Setup and Implementation of an Automated Testing Pipeline for a DataOps Use Case

Bachelor's Thesis (T3201)

presented to the
Department of Computer Science

at the
Baden-Wuerttemberg
Cooperative State University
Stuttgart

by
Oliver Rudzinski

submitted on
September 7th, 2020

Project Period (CW)	25/2020 – 36/2020
Matriculation Number, Course	`5481330`, TINF17A
Training Company	Hewlett Packard Enterprise
Internship Company	DXC Technology
Project Supervisor	Dipl.-Ing. Bernd Gloss
University Supervisor	Jamshid Shokrollahi, Ph.D.

Erklärung

Declaration of Authorship

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema:

I hereby declare that I am the sole author of this bachelor's thesis on the topic:

Setup and Implementation of an Automated Testing Pipeline for a DataOps Use Case

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

and that I have not used any sources other than those listed in the bibliography and identified as references.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

I further declare that the electronically submitted version of this thesis is identical to the printed version.

_____, _____ _____

Ort Place | Datum Date | Unterschrift Signature

Abstract

Contents

List of Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
ARTS	Association for Retail Technology Standards
AWS	Amazon Web Services
BI	Business Intelligence
CI/CD	Continuous Integration & Deployment
CLI	Command Line Interface
CSV	Comma-Separated Values
DAG	Directed Acyclic Graph
DAMA	Data Management Association
DDL	Data Definition Language
DWH	Data Warehouse
EC2	Elastic Compute Cloud
ECR	Elastic Container Registry
ECS	Elastic Container Service
ELT	Extract-Load-Transform
ETL	Extract-Transform-Load
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure as Code
IAM	Identity and Access Management
JSON	JavaScript Object Notation
MBA	Market Basket Analysis
ML	Machine Learning

MVP	Minimum Viable Product
`pip`	Pip Installs Packages
POS	Point of Sales
S3	Simple Storage Service
SPC	Statistical Process Control
SPOF	Single Point of Failure
SQL	Structured Query Language
UI	User Interface
URI	Uniform Resource Identifier
VCS	Version Control System
XML	Extensible Markup Language

List of Figures

List of Tables

List of Source Code Excerpts

1. Introduction

1.1. Relation to Project Environment

The project is conducted within the *Analytics* department of *DXC Technology Company*. This division is currently implementing DataOps as a competence inside its service portfolio and is working on DataOps realization projects with several clients of different business areas. A proof of concept, outlining the features and advantages of DataOps, is desired. On the one hand, it can help to get potential customers interested in DataOps and DXC realizing it. On the other hand, such a project could also be used in existing client workshops and for employee education purposes. The latter could also improve the performance of current and future DataOps projects conducted by DXC Analytics.

1.2. Project Scope

The goal of the superordinate project of this bachelor's thesis is to design and implement a visual, interactive DataOps use case. Specifically, a fictional, idealized retail business use case has been ideated and is now subject to implementation. The use case idea works with a number of retail branches of a retail company that sells different goods to its customers. Based on the purchases, the Point of Sales (POS) data (i.e., receipts) can be used for analytics and Business Intelligence (BI) purposes. Specifically, these data can be leveraged for Market Basket Analysis (MBA). The use case at hand remains entirely fabricated, allowing for an isolated, non-business-critical proof of concept design environment. Apart from the pure data analytics part, the desired solution includes a data generator of pseudoreal input data as well as a web UI for visualization and presentation purposes. These aspects lie outside the scope of this thesis' project. It focusses on the analytics-driven area of the use case. Specifically, the analytics solution of the demonstration use case is a data pipeline which receives input data and performs individual steps in order to generate an MBA, which results could then be visualized inside the User Interface (UI).

1.3. Task Definition

The primary target of this thesis' project is to evaluate the paradigm of DataOps testing within a given use case. The previously described use case analytics solution has already been developed. This solution needs to be evaluated from a DataOps perspective, redesigning it to comply with state-of-the-art DataOps methodologies and standards. Then, the new solution needs to be enhanced with suitable testing frameworks, considering the use case circumstances and priorities. All required infrastructure for reaching the project targets needs to be realized and deployed within Amazon Web Services (AWS), the Amazon cloud platform. Additionally, for scalability reasons, all analytical processes are desired to be designed in a server-less approach.

1.4. Chapter Overview

This chapter, the Introduction chapter, presented the high-level issues of traditional analytics solution development and proposed DataOps as an enhancement.

Chapter 2, the State of the Art chapter, introduces DataOps and its key principles and methodologies as well as general testing frameworks for both software and data quality testing.

Chapter 3, the Actual State Analysis chapter, describes the preexisting analytics solution for the given use case and states present issues which stand in contrast to the task definition.

Chapter 4, the Testing Framework Design chapter, designs the method for implementing holistic DataOps testing inside the use case. It considers the findings from Chapters 2 and 3.

Chapter 5, the Implementation chapter, enables DataOps methodologies described in Chapter 2 and implements the testing framework from Chapter 4, taking missing aspects required by the project target definition into account.

Chapter 6, the Solution Evaluation chapter, evaluates the new solution based on predefined criteria on the functional requirements of its DataOps Testing capabilities.

Chapter 7, the final Conclusion chapter, summarizes all findings, proposes further research and enhancement, and concludes this thesis.

2. State of the Art

The discipline of testing can be found throughout the entire scope of DataOps [1, p. 42]. However, it does not provide specific testing measures or frameworks. This is because data analytics solutions are tailor-made based on the specific needs of their areas of application. Moreover, available DataOps testing foundations remain a set of guidelines and abstract requirements that need to be applied and customized individually within each DataOps solution.

In this chapter, general DataOps and testing foundations are deduced in order to understand the need for DataOps testing and aid the design process of a use-case-specific testing framework which is to follow in Chapter 4.

2.1. Introduction to DataOps

In order to understand the requirement of testing within DataOps, it is important to understand the principles and processes of DataOps itself. In general, DataOps is an approach within building and conducting data analytics which combines established methodologies originating from DevOps, Statistical Process Control (SPC) as well as *agile* software development [2, p. 24]. Several components from each of these methodologies are taken and applied to building and conducting data analytics. These processes aim to eliminate analytics issues found in the traditional development process of such solutions. These issues include but are not limited to slow development and adaptation of analytics solutions [3], error-prone analytics results, repetitive manual processes [2, pp. 11 sqq.], etc. Testing is a common component that supports these DataOps processes.

During the rise of DataOps, other terms, including *MLOps*, *AIOps*, etc., emerged. It is to mention that all data-related *Ops* underlie the general DataOps methodology and focus on specific subsets of data analytics applications [4].

2.1.1. SPC Heritage: Data Analytics Pipeline

Common data analytics solutions work by means of a pipeline: Data is acquired from various sources and flows through various steps of transformation, conversion, sanitization, and analysis before resulting in a valuable outcome, e.g., an analytics report. This can be compared to a manufacturing production line. For instance, raw materials from several input points are navigated through a number of steps, resulting in the output product. Issues that might occur during the production flow need to be recognized immediately. It does not suffice to notice issues at the end of a manufacturing process. This is why Statistical Process Control (SPC) is applied to the entire production line. It verifies that each step is conducted correctly and identifies deviations to expected, pre-defined values [5, p. 1]. Applicable tools can then perform recovery measures or stop the process entirely.

This methodology can be directly applied to data analytics pipelines [2, p. 27]. Each step should check if its input, processing, and output is valid and does not carry issues that might lead to unforeseeable problems during further analysis [6]. This could help solving the problem of incorrect analytics results since reverse-engineering the origin of the problem is often harder than performing individual checks and fallout measures [7]. These operational checks and measures are part of DataOps testing. Nevertheless, this project is going to focus on the functional testing aspects of the solution (e.g., when a new version of the solution is about to be deployed into production).

2.1.2. DevOps Heritage: CI/CD Pipeline Duality, VCS and Environment Management

DataOps' namesake, DevOps, originates in software development and aims to eliminate manual repetitive processes by automating them. It introduced Continuous Integration & Deployment (CI/CD) pipelines that take over processes taking place between solution development and deployment. This results in automatic building, testing, and deploying of software solutions [8, pp. 21 sqq.]. This does not only remove repetitive processes but also eliminates so-called *siloed organizations* (i.e., dedicated engineers, testers, operation teams, etc.) depending on each other during a development iteration [2, p. 56].

In DataOps, enabling CI/CD creates a pipeline duality. This duality is visualized in Figure 2.1.

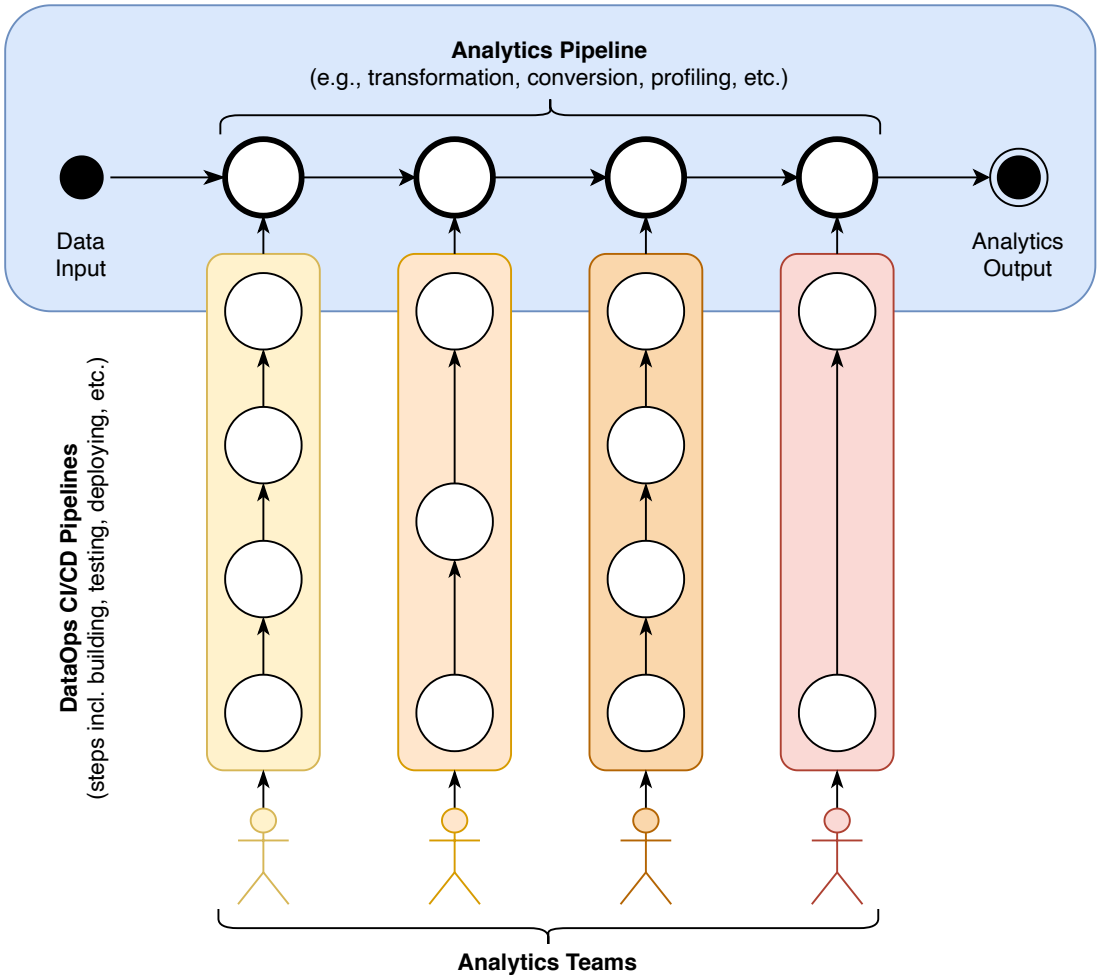

Figure 2.1.: DataOps Pipeline Duality (per [2, pp. 38 sqq.])

The data analytics (or data operations) pipeline is also called *Value Pipeline* since it is in charge of answering questions through analytical insights [2, pp. 32 sq.]. Its horizontal representation visualizes its continuous flow. On the contrary, the vertical pipelines, also called *Innovation Pipelines* represent the DataOps CI/CD pipelines [1, p. 66]. Whenever a new feature is developed for any stage of the pipeline, a number of preliminary steps is performed before finally deploying the solution into production [2, p. 33]. As with data analytics in general, the design of such pipelines highly depends on the analytics and quality requirements. In the scenario of Figure 2.1, the individual stages require different CI/CD steps and might also be worked on by different teams within the superordinate project.

As with DevOps, DataOps CI/CD ties in with the project's Version Control System (VCS). This allows for collaboration between developers as well as development environment management [9]. Usually, a developer uses an individual sandbox to make changes to the common source code. This sandbox is a highly isolated en-

vironment which can be used without impacting the development process of other developers. It includes the current common source code as well as processes for installing and running all required dependencies [2, p. 41]. When committing a change to the VCS, it triggers the corresponding CI/CD pipeline which performs its checks and reports potential issues. Otherwise, the updated source code becomes the new common source code since the deployment into production has been conducted successfully.

Another methodology originating from DevOps is *Infrastructure as Code (IaC)*. In a desired automated environment, IaC also enables automatic creation, provisioning, and re-instantiation of a (cloud) infrastructure [10, pp. 8 sqq.] which does not require repeated UI-based configuration, etc. This is enabled by programatic configuration of infrastructure resources, settings, as well as applications.

2.1.3. Agile Heritage: Fast-Paced, Iterative Development

One problem within data analytics is that traditionally developed solutions cannot keep up with the demand of changing requirements. The development process is slow such that valuable and time-dependent information cannot be processed on time [3]. In software development, agile development mostly replaced traditional waterfall-orientated processes. *Agile* in DataOps context means that development and improvement is iterative and fast-paced. It requires a Minimum Viable Product (MVP) which is continuously improved by means of previously mentioned SPC and CI/CD processes [2, pp. 19 sq.].

2.2. Introduction to Testing

The previous section introduced where testing aspects can be found within DataOps. This section presents the current state of the art for software and data quality testing. Both disciplines will be evaluated and used within the testing framework design process.

