OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

INF FAKULTÄT FÜR INFORMATIK

Exploring the Feasibility of Test-Driven Development in the Big Data Domain using a Microservice based Test-Driven Development Concept

Master-Thesis

Hetti Arachchige Maneendra Madushani Perera
Magdeburg, 3. Mai 2022

Supervisors: Daniel Gunnar Staegemann, Matthias Volk
Professor: Prof. Dr. Klaus Turowski

Otto-von-Guericke-Universität Magdeburg
Magdeburg Research and Competence Cluster
Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme
Very Large Business Applications Lab

VLBA Lab

Abstract

Big data analytics and big data applications have become hot topics in the recent past and using them in organizations have accelerated due to their benefits. These applications have spanned different domains and facilitated better decision-making due to better visibility and data usage. However, even though big data provides significant benefits, it has inherited its risks and challenges in data and models that need to be overcome for better accuracy and reliability. Therefore, testing should be performed as much as possible to overcome the negative impacts. 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 practiced in the big data application domain. Nevertheless, a microservice-based test-driven development concept is proposed in the literature, and the feasibility of applying it in actual projects is explored here. For that fraud, the detection domain has been selected as it has deep concerns for better accuracy and reliability by increasing the accurate predictions while minimizing false alarms that lead to many short-term and long-term negative impacts. Finally, the 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 on the number of tests, functionalities covered by the tests, and code quality. The results revealed that each testing level unit, sub-component, component, and the system is covered, and sufficient test coverage is present for each project. Furthermore, the automatic code analysis reports revealed that TDD had produced high reliable, maintainable, and secure code at the first attempt that is ready for production. Furthermore, best practices that will be beneficial for future projects are listed by the practical experience gained.

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 tools can make it more manageable.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CSV	Comma-Separated Values
GCP	The Google Cloud Platform
OOP	Object-Oriented Programming
PCA	Principal Component Analysis
REST	Representational State Transfer
RMSE	Root Mean Square Error
SDK	Software Development Kit
SOA	Service Oriented Architecture
TDD	Test-Driven Development
TLD	Test Last Development
SMS	Short Message Service
XML	Extensible Markup Language
XP	Extreme Programming
WAR	Web Application Resource

1 Introduction

With technological advancement, big data analytics has become a popular topic, and they have become an essential part of the day to day lives [1][2][3]. Today, more companies incorporate big data applications into their business operations due to increased revenue, improved intelligence in using technology, and significant innovation [3]. Both, customers and businesses benefit from big data applications because they improve customer services by delivering exact customer needs faster. In addition, it helps to make strategic decisions more often and faster for business due to having well-analyzed data about the organization with a clear picture.

In simple terms, big data can be described as a massive amount of structured, unstructured, or semi-structured data generated in a short period that conventional data management techniques are no longer capable of tackling [4]. People have shifted towards big data applications since it is hard to manage them using traditional or manual systems [3][4]. Today big data applications are not limited to a single domain; instead, they can be seen in many application domains like agriculture, education, governance, manufacturing, sports, and transformation [1]. Also, the use of big data in the financial fraud detention domain [5], tax fraud detection [6], NLP based applications [7] has increased. Big data analytics allows for better decision-making, which increases the company's competitive power while opening up the paths for company growth [3].

Even though big data applications provide more significant benefits to organizations, they have challenges in data provenance, data quality, data bias, model bias, model comprehension, and ethics inherited from their nature [2]. Therefore, it is essential to overcome or reduce the challenges for accurate and reliable decision-making. For that, rigorous testing is needed, which is a challenging task in big data applications due to its complexity and not being considered in traditional software engineering [8]. Therefore, to improve the quality, reliability, and decision-making power, adequate testing must be guaranteed, and for that test-driven development approach is proposed in the literature [1][2][8], which has not been practiced widely yet [1][8].

This thesis explores the feasibility of applying test-driven development for the big data domain using the microservice-based test-driven development concept proposed by Staegemann et al. [8] in "*Exploring the Applicability of Test-Driven Development in the Big Data Domain*." Here the proposed approach will be practiced through a prototype implementation. Best practices, patterns, anti-patterns, potential challenges, limitations, tools that can be used, etc., will be identified during the process. Here the fraud detection domain will be used to develop the big data application as the proof of concept. Fraud detection is one of the application areas where the software's quality and accuracy are vital deciding factors. Therefore, testing is one of the most critical aspects of such systems that can be improved via TDD because the quality improvement is more promising than the generally used testing strategy, Test Last Development (TLD) [9].

1.1 Aim and Goals

The ultimate goal of this project is to implement a fraud detection big data application in the cloud using a microservice-based test-driven development concept proposed by Staegemann et al. [8]. Cloud computing is one of today's hot topics, which provides all the computer services servers, databases, storage, and networks provided over the internet. Thereby following aspects will be identified and investigated.

- Identify the feasibility of implementing a fraud detection machine learning application in a cloud platform using TDD
- Identify the best practices, patterns, anti-patterns in TDD for implementing the big data applications
- Identify the tools that can be used in implementing tests
- Identify the potential challenges, limitations, or risks in the process
- As the final step, propose mitigations and future improvements from the lessons learned

While achieving the goals mentioned above, this thesis will answer the research question,

What are the best practices in applying test-driven development in implementing a big data application?

1.2 Keywords

- Fraud
  - Fraud is unauthorized access to someone's account or using someone's account or card, claiming as the owner of the account or card for a personal or financial gain [10]. Detecting these scams usually occurs in the financial sector, and it uses big data analytics to detect fraudulent patterns and anomalies.
- Big Data
  - Big data is large and complex data that is difficult to process via traditional data processing techniques due to its volume, variety, and velocity. Volume represents the size of the data, variety talks about the different formats of data, and finally, the velocity denotes the speed of data ingestion to the systems and its processing speed [4].
- Microservices
  - Microservices is an architecture style in software design that diverges applications into multiple services. This design aims to increase the maintainability and testability of the application. Furthermore, these split services can be deployed separately and furnish a loose coupling [11].
- Cloud Computing
  - Delivering all the computing services via the internet is called cloud computing, and it has been a hot topic in the recent past. These services include servers, databases, storage, and networks, accessed on demand [12].
- Google Cloud Platform (GCP)
  - Google Cloud Platform (GCP) is a computing service offered by Google, and it provides over 150 products, including cutting-edge artificial and machine learning platforms [13].

1.3 Background

1.3.1 Fraud Detection

Fraud activities are a big threat to financial institutions because they lose vast amounts of money each year due to fraud activities like unauthorized access and electronic card fraud [14].

Also, financial fraud detection can be of any category: online auction fraud detection, computer intrusion detection, credit card fraud detection, etc. [15].

Big data analytics helps discover hidden patterns and identify irregular activities in fraud detection. It adds tremendous value to the organization as it detects the fraud symptoms early and reduces operational business risks associated with the company [3]. The main goal of fraud detection is to predict fraud correctly while minimizing false alarms [15].

Identifying the outliers through data analytics techniques has helped banks or credit card companies to identify scenarios like fraud purchases and credit card thefts [14]. Today, credit card fraud detection using machine learning techniques has become very popular [10], and for fraud detection, different methods and techniques are used. For example, it includes supervised algorithms which use labelled transactions to train and test the model and then predict unseen transactions using a risk score etc. Another method is hybrid algorithms that use business rules with supervised algorithms like neural networks, Bayesian networks, decision trees, etc to predict fraud transactions. Furthermore, unsupervised algorithms like link analysis and graph mining also can predict fraud [15].

The quality in terms of accuracy, performance, reliability, and explainability of fraud detection applications are critical as incorrect fraud decisions can make a substantial financial loss if it fails to identify a fraudulent transaction. It is said that costs related to false-positive credit card trades are thirteen times more than valid transactions [16]. Only 0.25% are fraudulent transactions identified as fraud [17]. Also, it can damage the brand image when it terminates a valid transaction, and it adversely impacts customers' trust. Furthermore, this can result in losing potential customers as customers who had a bad experience can express their bad experience to the potential customers. So incorrect fraud tagging causes short-term negative impacts and long-term losses for the companies. Therefore, companies are putting much effort into reducing the false-positive rates, and improving efficiency and effectiveness is one of the critical steps taken in the industry. Such systems should be highly accurate and reliable, and for that, errors should be minimized, and expected outcomes should be carefully validated for every scenario by testing. In TDD, every production code is associated with a test, and therefore the testing coverage is higher than the TLD. Furthermore, it increases the accuracy and reliability of the system, which is highly anticipated in fraud detection big data applications. Therefore, TDD is well suited for fraud detection as it assures testing of every production code, thereby increasing precision and trustworthiness.

