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

Oʟɪᴠᴇʀ Rᴜᴅᴢɪɴsᴋɪ

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
SDK	Software Development Kit
SPC	Statistical Process Control
SPOF	Single Point of Failure
SQL	Structured Query Language
TDD	Test-Driven Development
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 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.
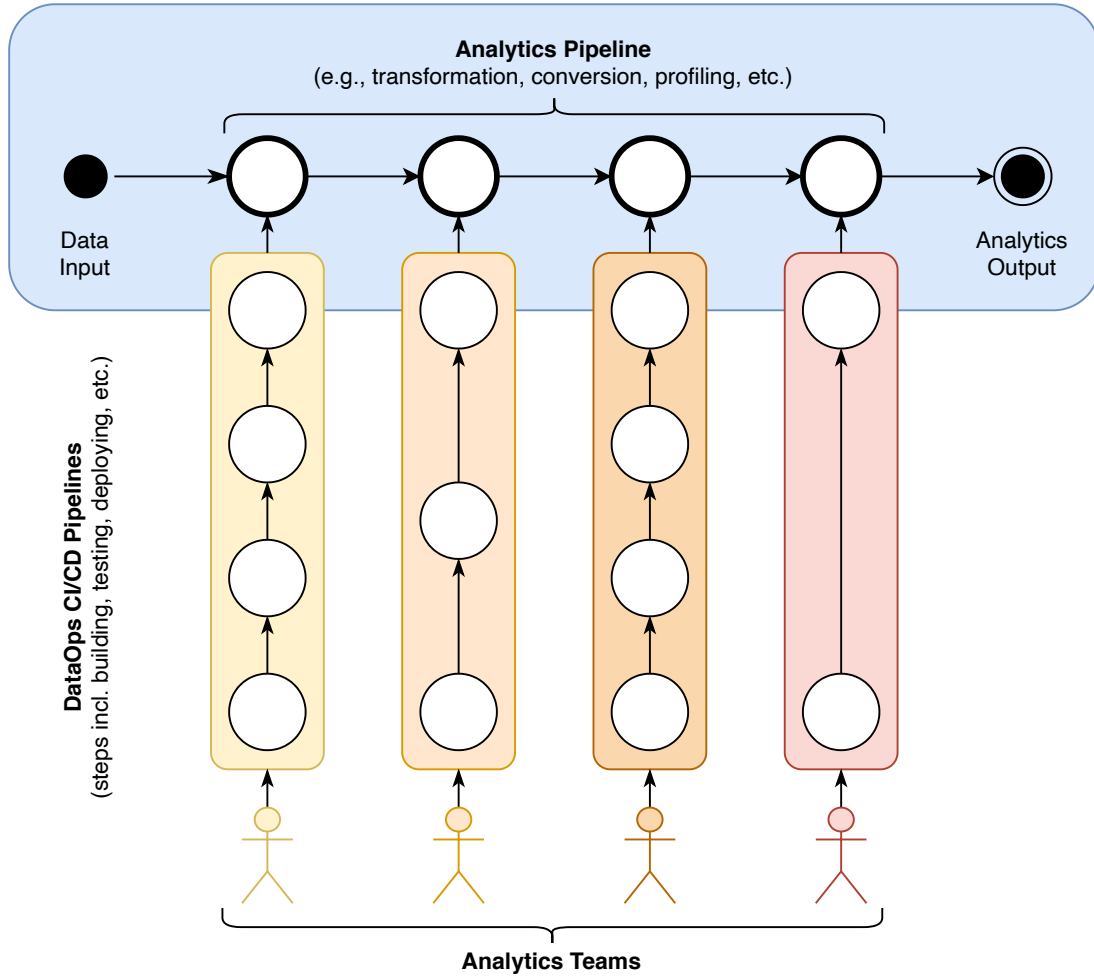
Figure 2.1.: DataOps Pipeline Duality (per [2, pp. 38 sqq.])

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, pre-defined) 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.

Complexity, Duration, Dependencies, Cost

Individual Test Size, Execution Speed

End-to-End Testing
incl.
System Testing

Integration Testing
i.e., component combinations
and interfaces

Unit Testing
i.e., singular components

Test Volume
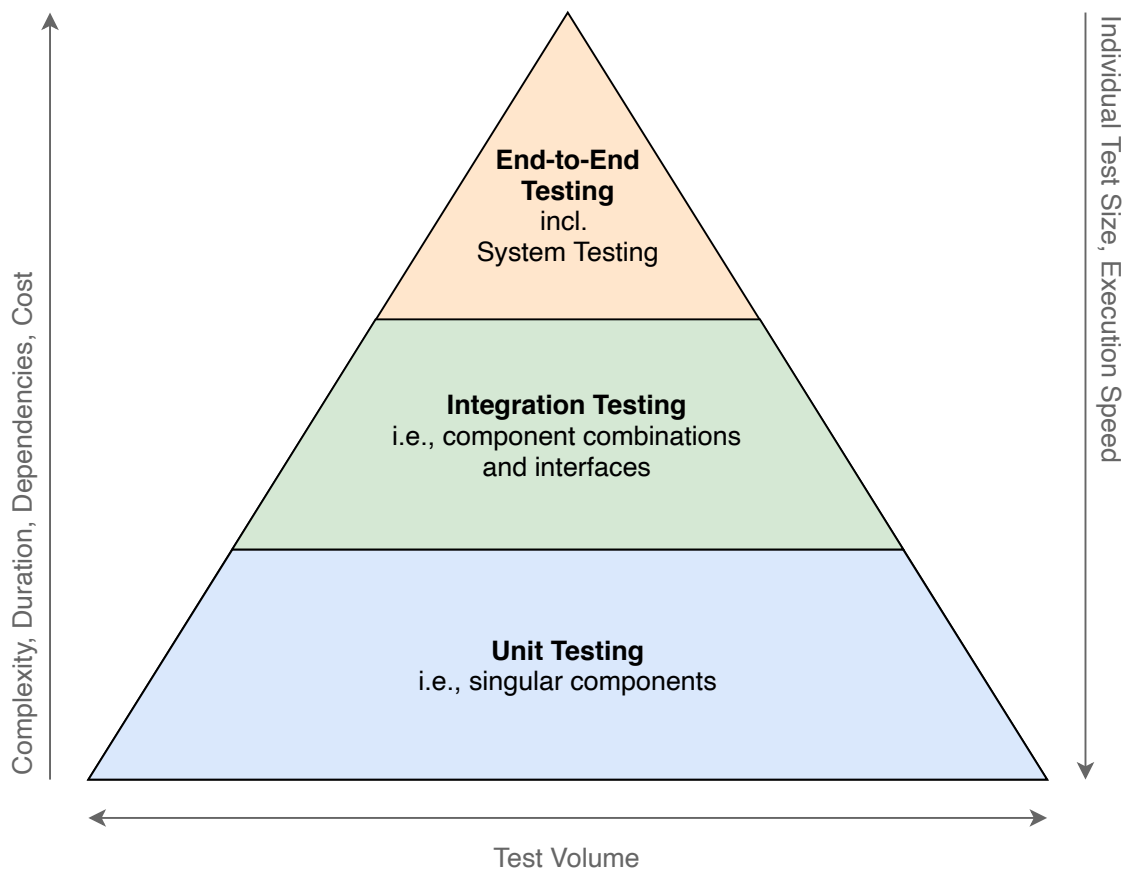
Figure 2.2.: Software Testing Level Pyramid

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].

In traditional DevOps CI/CD testing, the testing pyramid is seen as the ideal testing design structure [22]. It generally aims for discovering errors at the earliest testing stage possible, leading to a high volume of unit tests [23].

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 [24, 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][25]. 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 [25].

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 [26]. 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. 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 [27, 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 [28, 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 [29] 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` [30]. 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 [30] 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 XML format and contains the same information that is usually provided on a print-out receipt [31], 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
<?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 -->
```

```xml
        <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 [32]. The data lake structure can be seen in Figure 3.1.
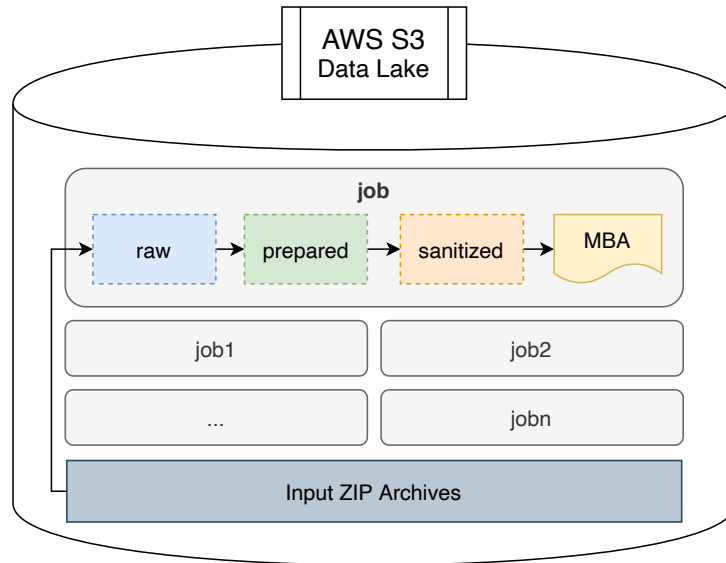
Figure 3.1.: Preliminary Data Lake Structure

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) [33]. 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 [34], 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.
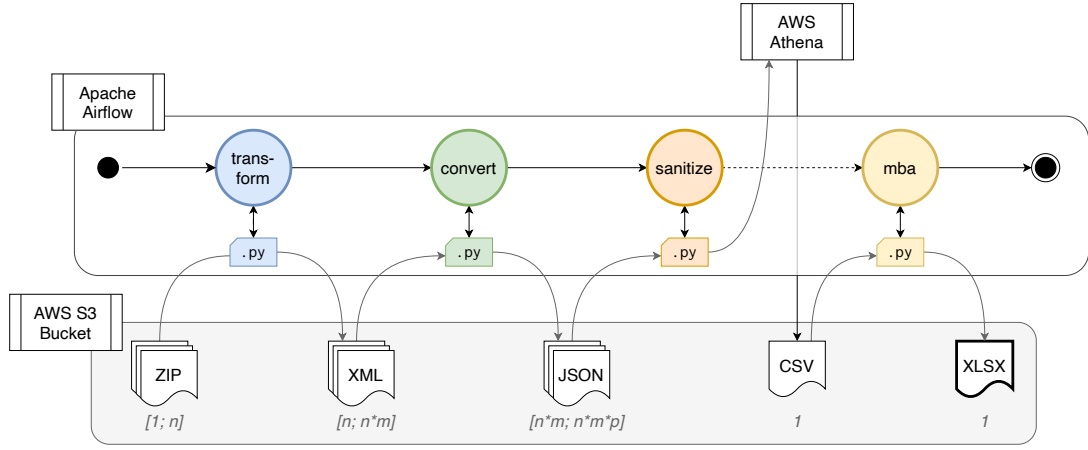
Figure 3.2.: Data Pipeline Overview

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.