2.2.1. Why Do We Need Testing?

In general, software development relies on testing principles to holistically and objectively validate the expected performance of a piece of software. Nowadays, organizations depend on data analytics more than ever [11]. BI and Data Warehouse (DWH)

solutions are designed to utilize data for business-required decision making [12]. While these systems are expected to generate value, many companies lose trust in their data analytics because it might be prone to unforeseeable errors [13]. This is because BI and DWH systems rely on high-quality data in order to provide representative analytics results and business insights [11]. Unfortunately, data quality issues of various sorts and manifestations lead to the systems generating false and potentially misleading reports [11][14][15]. Testing the solution for its expected performance at various analytics steps might help solve the underlying issues or even exhaust them completely.

From a pure software perspective, it is desired that the solution does not break or crash during analytics performance for an unforeseeable reason. Applicable software tests could recognize such issues prior to production deployment, reducing or completely removing crucial bugs inside the software [16, pp. 105 sqq.].

As previously mentioned in Section 2.1.2, the pipeline duality ensures continuous flow of both production-grade data analytics as well as improvement and enhancement of the solution. Both pipelines require individual testing measures. This is referred to as "The Duality of [DataOps] Testing." [2, pp. 40 sqq.]

2.2.2. Value Pipeline: Data Quality Testing

Since the Value Pipeline is in charge of the business-critical real-time analysis of various data, the solution-in-use must guarantee the recognition and handling of data quality issues prior, during, and after each individual processing step by means of its SPC capability. In other words, while the underlying analytics software is static, the variance of data is arbitrarily large and needs to be handled properly. In order to define this kind of event handling, unified data quality dimensions are required.

The Data Management Association (DAMA) in the United Kingdom defined "The Six Primary Dimensions for Data Quality Assessment" in 2013 [17, pp. 7 sqq.]. They consist of:

Completeness describes the proportion of the stored data against the potential of being *complete* by means of a use-case-specific completeness definition [17, p. 8][18].

Uniqueness is achieved when each unique data record only exists once inside the entire database at hand [17, p. 9].

Timeliness is the degree to which data represents the reality from the required point in time [17, p. 10].

Validity describes a data item corresponding to its expected (and therefore, predefined) format, schema, syntax, etc. This definition should also include a range of expected or acceptable variation thresholds [17, p. 11]. Testing for data schematics is one processes which allows for definitive objective differentiation between good and bad data [19].

Accuracy is the degree to which data *correctly* describes the actual object or event existing in the real world [17, p. 12].

Consistency describes the absence of difference when comparing multiple representations of the same real-life object against its actual definition [17, p. 13].

It can be seen that the areas of data quality are mostly covered when data complies with the corresponding data governance definitions [19]. These definitions build the foundation for a valid testing framework.

In practice, checks based on these dimensions have to be embodied inside the data analytics solution. Based on mentioned, use-case-specific requirements, a data flow can be checked by means of those dimensions, leading to recovery measures inside the system, or a system failure with appropriate error reporting.

2.2.3. Innovation Pipelines: Software Testing

Section 2.1.2 describes the Innovation Pipelines as DataOps-specific CI/CD pipelines. Apart from the build and deployment process, a major part of these pipelines is represented by software testing. Whenever a developer intends to update the codebase, a number of tests of various levels is conducted, visualized in the pyramid graphic in Figure 2.2 below.

Depending on the type of software, the types and levels of testing can vary. For data analytics solutions with UIs outside their scope, the three foundational levels suffice.

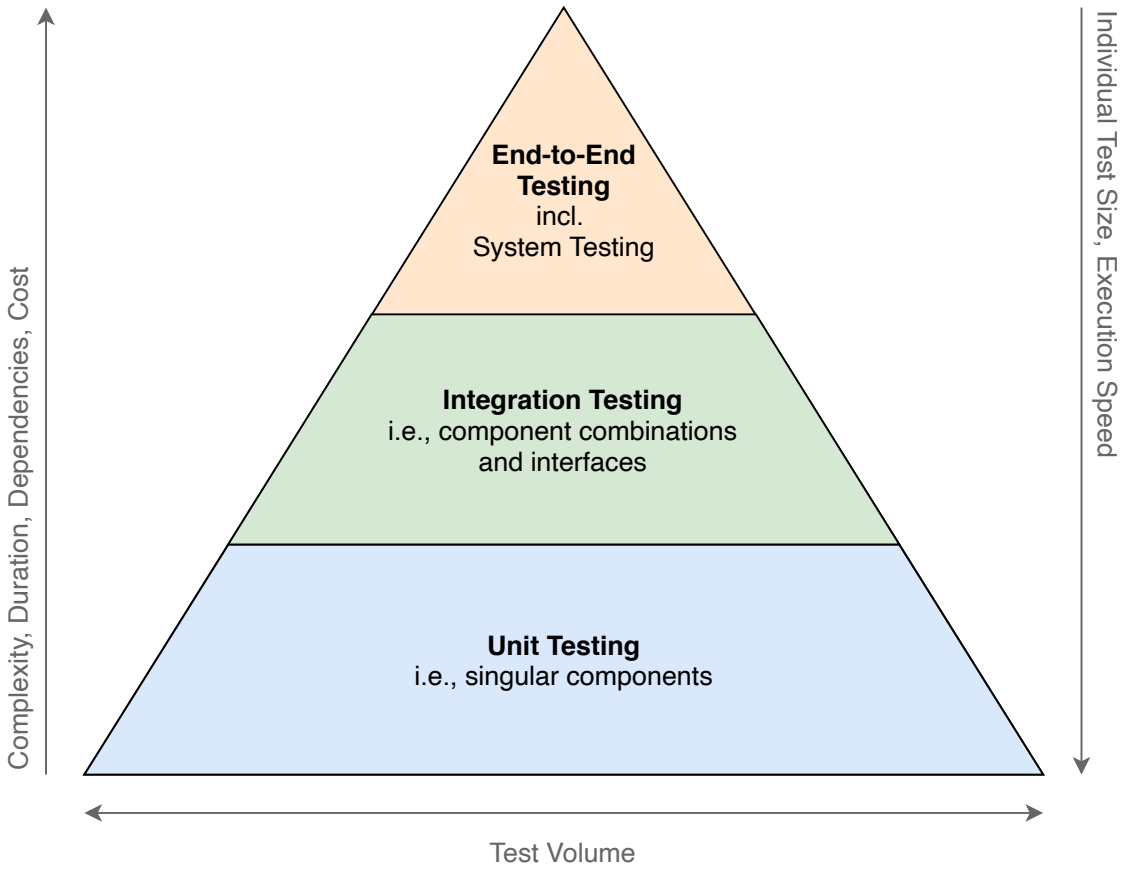

Figure 2.2.: Software Testing Level Pyramid

Unit Tests take individual components (i.e., *units*) from the source code and check their behavior. Unit tests are pieces of code that invoke the chosen test unit and compare its actual result with the expected outcome [20, sec. 1]. They are characterized as granular, thus highly voluminous, repeatable, isolated, and idempotent [20, sec. 2].

Integration Test verify that multiple combined units are working together as expected. They focus on testing of interfaces that connect singular components [21, p. 66]. Integration tests are performed in a less isolated environment, resulting in more outside dependencies [20, sec. 3].

End-to-End Tests describe the highest level of software testing [21, p. 67]. For the sake of simplicity, system tests are treated as a part of end-to-end tests within this work. End-to-end tests are performed under a (close-to) deployment situation. This creates an idealized real-life scenario. Thus, end-to-end testing is the least isolated level and serves as the final test stage before actual deployment. [21, p. 67].

Those three layers represent the *Fuctional Testing* type since they are built based on the functional requirements of a software solution [21, p. 69]. Its counterpart, *Non-Functional Testing*, covers aspects like load, performance, security, etc. [21, p. 70].

Apart from that, other testing methods can be applied to each testing layer. *Smoke Tests* are a subset within the entire test suite that cover core functionality required for the solution to *just* run. Such tests can help to recognize the complexity of a software issue [22, sec. 5]. Conducting *Regression Tests* is also important. They aim to uncover errors with pre-existing components when other components were newly included, changed, or removed [21, p. 70][23]. This is especially crucial with data analytics solutions since historic data still needs to be processable after updating the analytics engine [18]. To achieve regression testing in practice, unit tests can be written in a more abstract manner and re-run for the entire software solution, requiring older unit tests still to pass [23].

Putting everything into DataOps perspective, the Innovation Pipelines need to cover source code changes of the analytics solution. These CI/CD pipelines need to embody traditional software tests to verify the correct behavior of the application. This also includes testing the SPC capability of the given solution [24]. For that reason, pathological test data needs to be provided that is able to invoke a majority of realistic events during the analytics process [2, p. 42]. In case of tests failing, the corresponding CI/CD Innovation Pipeline can report the issue back to the developer, allowing for further understanding and correction.

All in all, these tests are expected to invoke the majority of outcome possibilities, leading to a high test coverage and correct performance validation [6].

Considering DevOps testing (as opposed to the currently discussed DataOps testing) which is expected not to be data-driven, there exist many similarities to the Innovation Pipeline testing process. In general, the features of the solution need to be tested and validated based on their desired functionality. However, DataOps introduces a highly data-focussed testing process, requiring data governance consideration, test data management, etc.. Plus, it is expected to set different priorities in testing because of this data aspect. These priorities as well as further similarities and differences between DevOps and DataOps testing are discussed and expected to be deduced in the Solution Evaluation chapter.

3. Actual State Analysis

As stated in the Introduction chapter, the project is conducted based on a preexisting data pipeline solution where DataOps is to be enabled, including a holistic DataOps testing framework. This chapter performs an actual state analysis, describing the superordinate data analytics use case as well as the data pipeline specification prior to the project. The resulting aspects will then be used for the testing framework specification and reimplementation process.

3.1. General Information

The data pipeline at hand is four-stage pipeline that creates a Market Basket Analysis (MBA) for BI and reporting purposes. MBA is one of the key techniques used by large retailers to uncover associations between items. It works by looking for combinations of items that occur together frequently in transactions [25, p. 1]. The specific MBA performed at the end of this data pipeline provides two kinds of information. First, it describes item sets that are frequently purchased together. Specifically, the higher the *support* of an item set, the higher the probability that these items are purchased together. The *Apriori* algorithm is a widely used method to accomplish this [26, pp. 12 sq.]. Second, the MBA calculates other association metrics (i.e., *confidence* and *lift*) of the item sets, allowing for backed decision-making with regard to marketing campaigns, inventory stock-up, etc.

MBAs are based on a large amount of POS data. These data need to be integrated by means of the analytics engine requirements in order to be used for MBA purposes.

3.2. Analytics Solution Components

The data pipeline consists of various infrastructural and technical components. This also includes the input and output formats of the analyses. These components are described in the following.

3.2.1. MBA Engine: Python Library `mlxtend`

The Python library `mlxtend` provides functions that are required for conducting the previously described MBA. `mlxtend.apriori` conducts the analysis of frequent item sets. These are used for the second step, where `mlxtend.association_rules` is leveraged to calculate confidence and lift metrics [27] for all item sets with support $> 5\%$. Currently, the resulting associations are filtered such that only those with a high lift (> 6) and high confidence (> 0.8) are provided. The result is a Microsoft Excel spreadsheet document which lists all analytically promising item sets.

For better distinction between the actual association of the products, the MBA engine performs two analyses. Technically, they perform identically but the first analysis only considers items that are classified as food. The second analysis performs by only looking at non-food items. Thus, the analysis does not take cross-category associations into account, as they might be random and complicated to take advantage of.

The `mlxtend` library works with so-called data frames which are typically managed by another Python library, `pandas` [28]. The MBA algorithms require a data frame that flags what kinds of sub-categories of products (both food and non-food items) were purchased in every considered transaction. Thus, to provide these pieces of information, `pandas` requires an input of semi-structured data [28] where the following requirements are satisfied:

- each individual transactions can be identified: `InvoiceNo` attribute for each purchase
- each individual item is sub-categorized: `Description` attribute for each purchase
- each individual item is categorized (food or non-food): `ItemType` attribute for each purchase
- each individual purchase shows if and how much of an item was purchased: `Quantity` attribute

All these requirements can be satisfied by providing a single Comma-Separated Values (CSV) file which contains an arbitrary number of transactions. Based on the transaction information, the MBA is conducted.

3.2.2. Input Data: *ARTS POSLog 6* Standard

Transaction data is usually provided to the customer through a receipt that is printed out at the register. This data can also be leveraged for MBA. Since multiple registers across multiple branches of a retail company produce a large number of individual receipts every day, the format of those POS data needs to be unified. Additionally, the data needs to be centralized such that cross-branch MBAs are possible.

A widely used POS standard is the *POSLog 6* standard by the Association for Retail Technology Standards (ARTS). It is used by many established register systems. It is saved in Extensible Markup Language (XML) format and contains the same information that is usually provided on a print-out receipt [29], i.e., general transaction information (e.g., name and address of retail branch, cashier's name, currency) as well as information on each item purchased (category, brand name, quantity, price, etc.). The provided information allows to find out the required attributes for MBA.

