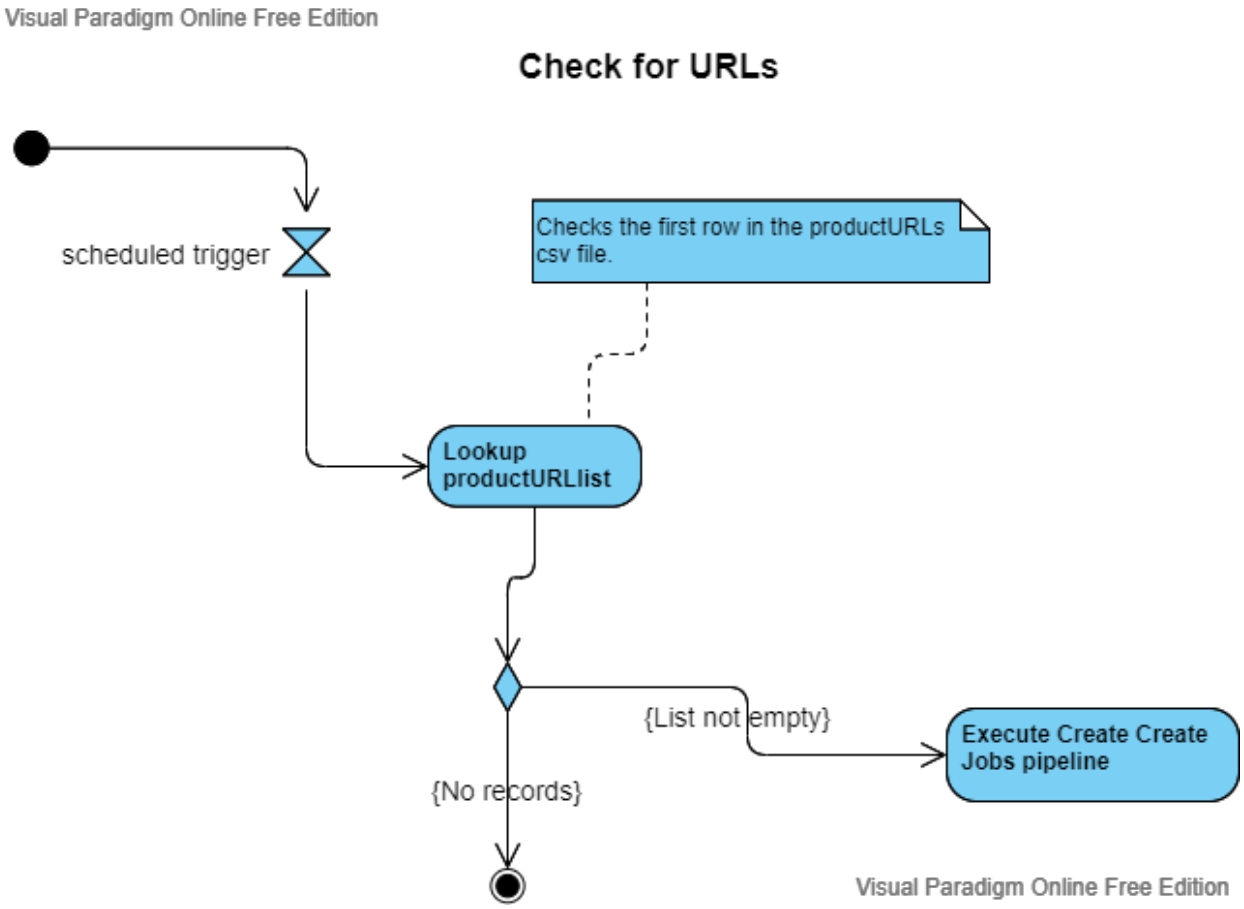


VIII. Planned Data Retrieval Pipeline Activity Diagram

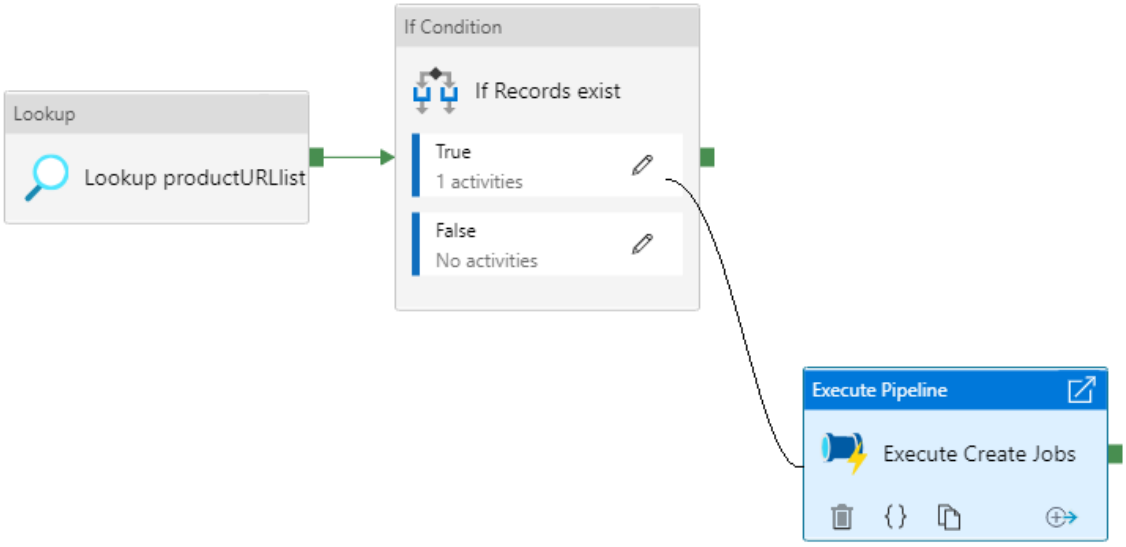


NOTE: The remainder of this appendix contains diagrams of implemented system components.