1.3.2 Big Data

Generally, big data can be described as a large amount of data, usually in different formats [22]. It is hard to handle these data with conventional data handling approaches [1][2][4]. Furthermore, big data can be explained using 5V models: volume, velocity, variety, value, and veracity. Here volumes refer to the massive amount of data that needs to be handled. Velocity refers to the streaming of data, which is the speed of data collection and the processing speed. Variety is the different types of format or data: text, images, video files, database files, etc. Value is the actual benefit of such data in terms of cost, such as storage cost, transformation cost, costs related to the acquisition of data, etc. Veracity refers to data precision and how good the data is at making correct predictions [4]. Today, organizations are trying to incorporate big data applications into their business due to its benefits, but it is not straightforward. Even though using big data applications in the organization is a challenging task, it benefits decision-making. To mitigate the challenges and produce accurate, reliable results, extensive testing of the big data applications is required, which can be fulfilled via test-driven development [1].

1.3.3 Microservices

Microservices is an architectural style that has gained much attention recently. The core objective of this architecture is to allow self-contained modules or services that can be deployed separately with their process, own storage, etc. [15]. Also, since they are different independent components, different programming languages can be used for the implementation, and generally, they will serve a single service in terms of functionality. Microservices has built upon share nothing philosophy and helps for loosely coupled, small autonomous services [19].

In the design and development of microservices, multiple repositories should be maintained as the application is divided into smaller self-contained components. Preserving these repositories is one of the main challenges in the microservices design pattern, and the other major hindrance factor is limited skills and knowledge in the area. Furthermore, identifying issues in design is difficult due to the distributed nature of the services, and debugging is also a bit difficult in the scenarios when multiple services are involved. By incorporating this architectural pattern, scalability, agility, and extendibility are aimed, and it is crucial to optimize security, performance, and response time for efficient usage [19].

1.3.4 Test-Driven Development (TDD)

The strategy of writing the tests before writing the actual functional code is considered test-driven development [1][8][18][20]. It is also known as test-first design, test-first programming, and test-driven design [18]. Even though the history of using TDD goes back several decades, the agile community still has a great interest in the idea. Kent Beck rediscovered TDD in 2003 as a software development process, but it first appeared in 1998 as an extreme programming methodology [9][21]. However, some argue that TDD was used in NASA’s Mercury project in the early 1950[18][21]. Also, TDD is considered the predominant practice among the practices introduced in XP [23].

In TDD, instead of writing the functionality code, tests are written first, and then the functional code is implemented incrementally and iteratively. Also, tests are designed as small as possible [1][8][18][20]. In TDD, higher quality is expected while reducing the risks and bias in the application. However, TDD requires more time in the pre-implementation phases which slows down the development process [1][24]. TDD has been used to develop conventional software, embedded software, cloud applications, web applications [25] etc, and it is said that TDD improves the quality of software [26][24] and increases productivity [23][25][26]. Furthermore, TDD improves reusability, flexibility, effectiveness, and risk reduction and decreases software complexity and rework cost [20].

In the literature, applying TDD has been considered a research topic by many researchers [18]. Also, there is interesting literature on the benefits and effects of TDD, and they have researched different aspects as effectiveness measures rather than measuring the effectiveness on a single criterion [23][24][27][28]. One of the papers researched the effects of TDD using some Java projects on *Github*. They have analyzed the impact of the rate of commits, commits related to bug fixes, and no reported bugs when using TDD vs. TLD. However, unfortunately, they could not find any statistical significance when using TDD [27]. Another examination of the effectiveness of TDD, examining the last ten years of research papers, was carried out in terms of time, cost, productivity, required effort, no issues found, etc. [24]. They have concluded that TDD has a positive impact on software quality but a negative impact on productivity as it needs more effort and training. In the paper “*On the Effectiveness of Unit Tests in Test-driven Development*,” Tosun et al. [28] found that TDD improves mutation score and branch coverage which are beneficial to finding and reducing bugs [28].

Many instances can be found in TDD with agile software development in the literature, but it is rare to find using TDD in big data applications.[1]. In TDD, four different units of testing need to be assured. System-level is the biggest unit, everything from start to end, method-level, which is the smallest part of the application; subcomponent level, which is a collection of methods; and component level, which is a collection of sub-components.

So, in this thesis, all four types of testing will be carried out following the proposed microservice-based test-driven development concept described in the Microservices section.

1.3.5 Test-Driven Development in Big Data

In the paper "*Exploring the Applicability of Test Driven Development in the Big Data Domain,*" Staegemann et al. [8] have designed a TDD testing approach that follows a microservice-based architecture that facilitates all four testing variations - system-level component-level, subcomponent-level, and method-level testing. Designing the development concept using microservice-based architecture allows high test coverage, flexibility, and dedicated development teams for different modules or components. In the paper "*Benchmark Requirements for Microservices Architecture Research,*" Aderaldo C et al. state that complexity in microservice-based architecture has shifted to the integration level where it was on the component level previously [29]. In the proposed TDD concept, Staegemann et al. [8] suggest testing the messaging flow of the entire system to ensure all the modules or components of the system are compatible. This idea reconfirms the statement by Aderaldo C et al. [29] that integration is the most complex task in microservice-based architecture, and it should be assured as the initial step. Table 1 summarizes four different testing types and the process of testing proposed in the paper [8].

Testing Level	Testing Process
Method	Unit testing. Assure the algorithm correctness.
Subcomponent	Through verification of all methods inside the subcomponent. Assure the performance requirements like processing speed and capacity with benchmarking.
Component	Through microservices. Assure compliance with the requirement also performance requirements with benchmarking.
System	Through testing all components.

Table 1 : Testing Types

1.4 Methodology

As the first step, an extensive literature review on the state of the art of test-driven development is carried out. It will include the TDD process, best practices, and general evaluation criteria. Then the proposed system is designed by identifying the components and modules. As the third step, a test list will be created before starting the project covering both aspects of cloud-related tests, big data application tests, and covering the four test types.

Then the implementation will be initiated using *Java* or *Python* or implementing some modules in *Java* and some in *Python* based on the outcomes of the literature review and exploring the tools supported in the two languages. The Google Cloud Platform (GCP) cloud platform will be used to develop machine learning models, and then the final application will be deployed in the cloud. One of the public fraud detection datasets from the Kaggle website will be used to train and test the machine learning model. Finally, test-driven development evaluation will be carried out by code coverage, revisiting the testing requirements and evaluation methods discovered in the literature review. The best-suited evaluation methods will be selected after the comprehensive literature review. After that initially created test list will be revisited to identify the tests that could cover and could not cover using TDD. Finally, future improvements will be presented after analysing the challenges and risks faced during development.

1.5 Outline

To fulfill the previously mentioned objectives thesis is structured as below.

Section 2 reviews the state of the art of TDD. Extensive research was conducted to discover current knowledge, and fifty-two papers were selected after filtering. Then the research is carried out on TDD, TDD process, limitations and challenges in TDD, best practices in TDD, tools used for writing tests during TDD application, and effects of TDD. Finally, in the effects of TDD, descriptive research is carried out to determine the effect of TDD on the three main factors: internal quality, external quality, and productivity.

Section 3 discusses the design and implementation of the big data application for online fraud detection. The main objective of this thesis is to explore the feasibility of applying TDD in developing a big data application. For that fraud detection domain is selected as described above. For fraud detection, several methodologies can be used, and, in this thesis, both algorithmic and rule-based methodologies are used to identify the transaction as fraudulent or legitimate. So, in this chapter, the design, design decisions, and the reasons are explained in detail. Implementing the online fraud detection application is divided into three main projects design and implementation of the classification machine learning model, rule engine, and online fraud detection streaming application. Therefore, all the implementation details and the implementation decision, tools, and frameworks are described with figures. Furthermore, a detailed discussion is done on how TDD is used in each project, what kind of tests are written for each service, and the tools and frameworks used to write tests. Sample test code snippets are attached for each service.