All communication between the analytics solution and the various AWS resources is done via the AWS Python Software Development Kit (SDK) `boto3` [35].

3.3.1. Transformation Stage

Figure 3.3 visualizes the Transformation Stage.
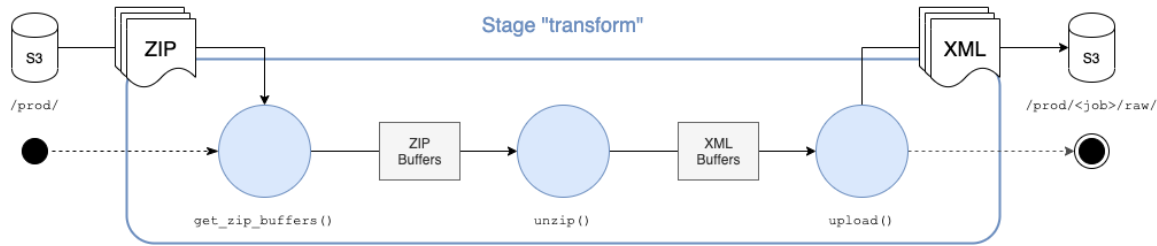
Figure 3.3.: Transformation Stage of the Data Pipeline

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.
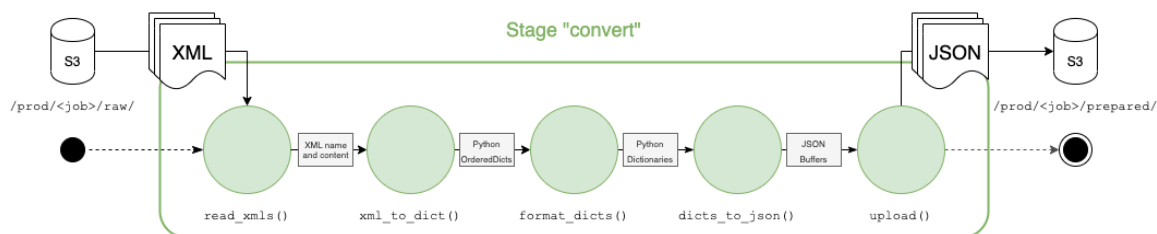
Figure 3.4.: Conversion Stage of the Data Pipeline

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.
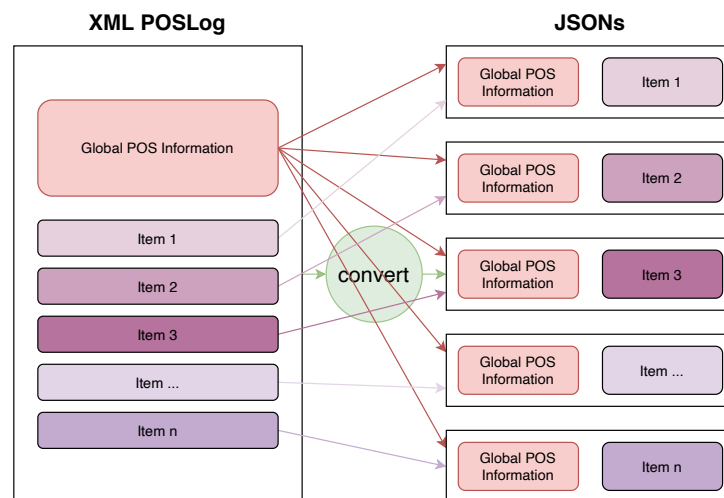
Figure 3.5.: Conversion of Nested POSLog XML Files