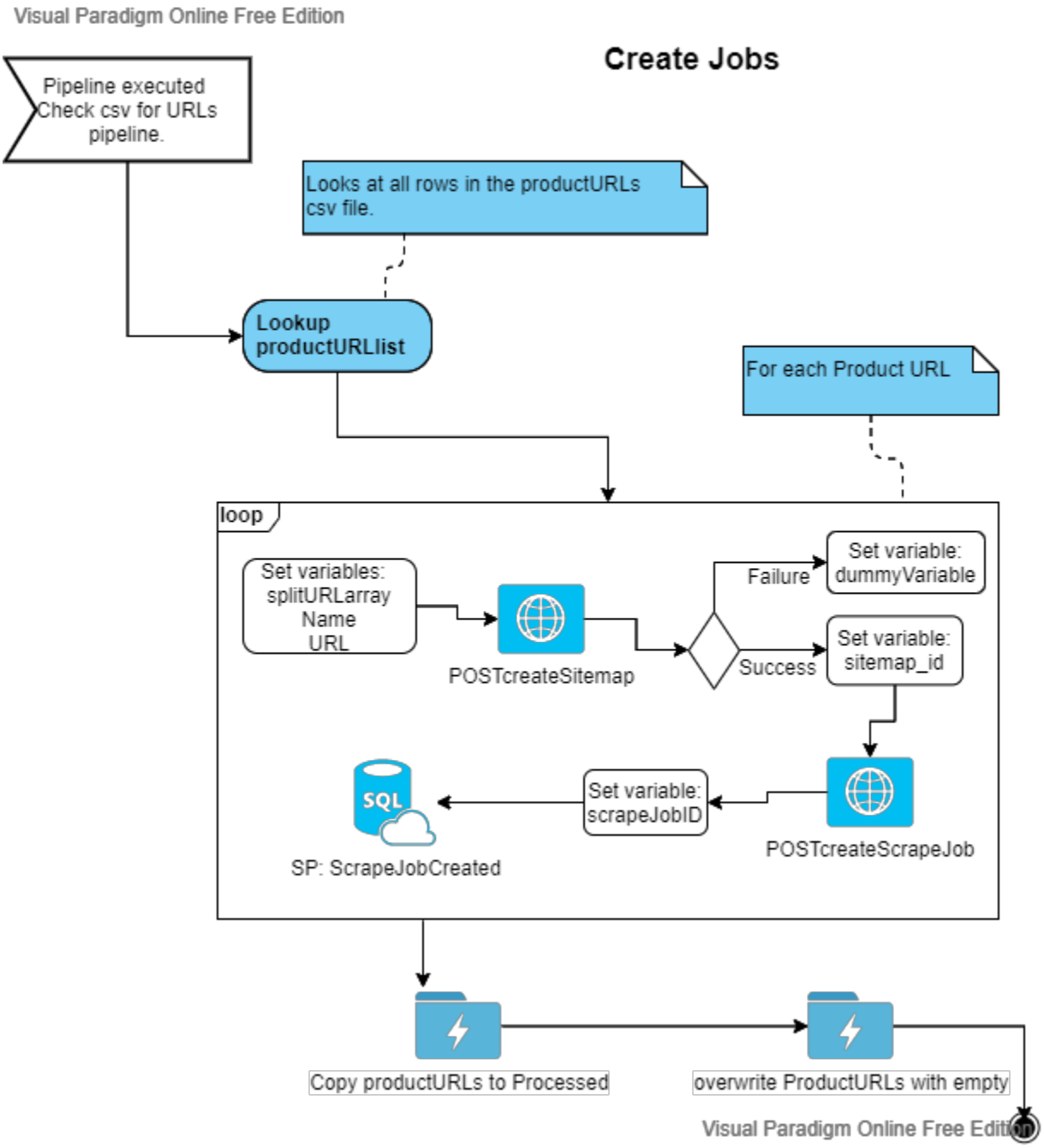
IX. Check for URLs Pipeline Activity Diagram



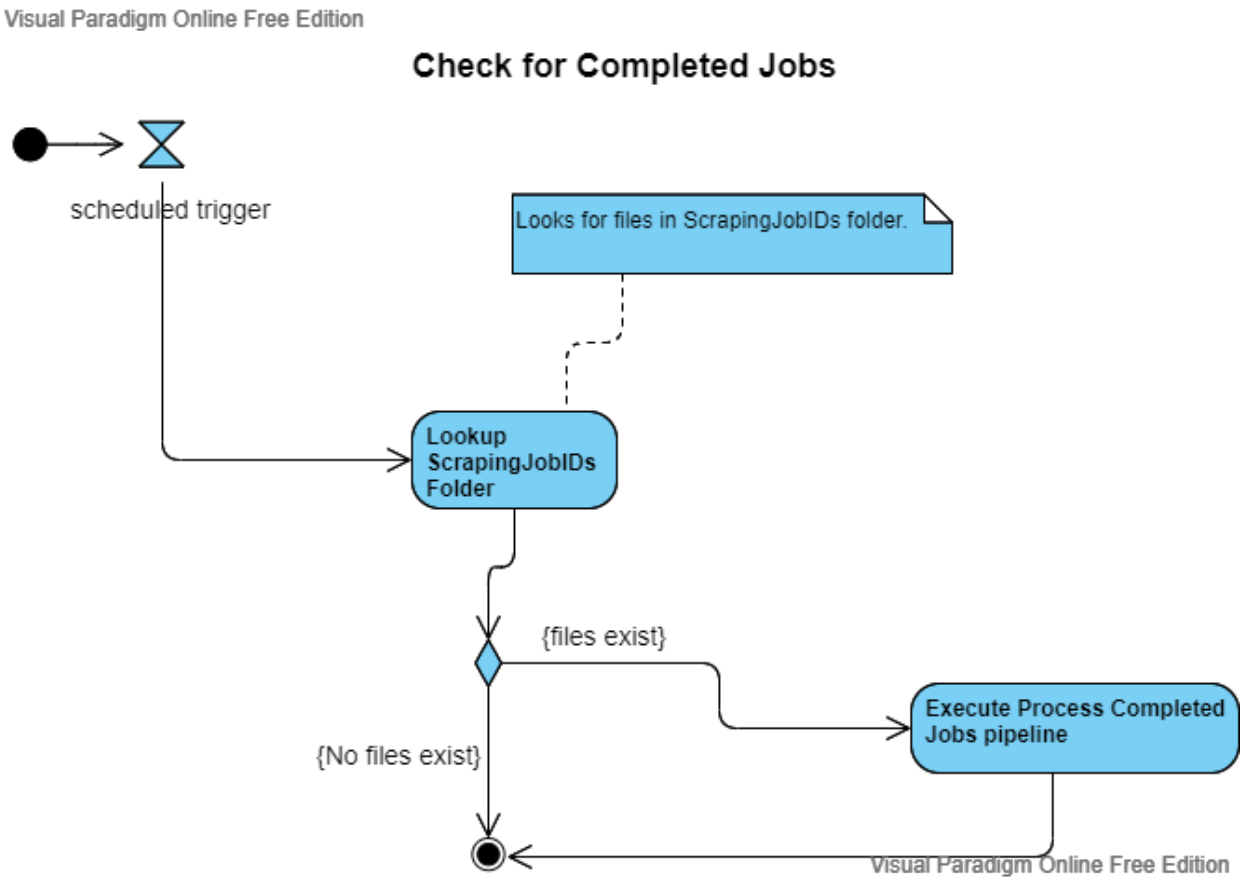
X. Check for URLs Data Factory Design View