Section 4 describes how the evaluation is conducted in the project. The best way to evaluate the TDD is to create two similar projects, one using TDD and the traditional test last approach, and then comparing TDD effects like internal quality, external quality, and productivity using suitable metrics. However, since it is not feasible in this thesis, the evaluation is carried out using evaluating the number of tests, their functionality, and testing types. Finally, the code quality aspects like reliability, maintainability, and security are measured using *SonarQube* and *Code Coverage* tools. Every aspect mentioned above is discussed in detail for each project, machine learning project, rule engine project and online fraud detection streaming application.

Section 5 concludes the thesis by presenting the best practices derived from the application of TDD in online fraud detection platform. Identifying the TDD best practices in big data applications is the main research question addressed in the thesis. Therefore, the practices derived from the literature review and the best practices identified in actual application are discussed here. Furthermore, this chapter presents the future work that needs to be carried out in this domain based on the experience gained from this project.

2 State of the Art

Before applying the TDD in the big data domain, it is essential to discover the current knowledge on TDD. Therefore, this report investigates the state of the art of the TDD, the TDD process, TDD limitations and challenges, and TDD best practices and tools. Scopus, a famous and reliable abstract and citation database, was used to find the literature, and the below query was used to retrieve the papers.

(TITLE (“test driven development” OR “test-driven development” OR “test-first design” OR “test first design” OR “test-first programming” OR “test first programming” OR “test-driven design” OR “test driven design”)) AND (LIMIT-TO (DOCTYPE , “cp”) OR LIMIT-TO (DOCTYPE , “ar”)) AND (LIMIT-TO (SUBJAREA , “COMP”)) AND (LIMIT-TO (LANGUAGE , “English”))

The query searches for the articles in the English language which have the words “test driven development” or “test-driven development,” or “test-first design,” or “test first design,” or “test-first programming,” or “test first programming,” or “test-driven design” or “test driven design” in the article title, abstract or keywords. Two filters are used to retrieve articles belonging to computer science and document-type conference papers and articles. The subject area was used to narrow the search and pick up the relevant articles mainly related to computer science, and document type was used as they are peer-reviewed. Therefore, the trustworthiness of the articles is higher than the others.

The search was carried out in October 2021, and according to the given criteria, 256 articles appeared on the list. However, not all articles are matched to the researched topic. Therefore, another filtering was needed to limit the articles further.

As the first step, the article's title was read, and some of the articles were removed that were not applicable because the main focus was deviating from the main subject. For example, papers like *“Towards test-driven development for FPGA-based modules across abstraction levels,” “Why research on test-driven development is inconclusive?*” filtered out as the main focus was on acceptance testing, etc. There were 126 articles after the filtering, and still, there can be irrelevant papers. In the second step, the abstract of the articles was read to identify the most suitable items for the purpose. While trying to read the abstract, it was noticeable that some articles were not openly accessible and therefore had to be removed. However, such an article count did not exceed 5. Finally, articles whose primary research area is not TDD and the research questions not aligned with the topics covered in this literature review are dropped. Finally, 56 articles were left to be reviewed in the literature. After reading the entire article, four papers were removed, deviating from primary researched subjects.

Table 2 summarizes the inclusion and exclusion criteria followed when selecting articles.

Inclusion Criteria	Exclusion Criteria
Is a conference paper or journal article	Not written in English
Is subject area "Computer Science"	After reading the content seems not relevant to research questions
Main research area is test driven development	
Is openly accessible	

Table 2 : Paper Inclusion and Exclusion Criteria

Figure 1 shows the process carried out for article selection.

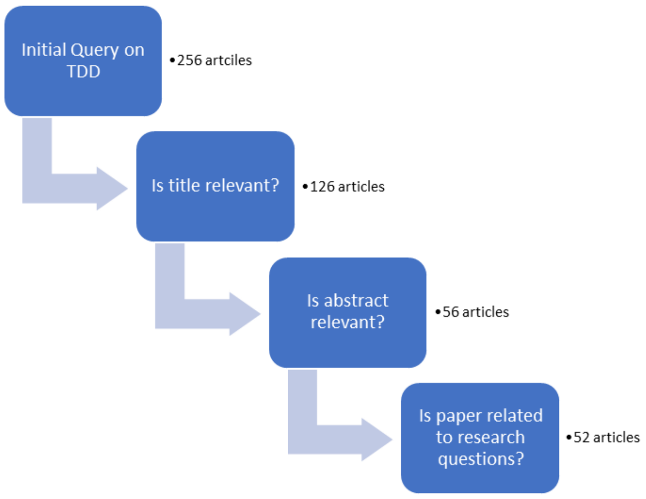


Figure 1: Paper Selection Process

As described above, finally, 52 papers are used in the entire literature review, and Table 3 depicts a summary of them.

Reference	Year	Title
[60]	2002	Experiment about test-first programming.
[40]	2003	Assessing test-driven development at IBM.
[52]	2003	Discipline and practices of TDD: (test driven development).
[18]	2005	Test-Driven Development:Concepts, Taxonomy, and Future Direction
[23]	2007	An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development
[55]	2008	Effective and Pragmatic Test Driven Development.
[65]	2009	Empirical investigation towards the effectiveness of Test First

		programming.
[61]	2009	Evaluating Test-Driven Development in an Industry-Sponsored Capstone Project.
[48]	2009	Long-Term Effects of Test-Driven Development A Case Study
[7]	2009	Towards agile and test-driven development in NLP
[34]	2009	Towards traceable test-driven development.
[42]	2010	Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers.
[37]	2010	*Selecting UML models for test-driven development along the automation systems engineering process*.
[64]	2010	Test-Driven Development - Still a Promising Approach?
[59]	2010	The impact of Test-First programming on branch coverage and mutation score indicator of unit tests: An experiment.
[39]	2010	What Do We Know about Test-Driven Development?
[41]	2011	Causal Factors, Benefits and Challenges of Test-Driven Development: Practitioner Perceptions.
[50]	2011	Critical Issues on Test-Driven Development.
[30]	2011	The effectiveness of test-driven development: an industrial case study.
[57]	2012	Adding Criteria-Based Tests to Test Driven Development.
[90]	2012	Binding Requirements and Component Architecture by Using Model-Based Test-Driven Development
[49]	2012	Quality of Testing in Test Driven Development.
[21]	2012	Test driven development: The state of the practice.
[63]	2012	*Unit Test Case Design Metrics in Test Driven Development*.
[51]	2013	TDDHQ: Achieving Higher Quality Testing in Test Driven Development.
[38]	2014	A successful application of a Test-Driven Development strategy in the industrial environment.
[14]	2014	Applying Acceptance Test Driven Development to a Problem Based Learning Academic Real-Time System.

[32]	2014	Designing a Framework with Test-Driven Development: A Journey.
[43]	2014	Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies.
[36]	2014	Obtaining better software product by using test first programming technique.
[47]	2014	On the Effects of Programming and Testing Skills on External Quality and Productivity in a Test-Driven Development Context.
[53]	2014	Test driven design.
[44]	2014	Understanding the Dynamics of Test-Driven Development.
[91]	2015	A Generic Testing Framework for Test Driven Development of Robotic Systems.
[62]	2015	Does test-driven development improve class design? A qualitative study on developers’ perceptions.
[31]	2015	Test-driven development of web and enterprise agents.
[58]	2015	Towards an operationalization of test-driven development skills: An industrial empirical study.
[45]	2016	The effects of test driven development on internal quality, external quality and productivity: A systematic review.
[46]	2017	A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?
[24]	2017	Evaluating the effectiveness of test driven development: Advantages and pitfalls.
[56]	2017	Teaching Test-First Programming: Assessment and Solutions.
[33]	2017	Test-Driven Development in scientific software: a survey.
[92]	2017	Testing Test-Driven Development
[28]	2018	On the Effectiveness of Unit Tests in Test-Driven Development
[89]	2018	Test First, Code Later: Educating for Test Driven Development.
[9]	2018	What Do We (Really) Know about Test-Driven Development?
[8]	2020	Exploring the Applicability of Test Driven Development in the Big Data Domain.

[35]	2020	Towards Interactive, Test-driven Development of Model Transformations.
[1]	2021	Applying Test Driven Development in the Big Data Domain – Lessons From the Literature.
[25]	2021	Test Driven Development in Action: Case Study of a Cross-Platform Web Application.

