Exploring the Test Driven Development of a Fraud Detection Application using the Google Cloud Platform

Daniel Staegemann[a], Matthias Volk[b], Maneendra Perera and Klaus Turowski
Magdeburg Research and Competence Cluster VLBA, Otto-von-Guericke University Magdeburg, Magdeburg, Germany

Keywords: Test Driven Development, TDD, Microservice, Big Data, Google Cloud Platform, GCP, Fraud Detection.

Abstract: The concept of big data hugely impacts today's society and promises immense benefits when utilized correctly, yet the corresponding applications are highly susceptible to errors. Therefore, testing should be performed as much and rigorous as possible. One of the solutions proposed in the literature is the test driven development (TDD) approach. TDD is a software development approach with a long history but has not been widely applied in the big data domain. Nevertheless, a microservice-based test driven development concept has been proposed in the literature, and the feasibility of applying it in actual projects is explored here. For that, the fraud detection domain has been selected and a proof-of-concept online fraud detection platform is implemented, which processes real-time streaming data and filters fraudulent and legitimate transactions. After the implementation, an evaluation was carried out regarding test coverage and code quality. The automatic code analysis reports revealed that TDD had produced very reliable, maintainable, and secure code at the first attempt that is ready for production. Finally, the evaluation revealed that it is highly feasible to develop big data applications using the concept mentioned. However, choosing suitable services, tools, frameworks, and code coverage solutions can make it more manageable.

1 INTRODUCTION

Due to the ever-increasing importance of knowledge and information, the concept of big data (BD) hugely impacts today's society and promises immense benefits when utilized correctly (Müller et al. 2018). While there are several slightly varying explanations of BD, the most prominent one is provided by the National Institute of Standards and Technology (NIST), which states that big data "consists of extensive datasets primarily in the characteristics of volume, velocity, variety, and/or variability that require a scalable architecture for efficient storage, manipulation, and analysis" (Chang and Grady 2019). The potential use cases for BD are manifold (Volk et al. 2020), leading to its adoption in numerous domains and industries. However, the corresponding applications are highly susceptible to errors. Therefore, testing should be performed as much and rigorous as possible. One of the solutions proposed in the literature is the test driven development (TDD) approach (Staegemann et al. 2020) that is further detailed in the following section.

To assure a comprehensive quality assurance, the testing should be performed on all levels, namely method, subcomponent (a single microservice), component (a group of microservices that contentually belong together), and the system in its entirety.

For the application of TDD in the BD domain, the use of microservices has been suggested (Staegemann et al. 2020). The core objective of those is to allow loosely coupled, self-contained modules or services that are created to solve one specific task, have their own resources and can be deployed separately. Several asynchronous communication patterns can be used among services and messaging; the event-driven approach and RESTful connections are some widely used patterns in software engineering. Also, since they are independent components, different programming languages can be used for the implementation (Shakir et al. 2021).

[a] https://orcid.org/0000-0001-9957-1003
[b] https://orcid.org/0000-0002-4835-919X

Staegemann, D., Volk, M., Perera, M. and Turowski, K.
Exploring the Test Driven Development of a Fraud Detection Application using the Google Cloud Platform.
DOI: 10.5220/0011559000003335
In *Proceedings of the 14th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2022) - Volume 3: KMIS*, pages 83-94
ISBN: 978-989-758-614-9; ISSN: 2184-3228

While the application of TDD in BD has already been demonstrated (Staegemann et al. 2022), the corresponding body of literature is still relatively sparse. Therefore, the publication at hand aims to expand it by presenting an additional use case from a complex and demanding domain. Therefore, fraud detection was chosen as the application area. Moreover, since cloud offerings such as the Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft Azure are highly popular, instead of building a solution completely from scratch, in this project, the GCP is used to explore the approach's applicability when using a cloud provider and (at least partly) its provided tools (e.g. Pub/Sub and Dataflow). Consequently, the focus of this paper is also on the realization of the TDD itself and not on the optimization of the fraud detection algorithm's accuracy.

2 TEST DRIVEN DEVELOPMENT

In the literature (Staegemann et al. 2021), the application of TDD is highlighted as a promising way to improve the quality of an implementation as long as the associated increase in development time and effort is considered an acceptable trade-off.

The approach aims at improving the quality of the product under consideration by mainly influencing two aspects. On the one hand, it aims to increase test coverage, which helps to find and subsequently fix problems that occurred during the implementation of the artifact in question. On the other hand, TDD also influences the design process itself by leading to a more manageable and pre-planned structure that helps to avoid bugs and incompatibilities (Crispin 2006; Shull et al. 2010). The main application area is software development, but the special case of implementing BD applications (Staegemann et al. 2020), process modelling (Slaats et al. 2018), or developing ontologies (Davies et al. 2019; Keet and Ławrynowicz 2016) are also found in the literature.

In the "traditional" way of software development, once a function or change is envisioned to be realized, it is implemented and then tested. In contrast, in the test-driven approach, the order of implementation and testing is reversed. That is, after the desired change is conceived, it is broken down into its smallest meaningful parts (Fucci et al. 2017). For these, one or more tests are written to ensure that the required functionality is provided. Then, the tests are executed and are expected to fail because the actual functionality is not yet implemented (Beck 2015). Only then is the productive code written to provide the new functionality. Factors such as the elegance of the code are not yet considered; instead, the simplest solution is sought. Once the code is ready, it must pass the previously written tests (Crispin 2006). If it is successful, the code is refactored to improve aspects such as readability or compliance with standards and best practices (Beck 2015). In the process, functionality is constantly validated against the tests.