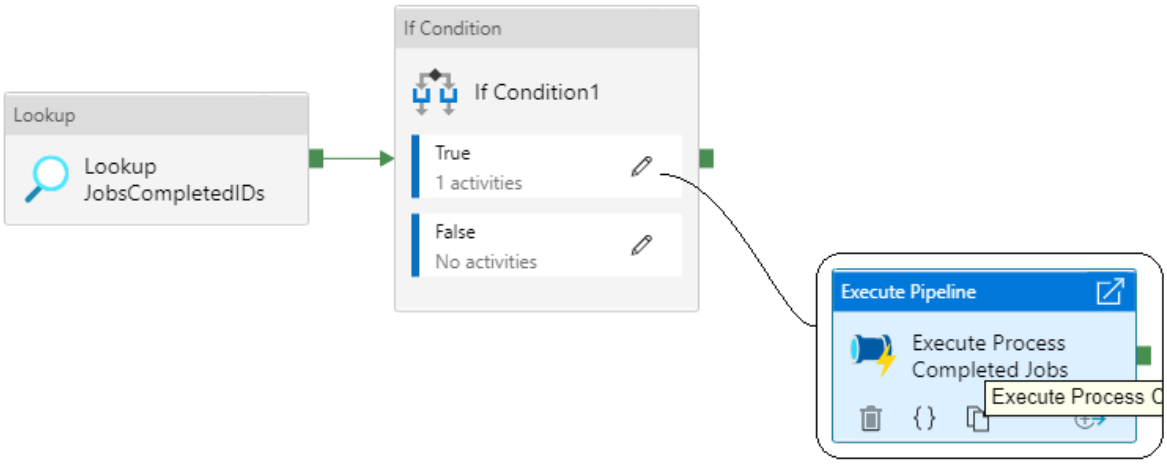
XI. Create Jobs Activity Diagram



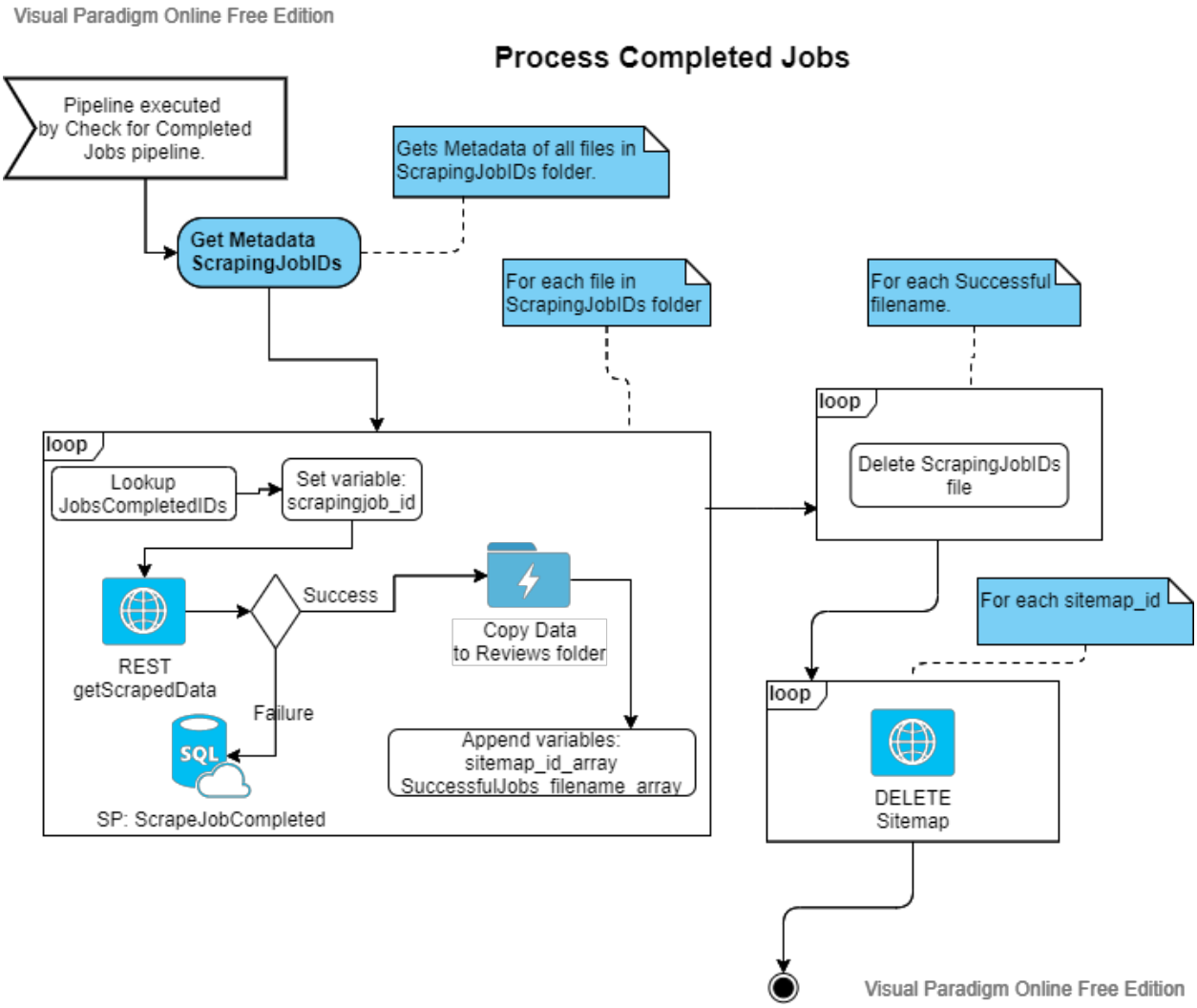
XII. Check for Completed Jobs Activity Diagram