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 [33]. 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.

```json
{
    "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.
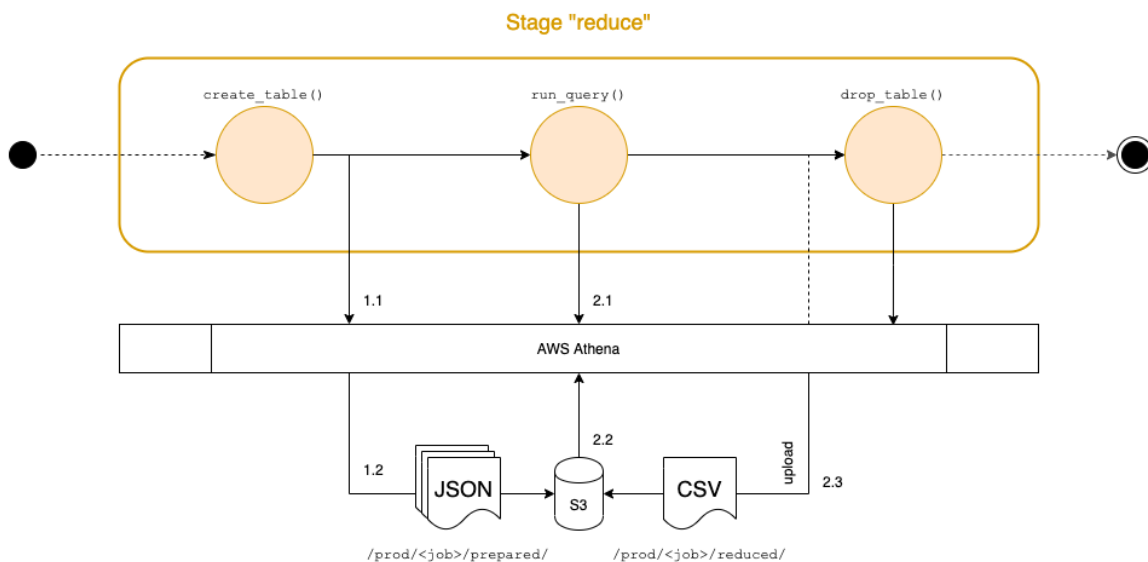
Figure 3.6.: Sanitization Stage of the Data Pipeline

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.
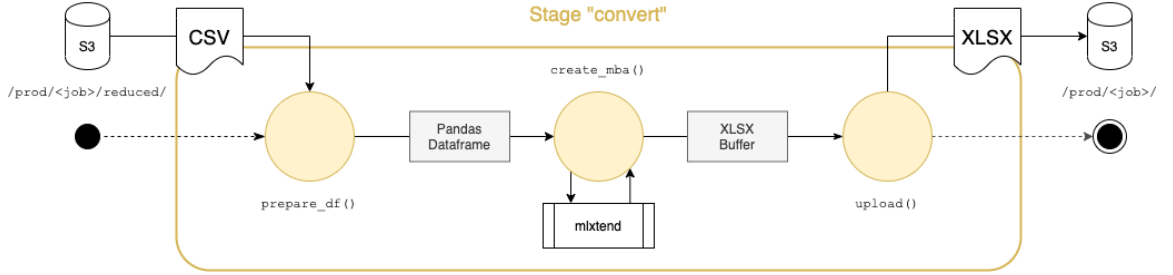
Figure 3.7.: MBA Stage of the Data Pipeline

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., `Description`s) 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` [30]. 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 [36] 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. Flag file (xmlCorr-err) 5. Log error including file information 6. 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. A file tag is added to the source file that caused the event invocation for troubleshooting purposes. 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. Again, the source data receives a file tag. 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.
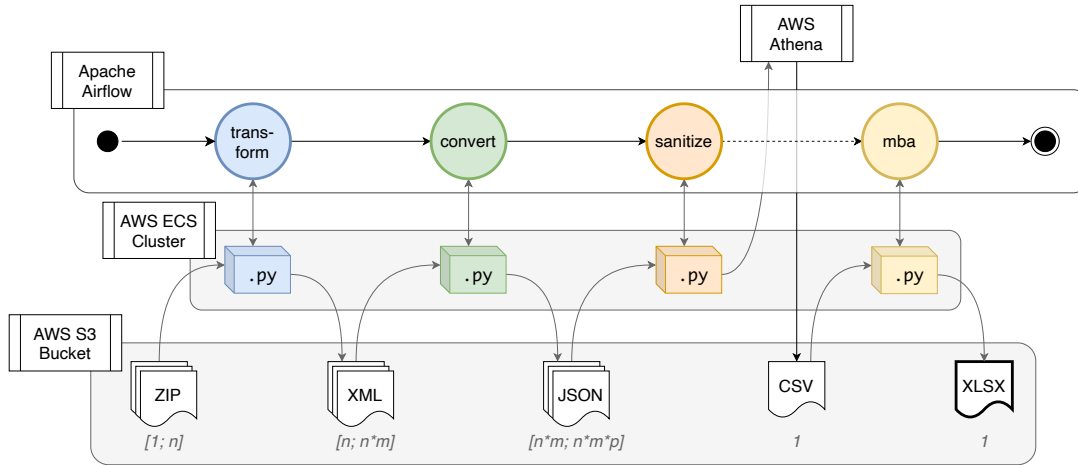
Figure 5.1.: Revisited Data Pipeline Architecture (Server-Less)

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 [37], 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 [38]. 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 [37]. Before running the container, specific configurations after the image has been defined.

Docker images are defined by means of a *Dockerfile* [37]. 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 defin-
ing 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 de-
velopment 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 ECR. ECR is an image repository that is designed to hold different versions of an image [39]. The image deployment and container execution strategy is depicted in Figure 5.2 below.
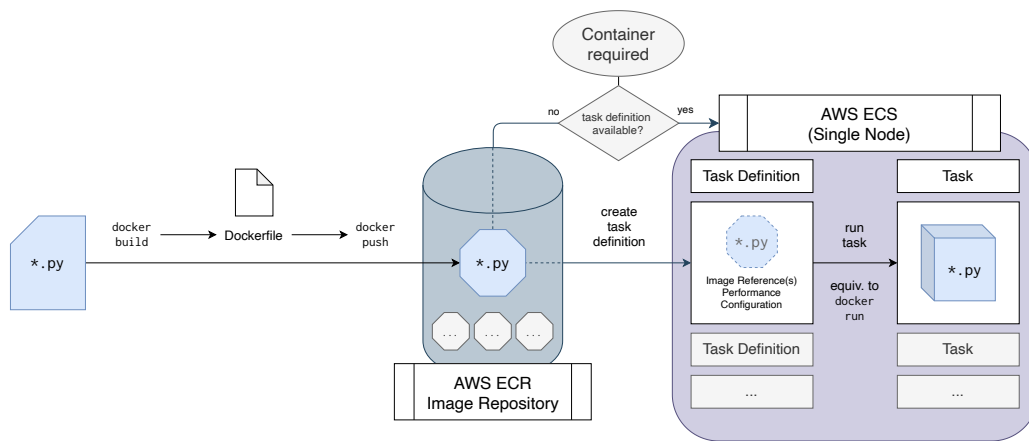
Figure 5.2.: Microservice Deployment Process

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.) [38]. 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* [40].

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 [40], 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 [41, 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:
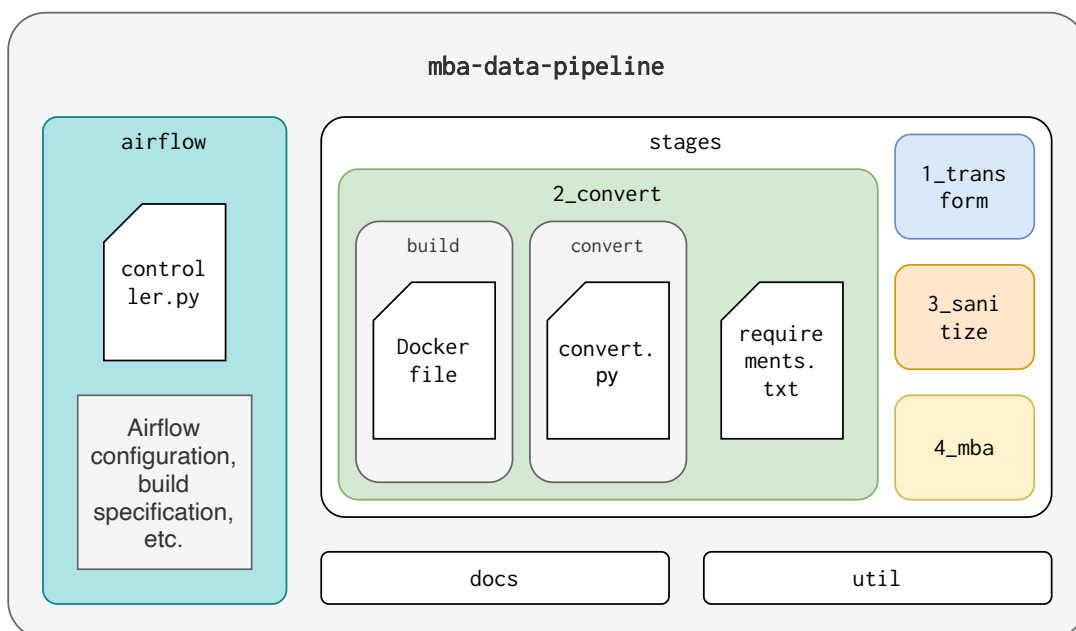

Figure 5.3.: Initial Project Repository Structure

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 [41, pp. 25 sqq.][42]. 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 [41, 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 on-going 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 [42]. 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* [43], depicted in Figure 5.4, below.
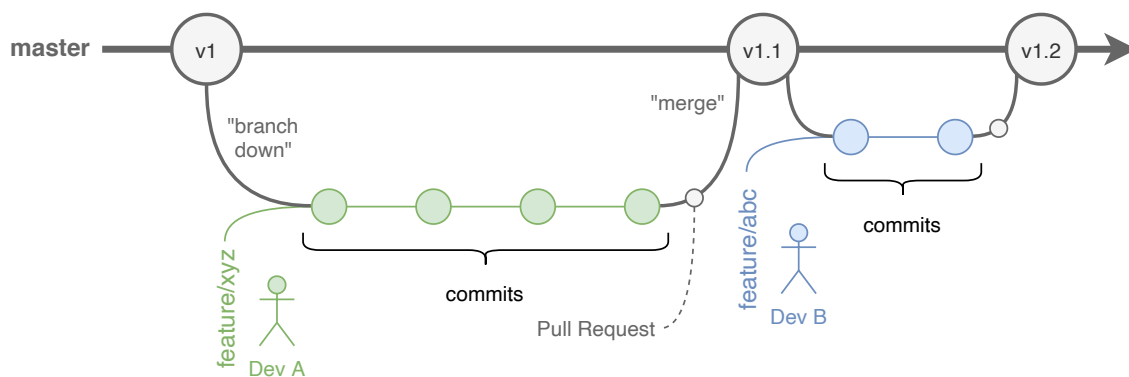
Figure 5.4.: *GitHub Flow* Branching Strategy (per [43])

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 [44].

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.
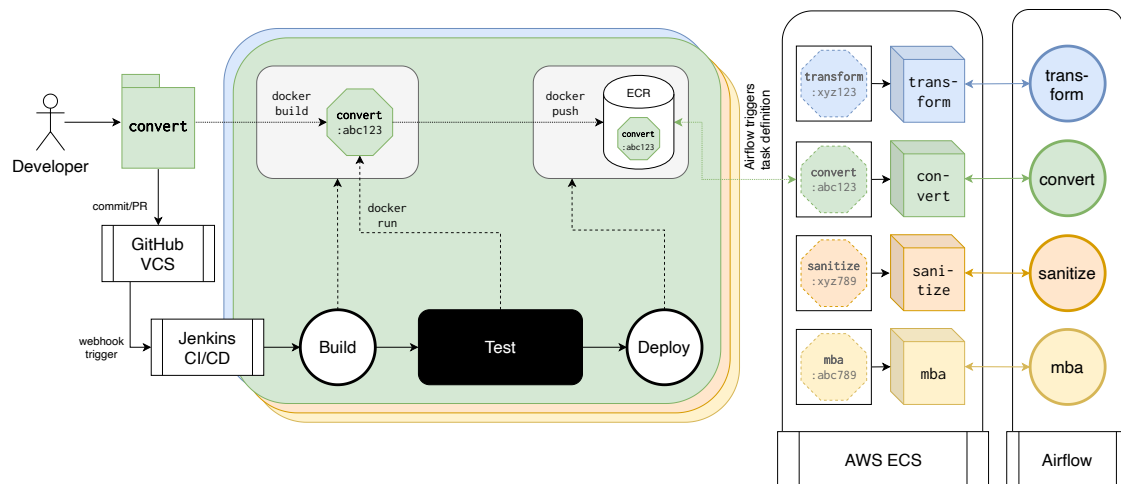

Figure 5.5.: DataOps CI/CD Architecture for the Conversion Stage

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 [45]. 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 [45]. 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 [45]. 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 [39]. This process is identical to other Docker registry services, independent from vendor.

Jenkins will then pass the encrypted credentials to the corresponding pipelines [39], 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 [45]. 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.
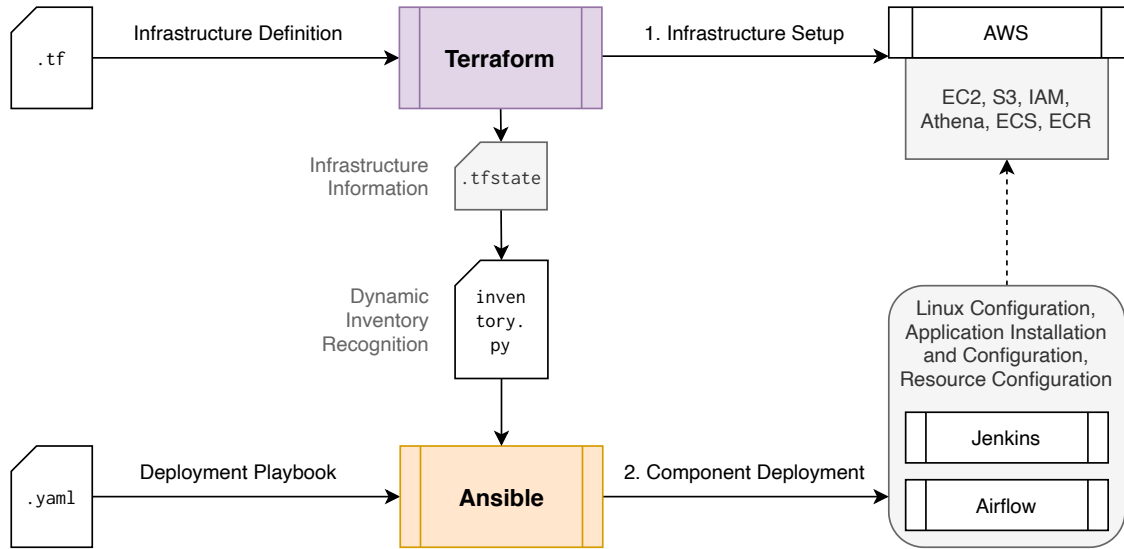
Figure 5.6.: IaC Infrastructure Deployment Strategy

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 [46], 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. [47]. 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

At this point, the DataOps capability of the MBA pipeline solution is sufficient to be enhanced by means of the testing framework, designed in Chapter 4. The practical implementation of the framework goes hand-in-hand with its design steps.

1. The data event handling mechanisms need to be implemented inside the productive analytics solution.

2. The test data (suites) need to be provisioned inside a separated data environment.

3. DataOps solution tests need to be written and executed by the Jenkins pipeline, making use of the available test data. The test outcome evaluation inside Jenkins will decide if the given version update may be deployed into production.

5.3.1. Data Event Handling Implementation

In general, the goal is that the data event handling inside the solution complies with its corresponding definition. When summarizing all event handling processes by means of their actions, the following capabilities need to be ensured:

- Logging of different event severities.

- Process termination for `CRITICAL` events.

- Threshold adjustment and evaluation for `ERROR` and `WARNING` events.

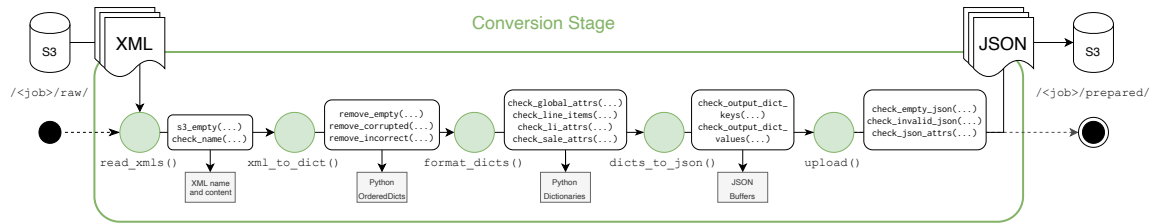- Dynamic File Flagging for `ERROR` and `WARNING` events.

Figure 5.7.: Conversion Stage Overview with Data Handling

Again, there are multiple ways to achieve this kind of data handling capability in practice. In order to keep the productive analytics source code readable and maintainable, data handling is outsourced to its own class `DataHandling` that is instantiated inside the solution. It receives the references to the data in question, threshold limits, etc. during initialization. Each data event definition receives its own function which is then called by the solution at the appropriate point during its runtime. The visualization of the Conversion Stage, including all its data event handlers, is visualized in Figure 5.7.

For the sake of clarity, the following paragraphs elaborate on the implementation of the list of general features inside the `DataHandling` class before presenting exemplary implementations of several data handling functions.

5.3.1.1. Event Logging

Event logging in Python can be realized with Python's standard `logging` library. The default log severities `INFO`, `WARNING`, `ERROR`, and `CRITICAL` correspond with the event severities of the testing framework. The `logging` object is instantiated inside the `DataHandling` class and configured to write log occurrences of severity `INFO` and above to a specified file which receives the name of the current analytics job. Each time a log is required, the `logging` object is called with the message and arguments to be contained.

5.3.1.2. Process Termination

The analytical process needs to be terminated whenever an event of `CRITICAL` severity occurs. In order to provide helpful monitoring information, the termination should not just shut down the process. Instead, it should prompt a message on why the process needed to be terminated. On the one hand, this is included to the log file by means of the previous section. On the other hand, each critical event receives its own custom Python *Exception*. This exception contains are log-like information message

as well as a traceback to the origin of the exception, and hence, termination. This is expected to allow for better understanding and recovery from the underlying issue.

5.3.1.3. Thresholding Capability

As previously mentioned, the threshold percentages are passed to the `DataHandling` class during initialization. Currently, a value of 0.05 for `ERROR` events and 0.1 for `WARNING` events, is utilized. This means that five percent of the input data may invoke `ERROR`s and ten percent of the data may invoke `WARNING`s. The solution does not take files with multiple data issues into account differently. Eventually, the most severe event is counted for threshold evaluation. At the end of the input data event handling process, the threshold values are evaluated. The process terminates when either one of the thresholds is exceeded. As previously described, transformation and output data is not subject to threshold evaluation but results in instant termination in case an event is invoked.

5.3.1.4. File Flagging

`WARNING` and `ERROR` events additionally flag files by means of their invocation event. The AWS Python SDK `boto3` is used for this matter since all data resides inside the S3 data lake. File flagging is implemented inside its own function which is called with the appropriate file reference (i.e., the file to be tagged) and the tag itself. `boto3` then requests all current tags for the file. In case the tag in question has not been set, it is added to the file now. The tagging information is provided inside the log file with `INFO` log severity.

5.3.1.5. Event Handling Implementation Examples

Now that the components of the event handing are defined, the individual handling functions can be implemented. The following examples go by the data event examples provided in Section 4.1.3.1, 4.1.3.2, and 4.1.3.3.

Example A: Data Source Empty   Source Code Excerpt 7 implements the data event handling for an empty data source. The function receives the file references as a parameter (l. 1) and makes use of a *guard* statement to rule out an empty data source first (l. 2). In case this condition does not apply, the function logs the `CRITICAL` event to the log file, providing information on the S3 bucket and bucket directory in