However, this approach not only affects test coverage, but also the design of the software by using small work packages instead of large tasks. In addition, this focus on incremental changes (Williams et al. 2003), which intertwines testing and implementation, provides more immediate feedback to the developer by resulting in short test cycles (Janzen and Saiedian 2005). Although most tests are written specifically for these small units, other tests such as integration, system, or acceptance tests can also be used in TDD (Sangwan and Laplante 2006). In addition, to fully exploit the potential of TDD without tying up the developer's attention by forcing him to manually execute the tests, TDD is often used in conjunction with test automation in a continuous integration (CI) context (Karlesky et al. 2007; Shahin et al. 2017). To ensure that the latest code addition or change does not negatively impact existing parts of the implementation, a CI server automatically starts and re-executes all applicable tests when a new code commit is registered by the versioning system.

3 THE IMPLEMENTATION

The developed solution consists of multiple microservices, and creating the application is divided into three major parts. The first step deals with the development of a *machine learning model*. It is used to identify a given transaction as fraudulent or genuine. Here, tasks like selecting suitable training data, pre-processing those data, creating the machine learning model, training the machine learning model, and deploying the machine learning model for online predictions are carried out.

Afterwards, the *rule engine*, which validates the transaction is developed. There are different ways to identify fraud, and traditionally rule-based fraud detection approaches have been used before introducing machine learning application approaches (Mekterović et al. 2021). Rule-based systems use conditions, and if the corresponding state is matched, the transaction will be marked as fraud or potential fraud. Therefore, this work uses machine learning and rule engine-related fraud detection to improve

prediction accuracy. Further, by introducing a rule engine, it is also aimed to identify the feasibility of applying TDD in the rule engine microservice.
Finally, the *online fraud detection platform* is developed. This is the place where the microservices are connected with each other for the final goal of identifying the transaction as fraudulent or not. An online fraud detection platform is a streaming application that initiates the analysis as soon as a transaction is received and sends the transaction status as the outcome. The design and implementation of the streaming pipeline are carried in this step.

3.1 Machine Learning Model

Fraud detection is a binary classification problem that categorizes the transaction as fraudulent or legitimate (Maniraj et al. 2019). The GCP platform provides several tools and services to build classification machine learning models. However, for building such a model, a suitable dataset is needed. Therefore, at first, a domain-specific dataset was chosen from *Kaggle*[1] was selected for the project, which is one of the most popular online communities for people interested in machine learning and data science. This dataset contains a simulated credit card transaction database that was created using the Sparkov data generator[2], and it contains legitimate and fraudulent transactions from 1st Jan 2019 to 31st Dec 2020. It covers credit cards of 1000 customers doing transactions with a pool of 800 merchants. The dataset's features are known, and the documentation of the dataset concerning the features is self-explaining. The dataset contains fraudTest.csv and fraudTrain.csv, dividing the dataset into two parts. However, the dataset is highly imbalanced because fraudulent transactions are only 0.52% out of all the transactions. This data distribution needs extra attention when building the machine learning model because the classification machine learning model's performance and accuracy strongly depend on the distribution of classes.

3.1.1 Pre-processing

The quality of machine learning models is highly dependent on the quality of the input provided for the model. Data is highly error-prone and needs cleaning and transformation before using them in the model training. Incomplete data, inconsistent data, and missing data are prevalent issues visible in datasets, and 70%-90% of project efforts are utilized for data wrangling, which is data understanding and transformation (Nauck 2019).

Pre-processing is the first step carried out in the model creation, and the GCP has several tools that can be used for cleansing and transforming data. Yet,

Table 1: The performed pre-processing steps.

Step	Activity
1	Scan raw dataset to check if there are any missing values in the TARGET column, *is_fraud*, and remove the rows where the TARGET is missing.
2	Search columns with a high number of missing values and remove those where half of the rows have missing values.
3	Delete duplicate rows and columns with a single value as they do not add value to the machine learning model's creation.
4	Categorical data columns are transformed into numerical. (Even though this was not used for the fraud detection dataset as the finally used *AutoML Tables* can handle absolute values, the tests and implementation are available for this pre-processing step.)
5	Missing values are imputed using the strategy *mean* in *sklearn.impute.SimpleImputer*, which replaces the missing values with the mean of each column.
6	Perform outlier removal using the *Inter Quartile Range mechanism*, the most trusted mechanism for removing unusual data points [4]. (The inter quartile range depicts a measure in descriptive statistics that tells the middle range of the dataset.)
7	Select the top features for the machine learning model by calculating correlations between the columns and the target. Apart from the above automatic pre-processing, manual pre-processing is also done to remove duplicate columns like merchant latitude and merchant longitude as longitude and latitude columns are already present.
8	Add a new data column named *data_split* (explained in the following section)
9	Write the cleaned data to a CSV file.

[1] https://www.kaggle.com/kartik2112/fraud-detection

[2] https://github.com/namebrandon/Sparkov_Data_Generation

when applying TDD, internal constraints would have increased the complexity and, therefore, another approach was needed. Using Python libraries like *sklearn*, *pandas*, and *numpy* to process and transform data is another popular and widely used method in machine learning, and it was well suited for TDD. Hence, the following cleansing and transformation steps were carried out using the abovementioned libraries, following the steps outlined in Table 1.