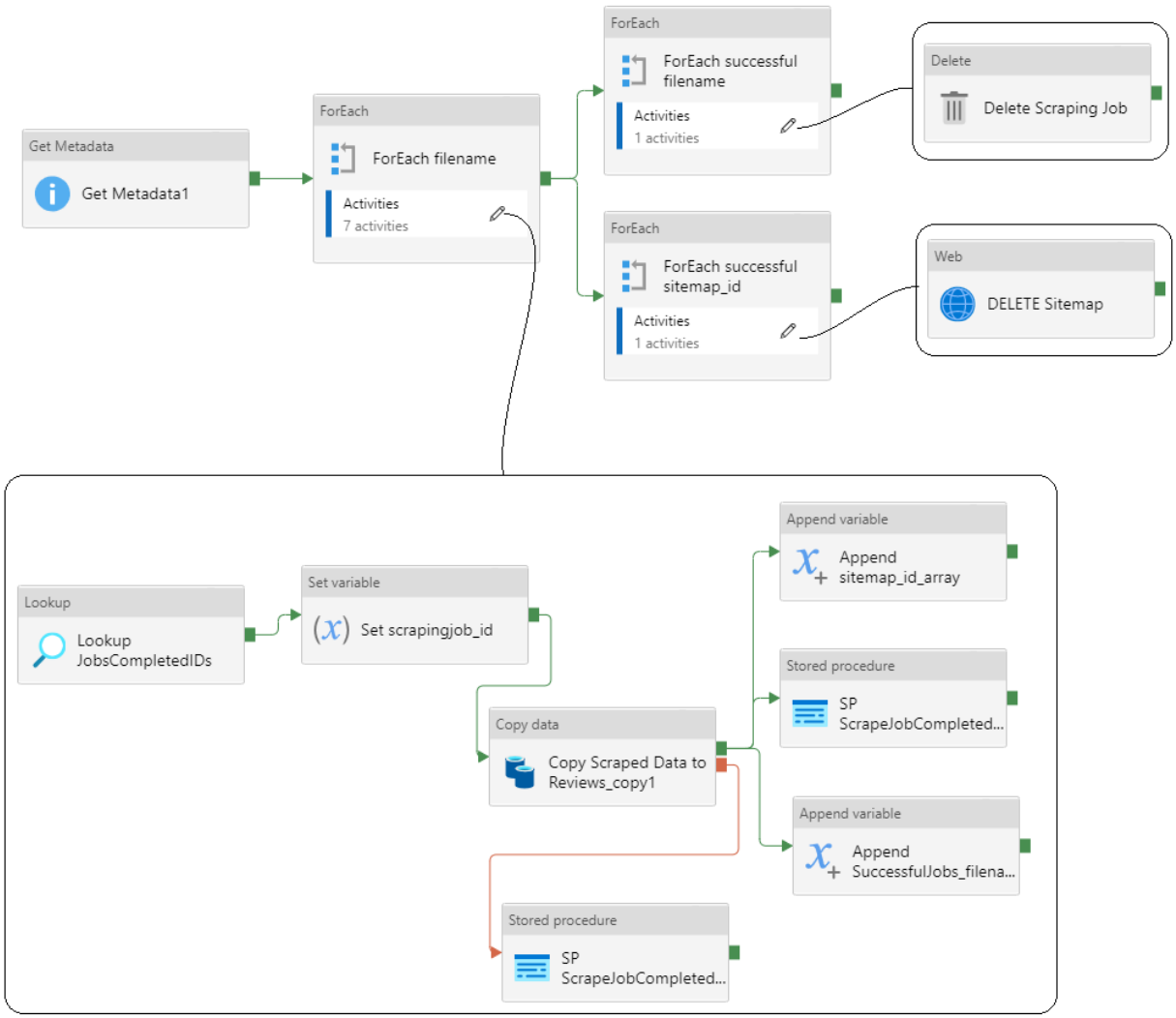
XIII. Check for Completed Jobs Data Factory Design View



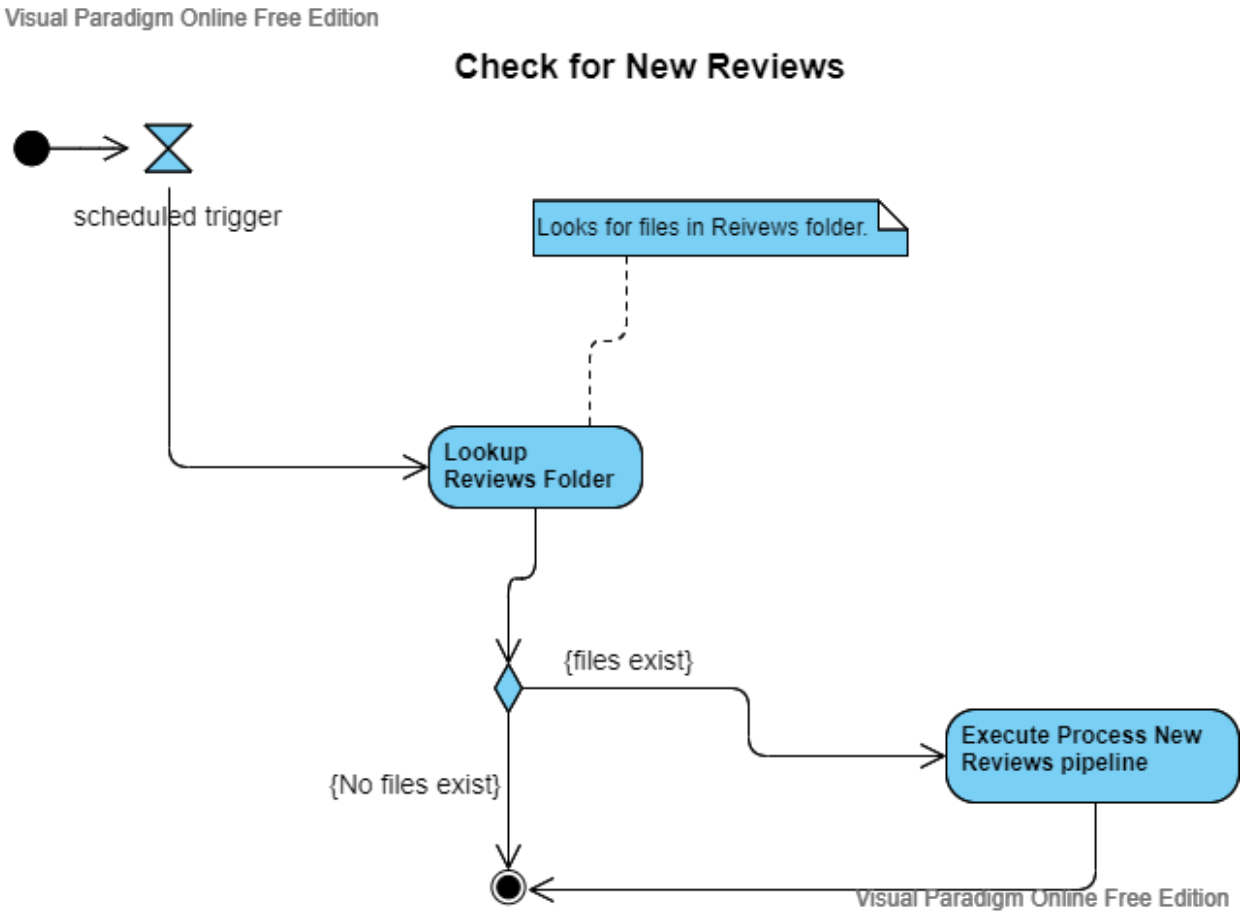
XIV. Process Completed Jobs Activity Diagram