An exemplary XML code snippet in POSLog format for the current solution looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsi:POSLog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Transaction CancelFlag="false" OfflineFlag="false" TrainingModeFlag="false">
    <RetailStoreID>High Street</RetailStoreID> <!-- retail store name -->
    <RevenueCenterID>6333-1221</RevenueCenterID> <!-- arbitrary -->
    <WorkstationID>POS5</WorkstationID> <!-- arbitrary -->
    <TillID>22</TillID> <!-- arbitrary -->
    <SequenceNumber>1000000000</SequenceNumber> <!-- POS receipt number -->
    <BusinessDayDate>2001-08-13</BusinessDayDate> <!-- POS date -->
    <BeginDateTime>2001-08-13T09:03:00</BeginDateTime> <!-- POS start datetime -->
    <EndDateTime>2001-08-13T09:05:00</EndDateTime> <!-- POS end datetime -->
    <OperatorID>Gosho</OperatorID> <!-- arbitrary -->
    <CurrencyCode>USD</CurrencyCode> <!-- arbitrary -->

    <!-- BEGINNING OF POS DATA -->
    <RetailTransaction OutsideSalesFlag="false" Version="2.2">
      <ManagerApproval>false</ManagerApproval>
      <ReceiptDateTime>2001-08-13T09:04:32</ReceiptDateTime>
      <LineItem VoidFlag="false">
       <SequenceNumber>1</SequenceNumber> <!-- incrementing sequence number (order) -->
        <BeginDateTime>2001-09-16T09:04:00</BeginDateTime>
        <EndDateTime>2001-09-16T09:04:03</EndDateTime>
        <Sale ItemType="Stock">
          <ItemID>FD7865</ItemID> <!-- first letter is type identifier (F = Food, N =
          ↪  Non-Food) -->
          <MerchandiseHierarchy Level="Department">Candies</MerchandiseHierarchy> <!--
          ↪  item category -->
          <ItemNotOnFileFlag>false</ItemNotOnFileFlag> <!-- arbitrary -->
          <Description>20oz HERSHEY'S</Description> <!-- specific product -->
          <TaxIncludedInPriceFlag>false</TaxIncludedInPriceFlag>
          <!-- PRICING -->
```

```
          <UnitCostPrice>1.23</UnitCostPrice>
          <UnitListPrice>1,79</UnitListPrice>
          <RegularSalesUnitPrice>1.63</RegularSalesUnitPrice>
          <InventoryValuePrice>1.23</InventoryValuePrice>
          <ActualSalesUnitPrice>1.63</ActualSalesUnitPrice>
          <ExtendedAmount>4.89</ExtendedAmount>
          <DiscountAmount>0.00</DiscountAmount>
          <ExtendedDiscountAmount>4.89</ExtendedDiscountAmount>
          <Quantity>3</Quantity> <!-- number of items purchased -->
          <POSIdentity>
            <POSItemID>01532226057966</POSItemID>
          </POSIdentity>
        </Sale>
      </LineItem>
      <LineItem VoidFlag="false">
        <!-- ... -->
      </LineItem>
      <!-- ... -->
      <LineItem>
        <Total TotalType="TransactionGrossAmount">44.01</Total>
      </LineItem>
    </RetailTransaction>
  </Transaction>
</xsi:POSLog>
```

Source Code Excerpt 1: Sample POSLog XML File

3.2.3. Data Location: *AWS S3 Data Bucket*

In a real-life retail scenario, the individual POS data of each branch are consequently uploaded to the branches' data infrastructure. Each day at end of business, this data needs to be provided to the global MBA engine. This is why the data is compressed into a single ZIP archive, identified by branch name and date stamp, and uploaded to the landing zone of the global retail company data lake. This data lake is realized with an AWS Simple Storage Service (S3) data bucket, a server-less data storage solution [30]. The data lake structure can be seen in Figure 3.1.

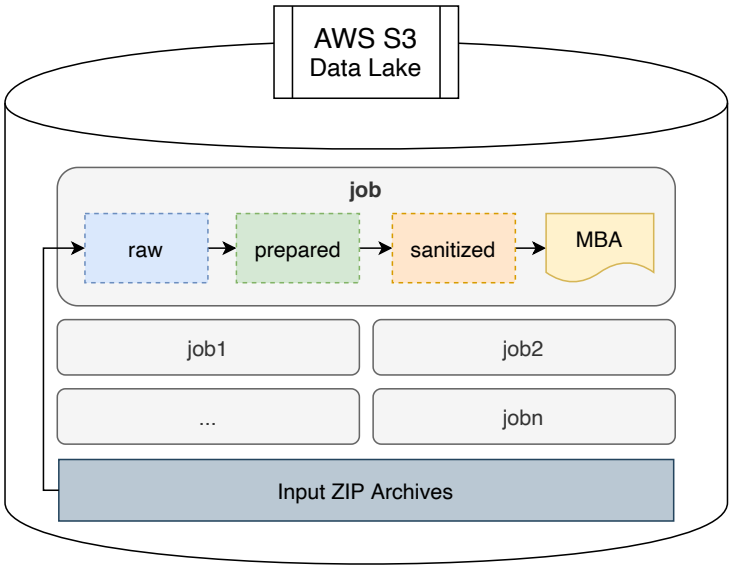

Figure 3.1.: Preliminary Data Lake Structure

For the data analytics pipeline which integrates the data and performs the actual MBA, the landing zone of production-grade data (as described in the previous paragraph) is considered the entry point. It receives the previously mentioned POSLog ZIP archives. It also contains the job sub area which contains all analysis jobs, e.g., an MBA every day after receiving new data. Each job area is again divided into sub areas which are used as intermediate destinations for the individual data pipeline stages, while the resulting MBA Excel spreadsheet file is uploaded to the root of its job area.

3.2.4. Centralized Data Processing Engine: *AWS Athena* Serverless SQL Database

It is desirable that a global MBA engine performs the analysis rather than every branch performing their own analyses since this might lead to inconsistencies between the individual results. Therefore, the data lake is not only used for storage purposes, but is rather the actual area where the analysis is performed based on the contents in its raw zone. Considering structure, all POS data across all branches is expected to be identical, which means that all data can be normalized and synthesized in the same way, allowing the data to be ingested into a relational database table for querying purposes.

AWS Athena makes use of the S3 data lake and does not require additional data storage resources. It rather considers a specific area of the S3 bucket its data source

and can query against it using traditional Structured Query Language (SQL) [31]. The query results are represented in CSV format that can be used by the MBA engine to perform its analyses.

The query for the information required by the MBA looks as follows:

```sql
--SQL
SELECT
  sequencenumber as InvoiceNo,
  itemcategory as Description,
  SUBSTRING(itemid, 1, 1) as ItemType,
  quantity as Quantity
FROM retail_transactions
ORDER BY InvoiceNo
```

Source Code Excerpt 2: SQL Query for Required MBA Information

This yields an output of the following format:

Attribute	Data Type	Corresponding JSON Attribute
SequenceNumber	bigint	SequenceNumber
Description	string	ItemCategory
ItemType	char (F/N)	ItemID (first character)
Quantity	int	Quantity

Table 3.1.: CSV Output Format

Using an entire external service (i.e., AWS Athena) might appear to have a high overhead but could be leveraged especially in situations where the MBA input requires different data. In that case, only an SQL query would need to be updated, as opposed to the entire source code of one or multiple stages of the pipeline. In case the MBA does not change its requirements, it might be possible to handle the XML-to-CSV conversion and reduction internally. The current solution applies AWS Athena for simplicity reasons.

3.3. Data Pipeline

The goal of the data pipeline is to integrate the input data (i.e., the POSLog XML files) and preparing the required accumulated, CSV-formatted data for the actual MBA. This process is divided into three preparatory data integration stages and followed by the fourth and final MBA stage.

In theory, a data pipeline can be understood as a Directed Acyclic Graph (DAG), since it defines a specific flow direction as well as a beginning and an end of one analytics period. The pipeline DAG at hand is technically encapsulated inside the *Apache Airflow* workflow software, residing on a AWS Elastic Compute Cloud (EC2) virtual server instance. It makes use of DAGs to define, manage, and operate (analytical) workflows. Airflow is Python-driven and provides a UI for real-time monitoring purposes [32], which makes it suitable for data analytics processes. It is important to mention that, currently, Airflow does not only orchestrate the analysis, but rather performs it. All analytics scripts are directly written into Airflow. The terms *pipeline* and *DAG* can be used interchangeably in the current state of the analytics solution. Its general structure is presented in Figure 3.2 below.

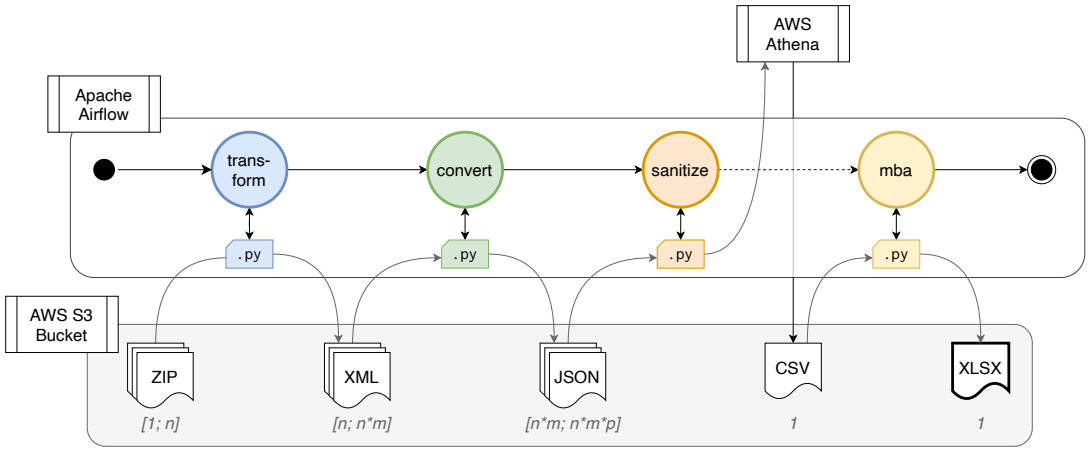

Figure 3.2.: Data Pipeline Overview

As shown in Figure 3.2, the Transformation Stage scans the pre-defined data lake landing zone for ZIP archives. It decompresses the ZIP archives and loads all XML files into a designated raw-file directory of the job-specific data lake space. The Conversion Stage takes the resulting data and prepares it for a serverless database integration, realized with AWS Athena. The Sanitization Stage queries the resulting table for the attributes required by the MBA engine. The query result is exported in CSV format and provided to the MBA engine, which performs the analysis in its designated MBA Stage, which represents the final stage of the data pipeline.

3.3.1. Transformation Stage

Figure 3.3 visualizes the Transformation Stage.

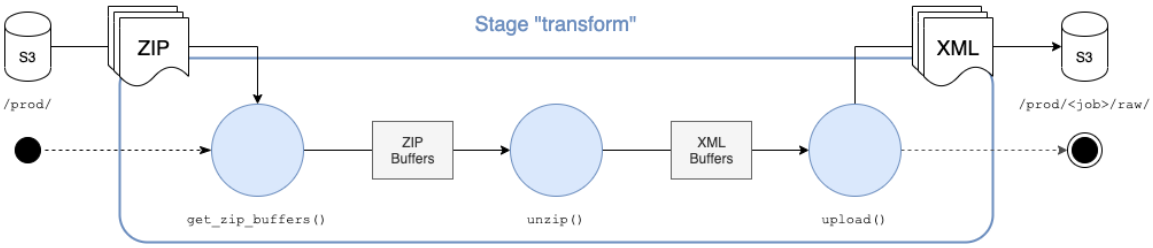

Figure 3.3.: Transformation Stage of the Data Pipeline

The archived XML data needs to be decompressed such that its contents can be read out and analyzed. In order not to contaminate the landing zone of the data lake, the data is simply decompressed and uploaded to a raw-file directory within the designated job data lake space. The XML data is nested, which is not suited for the direct integration into a relational database table, which is why the data needs to be converted in the next stage.

3.3.2. Conversion Stage

Figure 3.4 visualizes the Conversion Stage.

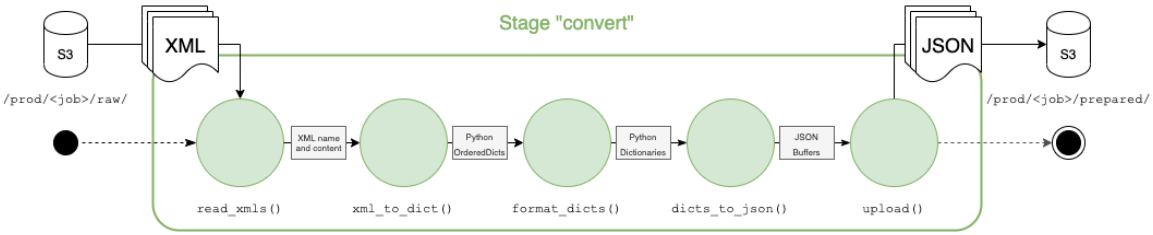

Figure 3.4.: Conversion Stage of the Data Pipeline

The XML files contain transaction-level information where the individual purchases are nested into. The MBA engine requires purchase-level information with a transaction reference which means that the XML data needs to be redundantly converted. The resulting data should still contain transaction information, but a single data entity needs to provide information on a single purchase within the referenced transaction and shall not be nested.

Before this conversion can take place, the XML data needs to be made processable in Python. Here, the `xmltodict` library is leveraged. The resulting Python dictionaries reflect the structure of the XML file. To achieve the goal of purchase-level dictionaries,

each purchase in a dictionary creates its own dictionary and also gets all transaction information provided. This process is visualized in Figure 3.5.

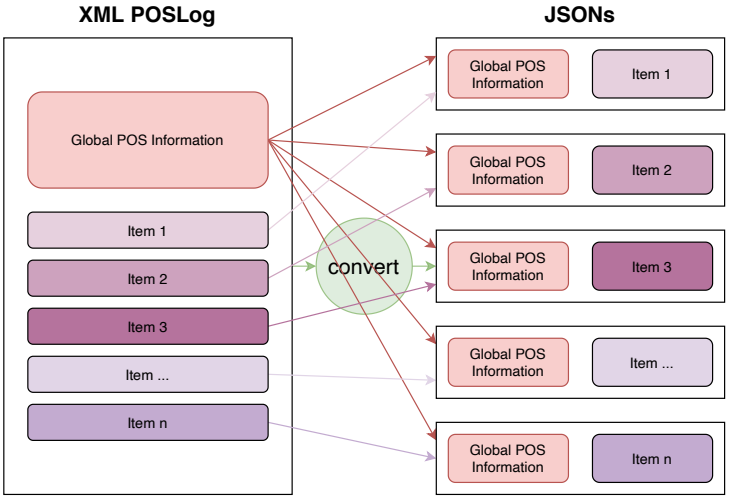

Figure 3.5.: Conversion of Nested POSLog XML Files

In simple terms, each new dictionary can be seen as a receipt for a single purchase since it contains all global transaction information as well as sale information for a single item.

The final step of this stage is to provide the data in a format that can be queried against using AWS Athena SQL queries. Per its documentation, JavaScript Object Notation (JSON) is a supported data format [31]. Thus, each dictionary is exported as its own JSON file to a prepared-file directory within the designated job data lake space. An example output JSON file can be seen below.