3.1.2 Machine Learning Model

After pre-processing, the second step of building the machine learning model is selecting a suitable GCP tool and implementing the fraud detection classification model. *AutoML Tables* is a popular GCP tool that automatically builds and deploys a machine learning model (Google 2022a). It has a simplified process, and it automatically selects the best model for the dataset provided. Furthermore, it maximizes the quality of the model by increasing the accuracy, and decreasing the Root Mean Square Error (RMSE), without manual intervention for feature engineering, assembling, etc. (Google 2022a). *BigQuery ML* is also another candidate for model creation. However, since the main focus is on increasing the model quality rather than model experimentation, the AutoML Tables are used.

In AutoML Tables, the dataset is split automatically with 80% for training, 10% for validation, and the other 10% for testing. Nevertheless, AutoML recommends a manual split for highly unbalanced datasets as an automatic split may result in fewer samples of the minority class for the testing split (Google 2022a). Therefore, a new column was added to the dataset called *data_split*, which contains three categorical values TRAIN, VALIDATE and TEST, constructing a custom split during model creation.

Finally, the updated pre-processed fraud dataset CSV file is uploaded to a *storage bucket* [3] via *Terraform* [4] to be able to access AutoML Tables. After uploading the CSV file, to create an AutoML Table dataset and create the model, Python SDK was used as Terraform still does not support AutoML Tables. Once the model is created, it is deployed (for this purpose, the *google-cloud-automl_v1beta1* Python module was used) in the Artificial Intelligence (AI) platform for online prediction. The GCP AI platform supports creating, training, and hosting the built machine learning model. Furthermore, it supports monitoring the online predictions and maintaining the model and versions.

3.2 Rule Engine

After creating the machine learning model and deploying it in the AI platform for online prediction, the second step was implementing the rule engine. There are two main models, rule-based and algorithmic models that are used to detect fraud. In rule-based models, multiple conditions identify the transaction as fraudulent or legitimate. This project uses the rule engine as a microservice hosted in a compute engine. It is the first component of the platform where the online transaction gets validated, and it has several rules for unusual attributes. In real-time, when a transaction is initiated, Pub/Sub will create a message and pass it to Dataflow. Then Dataflow will invoke the rule engine application.

The rule engine was developed as a RESTful web service using *Spring Boot*. It has a POST endpoint called "/api/isTransactionValid" and invoking the endpoint with the transaction details will return the transaction validity. It must be deployed on a server to use the web service. Therefore, a virtual machine is created in *Compute Engine* as the first step. Then *Tomcat*, the server software, and *MySQL 8.0* are installed as the web service uses it. After performing all these installations, the web service is bundled as a

Table 2: Checks performed by the rule engine.

Rule	Threshold
Check if the transaction amount exceeds the daily transaction amount	500
Check if the daily transaction total exceeds the daily transaction total	1000
Check if the daily transaction count exceeds the daily transaction count	5
Check if the time difference of the subsequent daily transactions is greater than the minimum time difference	5 seconds
Check if the distance of the subsequent daily transactions is lesser than the maximum distance	1000 km

[3] Storage buckets are containers that can hold data so that other GCP services can connect to these containers and access the data.

[4] Terraform is an infrastructure as code tool that enables the infrastructure's safe and efficient building and maintenance.

Web Application Resources (WAR) file and deployed on the server. When the transaction details are sent to the web service, it will validate the details against the five rules depicted in Table 2, which are created based on common factors used in fraud detection models (Huoh 2017).

After validating the transaction against the rules, the rule engine sends a status to the dataflow component. Regardless of the status being valid or invalid, it will be forwarded to the hosted ML model via Dataflow to further processing. The web service is developed using the Controller Service Repository pattern, a widely used design pattern. It breaks the business layer into the repository layer, which connects to the database, and the service layer, where actual business logic is implemented (Jones 2019). Moreover, it provides a more flexible architecture and allows to write unit tests quickly (Jones 2019).

3.3 Online Fraud Detection Platform

The final phase of creating the application is the development of the microservice-based online fraud detection platform using GCP products. It is designed to handle real-time streaming transaction data, process it immediately, and send the result to the relevant notification services. The selection decision for each service is discussed in the respective section.

3.3.1 Pub/Sub

In the fraud detection platform, transaction details come from different services like desktop computers, mobile phones, or laptops. Also, the final status of the transaction should be sent to different sources which initiated the transaction. Therefore, Pub/Sub messaging is appropriate for this scenario as all the initiating devices can send the transaction details to a single topic. Furthermore, once the processing is completed in the system, the same devices can subscribe to a single topic to determine the conclusive status of the transaction. Again, it is capable of ingesting data to streaming pipelines like Dataflow. Hence, Pub/Sub is selected, a real-time asynchronous messaging service provided by GCP. It is utilized to ingest transactions into the system and send the fraud transaction notifications from the system. Foremost, two topics and their respective subscripts are created in the cloud project via Terraform script. Then the Java code is written for message publishing and subscribing.

Here a topic called *transaction-topic* exists and all publishers are publishing transactions on this topic. Furthermore, this topic has a subscription called *transaction-subscription*, and Dataflow retrieves all the messages from this subscription and processes them either as fraud or legitimate. Then the fraud transaction details are written to the *fraud-status-topic*, and legitimate transaction details are written to the *transaction-status-topic*. Then a cloud trigger can be attached to the *fraud-status-topic* so that when a new message comes, a notification (e.g., a SMS) can be sent to the relevant parties.

3.3.2 Dataflow

Fraud detection should be done in real-time. Therefore, a service capable of handling transaction details in real-time, processing, and then sending the response to the relevant parties is required to fulfil this scenario. Dataflow comes in handy in this situation, and it is a service of GCP capable of processing different data patterns. It executes the batch and streaming pipelines developed using *Apache Beam SDK*, an open-source programming model (Google 2022b). Therefore, Dataflow is used in the project, and the message received from the Pub/Sub is transferred to process it and take necessary actions.