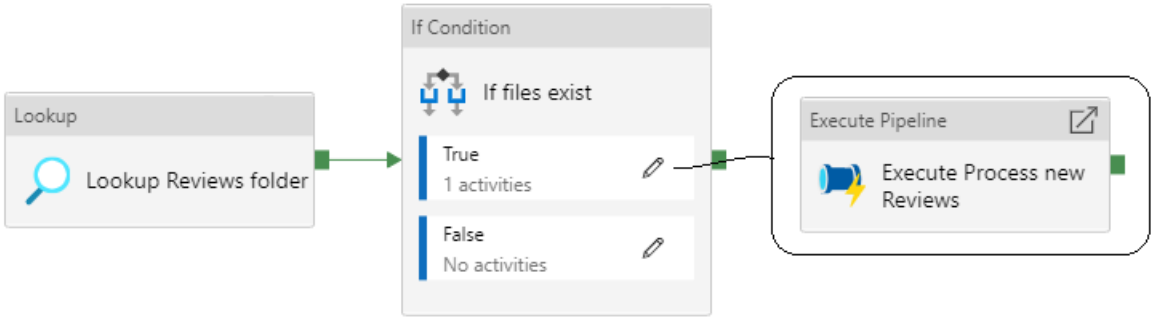
XV. Process Completed Jobs Data Factory View



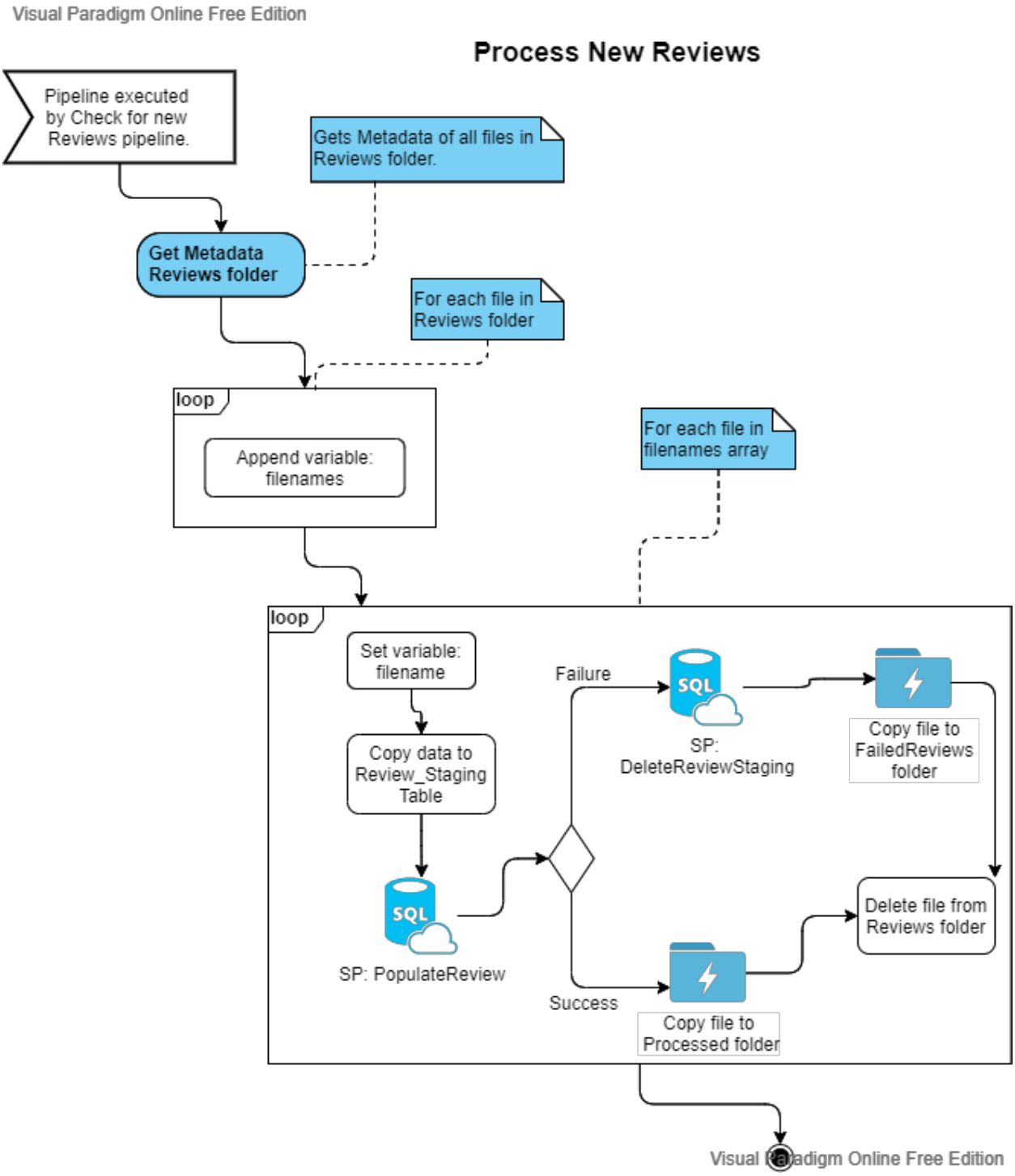
XVI. Check for New Reviews Activity Diagram