```
{
    "RetailStoreID":"High Street",
    "RevenueCenterID":"6333-1221",
    "WorkstationID":"POS5",
    "TillID":"22",
    "SequenceNumber":"1000000001",
    "OperatorID":"Gosho",
    "CurrencyCode":"USD",
    "ReceiptDateTime":"2001-08-13 09:04:32",
    "ItemSequenceNumber":"1",
    "ScanBeginDateTime":"2001-09-16 09:04:00",
    "ScanEndDateTime":"2001-09-16 09:04:03",
    "ItemID":"NO7722",
    "ItemCategory":"Wallets",
    "ItemNotOnFileFlag":"false",
    "ItemDescription":"Tommy Hilfiger Men's Extra-Capacity RFID Leather Tri-Fold
    ↪  Wallet",
    "TaxIncludedInPriceFlag":false,
    "UnitCostPrice":"1.23",
    "UnitListPrice":"1.79",
```

```
    "RegularSalesUnitPrice":"1.63",
    "InventoryValuePrice":"1.23",
    "ActualSalesUnitPrice":"1.63",
    "ExtendedAmount":"4.89",
    "DiscountAmount":"0.00",
    "ExtendedDiscountAmount":"4.89",
    "Quantity":"3",
    "POSItemID":"01244652347368"
}
```

Source Code Excerpt 3: Sample Converted Single-POS JSON File

The query itself is performed in the next stage.

3.3.3. Sanitization Stage

Figure 3.6 visualizes the Sanitization Stage.

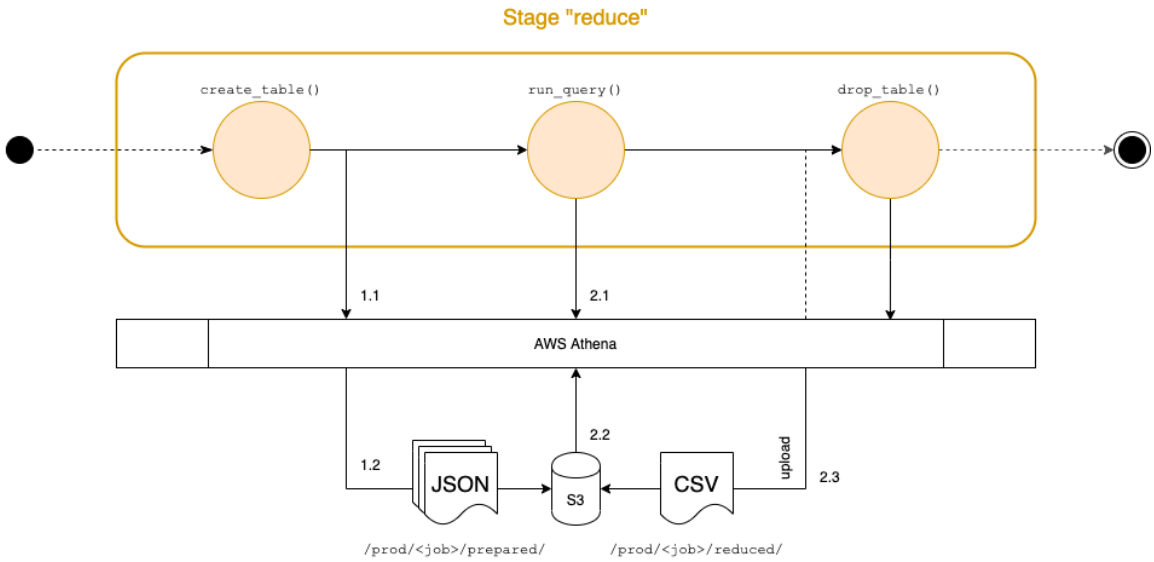

Figure 3.6.: Sanitization Stage of the Data Pipeline

The currently provided JSON files contain a significant amount of information which is not required for MBA. The data needs to be sanitized to only the required attributes for MBA and exported in CSV format.

Since each individual MBA has its own designated space in the data lake, the relational database table for each MBA needs to be created with its appropriate JSON file location. Thus, Athena Data Definition Language (DDL) is used to create the table and insert each JSON file as a row into the table.

After the table is populated, the Athena SQL query for `InvoiceNo`, `Description`, `ItemType`, and `Quantity` (cf. Source Code Excerpt 2) results in the representation

that is needed by the MBA engine. This query result is exported to the prepared-file directory of the job-specific data lake space. For the sake of a clean environment, the previously created table is dropped. The generated CSV file can now be used for MBA.

3.3.4. MBA Stage

Figure 3.7 visualizes the MBA Stage.

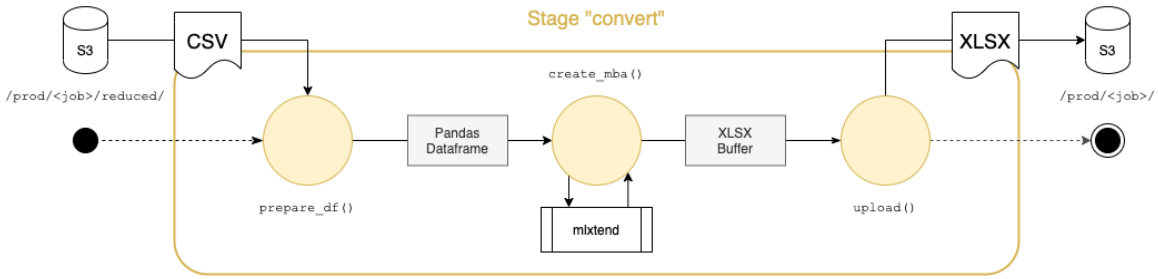

Figure 3.7.: MBA Stage of the Data Pipeline

As mentioned in its own section, the CSV file is provided to the MBA engine. In case the data is not in correct order (horizontal or vertical, respectively), it is reordered in the process of creating the `pandas` data frame. `apriori` is applied on the data frame, which is then used to determine `association_rules`. The `Quantity` attribute of the CSV file provides information on which items (i.e., `Descriptions`) where purchased in a transaction (identified by `InvoiceNo`), separated by `ItemType F` (i.e., food) and `N` (i.e., non-food). Both the food and non-food data frames and association rules need to be exported to a single Excel spreadsheet file. This functionality is also covered by `pandas` [28]. This file is finally uploaded to the root directory of the current analysis job data lake space. At this point, one MBA data pipeline process is finished.

3.4. Comparison to Target Requirements

From an infrastructure point of view, the preliminary solution meets the requirement of being implemented by means of AWS cloud services. On the contrary, the current architecture design is not server-less. More specifically, all analytics steps are performed on the same Airflow instance, resulting in bad scalability and a Single Point of Failure (SPOF) dependency (cf. Figure 3.2). Additionally, since some steps inside the pipeline might have different performance requirements, the server-driven architecture could lead to bottlenecks for some stages and to overhead for others.

From a DataOps perspective, the solution at hand lacks fundamental implementation of state-of-the-art methodologies, e.g. VCS support, CI/CD automation, etc. These aspects need to be covered, taking the use case circumstances into consideration. Developing the solution needs to be done in individual, isolated environments in order not to contaminate the production-grade environment as well as other developers' environments within the project. This can be achieved by VCS enablement. Additionally, since each pipeline stage is expected to be developed and maintained by different developers, the individual stages could have different requirements for their CI/CD automation. Therefore, four individual DataOps CI/CD pipelines need to be implemented by applying the different needs for every stage.

Finally, the current solution is entirely untested. This includes the absence of validating software tests as well as data quality checks within the solution. A use-case-specific testing framework based on the use case data definition and governance is required. This framework needs to be designed before it can be embedded inside the future DataOps processes of the solution.

4. Testing Framework Design

The creation of a DataOps testing framework aims to assure the data and software quality requirements of a solution. In general, based on the findings from Section 2.2, the following design process of such a framework is proposed:

1. Include data quality checks within the solution at hand based on pre-defined data requirements or data governance specifications.

2. Define test data suites that invoke the data quality checks, resulting in both positive and negative outcomes.

3. Create automated tests for the solution that make use of the test data suites to verify that the data quality and solution performance requirements are met. These tests will run whenever a new version of the solution is about to be deployed into production.

This chapter takes these steps, elaborates on them, and applies the project's MBA data pipeline solution on the process. It aims to provide a guideline for designing a DataOps testing framework based on a given scenario. For the desired pilot demonstration, the definition of a testing framework for a specific step within the data pipeline is sufficient. The Conversion Stage of the MBA data pipeline is suited for this task since it prepares the Athena database integration of raw POSLog data. This includes a variety of data transformation and processing, and thus, is prone to data quality issues throughout the entire flow of the stage.

4.1. Data Quality Check Design

In order to check for data quality issues within a data flow, it is important to understand the different steps the given solution performs during its flow. As stated in Section 2.2.2, this also includes the inspection of the raw input data and the final processed output data. In a multi-stage environment, it is also important to consider the required input for the next pipeline stages, in this case, the Sanitization Stage.

4.1.1. Conversion Stage Data Governance Review

Input and output data requirements should be defined by the use case data governance. In the Conversion Stage's case, the input data should consist of one or multiple POSLog XML files (cf. Section 3.3.2). The input data is also expected to be named in a unified format which contains retail branch store information, date information, as well as the identification number of the receipt (cf. Section 3.2.3). The data needs to originate from the `raw` subarea within the specific analytics job area inside the data lake (cf. Figure 3.1) The output data needs to be provided in JSON file format, where one POSLog XML file yields one or multiple JSON POS files, containing global receipt as well as single-purchase information (cf. Section 3.3.2), depending on the number of items from the source POSLog file. Thus, the data values of the individual files do not change but get rather split up. For the output data, a similar unified naming format including the order of purchases from the original POSLog file is also required. The data destination is the `prepared` subarea within the specific analytics job area inside the data lake. This data governance information can now be summarized in Table 4.1:

	Input Data	**Output Data**
Data Format	`XML` with POSLog formatting	`JSON` with defined single-POS formatting
Data Amount	$n > 0$	$m \cdot n \geq n$ where $m > 0$ is the number of individual items for each POSLog file 1 to n
File Naming	`<store-id>_<date>_pos<no>.xml`	`<store-id>_<date>_pos<no>_<sq>.json`
Location Format	`s3://<bucket-name>/<job-id>/raw`	`s3://<bucket-name>/<job-id>/prepared`

Table 4.1.: Conversion Stage Data Governance Specification Summary

The corresponding format examples have already been provided in Source Code Excerpt 1 and Source Code Excerpt 3.

As previously depicted in Figure 3.4, the Conversion Stage runs through the following data-focussed tasks to achieve the conversion:

1. Take the data from the S3 bucket and read its (binary) contents.

2. Map those contents to `OrderedDict`s using Python's `xmltodict` library

3. Remove nesting from the dictionaries by reformatting each `OrderedDict` to multiple, single-POS standard Python dictionaries
4. Export each single-POS dictionary to JSON (binary) format
5. Upload the files to the S3 bucket

Each task is now prone to different sorts of data quality issues, each of those neglecting one or multiple DAMA dimensions of data quality, mentioned in Section 2.2.2: *Completeness* issues appear, when data which was actually saved by the retail register is missing during analysis. Since this project is based on pseudorandomly generated data, this dimension falls out of the scope of the testing framework. A lack of *Uniqueness* happens when the analytics engine processes multiple identical POSLog files. This check should be moved to the testing framework of the next stage since distinction checks are simpler to perform within SQL. In case the data is otherwise valid, *Timeliness* problems can be found when the timestamp of the data file name does not correspond to the timestamp inside the POSLog file. The Conversion Stage is mostly prone to *Validity* issues, including the format of the input and output data, the schema of the corresponding files or specific values inside the files or intermediate transformation results. *Accuracy*, again, falls out of scope since technical issues with a real-life register system are not covered inside this project. *Consistency* in a retail BI solution is a two-edged sword, since product discounts, names, etc. can become updated over time.

For the sake of simplicity during implementation, data timeliness aspects are also treated inside the scope of data validity. For the Conversion Stage, this results in the data quality categories *Data Format Quality*, *Data Schema Quality*, and *Data Value Quality*.

4.1.2. Data Event Definition Process

The various potential data quality issues need to be defined in a unified format, creating a task sheet for implementation. These definitions are referred to as *Data Events* and include the following information:

Event Description Detailed information on the event

Event Category Category of the data event

Severity Extent of influence of the data event

Handing Detailed information on the handling of the data event

The degree of severity is important, since not every data quality issue has the same level of influence on the performance of the system. A corrupted file, which cannot be processed, should have a higher severity, potentially resulting in process termination. On the other hand, an exclusively incorrect file naming could have a minor impact on the analysis, but should not require a process termination. These severities need to be defined and justified individually based on the given use case, but a general classification of severity degrees could be useful. In the following, the severity levels based on a specification by Oracle [33] are used:

INFO An informative message, usually describing task activity. No action is necessary.

WARNING Minor derivation from expected performance. Might cause analytical mistakes when appearing more often.

ERROR Major derivation from expected performance, but still recoverable. Will cause analytical mistakes when appearing more often.

FATAL Severe derivation from expected performance that cannot be recovered from. Causes immediate task termination.

The `WARNING` and `ERROR` levels imply the implementation of thresholds that may reach a maximum appearance value for each level. Since errors are more severe than warnings by logic, the threshold for errors should also be stricter than the one for warnings.

4.1.3. Conversion Stage Input Data Events

The data event definition process is now applied to the Conversion Stage. For redundancy reduction, the following events are chosen to provide the most distinct data event cases. The entire list of data events for the Conversion stage can be found in Appendix A.1.

4.1.3.1. Example A: Data Source Empty

Description	**No data inside the designated data lake area**
Category	Data Format
Severity	`FATAL`
Handling	1. Log error including data lake location information 2. Prompt error information 3. Terminate analytical process

Table 4.2.: Data Event Example A: Data Source Empty

At the beginning of the Conversion Stage, the previously unzipped XML files need to be available for access. If, for any reason, the provided data source is empty, the process needs to terminate immediately, since the next steps (and, ultimately, the MBA itself) could not be performed without any data. Therefore, there is no recovery possible, which is why the process is terminated. The information on the empty S3 Bucket is prompted and logged. This data event mostly applies to the data format issue category.

4.1.3.2. Example B: XML File(s) Corrupted

Description	**One or multiple XML files corrupted**
Category	Data Format
Severity	`ERROR`
Handling	1. Remove corrupted file from analysis 2. Increase `ERROR` degree counter 3. Calculate threshold difference, terminate if exceeded 4. Log error including file information 5. Continue analytical process

Table 4.3.: Data Event Example B: XML File(s) Corrupted

After the data has been recognized, it needs to run through a process of checking the file validity. This does not take the actual content of the XML files into account but rather checks if the content is processable. If, for any reason, one or multiple source files are corrupted, they cannot be processed by means of the following steps. This represents an event with `ERROR` severity. Depending on the volume of source data, this data might be neglected and removed from the current analysis, or the analysis

needs to be terminated because the threshold for `ERROR`-throwing events has been exceeded. This data event also applies to the data format issue category. This event definition can be similarly applied to empty data files or files with incorrect headers (i.e., non-POSLog files) because of their comparable affects on the analytics outcome.

4.1.3.3. Example C: Missing Optional Attributes