```python
def s3_empty(self, files):
    return if len(files) > 0:

    self.logger.critical(
        'Data Source Empty | No files found at {}/{}'.format(
            self.src_bucket, self.src_dir
        )
    )
    raise S3EmptyError(self.src_bucket, self.src_dir)
```

Source Code Excerpt 7: Implementation of Data Event Example A: Data Source Empty

question (ll. 4–8). These have been passed to the class instance during initialization. At the end, it raises the custom S3EmptyError exception that terminates the process. The exception is prompted, again, with bucket and directory information (l. 9) for improved troubleshooting.

```python
def remove_corrupted(self, files):
    return if files == {}

    corrupted_files = []
    for name, content in files.items():
        try:
            et.fromstring(content)
        except et.ParseError:
            self.logger.error(
                'XML File Corrupted | File Name: "{}"'.format(name)
            )
            self.logger.info('Skipping file "{}"'.format(name))
            corrupted_files.append(name)
            self.error_files += 1
            self._set_file_tags(
                    self.src_bucket,
                    f"{self.src_dir}{name}",
                    "xmlCorr-err",
                )
    for file in corrupted_files:
        files.pop(file)
```

Source Code Excerpt 8: Implementation of Data Event Example B: XML File(s) Corrupted

Example B: XML File(s) Corrupted Source Code Excerpt 8 implements the data event handling for corrupted XML files. Again, the corresponding function receives file references. Since there is a possibility that other ERROR events have already occurred and removed faulty input data, the function checks if there are

any files left for evaluation (l. 2). Then, it loops through all files in question and attempts to parse their binary content to XML (ll. 6–7). The function makes use of the Python XML `ElementTree` sub-library (initialized as `et`). By design, the parser function raises an `ParseError` exception if the data is corrupted. By catching this library exception, the specific data handling can be processed. The log file receives a new line containing the corresponding file information (ll. 9-11). Additionally, the log also mentions that the file in question is skipped because of the `ERROR` event severity. The file name reference is added to another list that contains all corrupted files (l. 4 and 13) and the error file counter is incremented (l. 14). This will later be used for threshold evaluation. Finally, all corrupted files are removed from the original file reference dictionary (l. 15–16).

```python
def check_global_attrs(self, dicts):
    for name, poslog_dict in dicts.items():
        if all(attr in poslog_dict.keys() for attr in self.global_attrs):
            continue

        # Data handling for required attributes omitted...

        if not all(
            attr in poslog_dict.keys() for attr in self.opt_global_attrs
        ):
            missing_attrs = list(
                (set(self.opt_global_attrs) - poslog_dict.keys())
            )
            self.logger.warning(
                'Missing optional global attribute(s) | File '
                'Name: "{}" '
                'Missing Attribute(s): {}'.format(
                    name, ', '.join(missing_attrs)
                )
            )
            self.warning_files += 1
            self._set_file_tags(
                    self.src_bucket,
                    f"{self.src_dir}{name}",
                    "optArg-warn",
                )
```

Source Code Excerpt 9: Implementation of Data Event Example C: XML File(s) Corrupted

Example C: Missing Optional Attributes   Source Code Excerpt 9 implements the overall data event handling for missing global attributes. Since the event in question only covers the missing attributes, the handling of required attributes is omitted here (l. 6). At this point of the analysis, the binary XML data has already been parsed to Python standard dictionaries, which is why the function received a

dictionary of all POS dictionaries (l. 1) and loops over the root dictionary (l. 2). As with the first example, a guard statement is used to rule out the case of any missing argument (ll. 3–4). Then, the keys of the content dictionary are evaluated against the class constant list of optional attributes (ll. 8–10). In case this evaluation results in arguments missing, a list of missing attributes is created (ll. 11-13) and logged alongside the input file name (ll. 14–20). Since this is an event of WARNING severity, the WARNING event file counter is incremented (l. 21) and evaluated at the end of input data event handling. Finally, the source data tagged via the custom tagging function (ll. 22-26)

All further data events are implemented in a similar manner.

5.3.2. Test Data Provisioning

In order to test for correct data handling, appropriate test data is required to invoke the corresponding events. The design of the test data suited of various complexity degrees and use cases has been conducted in Section 4.2. The goal now is to provision these suites in order for the planned, holistic DataOps testing integration. Specifically, the test data needs to be available for continuous testing in every phase of development. Since a centralized data lake already exists, the goal can be achieved by adding the test data to the data lake.

5.3.2.1. Data Lake Architecture Revision

The current architecture of the data lake only considers production-grade analytics processes. Simply adding the test data inside this critical environment could be prone to compatibility and analytics correctness issues. Plus, it goes against the DataOps principle of distinct environments. This is why the following data lake architecture revision is proposed:

In the new structure, presented in Figure 5.8, the data lake is environmentally divided at the topmost level. One environment remains the otherwise unchanged production grade environment, containing raw input data as well as all analytics job subareas. The new testing environment now contains input data test suites for every stage of the pipeline. Since this data is also expected to be processed by the corresponding stage during testing, the area has also got job subareas for each conducted test.
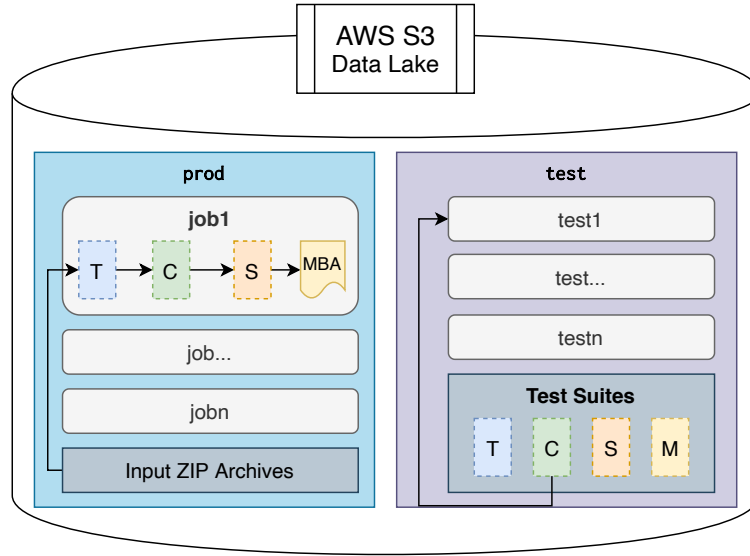
Figure 5.8.: Data Lake Architecture Revision

With this change, the stages do not conduct test-grade analysis with production-grade data, but remain in a separated testing environment. The corresponding S3 URIs, containing test data, are passed to the individual stages. In case all future tests pass, the stages are permitted to receive the production-grade data locations afterwards.

5.3.2.2. IAM Role Revision

With new environments, new AWS IAM roles are required. This creates another layer of maintaining analytical integrity. In case a test stage receives production-grade data references, a correctly configured IAM role prevents it from performing. Developer sandboxes and their IAM roles can also be changed here such that the personal environment does not have access to potentially sensitive data.

In practice, each stage receives a second IAM role for testing purposes. The corresponding credentials are passed as environment variables to the virtual stage container and replaced with the production-grade access credentials after the tests have passed.

With this method of test data provisioning, each environment (i.e., developer sandbox, local or CI/CD-integrated testing, etc.) has access to the test data suites while restricting access to sensitive, production-grade data at the same time.

5.3.3. DataOps Testing Implementation

The DataOps testing implementation is expected to ensure that a new version of a solution is running as expected before being deployed into production. This requires separate testing classes that evaluate the functionality of the solution based on expected outcomes. As described in Section 4.3, this includes all testing levels (i.e., unit, integration, and end-to-end testing) covering both software and data-driven aspects of the solution.

In general, the Python testing framework *PyTest* as well as applicable Python standard testing functions can be utilized for this purpose. With PyTest, individual testing classes are created. Each function inside a testing class can be seen as a single testing use case. The testing process in general performs a number of preparatory measures, runs the code piece at test and evaluates the outcome of the code under test. This happens by means of a comparison to the expected outcome. If these values correlate, the test passes and the next test case is performed. Otherwise, the test case fails and is prompted.

The following paragraphs implement the testing classes for all testing levels of the Conversion Stage.

5.3.3.1. Unit Testing Implementation

The unit tests for the Conversion Stage are expected to cover the granular, software-driven aspects of the solution. Per testing convention, the functionality of external libraries is expected to have already been tested such that these do not require additional testing. Therefore, only the extraction of data source and destination locations remain for the purpose of unit testing.

The outsourced `env` module covers this functionality. It received the corresponding values through a function given by the Python standard library `os`. There are multiple events that are caught at this point, which should not be mistaken with data events. The current events make sure that the S3 data location URIs are provided in the correct format. The remaining functions of the module extract individual parts of the URI that are required by the solution at the point where these functions are called. These functions represent the code under test for Conversion Stage unit testing.

The practical unit tests passes incorrectly formatted S3 URIs to the retrieval function and expects it recognize the issue and terminate the process. One of the test cases also provides a well-formatted URI and expects that no issue is reported. The component extraction functions (for S3 bucket name and path extraction) are only invoked when

the retrieval function reported no error. Thus, their test cases only pass correct URIs and compare their outcome (e.g., an S3 bucket name) to the hard-coded expectation based on the corresponding input URI.

5.3.3.2. Integration (Data) Testing Implementation

Integration testing of the Conversion Stage evaluates the data handling measurements based on the predefined test data suites residing in the testing environment of the S3 data lake. Therefore, it plays a major role for Conversion Stage testing because of the large number of data events and the data-driven nature of the stage in general.

Since the stage is expected to be capable of handing events regarding data format, schema, and value, three separate testing classes are created for the integration testing level. Now, each of the data event handling functions needs to be tested holistically. Specifically, the goal is to ensure the following aspects:

- The appropriate data event handling function is called at the appropriate point inside the analytics solution.

- The data event handling function performance complies with its definition (i.e., it performs all required measures if the event occurs).

The tests are supported by the previously defined test data suites, specifically the single-event and multi-event test data files (cf. Sections 4.2.1 and 4.2.2) that are now used to evaluate the performance of the data event handling. They are retrieved prior to the test execution and specified for each test case individually.

The following paragraphs present the testing implementation of the data event handling function that checks for corrupted XML files. This example provides the most individual test cases for a single data event out of the previously mentioned examples. Since corrupted XML files belong to the data format issue category, the tests are included inside the `TestCovertDataFormat` class. All of the test cases are performed with a single test data file that represents an actually corrupted XML file.