Table 3 : Selected Papers

The rest of the review is categorized into the following sections TDD, TDD process, TDD effects, challenges and limitations in TDD, TDD best practices, tools used in TDD.

2.1 Test-Driven Development(TDD)

The strategy of writing the tests before writing the actual functional code is considered test-driven development [1][8][18][20], and it has been used in agile software development for more than a decade [21][30]. TDD is considered a practice rather than a methodology that can be practiced with extreme programming or scrum [21]. It is also known as test-first design, test-first programming, and test-driven design [18].

There are several views on TDD like TDD is a process that enables thinking about the design before the functionality, TDD is merely a programming technique that enables clean code development, and TDD is not about testing the software but rather a process that enables to see through ambiguous requirements by collaborative process among customers and developers [30].

In TDD, tests are written first before writing the functionality code, and then the functional code is implemented incrementally and iteratively. Also, tests are designed as small as possible [1][8][18][20]. In TDD, higher quality is expected while reducing the risks and bias in the application. However, TDD requires more time in the pre-implementation phases which slows down the development process [1][24]. TDD has been used to develop conventional software, embedded software, cloud applications, web applications [25], interceptor and messaging-based middleware architectures [31], open-source frameworks [29], scientific software [33], backend and frontend based applications [25], model-driven engineering transformation applications [35], etc. It is said that TDD improves the quality of software [24][26] and the quality of the design process [36]. TDD also supports identifying the missing or vague requirements early by creating tests in the initial development phase [37][38] and helps for a clean design [39]. Furthermore, TDD highly improves reusability, flexibility, effectiveness, and risk reduction, decreases software complexity and rework cost [20], and improves code maintainability [38].

2.2 TDD Process

TDD is the reversed approach of the traditional software development process, where tests are written at the last phase after the production code is implemented [21][40]. Test-driven design, test-first coding, and test-first Programming are other names used in literature to describe TDD [30], and automated tests are written before implementing the actual production code [21]. Test-driven development is an incremental and iterative approach [41], and it does not require a detailed design of the system at the beginning of the project; instead, the design is developed while building the system by creating tests [23][40]. TDD is not merely writing the automated tests but helps for more fine-grained requirements as validated against tests [42]. This approach defines simple and small tests, but it does not validate functionality correctness and when to stop developing the function [21].

In the book *Test-Driven Development by Example*, Kent Beck describes the TDD approach using general guidelines and does not explain an exact process for the implementation [21]. According to him, the TDD mantra is Red/Green/Refactor, and the TDD cycle is to write a test, run it, and make it right. Table 4 describes the most common explanation of the TDD mantra using a traffic light [21][33][43].

Phase	Description
Red	Red light, initially there will be no production code, and the test is failing.
Green	Green light, after implementing the minimum production code to pass the test, the test is passing.
Refactor	Here, improving the production code happens without changing the functionality and usually removing duplicates and performance improvements are done here [48]. Refactoring is a continuous process until all the tests are turned green [25].

Table 4 : TDD Mantra

The TDD process is perceived in different ways by people. Mitrović et al. [31] explained TDD using three laws: 1) if there are no failing tests, new production code should not be written, 2) write a new test, 3) write new production code that is just enough to pass the test. According to them, any production code should not be written unless there is a failing test, and in a test failing scenario, a minimum production code should be written to make the failing test pass. Aniche et al. [42] explain the TDD using five steps, write a test, watch it fail, add code, check all tests, and refactor [42]. They are the exact five steps Beck explained in his second book on XP programming [4]. In the paper "*Understanding the Dynamics of Test-Driven Development*," Fucci et al. [44] explain the TDD in the same five steps but only changing the order of the last two steps. According to them, the flow is writing a test, watching it fail, adding the code, refactoring, and checking all the tests [44]. Bissi et al. [45] similarly explain the TDD process but add one more step. According to them, the flow is, writing a test, running all tests to see the failure of the previously created test case, adding production code to pass the above-created test case, running all the tests to verify the unit test passes and no regression issues have introduced, refactoring or improving the code or test, running all the tests to verify refactoring has not introduced any errors in tests.

The phases of TDD are described in similar steps by the above researchers, and Figure 2 shows the most precise process when summarizing all the literature gathered here. It is the exact six steps described by Bissi et al. [45]. After adding production code or refactoring it, it is crucial to check that all the available tests are passing and that changes have not introduced any regression issues. Therefore, it is the most reliable process that fits TDD.

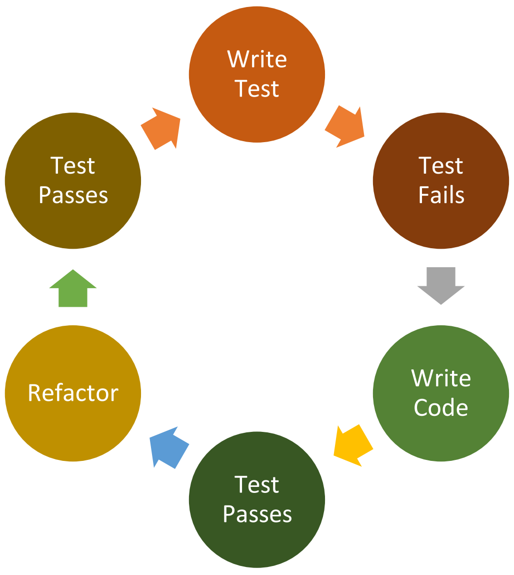


Figure 2 : TDD Process

TDD is a systematic process where it starts with a small feature. Once the feature implementation is finished and when all the tests related to the feature and all the regression tests are passed, the current cycle finishes, and again next round starts with another implemental feature [46]. As a result, TDD produces more test cases when compared with the TLD [47]. Fucci et al. [46] explain the above cyclic process as the cycle starts with writing the unit test, adding production code till test cases pass, refactoring the code to avoid repeating code, enhancing the design, and then starting the next cycle.

2.3 Limitations and Challenges in TDD

Multiple pieces of literature reveal limitations and challenges in TDD. Some of them are discovered during practical TDD applications [41][48]. Others have been derived from summarizing the past literature [21]. According to them, the most limiting factors of TDD are 1) the lack of experience, 2) knowledge, and competencies in applying the TDD in practice which hinders them from creating efficient and effective automated tests [9][33][41][49] and 3) switching to TDD mindset from traditional TLD method which requires high self-discipline [21][48][50]. Furthermore, many researchers have figured out that the absence of a detailed design of the system is challenging when applying the TDD [9][21][50]. Marchenko et al. [48] presented four real challenges they faced during their three-year-long TDD project in the Nokia Siemens Networks team, collected through a rigorous interview process conducted for the team members [48]. For them, strict regulation was needed to maintain the TDD practice and apply it in GUI development; configuration-related development such as creating XML files was challenging due to technical complexity. Furthermore, it was difficult for them to apply the TDD in their legacy code due to a lack of understanding of the requirements. In the paper “*Test-Driven Development in Scientific Software*”: a survey, Nanthaamornphong et al. [33] present the challenges discovered when applying TDD in developing scientific software by gathering details from the scientific community [33]. They have developed a survey with different questions and distributed it among 300 developers who had experience in TDD. From the survey; they found four main challenges, which are 1) more time to develop tests, 2) difficulty in writing test cases for complex functions, 3) writing tests for the functions where still the results are unknown since there are still in the research phase and 4) more time to adapt for TDD practice due to lack of skills and experience. Also, sometimes they had to set up a new environment for using TDD in their projects, and the lack of tools capable of creating tests was another hindrance factor for them. Buchan. et al. [41] discovered another interesting point during their three-year-long TDD project that the top-level decision-makers of the company were unhappy with spending a considerable time creating test cases. Their misconception was that more time is spent in TDD rather than working on the actual production code.

Even though TDD provides excellent benefits, usage in the industry is challenging [49]. In some industrial applications, the main challenge is not having previous exposure to TDD [38][49]. Causevic et al. [49] presented seven challenges for adapting TDD in the industry by conducting a systematic literature review. According to them, the barriers are increased development time, therefore, decreased productivity. Lack of theoretical and previous experience in using TDD in real projects. Lack of detailed design at the beginning of the project. Lack of skills in developing efficient and effective automated tests. Not utilizing the TDD practices and guidelines, and finally, ambiguities in applying TDD in the legacy code. In the same paper, they stressed an interesting fact that TDD has forgotten to provide guidelines for using it in legacy code. This limitation has created a massive barrier for the organizations with legacy codebases and hindered adopting the process. Kollanus et al. [50] also listed some of the literature's potential challenges, like lack of a detailed design that causes heavy refactoring and maintenance, insufficient skill levels to apply TDD in complex tasks, and not having the correct knowledge mindset to shift to a new paradigm. Furthermore, they revealed that the shift towards this new process has a high learning curve and management support is essential for continuing the process. Apart from the issues raised, they also pointed out that the applicability of TDD in complex scenarios and circumstances where limited automation tools are present causes more time since a high volume of test code causes high maintenance efforts and troubles. As described above Table 5 shows the major challenges discovered when applying TDD.

