JOURNAL OF INFORMATION SYSTEMS APPLIED RESEARCH

Volume 15, Issue 3
October 2022
ISSN: 1946-1836

Special Issue: Data and Business Analytics

In this issue:

iscap
Information Systems & Computing Academic Professionals

The **Journal of Information Systems Applied Research** (JISAR) is a double-blind peer reviewed academic journal published by ISCAP, Information Systems and Computing Academic Professionals. Publishing frequency is three to four issues a year. The first date of publication was December 1, 2008.

JISAR is published online (https://jisar.org) in connection with CONISAR, the Conference on Information Systems Applied Research, which is also double-blind peer reviewed. Our sister publication, the Proceedings of CONISAR, features all papers, panels, workshops, and presentations from the conference. (https://conisar.org)

The journal acceptance review process involves a minimum of three double-blind peer reviews, where both the reviewer is not aware of the identities of the authors and the authors are not aware of the identities of the reviewers. The initial reviews happen before the conference. At that point papers are divided into award papers (top 15%), other journal papers (top 30%), unsettled papers, and non-journal papers. The unsettled papers are subjected to a second round of blind peer review to establish whether they will be accepted to the journal or not. Those papers that are deemed of sufficient quality are accepted for publication in the JISAR journal. Currently the target acceptance rate for the journal is under 38%.

Questions should be addressed to the editor at editor@jisar.org or the publisher at publisher@jisar.org. Special thanks to members of ISCAP who perform the editorial and review processes for JISAR.

JOURNAL OF INFORMATION SYSTEMS APPLIED RESEARCH

Editors

2022 JISAR Editorial Board

Technical Implementation Case

A Cloud-based System for Scraping Data From Amazon Product Reviews at Scale

Ryan Woodall
Jrw5074@uncw.edu

Douglas Kline
klined@uncw.edu

Ron Vetter
vetterr@uncw.edu
Department of Computer Science

Minoo Modaresnezhad
modarsm@uncw.edu

Congdon School of Supply Chain, Business Analytics, and Information Systems
University of North Carolina Wilmington
Wilmington, NC 28403

Abstract

Amazon product reviews can provide a rich source of data for natural language processing research. We built a custom cloud-based system to support a related research project for obtaining Amazon product reviews. A third-party cloud-based scraping service automatically retrieved scraping jobs, then notified Azure Data Factory through an Azure Function. Raw scraping data was then transferred in batches to Azure Data Lake Storage, then custom SQL transformed the data for convenient queries in an Azure SQL database. The system obtained 17,962 product reviews and produced data sets in several formats. This paper fully describes the system and offers lessons learned from the experience.

Keywords: Data Pipeline, Cloud, Amazon Reviews, Big Data, Azure.

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. These systems need to be maintainable, adaptable, repeatable, and scalable. To explore modern cloud-based data-oriented system development, we created a system to gather large numbers of Amazon product reviews.

Amazon reviews are important for researchers exploring natural language processing. Each product has a distinctive dictionary, i.e., the words used in reviews change for each product category and each product. Reviews offer the ability for researchers to evaluate methods across

challenging situations such as dictionary, data quantity, data quality, themes, sentiment, etc.

Several Amazon Review data sets are freely available online. The data set used for McAuley, et al. (2015) and He & McAuley (2016) was primarily gathered for research relating the review text to product images. This data set is quite large, covering many categories and products. However, it is somewhat dated (May 1996 - July 2014) and does not include some of the review quality features implemented by Amazon, e.g., verified purchases. A subsequent dataset was collected by Ni, et al. (2019), providing more recent reviews (through 2018), but still lacks the newer review quality indicators and details about the reviewer that can filter fake reviews.

Amazon (2019) offers its own large review data set, which includes whether the review was tied to a verified purchase. This still leaves out many data items that can be used to evaluate a review's quality and reliability, such as reviewer name, reviewer rating, reviewer social media names, etc.

Web scraping of publicly accessible web content has been contested in recent years. The data analytics company HiQ filed a complaint against LinkedIn's practice of restricting access to users' profiles (Katrix & Schaul 2019). HiQ argued that the practice was anti-competitive and violated state and federal laws. LinkedIn argued that web scraping violates the Computer Fraud and Abuse Act (CFAA), which prohibits accessing a protected computer without authorization. The court favored HiQ, stating that users, rather than LinkedIn, held the copyright of their own data and that users clearly intended for their data to be publicly accessible. Furthermore, giving companies exclusive rights to users' data would create "information monopolies" that harm the public interest. This ruling was upheld in 2022 (Tse & Brian 2022).

A few for-pay services exist, providing functionality similar to that created in this project. However, the actual data pipeline is opaque, with no visibility of the transformations occurring to the data. Amazon itself has recently begun offering a for-pay API for accessing their data. However, API access tokens are given only to developers from vetted companies, and we did not explore this avenue.