Description	**One or multiple XML files is missing *Optional Attributes***
Category	Data Schema
Severity	`WARNING`
Handling	1. Calculate occurrences 2. Increase `WARNING` degree counter accordingly 3. Calculate threshold difference, terminate if exceeded 4. Flag file (`optArg-warn`) 5. Log warning including file information 6. Continue analytical process

Table 4.4.: Data Event Example C: Missing Optional Attributes

After the data has been validated for processing, its content needs to be evaluated as well. Considering the upcoming Sanitization Stage, it requires valid values for its `SequenceNumber`, `ItemCategory`, `ItemID`, as well as the `Quantity` attribute (cf. Source Code Excerpt 2), hereinafter referred to as *Required Attributes*. All remaining attributes (i.e., *Optional Attributes*) are not relevant for the analysis but are still desired to be valid to rule out any unforeseen performance issues. Since the data is still processable by means of the current and upcoming pipeline stages, this data event is classified to have a rather minor impact on the overall outcome. Nevertheless, it should not occur too often since this could point to other issues inside the source data. Therefore, the `WARNING` event severity is applied here. In case its threshold is exceeded, the analysis will be terminated, similar to the `ERROR` event threshold. This data event applies to the data schema issue category. This event definition can be similarly applied to a *data naming* or *data extension* issue since it has a similarly low degree of severity for the analytics outcome.

Each further data event should be defined in such a manner that the severity of it is evaluated based on the data governance which can then propose handling measures for the implementation. It is also important to consider the order of the performed

data quality checks. For example, checking for required arguments inside a file cannot happen before a data corruption check.

4.1.4. Conversion Stage Transformation and Output Data Events

The same procedure needs to be applied for transformation and output data within the specific stage. This might include similar or different threshold and event severity definitions. In the case of the Conversion Stage, the transformation of the data is purely focussed on its formatting. This means that the outcome for each further step is deterministic. If an error occurs during the transformation, this implies software-side issues within the solution that need to be tested and ruled out individually. From a data perspective, the input data has been validated prior to the first transformation step and, thus, cannot be the reason for incorrect, further performance.

This means that all upcoming data events resulting in errors during transformation or within output data need to be treated with `FATAL` severity level, resulting in instant termination once such an error occurs. This measure can help detect an underlying source code problem in the transformation steps during analysis.

In other cases (e.g. ML model training, which is not deterministic), the previous paragraph does not apply. Based on the use case and its circumstances, the handling of errors of all sorts need to be evaluated and defined appropriately.

4.2. Test Data Design

The next step is to create applicable test data suites in accordance to the data event definitions. During the DataOps Testing process, these data will be ingested into the analytics stage, determining if the data events are recognized and handled by the solution as expected. In general, the test data should remain as close to real production-grade data as possible, which also means the inclusion of production-grade values inside the POSLog file (i.e., actual product descriptions, prices, realistic quantities, etc.).

4.2.1. Single-Event Test Data

At first, it might be reasonable to check if all data event handling works individually. This means considering each defined data event as a single test case. The preparation of applicable test data can also be seen as the intentional violation of the data governance for each data event.

Remaining inside the Conversion Stage, Example A (cf. Section 4.1.3.1) should cover the occurrence of an empty data source. This can be tested by providing an empty job sub-area within a testing environment inside the data lake. It is expected that this case will trigger the event handling, resulting in process termination. For Example B (cf. Section 4.1.3.2), the appropriate trigger is a (binarily) corrupted XML file. The solution is expected to check for content validity. Such a corrupted data piece is expected to be removed from the current (test) analysis process. Finally, Example C (cf. Section 4.1.3.3) requires a valid POSLog XML file with an arbitrary missing optional attribute. Inside this stage of testing, it is important that this is the only issue inside the test data file since the individual data event handling performance should be evaluated first. Otherwise, another data event might unexpectedly be triggered. A missing optional attribute is expected to be flagged by the system, to log the warning, to calculate the threshold tolerance, etc.

It might also be useful to provide a generally clean and valid data piece in order to check that no issue event is triggered when no issues are present.

4.2.2. Multi-Event Test Data

Increasing complexity, the next step in defining test data is to combine several data issues in a single file. Based on an expected (i.e., desired) outcome, this might be useful for checking if the order of data validity checks is kept.

Inside the Conversion Stage, this might be the combination of missing both required and optional attributes. In case a required attribute is missing, the data file should be taken out of the analysis (cf. Appendix A.1). The missing optional attribute will only trigger a data flag. If, for any reason, the data is not removed, this might point to misimplementation of the data handling mechanisms. Still, each test data piece is considered its own test case.

4.2.3. Test Data Suites

After the individual test cases have been run through, it might be reasonable to include a (close-to) production-grade scenario inside the testing framework. This means that a testing suite of multiple test data files is created. The test data should be designed in a manner that it contains both correct and incorrect data. The incorrect data within the suite should also include different degrees of faultiness. This is expected to enable testing of the thresholding capability of the solution as well as general handling performance when dealing with multiple data files.

Given the Conversion Stage, approx. half of the data could be correct and valid. One fourth of the data might contain minor issues (e.g., incorrect file naming, missing optional arguments, etc.). The remaining fourth of the data might contain major issues (e.g., corrupted or empty files, missing required arguments, etc.). The predefined threshold values should be taken into account when designing such a test data suite.

4.3. DataOps Solution Testing Design

The final step of the DataOps Testing framework design is to validate the functionality of the data event handling and the general solution by means of automated tests through the application of the test data suites. This makes use of the classical software testing paradigm (cf. Section 2.2.3):

- Unit tests will cover all functionality outside of the data scope of the solution. This might include the setting of correct environment variables within the solution, validation of required credentials, etc.

- Integration tests will be used for the data handling validation. This includes the integration of the external data lake, containing the test data suites. On the one hand, the tests will ensure that the appropriate data handling measures are taken at the correct position within the source code. On the other hand, the tests will also validate that the data handling measures actually conform to the data event definitions.

- End-to-end tests will be built on top of the integration tests, running through the entire analysis process and taking all external infrastructure into account.

4.3.1. Conversion Stage Testing Architecture

4.3.1.1. Conversion Stage Unit Testing

Inside the Conversion Stage, Unit Testing does not play a major role. This is because the majority of tasks performed by the stage are data-orientated. Prior to the analytics performance, the stage evaluates the environment variables of the system that are expected to contain the Uniform Resource Identifier (URI) paths to the data source and data destination within the S3 data lake. This process is subject to unit testing.

4.3.1.2. Conversion Stage Integration (Data) Testing

Integration tests play the most important role inside the testing framework. Since all data (production-grade data as well as test data suites) reside on the external S3 data lake, this needs to be taken into account during testing. Integration testing within this solution can be divided into two parts: First, each data event is evaluated individually based on its predefined single-event test data (cf. Section 4.2.1). In this part, the tests validate that each data issue is *recognized* and *handled* in its appropriate context. When a corrupted XML file is provided, this needs to be recognized (i.e., logged and reported) as well as handled (i.e., corrupted data is removed from analysis).

When this validation is conducted successfully, the multi-event test data (cf. Section 4.2.2 can be induced for similar test cases. At this point, the general functionality of the individual data handling measures can be expected. Finally, the test data suite(s) (cf. Section 4.2.3) are used to test the thresholding capability of the solution, which is not testable with single-file test cases.

4.3.1.3. Conversion Stage End-To-End Testing

The final testing stage inside the Conversion Stage should also take all remaining infrastructure into account. In this case, this includes Airflow, which should be triggered by the end-to-end test and receive the test data suites. Airflow runs through the entire data pipeline and reports its production-like outcome which is then evaluated by the end-to-end test. This will ensure that no dependencies have been left out during testing.

4.3.2. Application of DataOps Testing

The testing framework is applied whenever a new version of the analytics solution is about to be deployed into production. In case a new feature is developed or an existing feature is updated or removed, the tests ensure that the core analytics capacity is maintained. Without these tests, version dependencies or classical software bugs could be deployed into production, resulting in analytics result issues or unresolvable crashes of the system. In case one of the test cases does not pass, the deployment is not continued which allows the correction of the underlying process. This procedure is repeated until all tests have passed. In case of the development of entirely new features or data handling measurements, applicable test cases need to be created to validate these features, as well. This includes regression testing, taking requirements from other stages into account.

Again, this testing framework design process needs to be individually applied to the context of the given use case. Other solutions might require more extensive unit testing or have a more complex end-to-end testing structure. These requirements need to be evaluated prior to design and implementation.

The testing implementation process itself is even more individual. This is because different ways of implementation might lead to the same outcome. The implementation of the testing framework for the Conversion Stage will be described in the following chapter.

5. Implementation

This chapter accompanies the process of practical implementation of the target solution. It aims to enhance the current solution by means of the task definition (cf. Section 1.3) and the findings from the actual state analysis (cf. Chapter 3). Specifically, three main aspects are expected to be added to the solution after implementation, being

1. the decapsulation of the analytical scripts from the Apache Airflow instance,
2. the inclusion of technical DataOps standards, specifically IAM, VCS, CI/CD and IaC, and finally
3. the implementation of the DataOps testing framework (cf. Chapter 4).

The order of documentation does not necessarily reflect the implementation sequence of the project. Since this thesis can also be seen as a guideline for future DataOps projects, the implementation steps are provided in order to build on one another in the most useful way. The first two implementation tasks are performed for the entire data pipeline, while DataOps testing is exemplarily implemented for the Conversion Stage.

5.1. Server-Less Architecture Enablement

In order to achieve the goal of a (close-to) server-less architecture, the current architecture of the Value Pipeline needs to be revisited. Currently, the analysis is directly performed on the Airflow server instance (cf. Section 3.3). Instead, it is desired that each analytical step can be performed and configured independently while also complying to the server-less philosophy. This results in a high-level design change depicted in Figure 5.1.

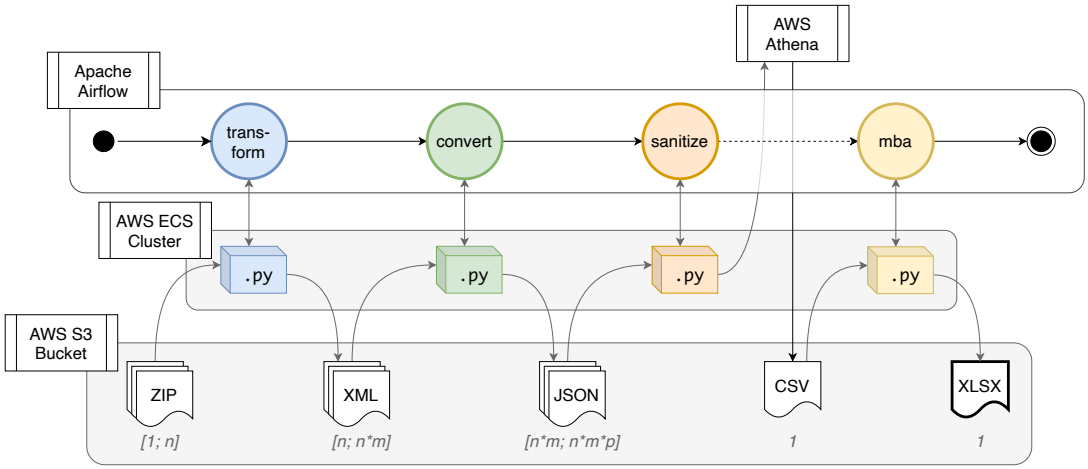

Figure 5.1.: Revisited Data Pipeline Architecture (Server-Less)

As visualized above, the new architecture outsources the Python analysis scripts outside of the Airflow instance. Instead, they new create individual microservices, deployed as virtual containers. These containers, typically realized with the *Docker* containerization software [34], contain all required dependencies to run their services. AWS' *Elastic Container Service (ECS)* can be used for the server-less deployment and execution of these container tasks [35]. With this architecture change, Airflow only remains an orchestration tool that starts the appropriate container tasks. The tasks report their statuses back to Airflow, resulting in Airflow either triggering the next service or terminating the process if an error occurs.

5.1.1. Microservice Containerization

In microservice virtualization, containers are the final desired product. The goal is to have a sequence of commands that encapsulate the Python program code of each individual stage inside their own containers, considering individual stage requirements. To achieve this goal, the intermediate creation of a so-called *image* is required, which provides basic components for making the service runnable [34]. Before running the container, specific configurations after the image has been defined.

Docker images are defined by means of a *Dockerfile* [34]. The tasks inside this file are sequentially executed when creating the image. Exemplarily, the Dockerfile of the Conversion Stage can be seen below.