First, transactions should be ingested into the real-time stream pipeline to start the streaming job. Then, the transaction details are published to the described transaction topic, and Dataflow gets them for further processing. Therefore, Dataflow reads the messages in the transaction topic as the first job.

After getting the transaction details, the actual business process of identifying the transaction as fraudulent or legitimate should be initiated. For that, Dataflow invokes the Rule Engine hosted in the Compute Engine. Then the response is retrieved from the RESTful endpoint about the transaction validity and whether the transaction is valid. This information is kept in the memory to initiate the next step.

Here, the transaction fraud status is retrieved from the machine learning model. Both algorithmic and rule-based methods identify fraudulent transactions in this online fraud detection platform. In the previous step, the rule engine was invoked, and here the same is done with the machine learning model, which is hosted in the AI platform to predict whether the transaction is fraudulent. Then the response retrieved from the machine learning model is kept in memory, and both the machine learning response and the rule engine response are carried to the next step of the job.

The responses obtained from both models are essential to identify the transaction as fraudulent or legitimate. Furthermore, it is important to persist the results and the transaction details for future use.

Therefore, in the following step, transaction details are written to the BigQuery table called *transaction_fraud_status_data_table* for future reference. For example, when the table row count exceeds a certain threshold, a new machine learning model can be trained using the old dataset and the persisted data in the BigQuery ML table.

After saving the fraud status, it is essential to send the transaction status to the platforms that initiated the transactions, whether to continue processing or terminate the process if the transaction is potentially fraud suspicious. Therefore, filtering of the transactions happens as the next step. First, transactions are filtered as fraud or genuine, and in this step, Dataflow produces a new message with fraud transaction details along with the fraud status and sends it to the topic called *fraud-status-topic*.

In the filtering process, if the transaction is considered genuine based on the machine learning response and rule engine response, Dataflow produces a message with transaction details and transaction status. It is published on a topic called *transaction-status-topic*. Then the platforms subscribed to this topic can retrieve the data and continue processing the transaction.

3.3.3 BigQuery

Fraud detection is a dynamic process that does not have an ending. Organizations should continually monitor fraud detection systems and make necessary enhancements to the systems to adapt to the trends (Wedge et al. 2019). Continuous monitoring, learning from the incidents, and incorporating knowledge gained from the past incident are essential. Therefore, keeping track of indicants and their response should be mandatory, and for this, BigQuery can play a significant role. BigQuery is a data warehouse that is cost-effective and scalable. Also, it is the input source for most of the model creation services like BigQuery ML and AutoML Tables. After invoking the rule engine and machine learning model in this pipeline, the transaction details with their validation and prediction status are written to a BigQuery table for future reference and use. In the project, a table called *transaction_fraud_status_data_table* is created using Terraform and writing data to the table is implemented using Java inside the Dataflow transform. The data written to tables can be harnessed for different analytical purposes.

3.4 The Testing

After the previous section introduced the developed application, in the following, it is outlined how the test driven approach has been realized.

3.4.1 TDD in the Machine Learning Model

Since the focus of the conducted project is on the application of TDD, it is essential to understand the test structure and test cases precisely. Therefore, *Python unittest* is employed instead of *Pytest* as the tests are easily understandable. Tests written for pre-processing functions are executed locally in the Eclipse IDE. However, this is not possible for all parts of the application, leading to varying testing approached outlined in the following. However, it is impossible to run the AutoML Table's related tests locally as they need to be executed in real environments. Some of the GCP services like Pub/Sub, Bigtable, and Spanner provide emulators which provide the capability of developing and testing the application locally in a simulated environment (Google 2022c). However, since the AutoML Tables do not have the emulation option, tests were executed in the real environment[5]. Before creating the AutoML Table model, datasets should be imported to a storage bucket or a *BigQuery* table. In the project, the datasets are uploaded to a storage bucket called *fraud_detection_data_bucket* and then imported to the AutoML Table. To create the storage bucket and upload the CSV files, Terraform was used, and before that, *Chef InSpec*[6] tests (controls) were written to verify the outputs. Chef InSpec GCP is a resource pack that can be used to write tests for GCP resources. Afterwards, the Terraform scripts are written and executed, and then again, Chef InSpec tests are executed to verify that the resource creation is successful.

3.4.2 TDD in the Rule Engine

During the development of the rule engine, TDD is used, and *JUnit Jupiter* and *spring-restdocs-mockmvc* libraries are used to write the tests. They are the general libraries used in writing tests in Spring Boot RESTful web service applications. Since the *Controller Service Repository pattern* is used, the repository tests are first created, and then the repository layer is implemented. Afterwards, the

[5] To save costs, this can be done using a small test dataset, since only the connectivity but not the AutoML can and need (since it is a proven component) to be tested.

[6] https://github.com/inspec/inspec-gcp

service tests are written mocking the repository layer, and then the service layer is implemented. Finally, controller tests are written mocking the service and repository, and then the controller is implemented, which has the POST endpoint. After writing the code, the controller is tested using POSTMAN.

Following the local testing, the code is bundled as a WAR file (a collection of Java classes and all necessary resources to be deployed on a server) and deployed in the compute engine. Before creating the compute engine virtual machine instance in the GCP, the Chef InSpec test is written for testing the resource creation. Subsequently, the compute engine instance is created using Terraform. After creating the virtual machine, the Chef InSpec test was executed to verify the resource exists in GCP with specific properties (e.g., machine type). Then in the VM, Java, Tomcat, and MySQL are installed to be able to run the web service and the WAR file is deployed. After deployment, POSTMAN was used to send the requests to the REST API and verify that the deployment was successful.