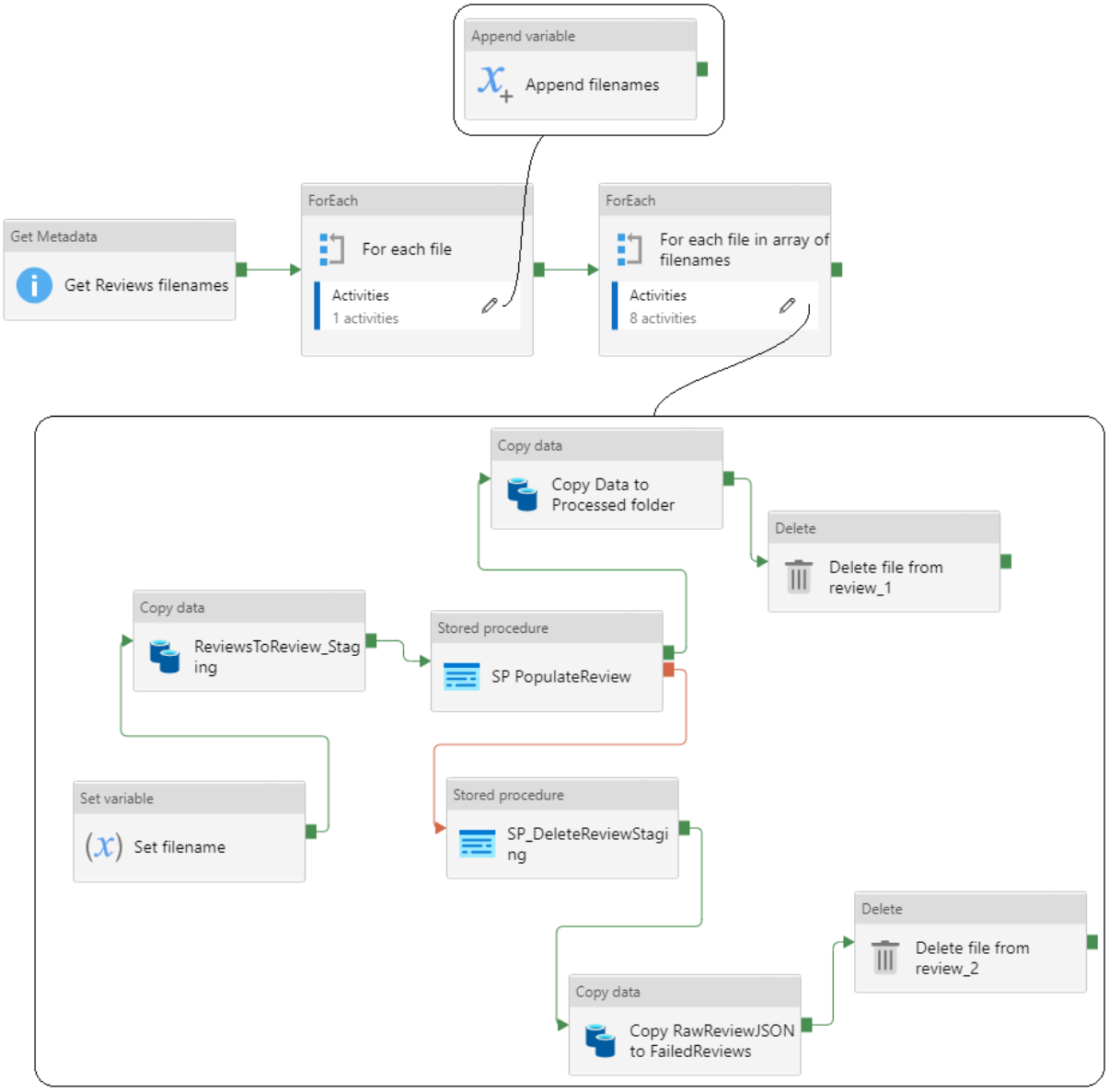
XVII. Check for New Reviews Data Factory Design View



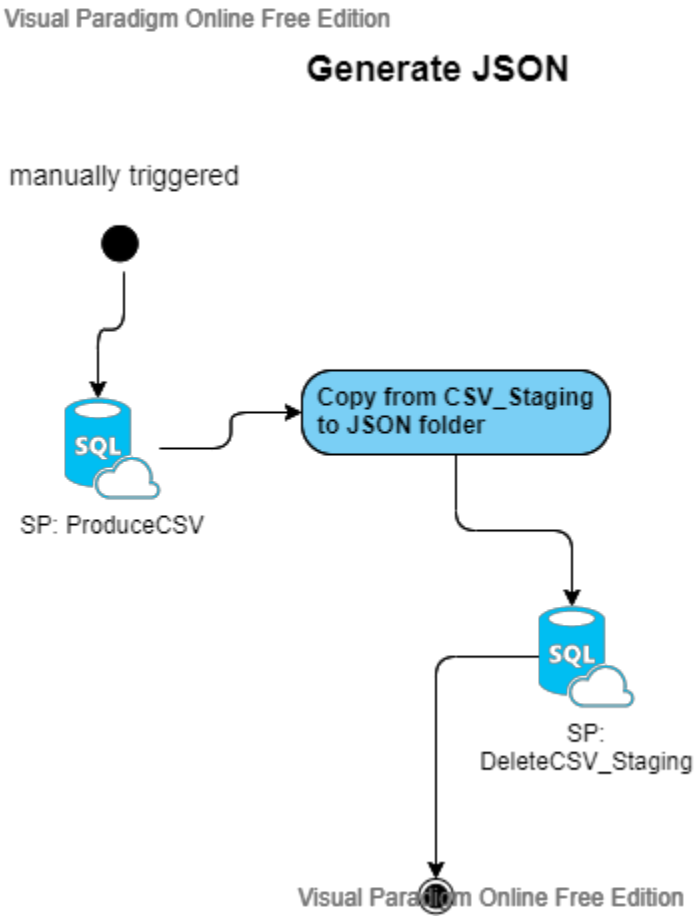
XVIII. Process New Reviews Activity Diagram



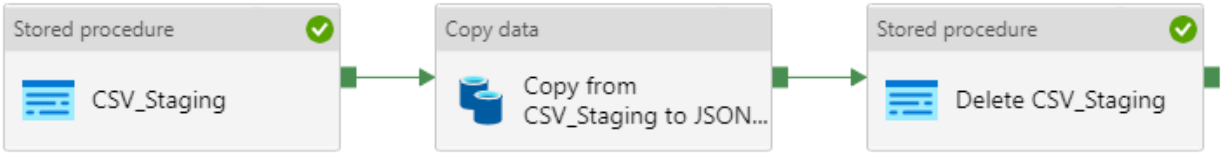
XIX. Process New Reviews Data Factory Design View