```
1  FROM python:3.7-alpine
2  WORKDIR /usr/src/app
3
4  ARG aws_id
5  ARG aws_key
6  ARG data_src
7  ARG data_dst
8  ENV AWS_ACCESS_KEY_ID=$aws_id
9  ENV AWS_SECRET_ACCESS_KEY=$aws_key
10 ENV DATA_SOURCE=$data_src
11 ENV DATA_DESTINATION=$data_dst
12
13 ENV PYTHONPATH=.:./convert
14 RUN echo $PYTHONPATH
15
16 ADD . .
17
18 RUN pip install -r ./requirements.txt
19 RUN pip install -r ./test/test-requirements.txt
```

Source Code Excerpt 4: Dockerfile of the Conversion Stage

As can be seen in Source Code Excerpt 4, the image definition begins with referencing the base image (l. 1). This contains a slimmed-down operating system with the capability of running Python. The Dockerfile also specifies the working directory (l. 2) and prepares environment variables (ll. 4–11) for the container that contain AWS credentials as well as designated S3 path URIs for input and output data locations. The credentials are required for the service to perform its steps on the designated AWS infrastructure. These environment variables will receive their values during the build process of the container (i.e., when executing the image). Finally, the local package directory is added to the working directory of the image (l. 16) and all required Python dependencies (for both running and testing) are installed via the `pip` Python package manager (ll. 18–19). These dependencies are specified in the applicable `requirements.txt` files inside the package directory.

The environment variables will prove valuable during DataOps enablement. By defining granular AWS credentials, each stage is only authorized to read and write to their designated S3 data lake subareas. This creates separated environments for each development stage.

The image can now be build using the `docker build` command. The design of the Dockerfile requires that the environment variables are passed before container execution (i.e., via arguments in the `docker build` or `docker run` command). The image receives a unique tag and the Dockerfile is referenced for correct handling of relative paths. Finally, the image can be used for running the container. This is

achieved using the `docker run` command, specifying which command to run inside the container (e.g., for the Conversion Stage, `python ./convert.py` for executing the analytics script). This runs the container inside the local system.

5.1.2. Image Deployment

Instead of the container running locally, it is desired to execute it inside the serverless ECS. This requires the preliminary deployment to an associated AWS service, the Elastic Container Registry (ECR). ECR is an image repository that is designed to hold different versions of an image [36]. The image deployment and container execution strategy is depicted in Figure 5.2 below.

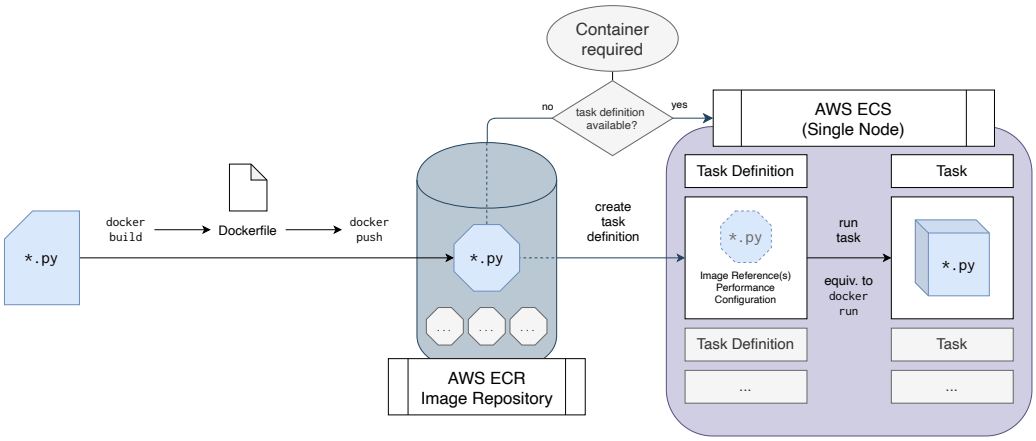

Figure 5.2.: Microservice Deployment Process

Following the flow of the figure, the image is build by means of its Dockerfile and pushed to the designated ECR. The octagon shape is used to describe an image. The execution requires another preliminary step which is the task definition of the container. This is because ECS can also perform container execution sequences, periodic repetitions of the task, etc. Plus, each task can have different performance requirements. The task definition allows for individual hardware specification (e.g., processing power, memory, etc.) [35]. Thus, the task definition can be seen as a blueprint for the (more or less complex) container service. For the data pipeline stages, a task with basic performance specifications as well as one image reference for running the applicable container once, suffices. The task can now be run based on its task definition, providing information on which Python script to run. This executes the task and tears down the container after the task has been worked through or an error has occurred. When the container service is required again and the image repository has not changed, the previous task definition can be directly executed.

5.1.3. Server-Less Container Execution via Airflow

Returning to the high-level overview, creating and running the task definition is performed by Airflow through the AWS Python API, `boto3`. The Airflow DAG controller is redesigned to constantly evaluate the state of ECR and ECS. It checks for images inside each stage's image repository and builds the applicable task definitions. When all task definitions of all four pipeline stages are present, the Airflow DAG is ready for execution. The only task inside each individual step of the DAG is to execute the task definition and evaluating the container response.

Because of the periodic review, the most recent images and task definitions need to be saved inside Airflow variables to prevent the system to constantly recreate the same task definitions, resulting in an endless loop and, thus, in infrastructure overload.

In case a new version for a stage is recognized, a new task definition is created and re-referenced inside the DAG, enabling automatic updating of the pipeline solution.

5.2. DataOps Enablement

The next step is to include DataOps-related technologies to the project. They have mostly to do with general solution automation and environment management. Specifically, the following aspects are implemented and configured:

1. Enhanced permission management via AWS Identity and Access Management (IAM),

2. Version Control System (VCS) by using *Git* and a *GitHub* repository,

3. Continuous Integration & Deployment (CI/CD) by using the *Jenkins* automation software, and

4. Infrastructure as Code (IaC) by using the *Terraform* infrastructure automation tool as well as the *Ansible* infrastructure provisioning software.

These processes will support and build on each other during development for both automation and environment management disciplines.

5.2.1. Enhanced Permission Management: AWS IAM Roles

AWS IAM roles are used to provide different access permissions to different users and resources. If a user or resource tries to perform actions on AWS services (e.g., via CLI or API), the system checks if the required credentials are present to perform this action. The credentials are provided in the form of asynchronous encryption, resulting in a public *Access Key ID* and in a private *Secret Access Key* [37].

Because of the static nature of the current solution, only one full-access IAM role is used. This is not desirable from multiple perspectives. Each developer should be provided an IAM role that is sufficient for his or her needs. The underlying permissions should not exceed these needs in order to prevent collision or changes within AWS resources outside of the scope of the developer. The same aspect applies to resources that automatically perform actions on other resources. Another important factor is data privacy. In a production-grade environment with sensitive data, it is crucial and binding by law that the data is only seen and processed by authorized entities.

For instance, if a data pipeline stage is misconfigured and tries to read from another area of the S3 data lake, the process will terminate when the given IAM role prohibits this action. Otherwise, the stage is granted access to data that is not required for its task. In that case, correctly configured IAM roles can also prevent errors in case the content of the incorrect S3 area cannot be processed by the current analytics stage.

Considering the MBA data pipeline, the present variety of resources requires different permissions on a number of other different resources. Table 5.1 below describes all permissions required by different components.

Component	Resource	Type	Scope
Airflow	ECR	Full Access	designated stage image repositories
	ECS	Full Access	designated container cluster
Stages			
Transformation	S3	Read	`prod/landing/`
	S3	Write	`prod/**/raw/`
Conversion	S3	Read	`prod/**/raw/`
	S3	Write	`prod/**/converted/`
Sanitization	Athena	Full Access	designated database
	S3	Read	`prod/**/converted/`
	S3	Write	`prod/**/sanitized/`
MBA	S3	Read	`prod/**/sanitized/`
	S3	Write	`prod/**/`

Table 5.1.: AWS IAM Role Overview

These roles are setup within the AWS IAM Console and passed to the individual components as environment variables [37], depending on their individual deployment.

5.2.2. VCS Enablement: *Git* and *GitHub*

Embedding the project inside a VCS allows for collaborative development. When working with such a system, the common codebase is located inside a distributed repository. This repository can be cloned (i.e., copied) to the developers local development environment. *Committing* (i.e., performing and saving changes of the personal version of ones source code) does not collide with the source code versions of other developers [38, pp. 9 sqq.]. Plus, since the project at hand relies on external programming and developing dependencies (e.g., Python packages and libraries, environment configuration processes, etc.), the repository can also contain configuration utilities for setting up the developing sandbox.

5.2.2.1. Repository Structure

A cloned repository is a typical project directory. The repository for the MBA data pipeline solution looks as follows:

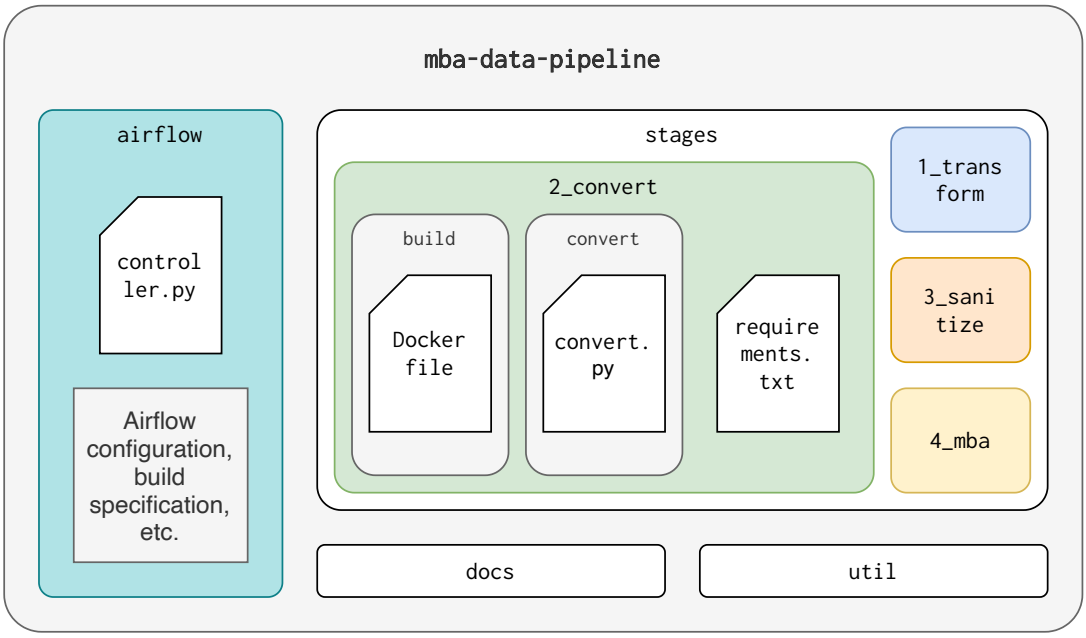

Figure 5.3.: Initial Project Repository Structure

The repository presented in Figure 5.3 is a so-called *monolithic* repository. Since the project is expected to be developed by separate stage teams, this repository

de facto consists of multiple projects. Starting at the root of the directory, the repository contains subdirectories for Airflow pipeline management, the individual stages of the data pipeline, documentation as well as miscellaneous utilities. The Airflow pipeline management consists of the pipeline controller script, as well as supporting build files that are outside the scope of this thesis. More importantly, the individual stages are, again, divided into subdirectories. Each stage has its own build directory (currently including its Dockerfile), a package directory (currently including all productive source code), as well as dedicated requirements files required by the source code).

5.2.2.2. Repository Setup

The goal is to create a GitHub repository for the MBA data pipeline project. First, the project needs to become a Git repository, enabling the version control capability. GitHub is used for the distribution and collaborative management of the code base [38, pp. 25 sqq.][39]. First, the project directory needs to be initialized by means of Git. Then, the repository can be created within the GitHub web UI. Lastly, the local Git repository is pushed to the recently created GitHub repository.

In order to allow for the automatic configuration of development sandboxes, each stage directory receives a *Makefile* that creates shortcuts for configuration commands. This includes local building and executing of a Docker container. Other development conventions (e.g., for Python, that development should be conducted in Python *virtual environments*), cannot be enforced but only suggested within the project documentation.

5.2.2.3. Branching Strategy

Currently, an authorized developer is able to clone this repository, make any kind of changes, and push the changes back to the repository. This kind of workflow is not desired which is why *branching* is introduced. Branching is a GitHub feature that supports development environment management within the repository [38, pp. 62 sqq.]. At this point, only one branch, the `master` branch, exists. It holds the history of all versions of the source code. These versions change when updated code is pushed to the brach. In practice, the `master` branch is supposed to be the communal *single version of the truth*. It is expected to be runnable and should reflect the source code of the solution which is currently used in production. When changes of all sorts are possible inside the `master` branch, the validity of it gets lost. This is why this branch

needs to be protected such that only evaluated changes can be pushed to this area of the repository.

Branching is a matter of project convention definition. The general goal is that ongoing development of an arbitrary feature is conducted outside of the `master` branch (i.e., on separated feature branches). Each (sub-)team working on an individual feature *branches* from the `master` branch, resulting in an exact copy of its origin in the beginning. All changes pushed to the new branch do not affect the *master* brach. When the feature is considered done, a GitHub *pull request* is created [39]. This pull request is evaluated based on its configuration. If the evaluation passes, the feature branch gets *merged* with the `master` branch, meaning that all changes are applied to the source code inside the `master` branch at once. Finally, the feature branch is deleted. From now on, the updated `master` branch is used as the starting point for further development. This branching workflow is commonly referred to as the *GitHub Flow* [40], depicted in Figure 5.4, below.

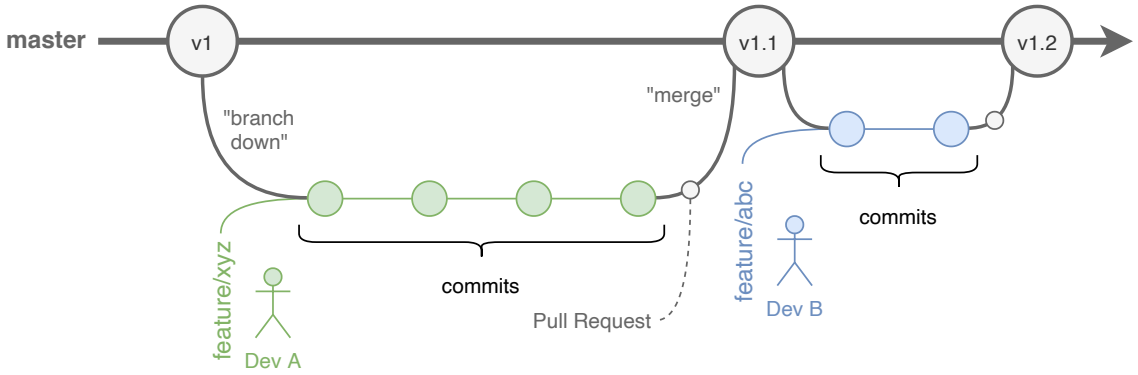

Figure 5.4.: *GitHub Flow* Branching Strategy (per [40])

The branching strategy, as presented above, is typical for a fast-paced development cycle. Each approved change, no matter how small, is directly published into production. Apart from that, other branching strategies could be applied. It might be reasonable to include separate `development` and `release` branches. The `development` branch could become the starting point for new features and could contain features that have not been published yet. In case of a slower release cycle, the development branch could be merged with the *release* branch for compatibility testing purposes and deployed into production (i.e., merged with the `master` branch) afterwards. Another `hotfix` branch might be used for quick bug fixes that branches from the *master* branch, fixes the bug, and merges back into production [41].

For the sake of simplicity and demonstrability, the previously described branching strategy is chosen for the MBA data pipeline. The `master` branch is configured inside GitHub to only accept pull requests for incoming changes, not direct pushed

onto the `master` branch. Later, the pull request functionality will be enriched with CI/CD. In case the process runs through without errors, the changes are evaluated positively and cleared for the merging process.

5.2.3. CI/CD Enablement: *Jenkins*

CI/CD is expected to enhance and automate two aspects inside the development process of the MBA data pipeline solution:

1. It handles the build and deployment of virtual Docker images and containers by means of Figure 5.2. This also includes providing each task with its designated environment variables for infrastructure access as well as analysis data input and output locations.

2. It automates the execution and reporting of test cases, only allowing for approved deployment of tested versions of the solution. This process supports the previously mentioned pull requests which will fail when tests are evaluated negatively.

The latter aspect will be implemented in Section 5.3.

The desired DataOps CI/CD workflow is depicted in Figure 5.5. It considers testing a black box for now. It is noteworthy to mention that each stage of the data pipeline could have different requirements for its deployment, which is why each stage receives its own CI/CD pipeline. This becomes clear when considering testing since each stage performs different tasks and requires different test cases.

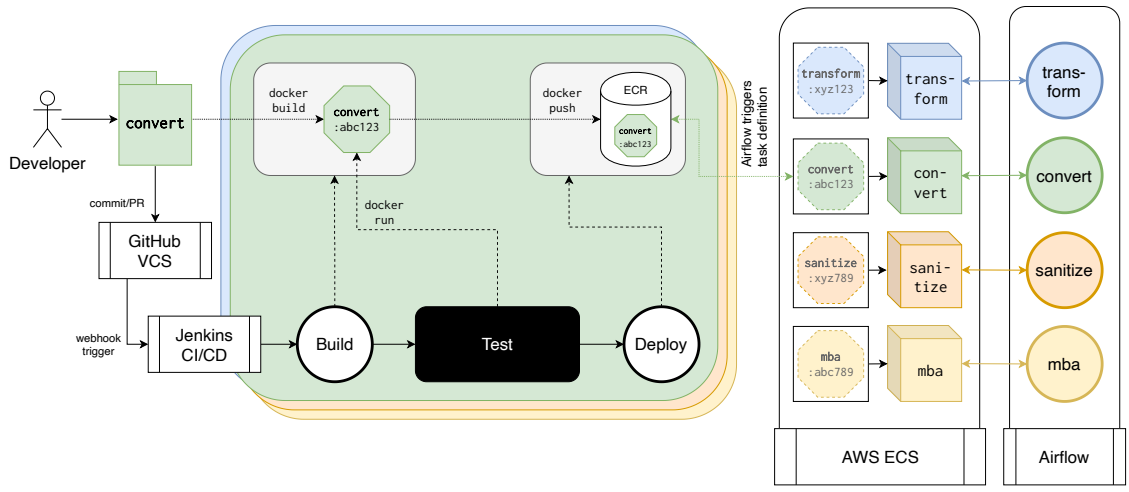

Figure 5.5.: DataOps CI/CD Architecture for the Conversion Stage

Figure 5.5 represents the CI/CD workflow of the Conversion Stage. It is meant to be executed whenever a GitHub pull request of a Convert Stage-related feature is opened. This pull request appearance should trigger the corresponding pipeline, realized with the *Jenkins* CI/CD software [42]. The first step needs to build a virtual Docker image using the target source code files from the pull request. Again, this build process is done by means of the stage's specific Dockerfile and requires additional arguments for the containers environment variables. When the image is built, the build stage of the CI/CD pipeline is completed. Later, this this image will be run on the Jenkins instance for different testing purposes. After these tests pass, the image is ready for deployment. Airflow expects runnable images inside their designated ECR repositories which is why the image, individually tagged, is pushed to its repository. The deployment process is done at this point. When Airflow needs to run a new analysis, the controller script will scan the registry for the latest image, recognize a new image, create the applicable task definition and run the container based on the newly deployed image. Every further analysis will be done utilizing this image until a new image is deployed again.

5.2.3.1. Jenkins Software and Pipeline Setup

Comparable to Apache Airflow, Jenkins needs to be considered as an external application that needs to reside on some kind of infrastructure. For efficiency reasons, the EC2 instance already running Airflow also receives Jenkins. Both web servers now run simultaneously on different ports for HTTP access.

Jenkins provides different blueprints for CI/CD pipelines. The DataOps CI/CD pipeline of the MBA data pipeline needs to recognize pull requests that originate from different feature branches within GitHub. This is why the *Multibranch Pipeline* is chosen [42]. Such a pipeline is created for each of the four stages of the data pipeline. The branch needs to be granted access to the GitHub repository. Since the project is working with a monolithic repository, each CI/CD pipeline needs to be configured in such a way that it only recognized changes and pull requests to specific areas of the repository.

There are multiple ways to achieve this configuration. Jenkins distinguished between branches and pull requests, meaning that a branch with a pending pull request is listed and evaluated by Jenkins *twice*. Jenkins is therefore configured to disregard branches that are also filed as pull requests. The only non-pull request branch should be the `master` brach since all other branches are expected to be in development if no pull request is present. Thus, Jenkins should only consider `master` a long-lasting branch. The pipeline should only consider stage-specific changes, which is why Jenkins should

filter each pull request by its GitHub label. This requires the developer to add the specific labels when filing a pull request. Finally, Jenkins should automatically run a CI/CD job inside the corresponding pipeline when a change is recognized. This needs to be enabled in both Jenkins and GitHub. GitHub needs to send a `POST` request to Jenkins when a pull request is filed, whereas Jenkins needs to trigger the corresponding pipeline on `POST` request arrival.

5.2.3.2. Jenkins Credential Management

The virtual microservices need to receive their AWS IAM access credentials during their build process. Since this process is expected to be covered by CI/CD, Jenkins needs to hold these credentials and pass these to the corresponding stage containers inside its CI/CD pipeline workflow. This is achieved by adding the individual credentials to Jenkins' encrypted credential storage [42]. These need to include all stage-related credential key-value pairs that provide appropriate access to the analytics resources as described in Table 5.1. Additionally, Jenkins needs to be granted permission to push images to the ECR repositories. The corresponding credential should not be mistaken with a separate IAM role. Rather, Jenkins needs to hold a login token for ECR that is retrieved via AWS CLI [36]. This process is identical to other Docker registry services, independent from vendor.

Jenkins will then pass the encrypted credentials to the corresponding pipelines [36], resulting in no hard-coded credentials in any configuration file.

5.2.3.3. Pipeline Workflow Declaration

After the pipeline preferences have been configured, the actual pipeline workflow needs to be declared for each stage. In Jenkins, this is done via a so-called *Jenkinsfile*. It is written in *Groovy* programming language and contains instructions for each stage of the pipeline [42]. A sample Jenkinsfile for the Conversion Stage is shown below.