3.4.3 TDD in the Online Fraud Detection Platform

Corresponding to the description of the platform, its testing also comprises three components, namely Pub/Sub, Dataflow, and BigQuery. Prior to the creation of the Pub/Sub topics and subscriptions, Chef InSpec tests are written to assure the resource exists in the GCP project. Before executing the Terraform scripts they failed but were passed after running it. Then, before writing the Java code to publish and subscribe messages, *JUnit tests* were written for every function. The tests are executed locally and in the actual environment. GCP provides a Pub/Sub emulator, which helps develop and test applications locally without connecting to the actual production environment. The emulator supports the creation of topics and subscriptions, publishing messages, and subscribing messages (Google 2022a).

Dataflow is the streaming platform that integrates all other services and produces the final result. First, it receives input data from the Pub/*Sub*, and finally, the result is written to *BigQuery* and relevant *Pub/Sub* notification channels. Different services use different data formats, and *Dataflow* transforms them accordingly to be used in the job. Handling different data formats and transforming them is complex and highly error prone. Thus, it is essential to test all these transformation steps to verify that no issues are introduced during the process. Furthermore, *Dataflow* integrates all the services, and it is essential to verify that all the integrations happen successfully without any issues. Consequently, transformations and integrations need thorough testing. The *Apache Beam SDK* supports these two types of local testing, testing transformers that transform input data to another format to be processed in the next step and testing the end-to-end pipeline (The Apache Software Foundation 2021). It has a runner called *DirectRunner*, which runs the pipeline locally on a small scale. In the project, an instance of *TestPipeline* is created, primarily used for testing transformers. All the data transformations are tested using *PAssert* statements, which can verify the content inside collections. Once all the PTransforms are tested, the end-to-end pipeline testing was done, using a test that executes all the PTransforms in the pipeline. This verifies that all the integrations are working without any issue. However, all these above tests are executed locally, and it is necessary to test the entire pipeline in the actual environment. Therefore finally, a test has been written to execute the entire pipeline in the actual GCP project and verify the successful execution via asserting the pipeline result state.

Dataflow writes the transaction details and the fraud status to the BigQuery table for future reference in the online fraud detection platform. First, verifying that the transaction fraud data table exists in the GCP environment is necessary. Hence, as the first step, the Chef InSpec test is written to verify the table's existence. Initially, the test failed before executing Terraform script, which creates the BigQuery table, and it was passed after the resource had been created.

Once it is assured that the table exists in the cloud, it is essential to test the functionality of converting the transaction details to a table row. In the Dataflow job, transaction details are converted to a table row, the input for a BigQuery table, and written using the following properties: *BigQueryIO.Write.CreateDisposition.CREATE_NEVER* and *BigQueryIO.Write.CreateDisposition.WRITE_APPEND*.

The *Apache beam SDK* provides some classes like *FakeBigQueryServices* (a fake implementation of BigQuery's query service), *FakeDatasetService* (a fake dataset service that can be serialized for use in testReadFromTable), *FakeJobService* (a fake implementation of BigQuery's job service.), which simulate the real environment without connecting to the GCP project to test the BigQuery related functionalities locally. Therefore, JUnit tests are written using the above libraries to test the BigQuery table writing options.

4 DISCUSSION

By implementing the above-described project, it has been shown that the application of TDD in the given context is possible. However, to further evaluate its use, several aspects, such as the test coverage or the code quality, are regarded in the following, which is succeeded by some further observations.

4.1 Evaluation

While it is not feasible to specify each developed test case, as it would go beyond the scope, Table 3 gives a quantitative overview of the distribution of the tested aspects (those might, in turn, have more than one test case devoted to them).

In machine learning, testing as many aspects as possible is recommended (Nauck 2019), therefore, for the machine learning model, the tests comprise, inter alia, data quality and completeness, feature quality, and checks regarding data errors and consistency.

Dataflow is the main component of the online fraud detection model, and many tests have been written for it. Thorough pipeline testing is crucial to building effective data processing; testing the transformation functions and the end-to-end pipeline are the most critical steps for building a pipeline. In this project, tests are written for every transformation function, composite transformation function, and finally, test the end-to-end pipeline as suggested in the Apache Beam documentation (The Apache Software Foundation 2021).

The rule engine is created using the controller, service, and repository design pattern. It is essential to write tests for all three layers independently and then finally test as a whole. Therefore, tests are written for each layer mocking the other interacting layers, and finally, the entire system is tested using POSTMAN.

Overall, as shown in Table 4, the four testing levels (method, subcomponent, component, system) that are outlined in (Staegemann et al. 2020) could all be covered, emphasizing the feasibility of the proposition.

While the indicators of code coverage are somewhat tangible, assessing the code quality is less clear. There are various metrics in code quality, and different projects use different metrics based on their context.

For example, in TDD, overall quality, maintainability (Madeyski 2010; Shrivastava and Jain 2011), number of bugs (Borle et al. 2018; Khanam and Ahsan 2017), reliability, and code coverage by tests (Causevic et al. 2012; Madeyski 2010) are some of the widely used metrics. There are many online tools available to measure the code quality, and in this project, *SonarQube* is used, an open-source code review tool developed by *Sonar Source*. The tool has a free community version, and it can be installed locally to inspect the local project's code. It supports analyzing both Java and Python source code used in the project. Therefore, it was well suited for this context. Furthermore, SonarQube has numerous static code analysis tools supporting code quality and code security, considering aspects such as complexity, duplications, detected issues,

Table 3: The implemented test cases.