Limitation Type	Description
People	Lack of knowledge, experience, and competencies in applying TDD [9][33][38][41][49][51] Difficulty to shift to the TDD mindset [21][48][50][51] Not having a proper understanding of the TDD practice by senior-level management [41][51]
Software	Technical complexity in applying TDD in some scenarios like GUI development, configuration development, complex functions, etc. [48][51] Lack of software tools to create tests [51]
Productivity	Increased development time and effort [49]
Process	Lack of detailed upfront design [9][21][50][51] Not having proper guidelines for using TDD for legacy code [49] High test code volume [51]

Table 5 : TDD Challenges Summary

2.4 Best Practices in TDD

TDD does not provide strict guidelines but contains general guidelines [21]. Therefore, no hardcoded rules, and best practices are derived from the experiences and mistakes in using it in real projects. In the paper "*Assessing Test-Driven Development at IBM,*" Maximilien et al. [40] presented some compelling facts witnessed during the adaptation of TDD in their project. Starting TDD at the beginning of the project is vital to increase productivity; it is better to introduce automated build tests at the later development phase. Also, it is crucial to encourage the development team to continue using TDD practice while providing them the knowledge to uplift their skills to design and create efficient and effective unit tests. Furthermore, adding a new test to each issue found is essential, and the initial unit test plan should be reviewed thoroughly.

One of the members bringing up the paper "*Discipline and Practices of TDD*" [52] mentions some of the best practices according to his personal experience. According to him, tests should be as small and simple as possible, tests should be refactored continuously, and they should validate the stakeholder's actual needs rather than verifying the specified requirements in the system. Also, tests should be grouped based on the fixture, and developers should spend equal time creating tests as developing the actual production code, which emphasizes that tests are equally crucial as code in the TDD process.

Online surveys are also an effective way to capture the opinions of developers in each area. For example, Aniche et al. [42] conducted an online survey, and 218 developers voluntarily contributed. The survey results pointed out the most frequent mistakes that the practitioners made when applying TDD to their projects, and they are as follows. Not being vigilant about the failing tests and starting the next test without considering the previous test fail, forgetting the refactoring of production code, refactoring the production code while working on a test, and refactoring the code when there are test failures are several of them. Furthermore, they found out that using bad test names, not creating simple tests, executing only the failing test without running all other tests, writing complex tests without refactoring them, incorporating undesirable complexity into the code are other repeated mistakes done by developers.

Refactoring is one of the most critical steps in TDD, which improves the code without changing the external behaviours, and it is considered cleaning, which improves the readability of the code [42]. Nanthaamornphong et al. [33] list some of the refactoring best practices that can be followed that were discovered when applying TDD in scientific software literature. Creating small and simple classes and methods and using proper names in classes, variables, easily recognizable methods, using Object Oriented Programming (OOP) concepts like *encapsulation*, *inheritance* in the code, and employing appropriate design patterns are the mentioned best practices during the refactoring phase.

Lessons learned from an actual industrial project are great assets for future endeavours. In the lessons learned section, Buchan et al. [41] presented several points gathered from a 3-year long medium-sized software development project. The project adopted TDD as the extreme programming methodology, and the experience received during the project was collected by interviewing the five key team members who were contributing to the project mostly. According to the interviewees, every team leader must have sufficient skills and experience in TDD to strictly follow TDD in the project, create effective test cases, and mentor subordinate team members who have less experience or difficulties in adopting TDD. Also, it is essential to follow the TDD practices strictly, avoid common mistakes to gain the maximum benefits it is promising, and realize the benefits of TDD by management is vital to gain the support by minimizing the pressure they produce due to increasing time and efforts. Furthermore, it is essential to measure the TDD benefits using metrics and let both team members and management understand the application of the TDD in the project.

Furthermore, as other researchers describe [40][41], it is good to provide necessary training to developers with fewer TDD experiences and uplift their skills for a smoothly running project. Finally, it is necessary to integrate automation tools and techniques into the day-to-day development process for better productivity. It will help motivate the developers by reducing the challenges they face during their development phase. Apart from the above mentions, they also emphasize that using meaningful names in tests and production code is a best practice in TDD, and creating small classes improves the low coupling and high cohesion in the code.

In the introductory phase of the thesis, a search is carried out to grab the initial knowledge of TDD. During that time, a master thesis [54] based on best practices of TDD was impressive because it was directly about the thesis research question. The researcher listed out best practices like adopting TDD for the whole project rather than partially adopting it. Everyone in the project should adopt TDD rather than teams using another programming methodology. He pointed out an essential practice of creating a simple test list before each feature. Kawadkar. [53] emphasized the same best practice of creating a detailed test list during the design phase uncovered in his two-year case study of test-driven design. It will help to brainstorm the feature to be implemented, and it also can be used to check the progress and completion of the specific feature. Implementing the tests can be started with the tests that provide more feedback or from simplicity. Furthermore, he pointed out that automated tests improve reliability and reproducibility. Again, designing small, simple, self-explaining tests is beneficial for the execution time. Automated tests should be executed frequently. Therefore, it is essential to consider the total execution time to improve the maintainability of test cases and avoid some test cases being abandoned.

The success of a project depends on the encouragement given to the people involved in it. An active team discussion about the TDD benefits and implementation to encourage applying TDD is one of the best practices derived from a real-time project [55]. Furthermore, the experience of that project revealed that TDD should be strictly followed, and the automated code coverage tools should be used to measure coverage frequently to find gaps. In the project, the *Cobertura* code coverage tool was used.

Gaining TDD skills is helpful for its better adaptation [40][41], and Missiroli et al. [56] presented an interesting fact that proving TDD education in high school is beneficial. They pointed out that early exposure to TDD avoids mental, behavioural change limitations. However, they also conclude that teaching TDD in schools is considerably high.

When concerning the literature, most of the best practices have derived from the actual experiences of using TDD in projects [40][52][55]. When summarized, it is necessary to fully adopt the TDD at the beginning of the project and provide knowledge or necessary training to uplift TDD skills to create effective test cases [40][41] while encouraging them. Also, it is crucial to have an initial test plan [40] or test list [54] and add more test cases for each issue found [40] to better test cases. Furthermore, small, simple tests [42] should be created to validate actual user requirements, and equal time should be spent on creating tests as they are equally important as production code [52]. Furthermore, it is better to group tests based on the fixture, and production code [42] should be continuously refactored [52]. Using design patterns for code is also a good practice [33][42], and small, simple classes, methods, variables with proper names using OOP concepts [33] should be created. Finally, it is better to use automation tools [54] and measure TDD effects to make it more visible and gain management support [41].

Table 6 shows the best practices derived from the literature.

Best Practice Type	Description
People	Encourage developers for continuous usage of TDD and provide necessary resources to gain the required skill set using adopt TDD [40][41]
	Every Team leader should be a promoter of TDD [41]
	Awareness of TDD benefits among the management [41]
	Team discussions about the TDD benefits and implementation [55]
Software	Use readable test names which describes the tested functionality [41]
	Use meaningful names in functional code [41]
	Keep the production code simple as possible [54]
	Use design patterns for the production code for low coupling and high cohesion [33][41]
	Use OOP concepts when possible [33]
	Use automation tools and techniques [41][54]
	Use code coverage tools [55]
Process	Create a proper test plan at the beginning of the project [40]
	Start TDD at the beginning of the project [40]
	Follow TDD practices strictly [41][55]
	Avoid partial adoption of TDD [54]
	Prepare a test list for each feature [54]
	Write tests to validate end-user requirements [52]
	Group tests according to fixture [52]
	Spend sufficient time in creating tests as tests are equally important as the production code [52]
	Write small and simple tests, mostly with a single assertion [52][54]

	Watch for test features and work on a single test at a time [42]
	Refactor the production code [42]
	Refactor the production code only when all the tests are green [42]
	Run all the tests after fixing a single test failure [42]
	Consider test execution time [54]
	Refactor the test code [52]
	Measure TDD benefits, make them more visible using metrics [41]

Table 6 : TDD Best Practices

When considering the importance of the above points, Figure 3 shows them in three categories. First, test-driven development is a systematic process, and better results come with the best adaptation [45]. Therefore, all the process-related best practices are given high importance. The procedures related to automation tools and the design patterns are given medium priority. Even though it improves the process, not using them may not fail the entire process. Finally, management support is beneficial for the success of any project, but not having enough support will not degrade the function as a whole. Consequently, grouping tests based on fixtures and measuring execution time are team choices therefore categorized as low priority when adopting and continuing TDD in the actual projects.