```
1  pipeline {
2    /* Preamble omitted */
3
4    stages {
5      stage('Build') {
6        steps {
7          sh 'echo Building "convert" stage image...'
8          withCredentials([usernamePassword(
9            credentialsId: 'mba-pipeline_conversion',
10           passwordVariable: 'AWS_KEY',
11           usernameVariable: 'AWS_ID')]) {
12           sh "
13             docker build -t <aws-user-id>.dkr.ecr.eu-central-1.amazonaws.com/
14                                  conversion:${GIT_COMMIT_SHORT}
15                           --build-arg aws_id=$AWS_ID
16                           --build-arg aws_key=$AWS_KEY
17                           -f stages/2_convert/build/Dockerfile
18                           stages/2_convert
19           "
20         }
21       }
22     }
23
24     /* Test stages... */
25     /* Deploy stage... */
26   }
27 }
```

Source Code Excerpt 5: Jenkinsfile Build Stage for the Conversion Stage

Source Code Excerpt 5 shows the build stage inside the Jenkinsfile. It retrieves the corresponding stage credentials from Jenkins' credential storage (ll. 8–11) and executes the build process of the Docker image (ll. 12–19). The image needs to be tagged in an AWS-provided ECR format such that it is assigned to the correct repository during the deployment phase. This tag includes the corresponding Git commit ID for versioning purposes (l. 14) which is passed to Jenkins by the previously mentioned GitHub `POST` request trigger. Prior to the build, the arguments for the AWS IAM credential pair are passed (ll. 15–16) and set as environment variables by means of the Dockerfile specification. Finally, the Dockerfile (l. 17) and the build context for relative path recognition (l. 18) are specified. This command is executed by the Jenkins CI/CD pipeline job. In case of a correct execution, the next stage is performed. Since testing is omitted for now, it directly continues with the deployment, shown below.