Service	Type of test case	Tested aspects
Machine Learning Model	Pre-processing test cases	16
	Model creation test cases	12
	Infrastructure test cases	2
Rule Engine	Infrastructure test cases	3
	Controller test cases	2
	Service test cases	23
	Repository test cases	14
Pub/Sub	General	4
Dataflow	General	1
	Data transforming test cases	15
	Pipeline test cases	5
BigQuery	General	3
	Infrastructure test cases	4
Online Fraud Detection Platform	General	4
Total		108

Table 4: The test levels.

Testing Level	Testing Process
Method	Unit tests are written for each microservice to assure the algorithm's correctness. They covered each feature like pre-processing and machine learning model creation. Furthermore, tests are composed for infrastructure creation and availability. Unit tests are written for each layer mocking the other dependencies in the rule engine microservice. In the fraud detection application, tests are created to verify dataflow pipeline creation, BigQuery table insertions, push notification sending, and subscriptions.
Subcomponent	Complete pre-processing and AutoML Tables functionality can be considered subcomponents of the machine learning part, and these are covered using the collection of unit tests. In other parts, the collection of unit tests can also be considered subcomponent tests when summed up. For example, the subcomponents tests will be all tests related to the layers of the rule engine, BigQuery table functionality, Pub/Sub functionality, and Dataflow functionality.
Component	The communication between the microservices is tested using Dataflow Pipeline, and different scenarios are written to verify that fraud transactions are identified accurately. In addition, the POSTMAN tool was used to test whether the REST endpoint is working correctly in the rule engine. While(Staegemann et al. 2020) has mentioned the assurance of the performance requirements like processing speed and capacity with benchmarking. However, no tests are written separately for speed and capacity as they can be directly monitored via GCP reports.
System	Finally, system tests are written considering the end-user requirements to validate the system as a single unit. The Cucumber framework was used to verify that the system works as expected.

maintainability, quality gates, reliability, security, size, and test coverage. In the following, the analysis the initial code quality of the projects without fixing any issues highlighted in the analysis.

Concerning the analysis for the rule engine, *QUALITY GATE STATUS=PASSED* assures that it is production ready.

Moreover, the analysis report reveals that its maintainability and security are at the highest level. In the report, two issues are visible, related to asserting dissimilar types.

However, those are minor issues that can be fixed quickly and do not impact the actual fraud detection logic when reviewed. Furthermore, the analysis shows 121 lines to cover. Yet, when reviewed, they are Plain Old Java Object (POJO) classes, servlet initialization code, and object transformation code that are not generally covered via tests and have a very low priority in testing. Therefore, the test coverage of the actual business logic can also be considered at a higher level.

The machine learning model is created using Python and SonarQube does not directly support the code coverage and analysis. Hence, the Coverage Tool is incorporated to assess the code coverage. The Code Coverage Tool generates a coverage XML which records the code coverage by tests and is then imported to SonarQube to view the complete analysis. For the analysis, thirty-seven python tests are executed via the tool. According to the results, the machine learning model creation project is also *QUALITY GATE STATUS=PASSED* which assures that the project is production-ready at the first development version without modifications.

Additionally, reliability, security, and maintainability ratings are at the highest level, being rated with an A, according to the analysis. The analysis report shows a code coverage of 67.2%, but when checking the not covered lines, some of them are Chef InSpec controls files, which are the tests for Terraform scripts, and some of the lines shown in the analysis are print statements that do not need the tests. So finally, 173 lines needed to be covered, which show as 485 in the report. When excluding the files whose tests are not essential, code coverage increased to 78.9%, which is good coverage for the first attempt. Further, it is very close to the code coverage rating A level, which is 80%.

The online fraud detection platform was also developed as a Java application, and according to the analysis, it also shows as *QUALITY GATE STATUS=PASSED*. That reveals it is ready to be deployed in production. However, the analysis shows seven bugs, but five are related to exception handling, which can be fixed easily when deep into the issue. The other two issues are also not critical and can be fixed easily. Security is an essential aspect of fraud detection systems as it deals with sensitive data, which needs higher protection. Measuring the security aspects with the code coverage tool, the

analysis shows ten security hotspots. When checking the detailed analysis, all of them are generated because of the statement *e.printStackTrace()*. In the analysis, maintainability shows as a rating A, which is one of the factors expected to be achieved via TDD. The test case count is different from the actual count as Cucumber tests and Inspect GCP controls are not picked up by the coverage tool. Three hundred sixty-nine lines needed to be covered in the code coverage, but when checked, some lines are covered via TestPipeline tests. They are not identified via the code coverage tools due to its handling in Apache Beam SDK. Therefore, considering the above fact, the precise lines to be covered are less than 100, which could be achieved quickly.

4.2 Further Observations

Many tools and services are available in GCP for developing big data applications. However, not all tools support TDD for different reasons, like not having well-defined APIs and the design of the service being unsuitable for TDD. Consequently, tools should be evaluated before using them in big data applications, and only the most suitable tools should be used for the purpose. Further, the GCP only provides limited support to test apps locally, and the provided emulators are also in the beta stage (Google 2022a). For this reason, most tests are executed in the actual production environment, and they incur costs. Furthermore, it will create the same resource multiple times, which is not demanded. Thus, not having a test environment to test the big data application is a limitation that needs to be resolved in the future.