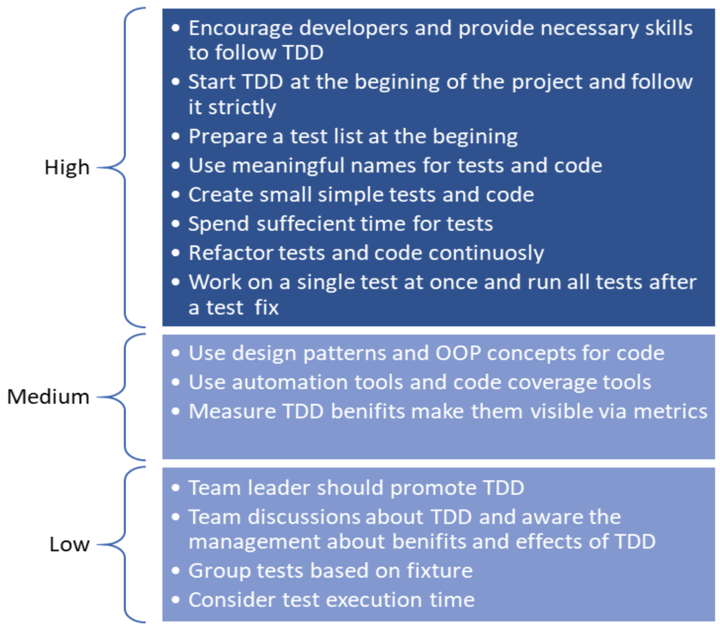

High
• Encourage developers and provide necessary skills to follow TDD
• Start TDD at the begining of the project and follow it strictly
• Prepare a test list at the begining
• Use meaningful names for tests and code
• Create small simple tests and code
• Spend suffecient time for tests
• Refactor tests and code continuosly
• Work on a single test at once and run all tests after a test fix
Medium
• Use design patterns and OOP concepts for code
• Use automation tools and code coverage tools
• Measure TDD benifits make them visible via metrics
Low
• Team leader should promote TDD
• Team discussions about TDD and aware the management about benifits and effects of TDD
• Group tests based on fixture
• Consider test execution time

Figure 3 : TDD Best Practices

2.5 Tools

Automated testing tools play a significant role in TDD as the unavailability of good automated testing tools makes it challenging the proper application of TDD [9][33]. The majority of the developers use automated testing tools when developing tests [33]. The most used testing tool in the literature is *JUnit* [30][33][38][40][41][49][55][57][58][59][60], and the following are the testing tools that have been used in creating the tests.

- Jython[40]
- Cobertura to measure statement and branch coverage [4]
- Cmake, CTESTS, Google Test or GTest, Python nose, Python unit tests, CXX tests [33]
- EMMA - Java code coverage tool, Chidamber and Kemerer Java Metrics (CKJM) - software metric tool for java, Metrics 1.3.6- software metrics tool for Eclipse [61]
- Clover code coverage measurement tool, Judy mutation testing tool [59]

2.6 Effects of using TDD

Day by day, the complexity of the software is increasing, and producing quality software has become a significant challenge [25]. Therefore, it is crucial to test the software to reduce defect rates thoroughly, and TDD believes it can support [25][56]. Furthermore, TDD claims to catch bugs early and reduce fixing costs while improving the customers' trust [38]. Maximilien et al. [40] found out that their defect rate dropped by 50% when they used TDD in IBM [40], and in the survey, 60% of the participants were convinced that TDD reduces the defect rate of the software [42]. The benefits of TDD are not limited to lower defect rates but the improved design of the software and integration effects [40]. In the literature, applying TDD has been considered a research topic by many researchers [18]. Also, there is exciting literature on the benefits and effects of TDD. They have researched different aspects as effectiveness measures rather than measuring the effectiveness on a single criterion [23][24][27][28]. Many researchers have categorized effects into three main categories: internal quality, external quality, and productivity in the literature.

2.6.1 Internal Quality

Internal quality refers to the way the system has been constructed, and code coverage, reusability, code complexity are the measures that are used to measure the internal quality. Class design is also one of the internal quality factors as it creates simple, low coupling, and high cohesion classes that are easy to maintain, easy to extendable, and easy to understand [62][63]. Aniche et al. [62] conducted a qualitative survey on developers' perception of whether the class design improved with TDD. Twenty-five participants with six companies in Brazil participated in this research, and they were given an exercise that was carried out on their premises. After the exercise, the researchers gave them a survey regarding designing the code and the challenges faced while applying TDD. Then an interview procedure was conducted for some of the selected developers and allowed them to talk openly. According to the results, TDD did not directly help for a better class design but creating tests first indirectly, and constant feedback on tests helped them design testable classes. Furthermore, another industrial experiment [55] revealed that TDD improves code quality with more cohesive, less coupled classes. Also, during the implementation, it was noticeable that TDD was encouraged to implement only the features needed by the end-users and to have more tests gave developers great confidence about the functional correctness of the system. Even though the above research claims an increased code quality using TDD, some argue that code quality depends on the actual testing [23].

The test case design is also another internal quality factor which is measured using the metrics like Weighted Methods per Class (WMC) which measure class's complexity, Depth of Inheritance Tree (DIT), which measure the complexity and reusability of the class, Number Of Children (NOC) which measure class's reusability and testability effort of the class, Response For a Class (RFC) which measure testing and debugging efforts of a class, Line of codes (LOC), and Dependency Inversion Principle (DIP) which measure class's dependability. Shrivastava. [63] researched the effects of applying TDD on test case design using six metrics mentioned above. In the experiment, two projects were implemented, one using TLD and the other using TDD. TDD was significantly better in Weighted Methods per Class (WMC), Response for a Class (RFC), and Depth of Inheritance Tree (DIT), which means the classes implemented using TDD were less complex and reusable.

Branch coverage or code coverage and mutation score of unit tests are complementary metrics used to measure the internal quality of the software. Branch or code coverage measures the effectiveness or thoroughness of the test cases and a mutation score that reveals how well the code is covered using test cases and how effective the test cases are to find detectors [49][59]. Madeyski. [59] evaluated the impact of unit tests in TDD using a group of students and found out that there is no statistically significant difference between the two approaches. However, in an experiment, Tosun et al. [28] found that TDD improves mutation score and branch coverage which are beneficial to find and reduce bugs. Causevic. [49] conducted a similar experiment, but he considered both internal and external quality. He experimented with the quality of test cases and estimated how test cases contributed to the internal and external quality of the software. The internal quality effectiveness of the test cases is measured using the code coverage and the mutation score. Then, the test cases' external quality defect detection capability was measured by counting the failing test cases. In the experiment, the overall quality of the test cases was nearly the same for TLD and TDD, but they had the overall impression that code quality is improved when using TDD.

2.6.2 External Quality

External quality refers to how properly the system is when viewed externally. Defects found during the testing [24][27] or after the production deployment, the number of test cases passed after the development phase [45], requirement validation [58] are the primary criteria used to measure the external quality. Fucci et al. [58] researched how developers' TDD skills help external quality by requirement validation. Thirty developers are clustered according to their TDD experience and then asked to create a feature using TDD. Then the code was evaluated for external quality; the research could not find a statistical significance between TDD skills and external quality. However, another research [24] that explored the effectiveness of TDD by examining the last ten years of research papers showed a positive impact on software quality.