```python
1  def test_remove_corrupted_calls(self):
2      with patch("datatest.DataTest.remove_corrupted") as remove_corrupted:
3          xml_to_dict({})
4          assert remove_corrupted.called
```

Source Code Excerpt 10: Data Handling Function Call Test for Corrupted XML Files

Data Handling Function is Called   Source Code Excerpt 10 represents the test case function that evaluates if the `remove_corrupted(...)` function is called within the `xml_to_dict(...)` function of the analytics solution. Since the function call itself is always required, the input data for the function does not play a role. Since the data handling function is not explicitly known by the test function, it is mocked by the PyTest framework (l. 2). Then, `xml_to_dict(...)` is called with no data (which is arbitrary here) (l. 3). Finally, the test *asserts* that the data handling function was indeed called (l. 4). An assertion in general returns `True` if the given condition (here, the function call) applies. If not, it returns `False`.

```python
def test_xml_corrupted_skips(self, caplog):
    test_collection = prefix.format("corrupted.xml")

    mock_dict = {
        obj.key.split("/")[-1]: obj.get()["Body"].read()
        for obj in bucket.objects.filter(Prefix=test_collection)
        if not obj.key.split("/")[-1] == ""
    }

    with caplog.at_level(logging.INFO, logger=__name__):
        xml_to_dict(mock_dict)

    assert "Skipping file" in caplog.text
    assert len(mock_dict) == 0
```

Source Code Excerpt 11: Data Handling Function Test for Skipping Corrupted XML Files

Data Handling Function Skips Corrupted File   Source Code Excerpt 11 represents the test case function that evaluates if the a corrupted XML file is skipped within the `xml_to_dict(...)` function of the analytics solution. The implementation does not explicitly test the data handling function but its actions. This approach was chosen since the implementation of such a function might change, but the fact that corrupted XML files cannot be processed does not change. The function specifies the corrupted XML file (l. 2) and loads it inside a Python dictionary from the S3 bucket (ll. 4–8). This dictionary is required by the `xml_to_dict(...)` function as its parameter. The function is then called with the prepared dictionary (l. 11). Since the corrupted file has been the only file in scope of the test, the test finally asserts if the dictionary is empty after the function call which would mean that the file has been skipped successfully. Since this process is also logged, the test function mocks the logging mechanism (l. 10) and asserts that the information has been added to the log file (l. 13).

```python
def test_xml_corrupted_logs(self, caplog):
    test_collection = prefix.format("corrupted.xml")

    mock_dict = {
        obj.key.split("/")[-1]: obj.get()["Body"].read()
        for obj in bucket.objects.filter(Prefix=test_collection)
        if not obj.key.split("/")[-1] == ""
    }

    with caplog.at_level(logging.ERROR, logger=__name__):
        xml_to_dict(mock_dict)

    assert "XML File Corrupted" in caplog.text
```

Source Code Excerpt 12: Data Handling Function Test for Logging Corrupted XML Files

Data Handling Function Logs Corrupted File   Source Code Excerpt 12 represents the test case function that evaluates if a corrupted XML file is logged when the `xml_to_dict(...)` function is executed. The process is very similar to the example above. The data is received from S3 (ll. 4–8), the logging mechanism for `ERROR` logs is mocked by PyTest (l. 10), and the `xml_to_dict(...)` function is called with the resulting dictionary (l. 11). The assertion evaluates the predefined string for corrupted XML files matches with the actual outcome of the function call.

```python
def test_xml_corrupted_tags(self):
    s3_client = client("s3")
    test_collection = prefix.format("corrupted.xml")

    mock_dict = {
        obj.key.split("/")[-1]: obj.get()["Body"].read()
        for obj in bucket.objects.filter(Prefix=test_collection)
        if not obj.key.split("/")[-1] == ""
    }

    xml_to_dict(mock_dict)
    tag = s3_client.get_object_tagging(
        Bucket=bucket.name, Key=test_collection,
    )["TagSet"]