Amazon reviews can be used for many natural language processing (NLP) research purposes, such as sentiment analysis, bot detection, theme analysis, summarization, and recommender systems. Furthermore, reviews are the primary method for consumers to evaluate products for purchase.

For a related research project (Gokce, et al., 2021) we decided to collect a custom data set that would be more current and include the missing items identified above. We used this opportunity to review modern methods for the large-scale collection of data in the cloud. Rather than filtering the existing massive data sets for the needed data, we would create smaller targeted sets for our tasks. Another concern we had was that the actual processing steps performed on the raw reviews are unclear and perhaps not repeatable. Ni, et al. (2019) offers their data in several forms with different levels of "aggressive" removal of reviews for various reasons. We took this opportunity to curate our own data set in a fully auditable, repeatable manner, where every data modification was explicitly described by code.

To fully explore modern data pipeline methods, we established the following goals:

- Cloud-based – the system should entirely reside in the cloud
- Automated – the system should orchestrate tasks without the in-progress intervention
- Scalable – the system should be scalable
- Formats – the system should offer the data set(s) in several formats

2. TECHNOLOGIES

In this section, we describe the technologies that we used and why they were chosen. Many vendors offer cloud-based services. To limit the overwhelming options and align with our existing technology skillsets, we used Azure cloud services as much as possible. Generally, equivalent services exist on most cloud provider platforms.

We used a third-party cloud-based web scraping service called WebScraper (2020) because of its convenient low-code nature and our prior experience with its desktop-product. Scrape job definitions can be authored in the Chrome web browser and exported in JSON format. Scrape jobs can traverse paginated web pages, drill-down and up through pages, and gather related data entities such as products, reviews, reviewers, etc. Bot-detection-avoidance features are included, such as pauses between page

requests and the use of multiple IP addresses for requests. In the cloud-based service, a full API is provided for the programmatic definition of jobs.

Azure Data Lake Storage Gen2 was used to store raw files and also as the storage area for the Azure SQL instance. This service offers a hierarchical namespace, high scalability, metered fees, and can support map-reduce style operations.

Azure Data Factory (ADF) integrates seamlessly with Data Lake Storage and can be used to orchestrate data workflows among various locations and services in the cloud. Our initial intent was to use ADF for the bulk of the data operations, but ultimately, it was mainly used to move data from one location to another.

An Azure SQL relational database instance was used to deliver final data in an easily query-able format. In addition, relational models offer a quite compact representation of the data, reducing storage costs, and improving performance.

Azure Functions were used to trigger events across distributed components of the system. With Azure Functions, a secure HTTPS endpoint could be called with proper credentials, triggering events in a remote system and/or passing data between systems.

We planned to use Azure Key Vault to manage the secrets needed for secure communication between distributed system components. Secrets we planned to keep in the Azure Key Vault included connection strings, credentials (username/password), and API tokens. Ultimately, Azure Key Vault proved unwieldy for our relatively small project and required a full Azure Active Directory Domain and high-level domain credentials. For expedience without sacrificing security, secrets were stored in each linked service defined in Azure Data Factory.

The many cloud services were declaratively defined in Azure Resource Manager (ARM) templates. ARM templates define the desired end state of a collection of services and manage the connections and security concerns among the services. ARM templates are text-based and declarative. With a system's ARM template, a complete replica of a complex distributed set of cloud services can be perfectly replicated. Furthermore, the entire system can be versioned in source control.

Even with the cloud-based services used, there were places where programming code was required. Transact SQL, Microsoft's procedural scripting language, was used to transform data from a staging table to relational tables in the database. Python code was used in the Azure function.

3. SYSTEM DESCRIPTION

This section demonstrates the system as it was ultimately built. The System Overview diagram in Appendix A gives a high-level view of the major components. The Webscraper component was the only non-Azure part. All other components are Azure Data Factory workflows and were housed in a single Azure Resource Group from which an ARM template could be generated, completely defining all components and their interactions.

The system operated as a set of Azure Data Factory workflows:

- Start Jobs (periodic)
- Scrape Data (external)
- Record Completion (episodic)
- Retrieve Reviews (periodic)
- Create Tabular Data (periodic)
- Create Flat Data (periodic)

Each of the above workflows is decoupled from the others so that work is "pulled" through the system rather than "pushed." The workflows marked as periodic are implemented via timers. Each periodic workflow wakes up at defined intervals and completes any waiting work. Waiting work is recorded in the ScrapeJob and ScrapeJob_Status tables (see the relational schema in Appendix B). The only episodic workflow, Record Completion, is implemented as an Azure Function. This Azure Function is called by the external Webscaper service to indicate that a scraping job has been completed. Azure Data Factory was mainly used to move data, record job states, and orchestrate the process.