After implementing the big data application, code coverage tools have been used to evaluate the project. For example, in the evaluation report of the fraud detection application, it was noticeable that the code related to Apache beam SDK transforms and function-related classes was marked as not covered in the report. However, those functions are covered using multiple tests and the code coverage tool was not always able to identify them accurately because of the structure of the code. Therefore, it is evident that code coverage tools cannot identify some of the complex code structures of big data applications. Hence, extra attention should be needed to select suitable tools for big data applications before selecting the tools.

The online fraud detection platform consists of different microservices. They are developed using different languages and using different frameworks. Because of this, a single test library is not sufficient for writing tests. In traditional projects, the JUnit and mocking libraries are sufficient to write all tests. However, different libraries are needed for big data applications due to their complexity. JUnit, unittest, Apache beam testing libraries, and BigQuery fake services are used to write tests in this project. Consequently, the developers should have the competencies to handle all these libraries.

Finally, in the literature it is considered best practice to execute all tests again after a single test failure. However, it was not feasible due to the incurring costs, and, therefore, the process was altered by executing only relevant feature tests after a test failure.

5 CONCLUSION

With today's society being more and more data driven, the concept of BD also gains significance. However, while the correct utilization promises immense benefits, assuring the quality of the corresponding systems is a challenging task. Aiming to facilitate it, the application of TDD in the BD domain has been suggested. Therefore, a project was conducted to further explore the general concept, show it in a fraud detection use case, and also examine it in the context of cloud provider services (such as the GCP in this case). In doing so it was shown that it is not only possible, but also yields good results regarding the test coverage and code quality, further substantiating the concept.

However, there were also some open challenges and starting points for future research that became apparent during the project. As indicated in the previous section, proper tooling is still an open issue that has plenty of potential for improvement. Further, some of the tests, like overfitting and underfitting, are not covered in the projects because they are not directly retrievable via AutoML responses. However, covering these types of tests adds more value to increase the quality, and in the future, research can be carried out to find a way to retrieve these values. For example, model hyperparameters-related information is logged in AutoML logs, and they can be analyzed to retrieve model training data. Finally, in this work, the feasibility of TDD is explored by applying it in the fraud detection domain. However, this approach for big data can be applied to other use cases so that more comprehensive insights can be gained, and collective insights will help for a better TDD process design.

REFERENCES

Beck, K. (2015). *Test-Driven Development*: *By Example*, Boston: Addison-Wesley.

Borle, N. C., Feghhi, M., Stroulia, E., Greiner, R., and Hindle, A. (2018). "Analyzing the effects of test driven development in GitHub," *Empirical Software Engineering* (23:4), pp. 1931-1958 (doi: 10.1007/s10664-017-9576-3).

Causevic, A., Punnekkat, S., and Sundmark, D. (2012). "Quality of Testing in Test Driven Development," in *Proceedings of the 2012 Eighth International Conference on the Quality of Information and Communications Technology,* Lisbon, Portugal. 03.09.2012 - 06.09.2012, IEEE, pp. 266-271 (doi: 10.1109/QUATIC.2012.49).

Chang, W. L., and Grady, N. (2019). "NIST Big Data Interoperability Framework: Volume 1, Definitions," *Special Publication (NIST SP),* Gaithersburg, MD: National Institute of Standards and Technology.

Crispin, L. (2006). "Driving Software Quality: How Test-Driven Development Impacts Software Quality," *IEEE Software* (23:6), pp. 70-71 (doi: 10.1109/MS.2006.157).

Davies, K., Keet, C. M., and Lawrynowicz, A. (2019). "More Effective Ontology Authoring with Test-Driven Development and the TDDonto2 Tool," *International Journal on Artificial Intelligence Tools* (28:7) (doi: 10.1142/S0218213019500234).

Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., and Juristo, N. (2017). "A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?" *IEEE Transactions on Software Engineering* (43:7), pp. 597-614 (doi: 10.1109/tse.2016.2616877).

Google. (2022a). "AutoML Tables-Dokumentation," available at https://cloud.google.com/automl-tables/docs, accessed on May 9 2022.

Google. (2022b). "Dataflow Documentation," available at https://cloud.google.com/dataflow/docs, accessed on May 10 2022.

Google. (2022c). "Testing apps locally with the emulator," available at https://cloud.google.com/pubsub/docs/emulator, accessed on May 10 2022.

Huoh, Y.-J. (2017). "Algorithmic and rules-based fraud models: A high-level look at two fraud models employed by financial service providers," available at https://fin.plaid.com/articles/algorithmic-and-rules-based-fraud-models/, accessed on May 9 2022.

Janzen, D., and Saiedian, H. (2005). "Test-driven development concepts, taxonomy, and future direction," *Computer* (38:9), pp. 43-50 (doi: 10.1109/MC.2005.314).

Jones, M. (2019). "The Repository-Service Pattern with DI and ASP.NET 5.0," available at https://exceptionnotfound.net/the-repository-service-pattern-with-dependency-injection-and-asp-net-core/, accessed on May 9 2022.

Karlesky, M., Williams, G., Bereza, W., and Fletcher, M. (2007). "Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns," in *Embedded Systems Conference,* San Jose, California, USA. 01.04.2007 - 05.04.2007, UBM Electronics.

Keet, C. M., and Ławrynowicz, A. (2016). "Test-Driven Development of Ontologies," in *The Semantic Web. Latest Advances and New Domains*, H. Sack, E. Blomqvist, M. d'Aquin, C. Ghidini, S. P. Ponzetto and C. Lange (eds.), Cham: Springer International Publishing, pp. 642-657 (doi: 10.1007/978-3-319-34129-3_39).

Khanam, Z., and Ahsan, N. (2017). "Evaluating the Effectiveness of Test Driven Development: Advantages and Pitfalls," *International Journal of Applied Engineering Research* (12:18), pp. 7705-7716.