    assert {"Key": "xmlCorr-err", "Value": "True"} in tag
```

Source Code Excerpt 13: Data Handling Function Test for Tagging Corrupted XML Files

Data Handling Function Tags Corrupted File   Source Code Excerpt 13 represents the test case function that evaluates if a corrupted XML file is tagged inside the

S3 bucket when the `xml_to_dict(...)` function is executed. Again, the preliminary steps (ll. 3–11) are identical to the previous test cases. Only a `boto3` *Client* (l. 2) for an S3 tag request (ll. 12–14) is added. This request returns a list of JSON strings. The test function asserts if the corresponding tag `xmlCorr-err` is correctly set to the value `True`.

All remaining data events are tested and evaluated in the same way.

Threshold Evaluation The threshold evaluation of the `DataHandling` class needs to be evaluated as well. This also includes checking if the data handling functions appropriately increase their file `ERROR` and `WARNING` counters. This is done by means of the test data suites that contain multiple data files with no, minor as well as major data quality issues (cf. Section 4.2.3). First, the test data suite is created and the expected threshold outcomes are calculated manually. The corresponding test case runs through the stage with the given test data suite and compares the actual threshold outcome with the expected one. On the one hand, this is evaluated by testing if the process is terminated because of the threshold being exceeded. On the other hand, the specific `ERROR` and `WARNING` percentages can be compared with the expected data issue percentages.

5.3.3.3. End-to-End Testing Implementation

Finally, end-to-end testing also brings Airflow into play such that all external resources are included in the testing process. It is important to remember that the areas covered by the unit or integration tests do not have to be repeated within end-to-end testing. Testing Airflow's impact on the analysis can be performed under the assumption that all other components have already been tested previously.

In practice, two final test cases is created that invoke the entire data pipeline with the new version of the Conversion Stage. This is done by pre-deploying the Conversion Stage test image to ECR and running the pipeline via the Airflow API [34]. One test case is run with a test data suite that is expected to terminate the process because of the threshold exceeding. This evaluates if Airflow correctly recognizes the process termination and stops the pipeline. The other test case is run with an acceptable test data suite. After the pipeline has run through, the exit status of the pipeline is evaluated, as well as the Conversion Stage output data in order to rule out any negative result impact resulting from Airflow.

5.4. Testing Process Automation

The current testing solution only allows for local test execution. This is valuable for developers who can run their tests prior to filing a pull request for their new feature. The actual goal is to automate the tests by means of the prepared Jenkins CI/CD environment. Additionally, all changes that have been made to the general project structure need to be included in the automation processes.

The current Jenkins CI/CD pipeline does not consider testing yet. In order to include testing inside the pipeline, the test files and classes need to be specified inside the project repository and Jenkinsfile. The following repository structure addition is proposed:
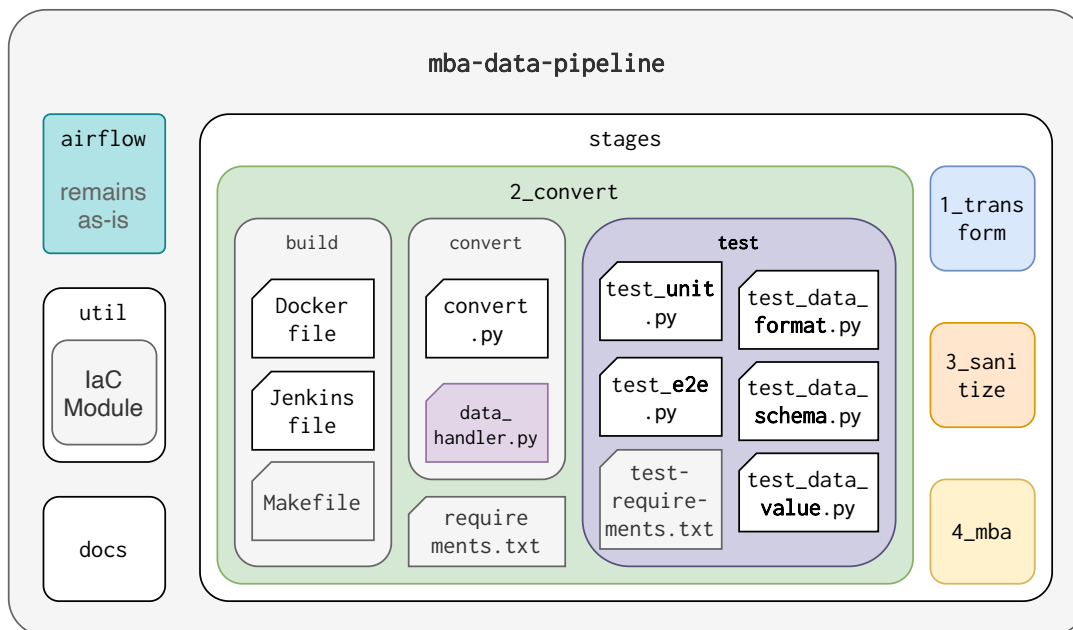

Figure 5.9.: Revisited Project Repository Structure (Testing Included)

Figure 5.9 visualizes the testing-enabled repository. Each stage receives its own `test` directory. This directory contains all testing class files as well as a requirements document, specifying required external installation for testing purposes. This document is similar to the already existing `requirements.txt` file.

With this change, Jenkins has now got access to the testing classes. Since the current Dockerfile already includes the entire stage module inside the image, the tests can already be run inside virtual Docker containers, which is desired. Currently, Jenkins only holds the production-grade IAM credentials inside is credential storage. The previously created IAM testing roles need to be added as well. Finally, the Jenkinsfile

can be manipulated in order to provide appropriate testing stages inside the process, resulting in the pipeline depicted in Figure 5.10
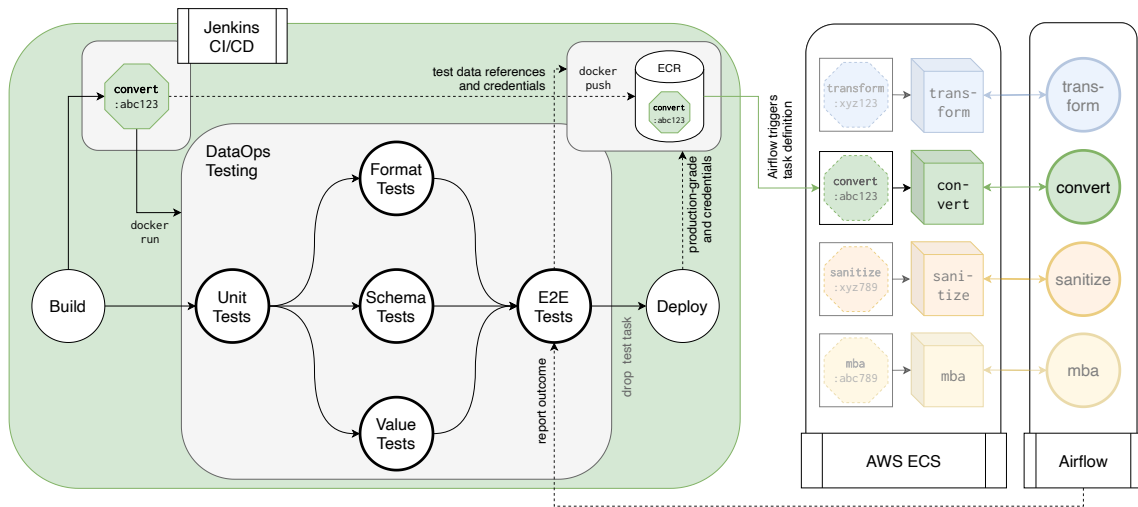

Figure 5.10.: Testing-Enabled CI/CD Pipeline Overview

The build stage is changed such that the image is built with testing credentials rather than production-grade credentials. Additionally, Jenkins already specifies the test data locations during the image build process. In production, Airflow configures the required environment variables inside the task definition. With testing, Jenkins uses the unique Git commit identifier to define the data output location. Then, testing stages are implemented. Their individual structures are very similar to each other. The testing stages of the Jenkinsfile are specified in Source Code Excerpt 14.

The unit testing stage calls the corresponding unit testing class file (l. 13) when running the Docker container (ll. 9–15). Additionally, a host-to-container volume mount is performed (l. 10) such that Jenkins receives access to the testing report file (l. 14). This file is used to display the testing report inside the Jenkins UI. The integration (data) tests are implemented similarly, but run in parallel (l. 24–30) for a more appealing visualization. This does not cause any problems since these tests are run in entirely separate containers. The end-to-end test performs a pre-deployment prior to test execution (ll. 35–40). This is required because the test calls the Airflow API to execute a test data pipeline including the new version image.

Finally, the test deployment is removed and the actual deployment, including production-grade credentials and no data location specification, is performed. The environment changes (i.e., data lake structure and additional IAM role definitions) are added to the IaC process of the project. In case of an infrastructure rebuild, these components will be also generated and embedded automatically.

This concludes the implementation task which is now subject to evaluation.