The Start Jobs workflow looks for new product review scraping jobs and calls the external scraping service to start them. New scrape jobs are entered (by humans or otherwise) in the ProductURLs.csv file in Azure Storage. The URL of a product defines a scrape job on Amazon. The same scrape definition is used on all products and can be found in Appendix C. The Start Jobs component is implemented as a periodic (every 15 minutes) azure data factory pipeline. A POST is made to the WebScraper service API for every URL in the file using secure credentials. Job-status information is stored in the Azure SQL tables and various files in Azure Storage. A single record is inserted into the Product table. Appendix D shows the actual ADF pipeline used for this

component. This is a good example of a simple ADF workflow. Similar simple workflows were used throughout the system.

The Scrape Data operation is performed over time by the WebScraper third-party service. When it is complete, the WebScraper service POSTs to the Record Completion, which is implemented as an Azure Function, is exposed as an HTTP endpoint. This message does not transfer the data but merely indicates that the scrape job has been completed. The Azure Function is implemented in Python and simply records the completion of the task in the SQL ScrapeJob_Status table in the SQL database.

The Retrieve Reviews component is implemented as a periodic azure data factory pipeline similar to the Start Jobs component. Each completed scrape job makes an API call to the WebScraper API for the resulting data and metadata and moves the data to Azure Storage. A single product produces a single scrape job, which produces a single set of reviews in a single file. This component places a potentially large JSON file in Azure Storage and creates related records in the ScrapeJob_status table.

The Create Tabular Data takes as input the JSON file from Azure Storage and inserts many records into the Review and Reviewer tables. The ADF activity diagram can be seen in Appendix E. This component moves a file of reviews into a staging table in the SQL database. The data is inserted into the staging table via an efficient bulk-load operation with no integrity checking. The data is unmodified from the WebScraper component and enters the staging table with all character-based data types. A stored procedure written in Transact-SQL transforms the data and inserts it into the Product, Review, and Reviewer relational tables. Notably, the star-rating is transformed from prose ("one star out of five") to an integer data type.

The Create Tabular Data component was initially implemented using Azure Data Factory. We found this to be slow, inefficient, and costly. The ADF processes appear to be implemented as record-at-a-time operations. ADF was still used to orchestrate the component, with stored procedures used strategically where efficient set-at-a-time operations were possible.

Finally, the Create Flat Data component executes straightforward SQL queries to produce a CSV flat file, as well as a hierarchical JSON file. This is for the convenience of users. Data Analytics professionals normally consume data in these formats. Because the data is in a relational schema with a database, any subset of the data can be readily produced in any format using straightforward SQL.

4. IMPLEMENTATION

In this section, we describe what is happening in the source code. URLs for Amazon product review pages are stored in a CSV file in the data lake. This file is regularly checked for new products from within the data factory. Appendix D shows a snapshot of the implementation in the data factory. Each box represents an activity from a menu of activities. Pipelines are built by simply clicking the activity and dragging it into the workspace. The pipeline that checks for new product URLs utilizes lookup, if condition, and execute pipeline activities. For data factory activities to be able to connect with other resources, linked services must be created. These are created in the data factory by clicking on the linked service and selecting to resource to be connected. This project only requires the creation of three linked services. Two are for the data lake and database. The third linked service is different in that it links to cloud storage outside of Azure. This type of linked service is called a REST service, where a GET request is configured to retrieve data from the WebScraper service API.

Appendix E shows a snapshot of the most involved pipeline responsible for taking the raw data uploaded to the data lake and copying it into a staging table in the database. Then a stored procedure activity is called that performs any necessary transformations while inserting the data into the relational model. The stored procedure can be found in Appendix I, and a relational model diagram is in Appendix B. Each box represents a type of activity selected from the data factory activities menu. Dragging the activity into the workspace and double-clicking it will open a wizard for configurations. For example, configurations for a copy data activity would involve selecting the linked service for the source and the target (these may use the same linked service). The datasets for the source and sink will also need to be selected as well. See Appendix H for an example of a copy activity that copies data from the data lake to the staging folder in the database.

The python code for the azure function used for the HTTP endpoint is in Appendix F. The purpose of the endpoint is to receive a POST notification from the Webscraper service API once a scraping job has been completed. The function receives

and parses the request body for the scrapingjob_id. The event log is generated, and a new blob is created in the data lake containing the scrapingjob_id. The id is later used to retrieve and store the scraped data. Three bindings are configured in a function.json file. These bindings are httpTrigger, blob, and http. The settings for each binding are in Appendix G.

5. RESULTS & DISCUSSION

To test the system, reviews for 15 products across nine subcategories in the Audio Books & Originals category were collected. The subcategories were:

- Bios & Memoires
- Self-Development
- Literature & Fiction
- Business & Careers
- Science Fiction & Fantasy
- Teen & Young Adult
- Health & Wellness
- Computers & Technology
- Kids

In total 17,962 reviews were collected and processed through the pipeline, producing data in CSV and JSON formats, as well as recording all entity instances in the relational database. Each form of the data was retained, providing a full audit-trail of all changes to the data.