Madeyski, L. (2010). "The impact of Test-First programming on branch coverage and mutation score indicator of unit tests: An experiment," *Information and Software Technology* (52:2), pp. 169-184 (doi: 10.1016/j.infsof.2009.08.007).

Maniraj, S. P., Saini, A., Ahmed, S., and Sarkar, S. D. (2019). "Credit Card Fraud Detection using Machine Learning and Data Science," *IJERT (International Journal of Engineering Research & Technology)* (08:09) (doi: 10.17577/IJERTV8IS090031).

Mekterović, I., Karan, M., Pintar, D., and Brkić, L. (2021). "Credit Card Fraud Detection in Card-Not-Present Transactions: Where to Invest?" *Applied Sciences* (11:15), p. 6766 (doi: 10.3390/app11156766).

Müller, O., Fay, M., and Vom Brocke, J. (2018). "The Effect of Big Data and Analytics on Firm Performance: An Econometric Analysis Considering Industry Characteristics," *Journal of Management Information Systems* (35:2), pp. 488-509 (doi: 10.1080/07421222.2018.1451955).

Nauck, D. (2019). "Test-Driven Machine Learning," available at https://www.infoq.com/presentations/tdd-ml/, accessed on May 9 2022.

Sangwan, R. S., and Laplante, P. A. (2006). "Test-Driven Development in Large Projects," *IT Professional* (8:5), pp. 25-29 (doi: 10.1109/MITP.2006.122).

Shahin, M., Ali Babar, M., and Zhu, L. (2017). "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," *IEEE Access* (5), pp. 3909-3943 (doi: 10.1109/ACCESS.2017.2685629).

Shakir, A., Staegemann, D., Volk, M., Jamous, N., and Turowski, K. (2021). "Towards a Concept for Building a Big Data Architecture with Microservices," in *Proceedings of the 24th International Conference on Business Information Systems,* Hannover, Germany/virtual. 14.06.2021 - 17.06.2021, pp. 83-94 (doi: 10.52825/bis.v1i.67).

Shrivastava, D. P., and Jain, R. C. (2011). "Unit test case design metrics in test driven development," in *Proceedings of the 2011 International Conference on Communications, Computing and Control Applications (CCCA),* Hammamet, Tunisia. 03.03.2011 - 05.03.2011, IEEE, pp. 1-6 (doi: 10.1109/CCCA.2011.6031205).

Shull, F., Melnik, G., Turhan, B., Layman, L., Diep, M., and Erdogmus, H. (2010). "What Do We Know about

Test-Driven Development?" *IEEE Software* (27:6), pp. 16-19 (doi: 10.1109/MS.2010.152).

Slaats, T., Debois, S., and Hildebrandt, T. (2018). "Open to Change: A Theory for Iterative Test-Driven Modelling," in *Business Process Management*, M. Weske, M. Montali, I. Weber and J. Vom Brocke (eds.), Cham: Springer International Publishing, pp. 31-47 (doi: 10.1007/978-3-319-98648-7_3).

Staegemann, D., Volk, M., Byahatti, P., Italiya, N., Shantharam, S., Chandrashekar, A., and Turowski, K. (2022). "Implementing Test Driven Development in the Big Data Domain: A Movie Recommendation System as an Exemplary Case," in *Proceedings of the 7th International Conference on Internet of Things, Big Data and Security,* Online Streaming/Prague. 22.04.2022 - 24.04.2022, pp. 239-248 (doi: 10.5220/0011085600003194).

Staegemann, D., Volk, M., Jamous, N., and Turowski, K. (2020). "Exploring the Applicability of Test Driven Development in the Big Data Domain," in *Proceedings of the ACIS 2020,* Wellington, New Zealand. 01.12.2020 - 04.12.2020.

Staegemann, D., Volk, M., Lautenschläger, E., Pohl, M., Abdallah, M., and Turowski, K. (2021). "Applying Test Driven Development in the Big Data Domain – Lessons From the Literature," in *Proceedings of the 2021 International Conference on Information Technology (ICIT),* Amman, Jordan. 14.07.2021 - 15.07.2021, IEEE, pp. 511-516 (doi: 10.1109/ICIT52682.2021.9491728).

The Apache Software Foundation. (2021). "Test Your Pipeline," available at https://beam.apache.org/documentation/pipelines/test-your-pipeline, accessed on May 10 2022.

Volk, M., Staegemann, D., Trifonova, I., Bosse, S., and Turowski, K. (2020). "Identifying Similarities of Big Data Projects–A Use Case Driven Approach," *IEEE Access* (8), pp. 186599-186619 (doi: 10.1109/ACCESS.2020.3028127).

Wedge, R., Kanter, J. M., Veeramachaneni, K., Rubio, S. M., and Perez, S. I. (2019). "Solving the False Positives Problem in Fraud Prediction Using Automated Feature Engineering," in *Machine Learning and Knowledge Discovery in Databases*, U. Brefeld, E. Curry, E. Daly, B. MacNamee, A. Marascu, F. Pinelli, M. Berlingerio and N. Hurley (eds.), Cham: Springer International Publishing, pp. 372-388 (doi: 10.1007/978-3-030-10997-4_23).

Williams, L., Maximilien, E. M., and Vouk, M. (2003). "Test-driven development as a defect-reduction practice," in *Proceedings of the 14th ISSRE,* Denver, Colorado, USA. 17.11.2003 - 20.11.2003, IEEE, pp. 34-45 (doi: 10.1109/ISSRE.2003.1251029).