XX. Generate JSON Activity Diagram



XXI. Generate JSON Data Factory Design View



APPENDIX D

Code Snippets

I. JSON Sitemap for Webscraper Job (_id and starter will be different for each job)

```
{
    "_id": "pny_drive_reviews",
    "starter": ["https://www.amazon.com/PNY-CS900-120GB-Internal-Solid/product-reviews
       /B0722XPTL6/ref=cm_cr_dp_d_show_all_btm?ie=UTF8&reviewerType=all_reviews"],
    "selectors": [
               {"id": "review_pagination","type": "SelectorLink","parentSelectors":
       ["_root","review_pagination"],
         "selector": ".a-last a",” multiple": false,"delay": 0},
        {"id": "review_link","type": "SelectorLink","parentSelectors":
       ["_root","review_pagination"],
         "selector": "a.review-title","multiple": true,"delay": 0},
        {"id": "reviewer_link","type": "SelectorLink","parentSelectors":
["review_link"],
         "selector": "a.a-profile","multiple": false,"delay": 0},
        {"id": "reviewer_name","type": "SelectorText","parentSelectors":
["reviewer_link"],
         "selector": "span.a-size-extra-large","multiple": false,"regex": "","delay":
0},
        {"id": "reviewer_location","type": "SelectorText","parentSelectors":
["reviewer_link"],
         "selector": "span.a-size-base.a-color-base","multiple": false,"regex":
"","delay": 0},
        {"id": "reviewer_helpful_review_count","type":
"SelectorText","parentSelectors":              ["reviewer_link"],
          "selector": "div.dashboard-desktop-stat:nth-of-type(1) span.a-size-large",
          "multiple": false,"regex": "","delay": 0},
        {"id": "reviewer_review_count","type": "SelectorText","parentSelectors":
["reviewer_link"],
            "selector": "div.dashboard-desktop-stat:nth-of-type(2) span.a-size-large",
            "multiple": false,"regex": "","delay": 0},
        {"id": "reviewer_heart_count","type": "SelectorText","parentSelectors":
["reviewer_link"],
            "selector": "div.dashboard-desktop-stat:nth-of-type(3) div.dashboard-
desktop-stat-value",
            "multiple": false,"regex": "","delay": 0},
```

```
        {"id": "reviewer_idea_list_count","type": "SelectorText","parentSelectors":
["reviewer_link"],
            "selector": "div.dashboard-desktop-stat:nth-of-type(4) span.a-size-large",
            "multiple": false,"regex": "","delay": 0},
        {"id": "reviewer_ranking","type": "SelectorText","parentSelectors":
["reviewer_link"],
            "selector": ".a-column .a-row span","multiple": false,"regex": "","delay":
0},
        {"id": "review_title","type": "SelectorText","parentSelectors":
["review_link"],
            "selector": ".a-size-base.a-link-normal span","multiple": false,"regex":
"","delay": 0},
        {"id": "review_stars","type": "SelectorText","parentSelectors":
["review_link"],
            "selector": "i.a-star-2","multiple": false,"regex": """delay": 0},
        {"id": "review_location_date","type": "SelectorText","parentSelectors":
["review_link"],
            "selector": "span.review-date","multiple": false,"regex": "","delay": 0},
        {"id": "review_verified","type": "SelectorText","parentSelectors":
["review_link"],
            "selector": "span.a-size-mini","multiple": false,"regex": "","delay": 0},
        {"id": "review_text","type": "SelectorText","parentSelectors":
["review_link"],
            "selector": ".review-text span","multiple": false,"regex": "","delay": 0},
        {"id": "review_helpful_string","type": "SelectorText","parentSelectors":
["review_link"],
            "selector": "span.a-color-tertiary","multiple": false,"regex":
"","delay":0},
        {"id": "reviewer_image","type": "SelectorHTML","parentSelectors":
["reviewer_link"],
            "selector": "div.updated-profile-image-holder","multiple": false,"regex":
"","delay": 0},
        {"id": "reviewer_bio","type": "SelectorText","parentSelectors":
["reviewer_link"],
            "selector": "span.read-more-text","multiple": false,"regex": "","delay":
0},
        {"id": "reviewer_hall_of_fame","type": "SelectorText","parentSelectors":
["reviewer_link"],
            "selector": "div.a-spacing-small:nth-of-type(1) span.a-size-
mini","multiple": false,
            "regex": "","delay": 0},
        {"id": "reviewer_top_1000","type": "SelectorText","parentSelectors":
["reviewer_link"],
            "selector": "div.a-fixed-right-grid:nth-of-type(2) span","multiple":
false,
            "regex": "","delay": 0},
        {"id": "reviewer_fb","type": "SelectorText","parentSelectors":
["reviewer_link"],
```

```
            "selector": ".a-fixed-right-grid-inner div:nth-of-type(1) .a-fixed-right-
grid-col img",
            "multiple": false,"regex": "","delay": 0},
        {"id": "reviewer_twitter","type": "SelectorText","parentSelectors":
["reviewer_link"],
            "selector": ".a-fixed-right-grid-inner div:nth-of-type(2) img",
            "multiple": false,"regex": "","delay": 0},
        {"id": "reviewer_instagram","type": "SelectorText","parentSelectors":
["reviewer_link"],
            "selector": ".a-fixed-right-grid-inner div:nth-of-type(4) img",
            "multiple": false,"regex": "","delay": 0},
        {"id": "reviewer_youtube","type": "SelectorText","parentSelectors":
["reviewer_link"],
            "selector": "div:nth-of-type(5) img.social-link-image","multiple":
false,"regex": "",
            "delay": 0}
    ]
}
```

II. Example POST request to create sitemap

Method: POST

```
URL: https://api.webscraper.io/api/v1/sitemap?api_token=<YOUR API TOKEN>
```

JSON:

```
{
        <The sitemap would go here>
}
```

Response:

```
{
       "success": true,
       "data": {
              "id": 123
       }
}
```

Note: “id” in response is referred to as “sitemap_id” from this point on.

III. Example POST form to Create New Job

(uses existing sitemap_id from sitemap creation)

Method: POST

```
URL: https://api.webscraper.io/api/v1/scraping-job?api_token=<YOUR API TOKEN>
```

JSON:

```
{
```

```
	"sitemap_id": 123,
	"driver": "fast", // "fast" or "fulljs"
	"page_load_delay": 2000,
	"request_interval": 2000,
	"proxy": 0, // optional. 0 - no proxy, 1 - use proxy. Or proxy id for Scale
plan users
	"start_urls": [		// optional, if set, will overwrite sitemap start
URLs
	    <sitemap start URL goes here>
	],
	"custom_id": "custom-scraping-job-12" // optional, will be included in webhook
	             notification
}
```

Response:

```
{
	"success": true,
	"data": {
		"id": 500,
		"custom_id": "custom-scraping-job-12"
	}
}
```

Note: “id” in response is referred to as “SCRAPING JOB ID” when deleting scraping job

IV. JSON From Webscraper POST form When Job is Completed

```
"scrapingjob_id": 1234
"status": "finished"
"sitemap_id": 12
"sitemap_name": "my-sitemap"
// Optional, custom_id will be passed to post data only if set
"custom_id": "your-custom-id"
```