The Webscraper service made requests in three parallel threads emanating from separate IP addresses, with adjustable delays between page requests. The number of reviews per product ranged from 293 to 2690, and scrape times per product ranged from 1.5 to 9.75 hours, with a combined scraping time of 67.5 hours. The number of records scraped per hour averages approximately 266.

In general, we accomplished the goals set out at the beginning of the project. The system can produce large amounts of Amazon reviews in a variety of formats in an automated manner.

By creating the system, we learned much and will relate some opinions and advice for developers implementing similar systems with these technologies.

Azure Data Factory & Azure Functions

It became clear that Azure Data Factory (ADF) and Azure Functions were not fully mature products. In some cases, the products were changed during the development of this system. Specifically, overnight changes in features broke the system multiple times during development, requiring rewrites of code. Second, integrations with products and languages were not complete. For example, when using Python to write Azure Functions, there was no direct access to Azure SQL, while there was direct access to CosmosDB.

We found Azure Data Factory jobs difficult and costly to debug, as well as expensive and inefficient. Processing appears to be row-by-agonizing-row, which was unnecessary in our situation. Ultimately, some parts we intended to write in Azure Data Factory were implemented as set-at-a-time operations in SQL. We found this to be more transparent, easier to write and debug, and much faster than the ADF pipelines.

Third-Party Cost

We used the cloud-based WebScraper service for its low-code approach and our familiarity with the desktop product. However, this service was the majority of the cost for producing reviews. The cost structure does not scale to the level we want. This is not critical of the service; it worked well, as advertised, and required minimal effort.

We anticipate custom coding the review collection component to continue collecting reviews at scale with reasonable costs. There are several libraries available, including python libraries Beautiful Soup (2021) and Scrapy (2021). We would not have to write a general-purpose full-featured cloud-based web scraping tool with all features of WebScraper, but could write code specific to our needs. Implementing a custom scraper in one or more virtual machines or containers or even in an Azure Function appears feasible. A less elegant approach would involve running the desktop browser-based free version of WebScraper on virtual desktops and collecting results as they are produced.

Cloud-based Considerations

The cloud-based distributed architecture comes with benefits and challenges. A distributed architecture necessitates decoupling and defining explicit interfaces between components, which generally produces a very clear transparent system. However, secure communication becomes onerous compared to a monolithic application. In systems with more components and services, the need for secrets management would be required through a product like Azure Key Vault.

ARM templates provide the ability to persist the exact definition of the web of services in a complex system in a text-based, version-able form. This opens the door to team development

and change management that didn't exist pre-cloud. However, each service we used was quite complex in its own right, each imposing its own learning curve. Ultimately, the logic of the entire system was spread across many places. In this cloud-based environment, it is very important to have clear roles for each component and explicit interface contracts to manage interactions.

6. ACKNOWLEDGEMENTS

We wish to recognize the support of the Congdon School of Supply Chain, Business Analytics, and Information Systems at the University of North Carolina Wilmington.

7. REFERENCES

Amazon Web Services Open Data, 2019. Multilingual Amazon Reviews Corpus https://registry.opendata.aws/amazon-reviews.

Beautiful Soup (2021) https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

Gokce, Y., Kline, D, Vetter, R., Cummings, J. (2021) Automated Text Reduction: Comparison of Reduced Reading List Creation Methods. Annals of the Master of Science in Computer Science and Information Systems at UNC Wilmington, 15(1) paper 2. http://csbapp.uncw.edu/data/mscsis/full.aspx.

He, R., & McAuley, J. (2016, April). Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In proceedings of the 25th international conference on world wide web (pp. 507-517).

Katrix, Basileios "Bill", & Schaul, Robert J (2019) Data Scraping Survives! (At Least for Now) Key Takeaways from 9th Circuit Ruling on the HiQ vs LinkedIn case. The National Law Review, 30 September 2019.

McAuley, J., Targett, C., Shi, Q., & Van Den Hengel, A. (2015, August). Image-based recommendations on styles and substitutes. In Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval (pp. 43-52).

Ni, J., Li, J., & McAuley, J. (2019, November). Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP) (pp. 188-197).