```
1  pipeline {
2    /* Preamble omitted */
3
4    stages {
5      /* Build stage... */
6
7      stage('Unit Test') {
8        steps {
9          sh "
10           docker run -v $WORKSPACE/test:/usr/src/app/test/out
11                      <aws-user-id>.dkr.ecr.eu-central-1.amazonaws.com/
12                        conversion:${GIT_COMMIT_SHORT}-test
13                      pytest ./test/test_convert_unit.py
14                      --junitxml ./test/out/unit-report.xml
15          "
16        }
17        post {
18          always {
19            junit "test/unit-report.xml"
20          }
21        }
22      }
23
24      stage('Data Test') {
25        parallel {
26          stage('Format Test') { /*...*/ }
27          stage('Schema Test') { /*...*/ }
28          stage('Value Test') { /*...*/ }
29        }
30      }
31
32      stage('End-to-End Test') {
33        steps {
34          script {
35            docker.withRegistry(
36              'https://<aws-user-id>.dkr.ecr.eu-central-1.amazonaws.com',
37              'ecr:eu-central-1:aws') {
38              sh "docker push <aws-user-id>.dkr.ecr.eu-central-1.amazonaws.com/
39                    conversion:${GIT_COMMIT_SHORT}-test"
40            }
41          }
42          sh /*"docker run..."*/
43        }
44        /* Test Report Handling... */
45      }
46    }
47  }
```

Source Code Excerpt 14: Testing Stages of the Revisited Jenkinsfile

6. Solution Evaluation

The following evaluation is expected to provide insights whether the solution works as intended during implementation. A focus is set on the testing capabilities of the CI/CD pipeline in order to find potential limitations of the solution and discuss possible improvement techniques in theory. Throughout the evaluation, DataOps is compared to DevOps while focussing on the CI/CD process and its testing features. This ultimately includes bottlenecks that cannot be covered by automated measures of the pipeline but rather need to be handled via conventions and best practices.

6.1. Workflow Demonstration

First of all, it might be valuable to evaluate the basic functionality of the CI/CD pipeline, including its testing and deployment capabilities when no errors occur. For this matter, the general development workflow is applied. Before the following changes, the latest production-grade image was tagged with the commit ID `3c75dd5` inside ECR.

A developer makes an arbitrary change to the Conversion Stage source code of the solution that is expected not to break the solution. This change is made in a separate feature branch of the VCS. The developer can verify this fact by running the pre-existing, automated tests locally on the developer sandbox. Then, a pull request is filed with the goal to merge the feature branch with the `master` branch, resulting in the new source code version being deployed into production. The pull request needs to include the appropriate GitHub labels, marking that the pull request should be treated as a new feature inside the Conversion CI/CD pipeline. The filing could look like the screenshot provided in Figure 6.1. A pull request description is well-desired but omitted here for the sake of simplicity.
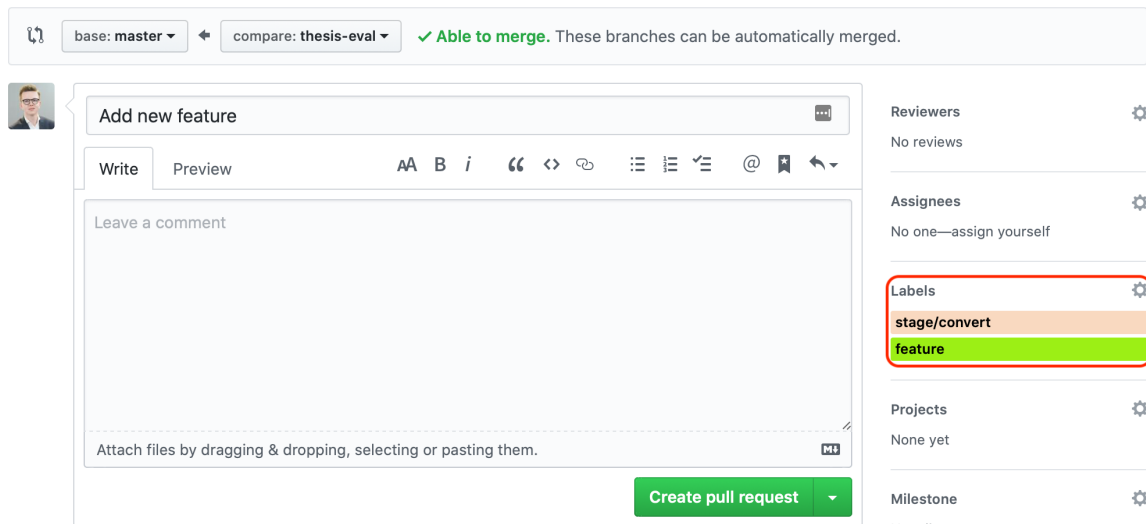
Figure 6.1.: GitHub Feature Pull Request for Conversion Stage (Screenshot)

The first minor issue can be found inside this workflow. The developer in charge of this pull request is required to set the appropriate labels. The `feature` label missing would cause the CI/CD pipeline not to run. The same applies to the stage specification label. An incorrect label could cause even more problems since this would trigger another pipeline with this stage source code, resulting in the build phase failing. The problem could be resolved by splitting up the repositories by stage but would remove the overall availability of the entire project's source code. This could have a negative impact on the productivity of the teams working on the project.

In case of correctly chosen labels, the Conversion Stage pipeline is triggered. A new pull request is automatically added to Jenkins and can be seen in the corresponding overview of the Jenkins UI, shown in the screenshot in Figure 6.2.
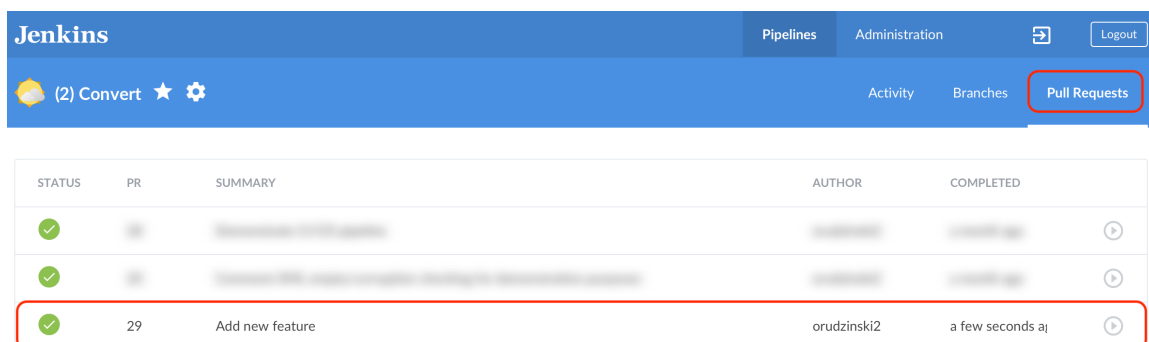

Figure 6.2.: Pull Request in Jenkins UI (Screenshot)

The green checkmark shows that the pull request was evaluated, successfully verified and deployed. The specific pipeline overview can be retrieved by inspecting the corresponding pull request. This yields the following screenshot in Figure 6.3.
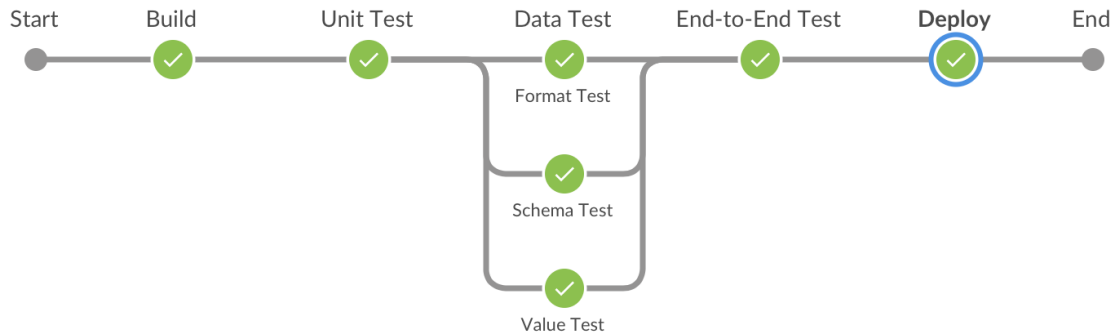
Figure 6.3.: Successful Pipeline Job in Jenkins UI (Screenshot)

Since the deployment has run through correctly, ECR should have received a new image. This image should be tagged with the last commit ID of the pull request as well as the `latest` tag. Inspecting the corresponding repository inside the AWS Console verifies this claim, presented in the screenshot of Figure 6.4

Figure 6.4.: Image List of the Conversion Stage AWS ECR Repository (Screenshot)

It can be seen, that the new image has been included with the appropriate tags. The old version of the image still resides in the repository for versioning reasons. Because of the correct tagging, Airflow will use this updated version of the Conversion Stage image for upcoming analyses.

The source code change is not reflected in the GitHub repository, yet. The feature branch still exists and the `master` branch has not been changed. Since the required CI/CD checks have run through successfully, the developer can now merge the feature branch with the `master` branch. This manual procedure is required by GitHub. This can be seen as another issue of lacking automation since the developer may not forget to merge the branches and remove the feature branch after the deployment has been conducted successfully. Existing copies of the previous `master` branch inside other development sandboxes need to be updated now since working with the outdated version might cause compatibility issues of upcoming versions as well as so-called *merge conflicts* within GitHub, where the feature might be correct by means of the CI/CD evaluation, but the code history of the branches to merge does not allow for an automatic merge and requires manual correcting.

At this point, one iteration of the CI/CD workflow is completed.

6.2. Testing Capability Evaluation

The previous evaluation use case did not contain on purpose in order to demonstrate the desired development workflow. Now, the workflow of the pipeline is evaluated for different kinds of issues. In general, this is divided into two categories:

1. issues that result from updating existing features, and

2. issues that result from creating new features.

6.2.1. Feature Update Testing

The most naïve approach is to remove a line of code that is required by the testing framework. This simulation is comparable to a scenario where an existing feature is updated but a required functionality was removed. The testing framework is supposed to be invoked by the CI/CD pipeline job and catch the issue. Specifically, the line checking for a corrupted XML file inside the analytics script of the Conversion Stage is removed. This data handling function has been presented in Section 5.3.3.2.

The process of filing a pull request remains identical. Jenkins' pull request job is triggered again, resulting in the job workflow depicted in the screenshot in Figure 6.5.
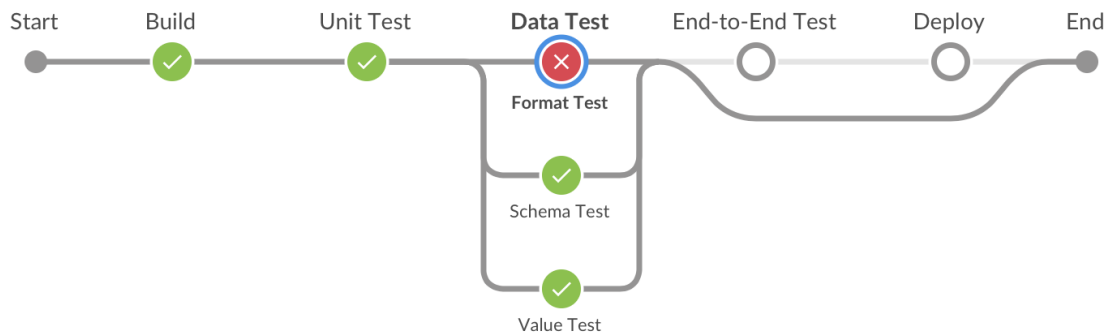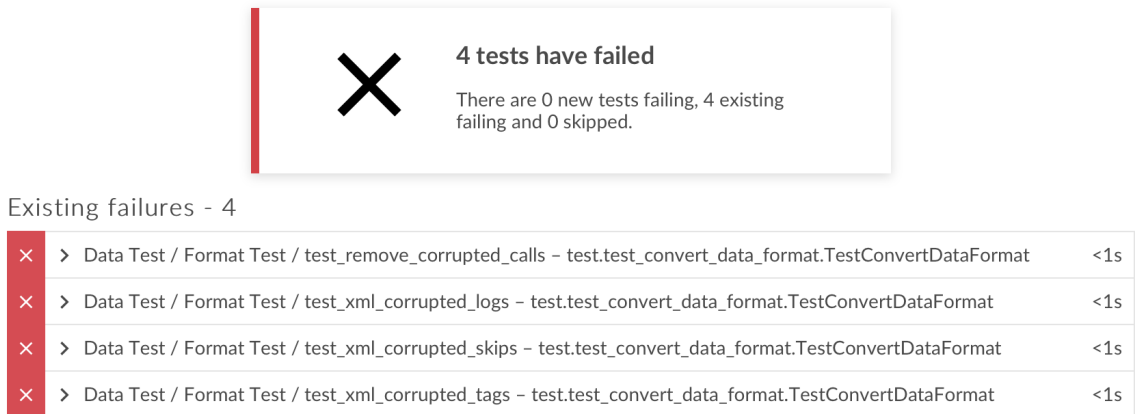
Figure 6.5.: Failing Pipeline Job in Jenkins UI (Line Missing) (Screenshot)

Since XML corruption testing is subject to the data format testing suite, the corresponding stage inside the CI/CD pipeline fails. The upcoming stages are skipped such that no flawed deployment is conducted. The *Tests* tab of the job inside the Jenkins UI provides insights to the tests that have been run during the testing stages and highlights tests that did not pass. This overview is shown in a screenshot in Figure 6.6.

4 tests have failed

There are 0 new tests failing, 4 existing failing and 0 skipped.

Existing failures - 4

✕	❯ Data Test / Format Test / test_remove_corrupted_calls – test.test_convert_data_format.TestConvertDataFormat	<1s
✕	❯ Data Test / Format Test / test_xml_corrupted_logs – test.test_convert_data_format.TestConvertDataFormat	<1s
✕	❯ Data Test / Format Test / test_xml_corrupted_skips – test.test_convert_data_format.TestConvertDataFormat	<1s
✕	❯ Data Test / Format Test / test_xml_corrupted_tags – test.test_convert_data_format.TestConvertDataFormat	<1s

Figure 6.6.: Failing Tests in Jenkins UI (Line Missing) (Screenshot)

It can be seen that all tests regarding XML corruption fail. This is because the centralized `remove_corrupted(...)` data handling function is not called inside the analytics script anymore. Since all handling measures are defined in that function, no measure is actually taken when the function call is missing. In case of the data handling function missing individual handling steps, the corresponding tests would fail. These test results are meant to provide valuable insights to the developer. The developer can analyze the problem, fix occurrences, and push a new commit to the pull request. When the test failures have been resolved, the updated version can be correctly deployed.

Similar failure detections occur with different changes of the source code. This shows that the pipeline is capable of stopping a deployment process based on the existing tests. This requires the assumption that all tests suites are complete and perform as expected.

6.2.2. Feature Creation Testing

The previous paragraphs covered the integration of an already existing feature. The following paragraphs deal with the creation and adequate testing of a new feature. The consideration of this practical area is important since the solution at hand can encounter changing requirements that need to be embodied accordingly.

For this evaluation, assume that the product owner of the MBA data pipeline requests a new feature that includes the weather data of the creation date of the purchase information to the analysis. This could be used to find a possible correlation between the purchasing behavior of the retail company's customers and the weather situation at the given point in time. The development teams find that the integration of weather data is mostly suited to be developed inside the Conversion Stage of the

data pipeline since it represents the first stage that has access to the raw purchasing data.

The naïve implementation of that feature is shown in Source Code Excerpt 15 below.