V. Example GET Request to Export Scraped Data in JSON format

Method: GET

```
URL: https://api.webscraper.io/api/v1/scraping-job/<SCRAPING JOB
ID>/json?api_token=<YOUR API TOKEN>
```

Response:

```
<CSV file>
```

VI. Example POST form to Delete Sitemap Job

Method: DELETE

```
URL: https://api.webscraper.io/api/v1/sitemap/<Sitemap ID>?api_token=<YOUR API TOKEN>
```

Response:

```
{
"success": true,
"data": "ok"
}
```

VII. Simple Structure of ARM Template

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-
             01/deploymentTemplate.json#",
  "contentVersion": "",
  "apiProfile": "",
  "parameters": {  },
  "variables": {  },
  "functions": [  ],
  "resources": [  ],
  "outputs": {  }
}
```

VIII. Parameters template

```
"parameters": {
  "<parameter-name>" : {
    "type" : "<type-of-parameter-value>",
    "defaultValue": "<default-value-of-parameter>",
    "allowedValues": [ "<array-of-allowed-values>" ],
    "minValue": <minimum-value-for-int>,
    "maxValue": <maximum-value-for-int>,
    "minLength": <minimum-length-for-string-or-array>,
    "maxLength": <maximum-length-for-string-or-array-parameters>,
    "metadata": {
      "description": "<description-of-the parameter>"
    }
  }
}
```

IX. Variables template

```
"variables": {
  "<variable-name>": "<variable-value>",
  "<variable-name>": {
    <variable-complex-type-value>
  },
  "<variable-object-name>": {
    "copy": [
      {
        "name": "<name-of-array-property>",
        "count": <number-of-iterations>,
        "input": <object-or-value-to-repeat>
      }
    ]
  },
  "copy": [
    {
      "name": "<variable-array-name>",
      "count": <number-of-iterations>,
      "input": <object-or-value-to-repeat>
    }
  ]
}
```

X. Variables template

```
"functions": [
  {
    "namespace": "<namespace-for-functions>",
    "members": {
      "<function-name>": {
        "parameters": [
          {
            "name": "<parameter-name>",
            "type": "<type-of-parameter-value>"
          }
        ],
        "output": {
          "type": "<type-of-output-value>",
          "value": "<function-return-value>"
        }
      }
    }
  }
],
```

XI. Resources template

```
"resources": [
  {
      "condition": "<true-to-deploy-this-resource>",
      "type": "<resource-provider-namespace/resource-type-name>",
      "apiVersion": "<api-version-of-resource>",
      "name": "<name-of-the-resource>",
      "comments": "<your-reference-notes>",
      "location": "<location-of-resource>",
      "dependsOn": [
          "<array-of-related-resource-names>"
      ],
      "tags": {
          "<tag-name1>": "<tag-value1>",
          "<tag-name2>": "<tag-value2>"
      },
      "sku": {
          "name": "<sku-name>",
          "tier": "<sku-tier>",
          "size": "<sku-size>",
          "family": "<sku-family>",
          "capacity": <sku-capacity>
      },
      "kind": "<type-of-resource>",
      "copy": {
          "name": "<name-of-copy-loop>",
          "count": <number-of-iterations>,
          "mode": "<serial-or-parallel>",
          "batchSize": <number-to-deploy-serially>
      },
      "plan": {
          "name": "<plan-name>",
          "promotionCode": "<plan-promotion-code>",
          "publisher": "<plan-publisher>",
          "product": "<plan-product>",
          "version": "<plan-version>"
      },
      "properties": {
          "<settings-for-the-resource>",
          "copy": [
              {
                  "name": ,
                  "count": ,
                  "input": {}
              }
          ]
      },
```

```
      "resources": [
          "<array-of-child-resources>"
      ]
  }
]
```

XII. Functions Template

```
"functions": [
  {
    "namespace": "<namespace-for-functions>",
    "members": {
      "<function-name>": {
        "parameters": [
          {
            "name": "<parameter-name>",
            "type": "<type-of-parameter-value>"
          }
        ],
        "output": {
          "type": "<type-of-output-value>",
          "value": "<function-return-value>"
        }
      }
    }
  }
],
```

XIII. Outputs Template

```
"outputs": {
  "<output-name>": {
    "condition": "<boolean-value-whether-to-output-value>",
    "type": "<type-of-output-value>",
    "value": "<output-value-expression>",
    "copy": {
      "count": <number-of-iterations>,
      "input": <values-for-the-variable>
    }
  }
}
```

XIV. Python boilerplate for function chaining

```
import azure.functions as func
import azure.durable_functions as df
```

```
def orchestrator_function(context: df.DurableOrchestrationContext):
    x = yield context.call_activity("F1", None)
    y = yield context.call_activity("F2", x)
    z = yield context.call_activity("F3", y)
    result = yield context.call_activity("F4", z)
    return result

main = df.Orchestrator.create(orchestrator_function)
```

XV. Power Query M Basic Structures

let expression

```
let
   Variablename = expression,
   #"Variable name" = expression2
in
   Variablename
```

if expression

```
if <condition> then
    <output if true>
else
    <output if false>
```

XVI. Jobs Completed Notification function app

function.json [Declares the bindings used in the function]

```
{
    "scriptFile": "__init__.py",
    "bindings": [
        {
            "authLevel": "anonymous",
            "type": "httpTrigger",
            "direction": "in",
            "name": "req",
            "methods": [
                "get",
                "post"
            ]
        },
        {
            "type": "blob",
            "direction": "out",
            "name": "outputblob",
            "path": "amazonreviewsdl/ScrapingJobIDs/{DateTime}.csv",
```

```
            "connection": "AzureWebJobsStorage"
        },
        {
            "type": "http",
            "direction": "out",
            "name": "$return"
        }
    ]
}
```

__init__.py [function main]

```
import logging
import azure.functions as func

def main(req: func.HttpRequest, outputblob: func.Out[str]) ->  func.HttpResponse:
    logging.info('Python HTTP jobCompletedNotification function processed a request.')

    req_body = req.get_body().decode('UTF-8')
    bodyList = req_body.split("&")

    scrapeParam = bodyList[0].split("=")
    scrapingjob_id = scrapeParam[1]

    siteParam = bodyList[2].split("=")
    sitemap_id = siteParam[1]

    logging.info("Scraping job id: " + scrapingjob_id)

    outputblob.set(f"{scrapingjob_id},{sitemap_id}")

    return func.HttpResponse(status_code=200)
```