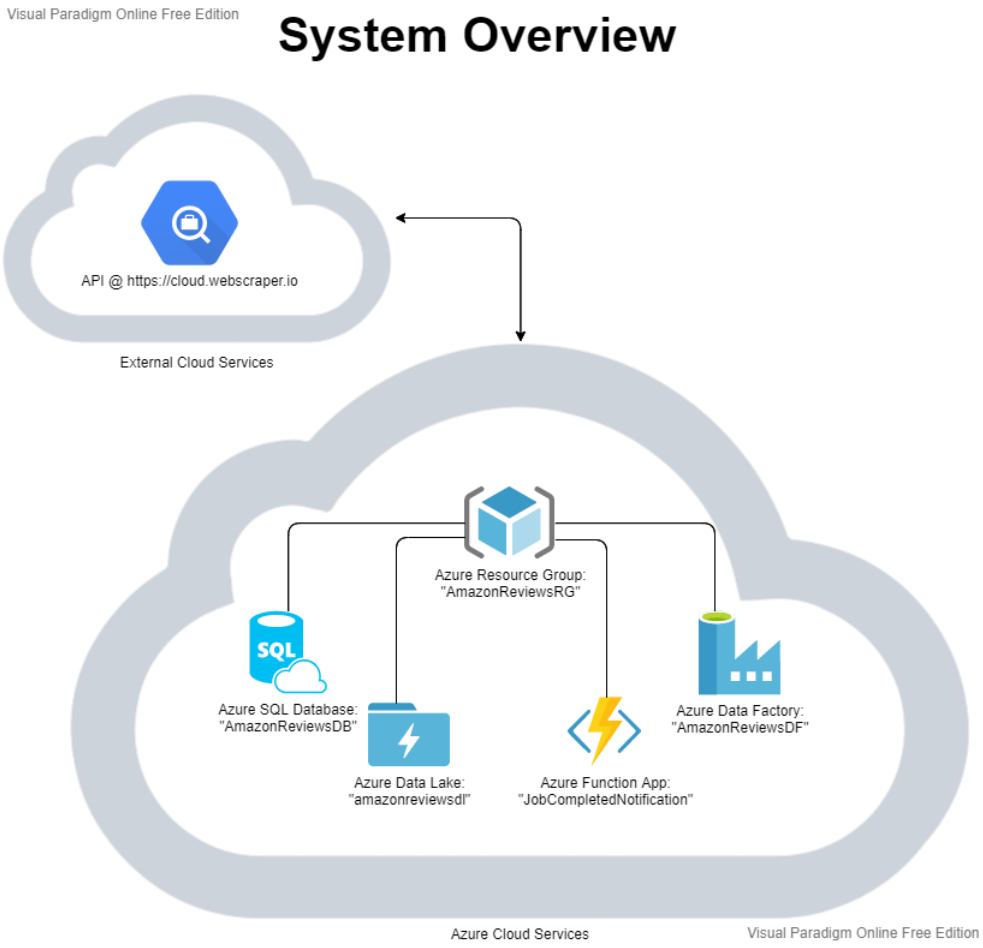
Scrapy (2021) https://scrapy.org/.

Tse, Shing, & Brian, Kristin (2022) HiQ vs LinkedIn. The National Law Review, 19 April 2022.

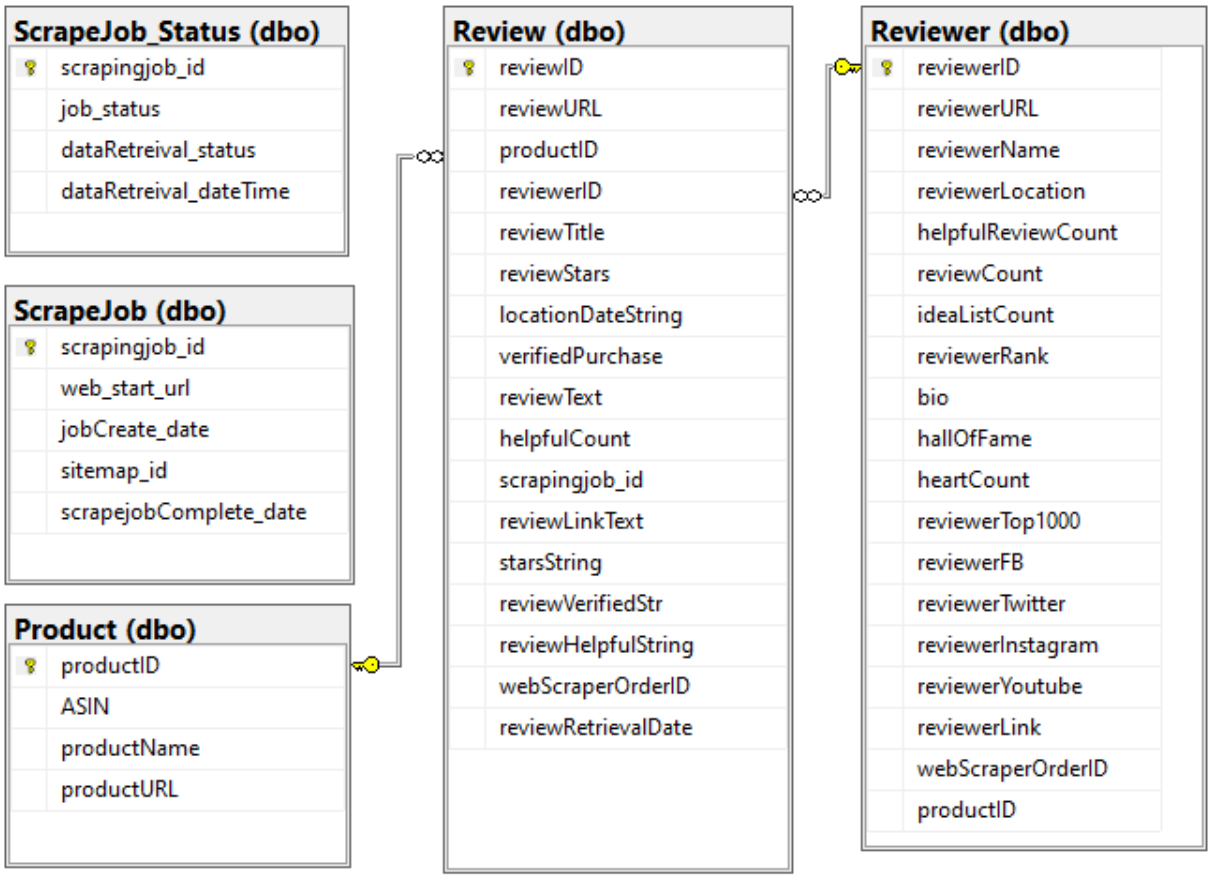
Webscraper Documentation. (2020). Retrieved September 5, 2020, from https://webscraper.io/documentation