2.6.3 Productivity

Productivity refers to the time and effort for developing the system and the number of person-hours [45]; the number of features for a single person-hour [30] is the widely used productivity metric in the literature. Multiple pieces of research can be found on the effect on productivity, and most of them either reflect decreased productivity [24][30][43][64] or do not show any significant difference [39][58]. The effect of TDD skills and productivity was evaluated by Fucci et al. [58], but he could not find a statistical significance between them.

Several pieces of research [43][45][61][64] can be found on how the efforts are changed when using test-first programming and test last programming. However, effort in terms of coding and effort in terms of testing is not much analyzed separately. However, Huang et al. [65]

experimented on the effectiveness of efforts separately, productivity, and external quality. Furthermore, they took an extra step by analysing the correlations between each other and adding more metrics when measuring external quality, which most researchers use as defect rate. In the experiment, they considered 10 (1) the presentation (2) user manual, (3) installation guide, (4) ease of use, (5) error handling, (6) understandability, (7) base functionality, (8) innovation, (9) robustness, and (10) happiness (overall satisfactory) factors which external clients assessed. The experiment revealed that TFP helps increase productivity but has high testing efforts. Regarding the external quality, both approaches had the same level of quality, and the only interesting correlation was the linear correlation between testing effort vs. coding effort, where coding efforts are decreasing with increasing testing efforts.

Some researchers have conducted experiments for all three directions [45]. Bissi. [45] conducted a systematic literature review on the effects of TDD on internal quality, external quality, and productivity. They have collected the articles between 1999 and 2014, and 27 filtered articles were used in this review. The review found an increase in internal quality both in industrial and academic settings except for two articles and found 16 studies that compared external quality between TLD and TDD. Most of the studies show a significant improvement. Most studies used the passed and failed count of test cases during the black box testing during the testing phase to measure the external quality. Furthermore, they evaluated how productivity changes when using TLD and TDD, and most studies have reported that productivity decreases during the TDD approach.

Another industrial experiment [61] of all three quality factors was conducted for a year-long project using undergraduate students. It measured the metrics like hours for a feature, production code size, test code size, code coverage, cyclomatic complexity, weighted methods per class, number of defects, and programmer opinion. The experiment revealed a productivity drop in TDD, but the production code size was the same in both TLD and TDD. Also, TDD produced seven times more test code than the TLD regarding the test code size, which is not a surprise. Cyclomatic complexity was similar for both approaches, but TLD had a more excellent code coverage and weighted methods per class than TDD. Furthermore, TLD had 39% more defects, and the collected programmers' perceptions favoured TDD.

In 2010 another systematic literature review on three quality factors was carried out by Kollanus, S. [64] to analyze the existing evidence on TDD. He filtered out 40 articles based on empirical evidence. He found that TDD increases the external quality but decreases productivity as it requires more development time for unit tests, which later becomes a benefit. However, the evidence regarding the internal quality was inconsistent in the literature. Therefore, he could not give a clear judgment on that factor.

A similar literature review [43] on empirical studies for three factors was conducted summarizing ten different quality factors, defect density, code coverage, code complexity, coupling and cohesion, size, effort, external quality, productivity, maintainability, and aggregation results. From the literature, it was evident that TDD helps to reduce the defects but increases the initial development time. Furthermore, they concluded that TDD produces small, less complex code which is easy to maintain and where maintaining efforts are less than code developed using TLD.

Dogša et al. [30] also experimented with the effects of TDD on three quality factors, external quality, productivity, and maintainability, by applying them in an industrial project. First, external quality was measured using failure density. The ratio of units of output divided by units of effort metric was used to capture productivity. Finally, the person-hours for fixing defects, improving the code, and maintaining the code were used to measure the maintainability factor.

From the experiment, it was evident that TDD improves external quality, maintainability of the code but decreases productivity and the main advantage of using TDD was maintainability.

A literature review [39] collecting all the literature from 1999 that researched TDD's effects quantitatively ended up with 22 articles. When summarized, TDD helps moderately improve external quality by encouraging developers to think about system design, preventing bugs by unit testing, and stressing the coding discipline throughout the development, which helps for more modularized and maintainable code due to consistent feedback and frequent refactoring. When summarizing the literature on productivity, the results were contradicting. Academic experiments favoured TDD, while industrial projects favoured the TLD approach. However, as a summary, they could not find strong adverse effects on productivity in large projects even though the TDD has a high learning curve and more maintenance of test cases apart from the production code.

Among the researchers surveys [33][62], academic experiments [23][59][60][61][65], industrial experiments[28][30][48][49][55][58][63], and literature reviews[24][39][43][45][64] are the most popular categories used to measure the effects of the TDD. Measuring the TDD effects using repository reviews [27] is not much found in the literature, and the research of Borle et al. [27] is the sole research encountered in the literature review. They have analyzed Java projects in GitHub and the impact of the rate of commits, commits related to bug fixes, and no reported bugs when using TDD vs. TLD. However, unfortunately, there was no statistical significance difference on the above factors when using TDD.

Effects of reliability and program understandability of using TDD are not much visible in the literature. However, Müller et al. [60] have done an academic experiment comparing the program reliability, understandability, and productivity and found out that there is no difference in productivity but has slightly high reliability of TFD program and improved program understandability.

The literature on the long-term effects of TDD application is minimal, and the Nokia Siemens Networks team project is one of them [48]. They used TDD for the three-year-long project, and the developer views are collected using an extensive interview process. According to them, code quality was improved using TDD, but the significant effect they witnessed was the increased maintainability of the code. In some of the previous experiments, TDD decreased productivity as it increased the initial development time [24][30][48], but here they did not examine any significant productivity drop over the three-year-long application.

Applying TDD in scientific software is not much research, and effects are also not widely known in the community. Nanthaamornphong et al. [33] surveyed scientific software developers who had prior experience in TDD, and 300 developers were engaged in the process. According to the responses, TDD improves code quality, changeability, code maintainability while identifying the defects early. Also, TDD helps for a better requirement analysis as it encourages thinking about the edge cases before implementing the actual production code.

Table 7 summarizes the papers found for TDD effects, and the effect column shows the impact of TDD for the specific quality factor.

	Literature	Experiment Type	Quality Factor	Effect	Metrics Used
1	[59]	Survey	Class design	TDD does not directly improve the class design but helps indirectly for a testable class design	-
2	[63]	Industrial Experiment	Class design	The classes designed using TDD were less complex and better reusable than TLD.	Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number Of Children (NOC), Response For a Class (RFC), Line of codes (LOC), Dependency Inversion Principle (DIP)
3	[45]	Literature review	Test coverage of production code	76% of the studies reported a significant improvement in internal quality	Code complexity metrics (McCabe), Lack of Cohesion in Methods (LCOM), Single Responsibility Principle (SRP), Open-Closed Principle (OCP), Coupling Between Objects (CBO)
			No of passed or failed test cases	88% of the studies reported a significant improvement in external quality	
			Time taken to implement a new feature	44% of the studies reported a decrease in developer productivity 28% of the studies reported an increased or same level of productivity	Time and amount of generated code
4	[58]	Industrial Experiment	External Quality	No statistically significant difference between the TDD skills	No of tackled user stories(should present at least one acceptance test to mark the user story as tackled)
			Productivity	No statistically significant difference between the TDD	Percentage of passing asserts

				skills	
5	[49]	Industrial Experiment	Quality of test cases	Test Case quality is the same in TDD and TLD	Code coverage, Mutation score, Defect rate
6	[59]	Academic Experiment	Internal Quality	No statistically significant difference between the unit test quality in TDD and TLD	Branch coverage, Mutation Score
7	[65]	Academic experiment	Productivity	TFP had 70% higher productivity than TLP	Output(lines of code (LOC)) per person hour,
			Coding Effort Test effort	More testing effort and less coding effort in TFP	Percentage of time spent on testing, Percentage of time spent on coding,
			External Quality	No difference on external quality	Defect rate, Client satisfaction on the 10 criterias
8	[27]	Review of Github repositories	No of defects	No significant difference between TDD and TLD repositories	No of commits
9	[24]	Literature Review	Software Quality	Increased the quality	Number of issues found
			Productivity	Decreased the productivity	
10	[28]	Industrial experiment	Unit test effectiveness	Higher defect detection capability More branch coverage	Mutation score, Branch coverage
11	[39]	Literature Review	Internal Quality	Moderate improvement in internal quality	
			External Quality	No evidence for external quality improvement	
			Productivity	No significant difference between TLD and TDD	
12	[30]	Industrial Experiment	External quality	Improves external quality	Defect density
			productivity	Decrease productivity	Output vs effort ratio
			Maintainability	Highly improves maintainability by reducing the overhead for regression testing, improving the code design, and reducing the debugging efforts	No of person-hours for defect fixes, improvements, and maintenance