```
1  pipeline {
2    /* Preamble omitted */
3
4    stages {
5      /* Build stage... */
6      /* Test stages... */
7
8      stage('Deploy') {
9        steps {
10         script {
11           docker.withRegistry(
12             'https://<aws-user-id>.dkr.ecr.eu-central-1.amazonaws.com',
13             'ecr:eu-central-1:aws') {
14             sh "docker push <aws-user-id>.dkr.ecr.eu-central-1.amazonaws.com/
15                   conversion:${GIT_COMMIT_SHORT}"
16             sh "docker push <aws-user-id>.dkr.ecr.eu-central-1.amazonaws.com/
17                   conversion:latest"
18           }
19         }
20       }
21     }
22   }
23 }
```

Source Code Excerpt 6: Jenkinsfile Deployment Stage for the Conversion Stage

Source Code Excerpt 6 shows the deployment stage inside the Jenkinsfile. It retrieves the ECR credentials (ll. 11–13) and performs the image deployment via `docker push` (ll. 14–17). This action is performed for two tags: the commit ID for versioning and the conventional `latest` tag such that Airflow recognizes it as the most recent, production-ready image.

All in all, the current CI/CD solution builds and deploys a new version into production when a feature pull request is filed to GitHub. Airflow recognizes the change based on its controller script and uses the new image for the corresponding stage for container execution and evaluation.

5.2.4. IaC Enablement: *Terraform* and *Ansible*

Another form of automation that is part of DataOps is automatic infrastructure deployment via IaC. The project at hand requires a variety of different AWS infrastructural resources, including the effort of configuration. In case that the infrastructure needs to be setup inside a different AWS account or it breaks during development, it is not desirable to be forced to start building up the infrastructure from scratch. Instead, the following IaC architecture is proposed.

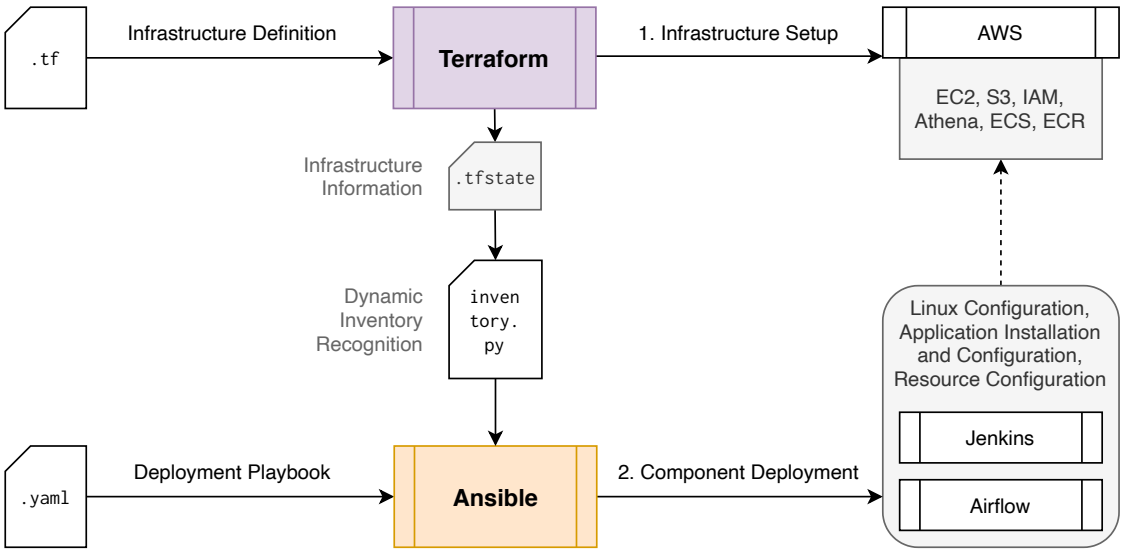

Figure 5.6.: IaC Infrastructure Deployment Strategy

The IaC architecture, visualized in Figure 5.6, is made out of two steps. First, all required AWS services are defined for the *Terraform* infrastructure provisioning tool. This also includes basic configuration of networking and access management. In general, all configuration that can be done via the web UI-based AWS Console or the AWS CLI or API can be done via Terraform. Terraform requires an individual IAM role that allows it to create the infrastructure. When the deployment script is performed via the Terraform CLI, it creates a `.tfstate` file in JSON format [43], summarizing all created infrastructure including public IP addresses, infrastructure IDs, etc.

This infrastructure needs to be configured via the *Ansible* infrastructure orchestration software. This includes the Linux configuration of the EC2 instance, the creation of S3 data lake (sub-)areas, etc. Basically, the goal is to run the corresponding commands for Terraform and Ansible, resulting in an up-and-running infrastructure, ready for production-grade usage.

Ansible requires an intermediate step before being able to perform actions on the infrastructure, which is the recognition of the resource inventory. This can be done manually, which is not desired, or via a dynamic inventory recognition script. This makes use of the previously generated `.tfstate` file and provides Ansible with information and access to the respective infrastructure. Then, so-called Ansible *Playbooks* are written and executed. These hold tasks for the configuration, platform and application installation, etc. [44]. Most importantly, Ansible is in charge of installing and configuring Airflow and Jenkins. Airflow is installed and receives its controller script as well as further configuration from the project repository. Jenkins is installed,

pre-configured, and required plugins are installed.

Unfortunately, these applications do not provide all necessary Ansible endpoints. This means that not all configuration steps can be done automatically but require manual adjustment. This includes Jenkins credential management, pipeline creation, etc. All in all, the infrastructure automation is valuable nonetheless since the manual setup process is significantly reduced.

5.3. Testing Framework Implementation

6. Solution Evaluation

7. Conclusion

Bibliography

[1] J. G. Schmidt and K. Basu, *DataOps – The Authorative Edition*. Austin, TX: Panther Publishing, 2019, ISBN: 978-0-980-21694-3.

[2] C. Bergh, G. Beghiat, and E. Strod, *The DataOps Cookbook*, 2nd ed. Cambridge, MA: DataKitchen, 2019.

[3] J. Lockner, "What is DataOps?", *IBM Big Data Analytics Hub*, Dec. 2019. [Online]. Available: `https://www.ibmbigdatahub.com/blog/what-dataops` (visited on 07/15/2020).

[4] M. Aslett, "DataOps: a passing buzzword, or the engine of the data-driven enterprise?", 451 Research, Tech. Rep., Aug. 2018.

[5] S. Knoth, "Statistical Process Control", *European University Viadrina Frankfurt (Oder)*, Jan. 2002. DOI: `10.1007/978-3-662-05021-7_11`.

[6] "Add DataOps Tests for Error-Free Analytics", *DataKitchen*, Apr. 2020. [Online]. Available: `https://medium.com/data-ops/add-dataops-tests-for-error-free-analytics-741ee48bd5cc` (visited on 07/20/2020).

[7] T. C. Redman, "To Improve Data Quality, Start at the Source", *Harvard Business Review*, Feb. 2020. [Online]. Available: `https://hbr.org/2020/02/to-improve-data-quality-start-at-the-source` (visited on 07/15/2020).

[8] A. K. Kaiser, "Reinventing ITIL® in the Age of DevOps", in. Berkeley, CA: Apress, 2018, ch. Introduction to DevOps, pp. 1–35.

[9] A. L. Davis, "Version Control", in *Modern Programming Made Easy: Using Java, Scala, Groovy, and JavaScript*. Berkeley, CA: Apress, 2020, pp. 127–130, ISBN: 978-1-4842-5569-8. DOI: `10.1007/978-1-4842-5569-8_16`.

[10] R. Chaganti, "Pro PowerShell Desired State Configuration", in, 2nd ed. Bengaluru, India: Apress, 2018, ch. Introduction to Infrastructure as Code and PowerShell DSC, pp. 3–11. DOI: `10.1007/978-1-4842-3483-9`.

[11] Munawar, N. Salim, and R. Ibrahim, "Towards Data Quality into the Data Warehouse Development", in *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, 2011, pp. 1199–1206.

[12] M. Souibgui, F. Atigui, S. Zammali, S. Cherfi, and S. B. Yahia, "Data quality in ETL process: A preliminary study", *Procedia Computer Science*, vol. 159, pp. 676–687, 2019, Knowledge-Based and Intelligent Information Engineering Systems: Proceedings of the 23rd International Conference KES2019, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2019.09.223.

[13] BI-Survey.com, *Data Quality and Master Data Management: How to Improve Your Data Quality*. [Online]. Available: https://bi-survey.com/data-quality-master-data-management (visited on 07/15/2020).

[14] J. Freudiger, S. Rane, A. E. Brito, and E. Uzun, "Privacy Preserving Data Quality Assessment for High-Fidelity Data Sharing", in *Proceedings of the 2014 ACM Workshop on Information Sharing Collaborative Security*, ser. WISCS '14, New York, NY: Association for Computing Machinery, 2014, pp. 21–29. DOI: 10.1145/2663876.2663885. [Online]. Available: 11.

[15] T. C. Redman, "Assess Whether You Have a Data Quality Problem", *Harvard Business Review*, Jul. 2016. [Online]. Available: https://hbr.org/2016/07/assess-whether-you-have-a-data-quality-problem (visited on 07/15/2020).

[16] G. O'Regan, "Software Testing", in *Concise Guide to Software Engineering: From Fundamentals to Application Methods*. Cham: Springer, 2017, pp. 105–121, ISBN: 978-3-319-57750-0. DOI: 10.1007/978-3-319-57750-0_7.

[17] N. Askham, D. Cook, M. Doyle, H. Fereday, M. Gibson, U. Landbeck, R. Lee, C. Maynard, G. Palmer, and J. Schwarzenbach, "Defining Data Quality Dimensions", *DAMA UK*, Oct. 2013. [Online]. Available: https://www.whitepapers.em360tech.com/wp-content/files_mf/1407250286DAMAUKDQDimensionsWhitePaperR37.pdf (visited on 05/17/2020).

[18] S. Shen, "7 Steps to Ensure and Sustain Data Quality", *Towards Data Science*, Jul. 2019. [Online]. Available: https://towardsdatascience.com/7-steps-to-ensure-and-sustain-data-quality-3c0040591366 (visited on 07/15/2020).

[19] I. Schieferdecker, "(Open) Data Quality", in *2012 IEEE 36th Annual Computer Software and Applications Conference*, 2012, pp. 83–84.

[20] R. Osherove, "The Art of Unit Testing", in, 2nd ed. Shelter Island, NY: Manning Publications, 2013, ch. The basics of unit testing.

[21] A. S. Mahfuz, "Software Quality Assurance", in. Boca Raton, FL: CRC Press, 2016, ch. Testing, pp. 59–71, ISBN: 978-1-4987-3555-1. [Online]. Available: https://learning.oreilly.com/library/view/software-quality-assurance/9781498735551 (visited on 07/17/2020).

[22] A. Tarlinder, "Developer Testing: Building Quality into Software", in. Crawfordsville, IN: Addison-Wesley Professional, 2016, ch. The Testing Vocabulary. [Online]. Available: https://learning.oreilly.com/library/view/developer-testing-building/9780134291109 (visited on 07/17/2020).

[23] A. P. Mathur, "Foundations of Software Testing", in, 2nd ed. Dehli, Chennai: Pearson India, 2013, ch. Test Selection, Minimization, and Prioritization for Regression Testing, ISBN: 9789332517660. [Online]. Available: https://learning.oreilly.com/library/view/foundations-of-software/9788131794760/ (visited on 07/17/2020).

[24] "Add DataOps Tests to Deploy with Confidence", *DataKitchen*, Apr. 2020. [Online]. Available: https://medium.com/data-ops/add-dataops-tests-to-deploy-with-confidence-4efde90869a6 (visited on 07/20/2020).

[25] Y.-L. Chen, K. Tang, R.-J. Shen, and Y.-H. Hu, "Market basket analysis in a multiple store environment", *Decision Support Systems*, vol. 40, no. 2, pp. 339–354, 2005, ISSN: 0167-9236. DOI: 10.1016/j.dss.2004.04.009.

[26] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, "Top 10 algorithms in data mining", *Knowledge and Information Systems*, vol. 14, no. 1, pp. 1–37, Jan. 2008, ISSN: 0219-3116. DOI: 10.1007/s10115-007-0114-2.

[27] S. Raschka, *mlxtend (Documentation)*, 2020. [Online]. Available: http://rasbt.github.io/mlxtend/ (visited on 08/04/2020).

[28] *pandas (Documentation)*, pandas Developer Team, 2020. [Online]. Available: https://pandas.pydata.org/docs/ (visited on 08/04/2020).

[29] *ARTS Transaction Concepts*, Object Management Group. [Online]. Available: https://www.omg.org/retail-depository/arts-odm-73/arts_transaction_concepts.htm (visited on 08/04/2020).

[30] *AWS Simple Storage Service (S3) (Overview)*, Amazon Web Services. [Online]. Available: https://aws.amazon.com/s3/ (visited on 08/04/2020).

[31] *AWS Athena (Overview)*, Amazon Web Services. [Online]. Available: https://aws.amazon.com/athena (visited on 08/04/2020).

[32] *Apache Airflow (Documentation)*, The Apache Software Foundation, 2019. [Online]. Available: https://airflow.apache.org/docs/stable/ (visited on 08/04/2020).

[33] *Error Severity Levels*, Oracle, 2010. [Online]. Available: https://docs.oracle.com/cd/E19225-01/820-5823/ahyip/index.html (visited on 08/08/2020).

[34] *Docker (Documentation)*, Docker Inc., 2020. [Online]. Available: https://docs.docker.com (visited on 08/14/2020).

[35] *AWS Elastic Container Service (Documentation)*, Amazon Web Services, Inc., 2020. [Online]. Available: https://docs.aws.amazon.com/AmazonECS/latest/developerguide (visited on 08/14/2020).

[36] *AWS Elastic Container Registry (Documentation)*, Amazon Web Services, Inc., 2020. [Online]. Available: https://docs.aws.amazon.com/AmazonECR/latest/userguide (visited on 08/14/2020).

[37] *AWS Identity and Access Management (Documentation)*, Amazon Web Services, Inc., 2020. [Online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide (visited on 08/14/2020).

[38] S. Chacon and B. Straub, *Pro Git*. Apress, 2020. [Online]. Available: https://git-scm.com/book/en/v2 (visited on 08/14/2020).

[39] *GitHub (Documentation)*, GitHub, Inc., 2020. [Online]. Available: https://docs.github.com/ (visited on 08/14/2020).

[40] "Understanding the GitHub flow", *GitHub Guides*, Jul. 2020. [Online]. Available: https://guides.github.com/introduction/flow/ (visited on 08/14/2020).

[41] V. Driessen, "A successful Git branching model", *nvie.com*, Jan. 2010, Edited in March 2020. [Online]. Available: https://nvie.com/posts/a-successful-git-branching-model/ (visited on 08/14/2020).

[42] *Jenkins (Documentation)*. [Online]. Available: https://www.jenkins.io/doc/ (visited on 08/15/2020).

[43] *Terraform (Documentation)*, HashiCorp. [Online]. Available: https://www.terraform.io/docs/ (visited on 08/15/2020).

[44] *Ansible (Documentation)*, RedHat, Inc., 2020. [Online]. Available: https://docs.ansible.com (visited on 08/15/2020).

A. Appendix

A.1. Conversion Stage Data Events

Description	**One or multiple XML file(s) are not named based on unified naming format**
Category	Data Format
Severity	`WARNING`
Handling	1. Calculate occurrences 2. Increase `WARNING` degree counter accordingly 3. Calculate threshold difference, terminate if exceeded 4. Flag file (`name-warn`) 5. Log warning including file information 6. Continue analytical process

Table A.1.: Incorrect Input File Naming

Description	**One or multiple XML file(s) do not have the appropriate file extension `.xml`**
Category	Data Format
Severity	`WARNING`
Handling	1. Calculate occurrences 2. Increase `WARNING` degree counter accordingly 3. Calculate threshold difference, terminate if exceeded 4. Flag file (`ext-warn`) 5. Log warning including file information 6. Continue analytical process

Table A.2.: Incorrect Input File Extension

Description	**One or multiple XML file(s) are empty**
Category	Data Format
Severity	`ERROR`
Handling	1. Remove empty file from analysis 2. Increase `ERROR` degree counter accordingly 3. Calculate threshold difference, terminate if exceeded 4. Flag file (`empty-err`) 5. Log error including file information 6. Continue analytical process

Table A.3.: Empty Input File

Description	**One or multiple XML file(s) not of POSLog format**
Category	Data Format
Severity	`ERROR`
Handling	1. Remove incompatible file from analysis 2. Increase `ERROR` degree counter accordingly 3. Calculate threshold difference, terminate if exceeded 4. Flag file (`noPoslog-err`) 5. Log error including file information 6. Continue analytical process

Table A.4.: Incompatible (i.e., Non-POSLog) File

Description	**One or multiple XML file(s) are missing required global attribute(s)**
Category	Data Schema
Severity	`ERROR`
Handling	1. Remove incorrect file from analysis 2. Increase `ERROR` degree counter accordingly 3. Calculate threshold difference, terminate if exceeded 4. Flag file (`glbAttr-err`) 5. Log error including file information 6. Continue analytical process

Table A.5.: Input File Missing Required Global Attribute(s)

f

Description	**One or multiple XML file(s) have insufficient number of item purchases**
Category	Data Schema
Severity	`ERROR`
Handling	1. Remove incorrect file from analysis 2. Increase `ERROR` degree counter accordingly 3. Calculate threshold difference, terminate if exceeded 4. Flag file (`item-err`) 5. Log error including file information 6. Continue analytical process

Table A.6.: Insufficient Item Purchases in Input File

Description	**One or multiple XML file(s) are missing required item purchase attribute(s)**
Category	Data Schema
Severity	`ERROR`
Handling	1. Remove incorrect file from analysis 2. Increase `ERROR` degree counter accordingly 3. Calculate threshold difference, terminate if exceeded 4. Flag file (`itemAttr-err`) 5. Log error including file information 6. Continue analytical process

Table A.7.: Input File Missing Required Item Purchase Attribute(s)

Description	**One or multiple XML file(s) are missing optional item purchase attribute(s)**
Category	Data Schema
Severity	`WARNING`
Handling	1. Calculate Occurences 2. Increase `WARNING` degree counter accordingly 3. Calculate threshold difference, terminate if exceeded 4. Flag file (`itemAttr-warn`) 5. Log warning including file information 6. Continue analytical process

Table A.8.: Input File Missing Optional Item Purchase Attribute(s)

Description	**One or multiple XML file(s) are missing required sale information attribute(s)**
Category	Data Schema
Severity	`ERROR`
Handling	1. Remove incorrect file from analysis 2. Increase `ERROR` degree counter accordingly 3. Calculate threshold difference, terminate if exceeded 4. Flag file (`saleAttr-err`) 5. Log error including file information 6. Continue analytical process

Table A.9.: Input File Missing Required Sale Information Attribute(s)

Description	**One or multiple XML file(s) are missing optional sale information attribute(s)**
Category	Data Schema
Severity	`WARNING`
Handling	1. Calculate occurences 2. Increase `WARNING` degree counter accordingly 3. Calculate threshold difference, terminate if exceeded 4. Flag file (`saleAttr-warn`) 5. Log warning including file information 6. Continue analytical process

Table A.10.: Input File Missing Optional Sale Information Attribute(s)

Description	**Formatted Python Dictionary is missing file-equivalent attribute(s)**
Category	Data Schema
Severity	`FATAL`
Handling	1. Log error including data information 2. Prompt error information 3. Terminate analytical process

Table A.11.: Formatted Python Dictionary Missing Attribute(s)

h

Description	**Formatted Python Dictionary value(s) do not comply with file-equivalent value(s)**
Category	Data Value
Severity	FATAL
Handling	1. Log error including data information 2. Prompt error information 3. Terminate analytical process

Table A.12.: Formatted Python Dictionary Value(s) Not Complying with File-Equivalent Value(s)

Description	**One or multiple JSON file(s) are corrupted**
Category	Data Format
Severity	FATAL
Handling	1. Log error including file information 2. Prompt error information 3. Terminate analytical process

Table A.13.: JSON Output File(s) Corrupted

Description	**One or multiple JSON file(s) are empty**
Category	Data Format
Severity	FATAL
Handling	1. Log error including file information 2. Prompt error information 3. Terminate analytical process

Table A.14.: JSON Output File(s) Empty

Description	**One or multiple JSON file(s) are missing file-equivalent attribute(s)**
Category	Data Format
Severity	FATAL
Handling	1. Log error including file information 2. Prompt error information 3. Terminate analytical process

Table A.15.: JSON Output File(s) Missing File-Equivalent Attribute(s)