Woodall, R., Kline, D, Vetter, R., Modaresnezhad, M. (2021) A Data Pipeline for Amazon Review Collection and Preparation. Annals of the Master of Science in Computer Science and Information Systems at UNC Wilmington, 15(1) paper 3. http://csbapp.uncw.edu/data/mscsis/full.aspx.

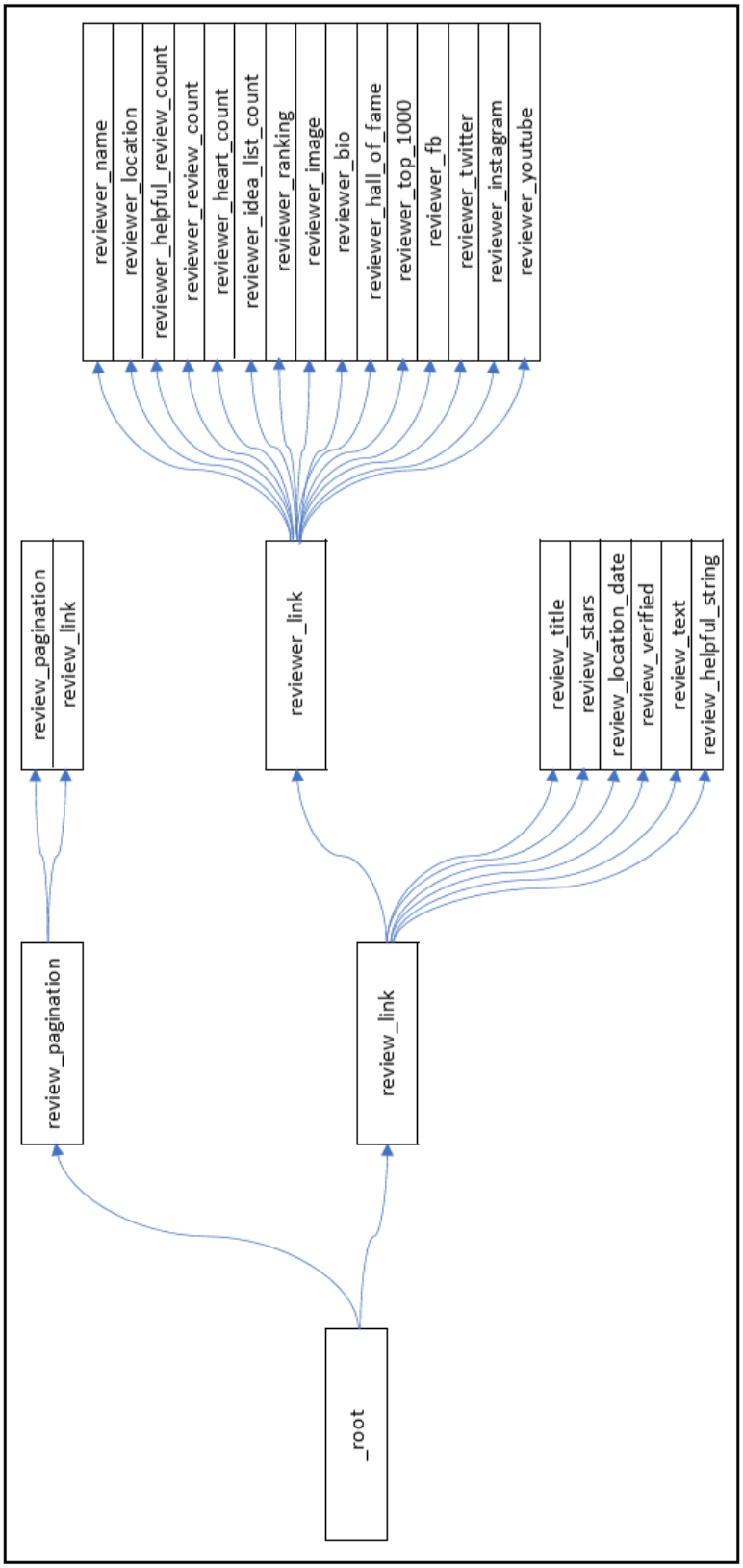
Appendix A - A high-level view of the system components