```python
def enrich_weather(dicts):
    for name, content in dicts.items():
        url = "http://api.weatherapi.com/v1/history.json"
        api_key = # omitted
        res = get(
            url,
            params={
                "key": api_key,
                "q": "48.781318, 9.180211",
                "dt": content["BusinessDayDate"],
            },
        )

        data = json.loads(res.text)["forecast"]["forecastday"][0]["day"][
            "avgtemp_c"
        ]

        content["temp"] = data
```

Source Code Excerpt 15: Naïve Implementation of a Weather Data Integration Feature

The feature's corresponding function is placed right after the reformatting of the POSLog dictionaries and, thus, right before the JSON file export. For each purchase, the weather that corresponds to the transaction date and place is requested from an external weather API (ll. 5–12). For simplicity reasons, the average weather data in degrees Celsius is extracted (ll. 14–16) and added to the dictionary (l. 18). There are multiple issues that can occur when such a code fragment is integrated to the solution.

6.2.2.1. Feature Testing Limitations

When performing an update of an existing feature, the validation of the update can be conducted by running the preexisting tests of the corresponding feature. However, the development of a new feature also means writing new tests. The CI/CD pipeline can only stop a feature from being deployed into production when corresponding feature tests fail. In case that these tests do not exist, the CI/CD pipeline cannot recognize any issues and publishes an untested feature. This situation is highly undesired and problematic for the integrity of such a solution. Plus, it yields discussions that go

beyond the technical aspects of data analytics development but need to deal with the developer's understanding and acceptance of an agile development workflow.

In agile development, a developer is always in charge of testing the feature that he or she works on [8, p. 18]. Principles like Test-Driven Development (TDD) have been introduced to support the importance of the testing process [48, p. 1]. DataOps testing principles might develop this into a data-driven approach (i.e., test data first, then solution test, then feature implementation). Nevertheless, from a general point of view, these principles can be suggested or required by convention, but not technically enforced. A separate tool could evaluate if tests exist for a given feature by executing all test suites and calculating the feature's occurrences in these tests. This measurement is referred to as the *test code coverage* [49, p. 15]. However, such a mechanism could be bypassed by simply executing a function inside a test case. This could presume the feature being tested and allow the change to be deployed into production. Other, more complex metrics (e.g., defect fix retest, test effectiveness, etc. [49, p. 15]) could be suited for more in-depth testing quality assurance, but would still be required to be implemented and reevaluated for the given use cases over time. All in all, DataOps requires the acceptance and consciousness of agile processes and does not only rely on technical measurements to control their fulfillment.

Even with appropriate testing, the issue is still present. This is because the feature and the corresponding tests are conducted by human developers. The person behind a feature might have a different understanding of the feature than the product owner or simply miss an edge case during testing that could lead to further issues. Specifically, the developer might implement the feature to provide weather data in degrees Fahrenheit and test the value correctness accordingly. The product owner or other developers might expect degrees Celsius instead. As with pure software development, such a feature change also requires updating the solution requirements and general collaboration and communication between developers and development teams of the separate stages. In general, all contributors should have the same understanding of a feature in order to prevent inconsistencies.

This issue also introduces the discussion of *testing tests* that are meant to test the functionality of the source code. The addition of testing validation based on previously mentioned complex testing quality metrics might mean more effort. With these validation systems, the question behind *their* validity can be posed and the discussion continues. This ultimately means that automated tests might not suffice for confident and correct deployment of such a solution. Apart from that, peer code reviews and other interdisciplinary measures might need to be taken in order to rule out mistakes that cannot be uncovered by traditional software solution testing.

6.2.2.2. Importance of Regression Testing

Another issue can be found in regression. Assume that a previous MBA needs to be re-conducted because an error inside its values is suspected. The data from the target MBA originates from a time before this feature was requested. The previous MBA did also not consider weather data for its analysis. When the analysis is repeated, the MBA might, purposefully, yield different results. However, this behavior is not desired. The addition of a new feature may not lead to inconsistencies between two versions of the same analysis. This would create a new data quality issue and result in further issues in the upcoming analytical stages. Instead, this feature may only be used for new analyses (e.g. by specifying a minimum date that needs to be present in the input data). This behavior correctness needs to be ensured by including a corresponding test to the testing suite of the solution.

Another form of regression testing in this case is the execution of previous tests. Their passing will ensure that the new feature does not change the behavior of previously existing features. Since this feature only adds an attribute to the output of the stage, the previous features and feature tests run correctly. Other feature implementations might require the change of preexisting features which then might require the adaptation of the corresponding tests, etc.

All in all, the general testing process, including its limitations and technical bottlenecks, can be compared to agile DevOps testing. On the other hand, regression testing plays a priority role in data analytics testing and needs to be taken into account for each feature. This finding preliminarily corresponds with the hypothesis from Section 2.2.3. Currently, it can be said that the DataOps testing process is similar to DevOps, but takes data-specific requirements into account.

6.3. Impact Analysis of Different Testing Levels

In order to evaluate further potential differences to the DevOps testing process, the following section deals with the different testing levels (unit, integration, and end-to-end testing) in the MBA DataOps pipeline environment. This part of the evaluation is expected to yield insights about the specific impact of the testing levels in such a data-driven environment, built up in a highly externalized technical infrastructure.

This evaluation is conducted based on testing use cases that occur at different hierarchical levels of the testing process. These use cases are not explicitly tested for

by the solution. The evaluation will provide information on how insightful the error recognition of the different testing levels is for recovering from the problem. The use cases are described as follows:

Non-Existing S3 Bucket URI Provided The format of the URI is correct, but the bucket does not exist (typographic error, deprecated bucket reference, etc.).

Invalid AWS IAM Credentials Provided The format of the IAM credential key pair is correct, but it either does not provide sufficient access to the resources, or it is deprecated .

Container Task Ran Out of Memory The container that an analytics stage is run in cannot finish its job because it requires more memory than expected.

6.3.1. Failure Case Execution

These cases are designed and invoked, leading to the following results:

6.3.1.1. Non-Existing S3 Bucket URI Provided

Non-Existing S3 Bucket URI Provided

Level	Unit	Integration	End-to-End
Result	passed	failed	*skipped*
Information		`boto3` Exception: `404 Not Found`	

Table 6.1.: Testing Evaluation: Non-Existing S3 Bucket URI Provided

As can be seen in Table 6.1, the corresponding unit test only tests if S3 URIs of incorrect format are caught and reported. Since the URI at hand is correct, the unit test passes. The integration (data) test actually connects to the bucket, resulting in an `boto3` exception, reporting that the provided S3 bucket does not exist.

The process of mocking the S3 connection service inside the unit test would not have helped here since the actual existence of the infrastructure can only be evaluated within the scope of the infrastructure.

6.3.1.2. Invalid AWS IAM Credentials Provided

Invalid AWS IAM Credentials Provided

Level	Unit	Integration	End-to-End
Result	passed	failed	*skipped*
Information		boto3 Exception: `Access Denied`	

Table 6.2.: Testing Evaluation: Invalid AWS IAM Credentials Provided

This case is similar to the incorrect URI and results in the same test outcome, shown in Table 6.2. Only the infrastructure itself can evaluate the credentials and report an issue. This is why, again, the unit test passes, while the integration test results in another exception, mentioning that the provided credentials cannot be used for the current operation.

6.3.1.3. ECS Container Task Ran Out of Memory

ECS Container Task Ran Out of Memory

Level	Unit	Integration	End-to-End
Result	passed	passed	failed
Information			boto3 Exception: `OutOfMemoryError: Container killed due to memory usage`

Table 6.3.: Testing Evaluation: ECS Container Task Ran Out of Memory

In this case, both the unit and integration tests do not recognize the problem. Only the end-to-end test provides the information that the corresponding ECS task has failed because of lacking memory capacity, shown in Table 6.3 This is because the integration tests are performed in containerized environments on the Jenkins server instance. These Docker containers do not have any resource limitations [37], other than the ECS tasks that require performance specifications in their definitions. Only the end-to-end test makes use of this infrastructural service inside the solution.

6.3.2. Testing Impact Evaluation

The presented use cases yield a number of insights. First, the typical, DevOps-inspired testing level hierarchy and its degrees of testing isolation do not hold in an

environment that is mostly created out of external resources. The cloud-driven infrastructure cannot be taken into account by unit tests since these are supposed to neglect these aspect and purely focus on the feature under test. When the infrastructure access is a crucial component of the feature, the unit test cannot recognize any issues.

Another aspect that follows is the strict separation of the testing levels. Taking the findings of the failure use cases into account, all infrastructure-related testing might be conducted during integration testing. This could also include URI and IAM credential format tests that do not require infrastructure access but categorically fit inside the same testing suite. This aspect has partly been considered during development: certain data handling function tests could have been created in the domain of granular unit tests. Since their overall features relate to the data-driven part of the solution, which has been tested during integration testing, the remaining tests are performed similarly.

Finally, the last failure use case that was only recognized by end-to-end testing opens up another discussion topic. In general, the performance measures could have been implemented into the integration testing level that makes use of Jenkins-driven Docker containers. This would require additional testing effort and, potentially, the redesign of the deployment structure within the CI/CD pipeline. While the overall reason behind testing should be the validation of feature functionality, strictly focussing on the definition of the testing levels could increase time and cost inside the development process.

As emphasized throughout this entire work, DataOps use cases and their corresponding automation and testing designs highly differ depending on the solution requirements, data governance definitions, etc. For the MBA data pipeline, unit tests provide little value to the actual testing expressiveness. There exists a general discussion questioning the value of unit tests [50][51]. This finding goes hand-in-hand with this discussion. The unit test cases might be included into the integration testing area and de-granularized for potentially better testing productivity. Further evaluation could give more insights on the impact differences between integration and end-to-end testing, leading to a generally more specialized, efficient and less tradition-driven testing architecture.

In generalization and comparison to the DevOps testing process, it can still be said that automated testing is performed inside its designated steps inside the CI/CD process prior to deployment. However, data analytics solutions often have to deal with externalized services, that cannot be neglected during testing. Instead of sticking to the traditional testing pyramid, each solutions' architecture needs to be evaluated

from a testability point of view in order to create a well-suited testing framework inside the DataOps use case.

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] *DevOps tech: Continuous testing*, Google, Jul. 2020. [Online]. Available: `https://cloud.google.com/solutions/devops/devops-tech-test-automation` (visited on 08/20/2020).

b

[23] H. Vocke, "The Practical Test Pyramid", *martinFowler.com*, Feb. 2018. [Online]. Available: `https://martinfowler.com/articles/practical-test-pyramid.html#TheTestPyramid` (visited on 08/20/2020).

[24] 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).

[25] 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).

[26] "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).

[27] 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`.

[28] 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`.

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

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

[31] *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).

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

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

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

[35]   *AWS boto3 (Documentation)*, Amazon Web Services, Inc, 2020. [Online]. Available: `https://boto3.amazonaws.com/v1/documentation/api/latest/index.html` (visited on 08/16/2020).

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

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

[38]   *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).

[39]   *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).

[40]   *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).

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

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

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

[44]   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).

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

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

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

[48]   I. Karac and B. Turhan, "What Do We (Really) Know about Test-Driven Development?", *IEEE Software*, Jul. 2018. DOI: `10.1109/MS.2018.2801554`.

[49]   T. Garrett, "Useful Automated Software Testing Metrics", *IDT, LLC*, 2011. [Online]. Available: `https://idtus.com/wp-content/uploads/2019/08/UsefulAutomatedTestingMetrics.pdf` (visited on 08/20/2020).

[50]   J. O. Coplien, "Why Most Unit Testing Is Waste", *RBCS*, [Online]. Available: `https://rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf` (visited on 08/20/2020).

d

[51] A. Golub, "Unit Testing is Overrated", *tyrrrz.me*, Jul. 2020. [Online]. Available: `https : / / tyrrrz . me / blog / unit - testing - is - overrated` (visited on 08/20/2020).

e

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)

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)

h

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)

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)

j