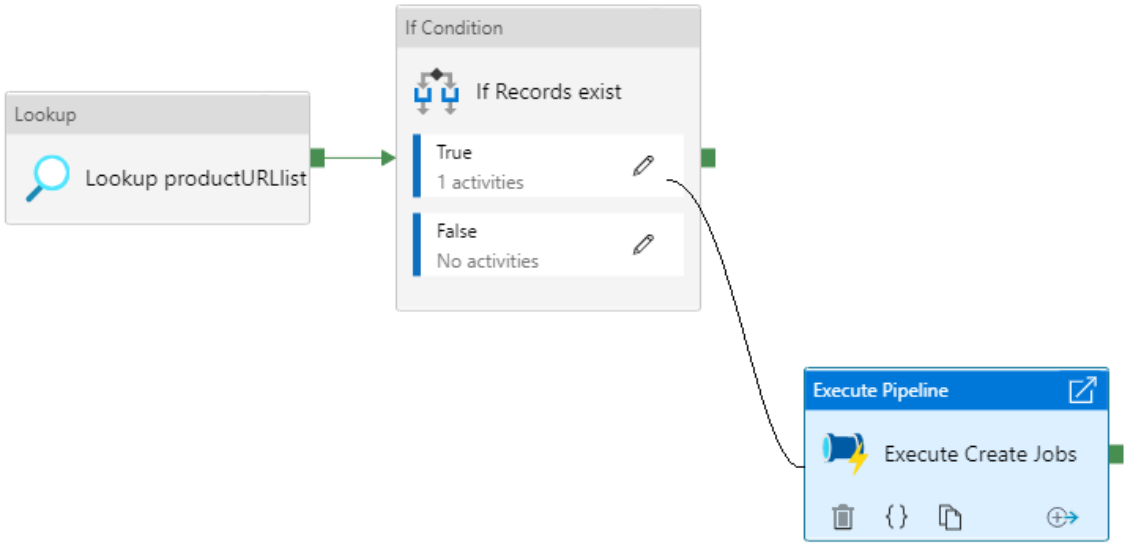
Appendix B – The relational schema



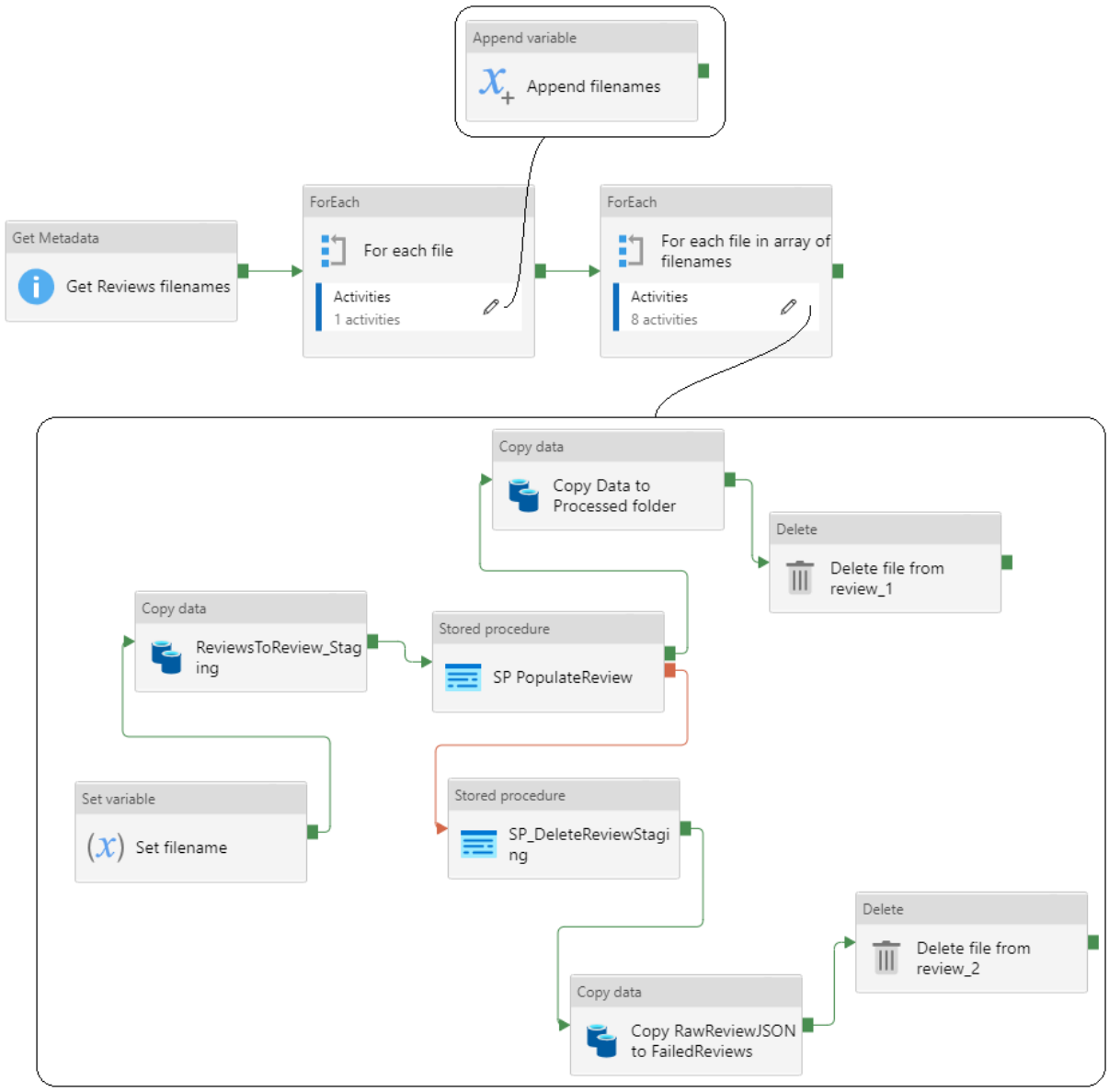
Appendix C – A scrape job definition



Appendix D – An Azure Data Factory workflow example: The data factory job responsible for checking new product urls in the data lake



Appendix E – An Azure Data Factory activity diagram



Appendix F - Azure function for accepting the completion message from the webscraper: The source code for the HTTP endpoint

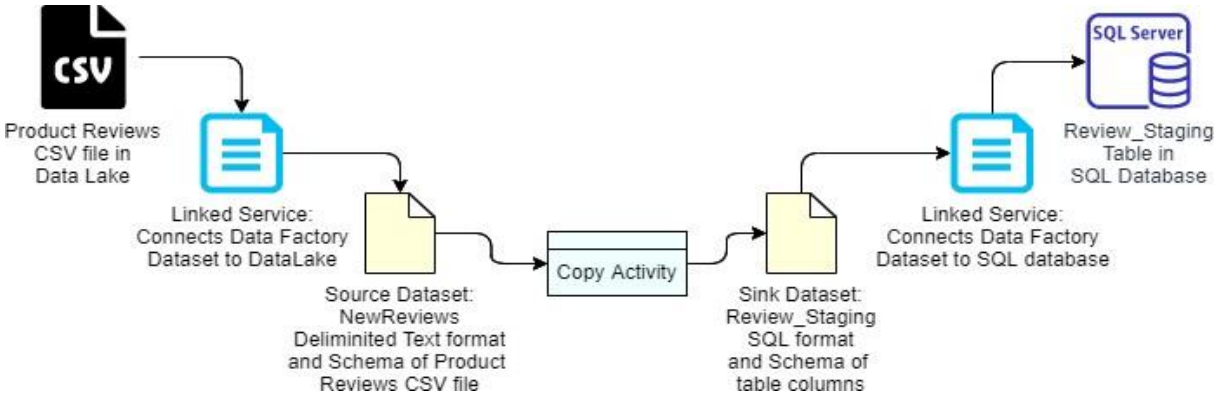
```
__init__.py [function main]
import logging
import azure.functions as func

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)
```

Appendix G - Azure function for accepting the completion message from the webscraper: The source code for setting the bindings

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"
        }
    ]
}
```

Appendix H – An example of a copy activity



Appendix I - The data factory job responsible for calling the stored procedure activity for inserting the data into the relational model

```
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 N
ULL
                      ELSE CONVERT(INT, REPLACE(SUBSTRING(Review_Staging.reviewer_help
ful_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_revi
ew_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_rank
ing,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_hear
t_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_Stagin
g.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
```