13	[64]	Literature review on empirical evidence	External Quality	Weak support for better quality	Number of defects found before the release or after the release
			Internal Quality	Contracting views on internal quality	Test coverage, number of test cases, Method size, cyclomatic complexity, coupling and cohesion
			Productivity	Increase the development time	Total development effort, Lines of code per hour, No of implemented user stories
14	[43]	Literature review on empirical evidence	External Quality	Reduce defects	No of defects
			Internal quality	Better maintenance code	McCabe's complexity, Code coverage, Complexity, Coupling and Cohesion, Code size, maintainability
			productivity	Increase initial development time	Effort, Amount of time spent on a task
15	[48]	Industrial Experiment	Code quality	Increase code quality	-
			Development speed	No significant negative impact in the long run	-
16	[33]	Survey	Code quality	Improves	-
			Code changeability	Improves	-
			Code maintainability	Improves	-
17	[23]	Academic Experiment	Code quality	Depends on actual testing efforts	No of acceptance tests passed
			Productivity	Reduce overall development effort and improve productivity	Code lines per person hour
18	[55]	Industrial experiment	Code quality	Improves, better cohesion, low coupling	-
19	[61]	Academic experiment	Productivity	Decreases	hours for a feature

			Internal quality	No clear view. Some factors favored TLD.	Production code size, test code size, cyclomatic complexity, weighted method per class
			External quality	Reduce no of defects	No of defects
20	[60]	Academic Experiment	Productivity	No difference between TFD and traditional approach	Time to implement the solution
			Program Reliability	Slightly high reliability in TFD	Percentage of passed assertions
			Program Understandability	TFD had high understandable code	No of reuse methods, No of failed method calls, No of method calls fails at least twice

Table 7 : TDD Effects Summary

TDD effects have been discussed in different types of articles and Figure 4 shows the article types and their count. According to the graph most of the TDD effects have been derived from the industrial experiments.

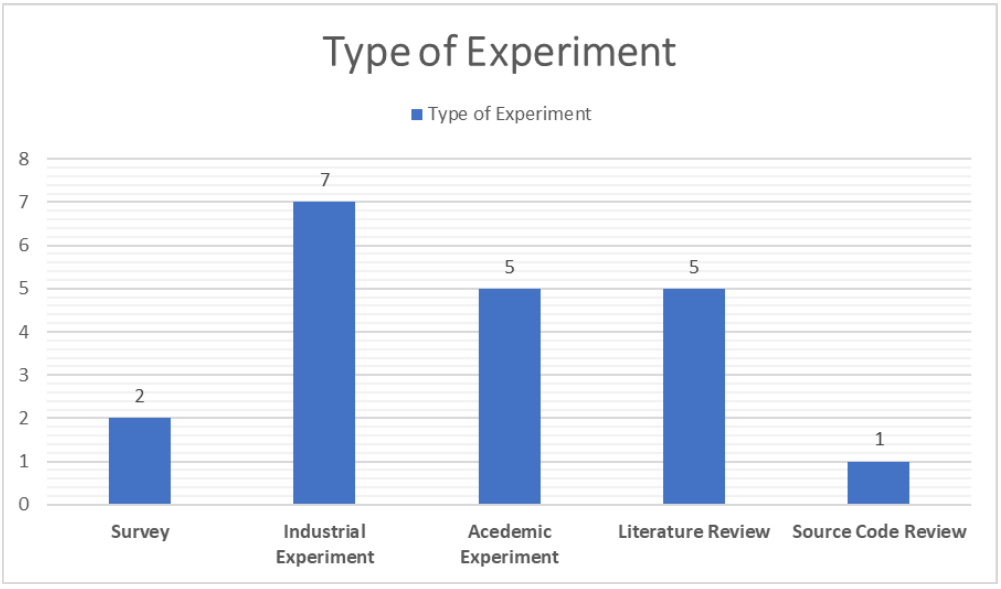


Figure 4 : TDD Experiment Types

As described above the impact of the TDD can be categorized in to the three main quality factors, 1) External Quality, 2) Internal Quality, 3) Productivity and Figure 5 shows how the opinions have varied for them in the literature.

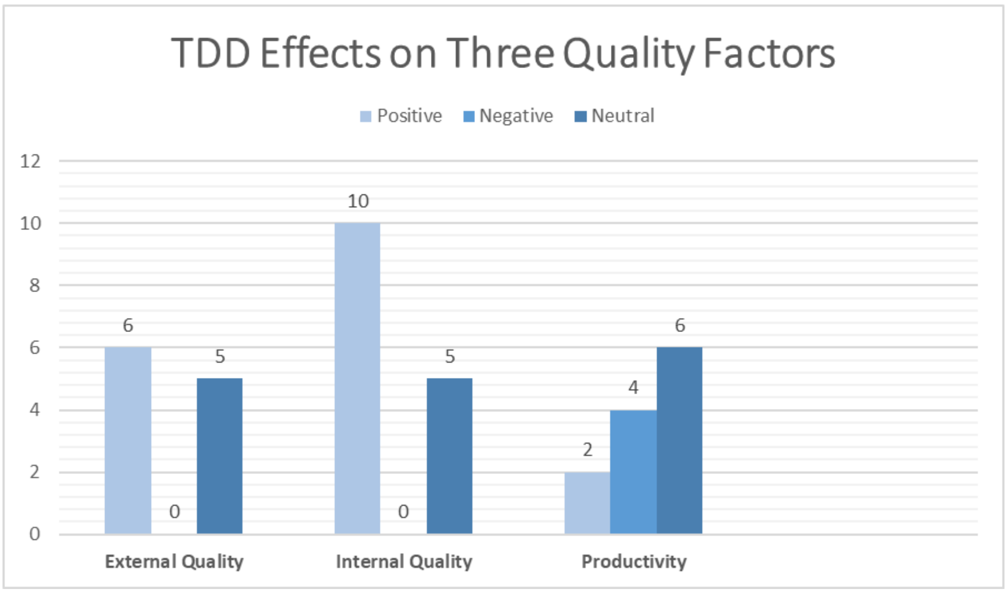


Figure 5 : TDD Effects on Quality Factors

Based on the review it is visible that TDD improves the external quality and internal quality of the project but requires more time than a TLD project.

3 Design and Implementation

This thesis aims to explore the feasibility of TDD in the big data domain using the microservice-based TDD approach. Therefore, an online fraud detection application is designed and developed using GCP services to explore the feasibility. The platform consists of multiple microservices, and creating the application is divided into three major parts.

The first step is developing a machine learning model. A machine learning model is needed to identify the transaction as fraudulent or genuine. In this project, tasks like selecting the data, pre-processing data, creating the machine learning model, training the machine learning model, and deploying the machine learning model for online predictions are carried out here.

Then developing the rule engine which validates the transaction is carried out. There are different ways to identify fraud, and traditionally rule-based fraud detection approaches have been used before introducing machine learning application approaches. Rule-based systems use conditions, and if the state is matched, it will mark the transaction as fraud or potential fraud. Therefore, this thesis uses machine learning and rule engine-related fraud detection to improve prediction accuracy. Also, it is aimed to identify the feasibility of applying TDD in the rule engine microservice by introducing a rule engine.

Finally, the online fraud detection platform is developed. The online fraud detection platform is the final product of this thesis. It is the place where all microservices connect 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 process 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 out here.

3.1 Machine Learning Model

Fraud detection is a binary classification problem that categorizes the transaction as fraudulent or legitimate [66]. GCP platform provides several tools and services to build classification machine learning models. However, the most crucial first step toward building such a model is finding a suitable dataset for creating the model. Therefore, an extensive search was carried out to find a dataset. First, open fraud detection datasets are searched on Google and then searched multiple fraud detection research papers [67][68][69][70][71] to investigate the databases they used in their research work. However, the search did not result in a suitable database. Then, the fraud detection datasets were searched in Kaggle, and a suitable dataset set was found: a simulated transaction dataset. The reason behind selecting the dataset is explained below. Then the machine learning model creation began, and Figure 6 shows the architecture of creating the machine learning model.

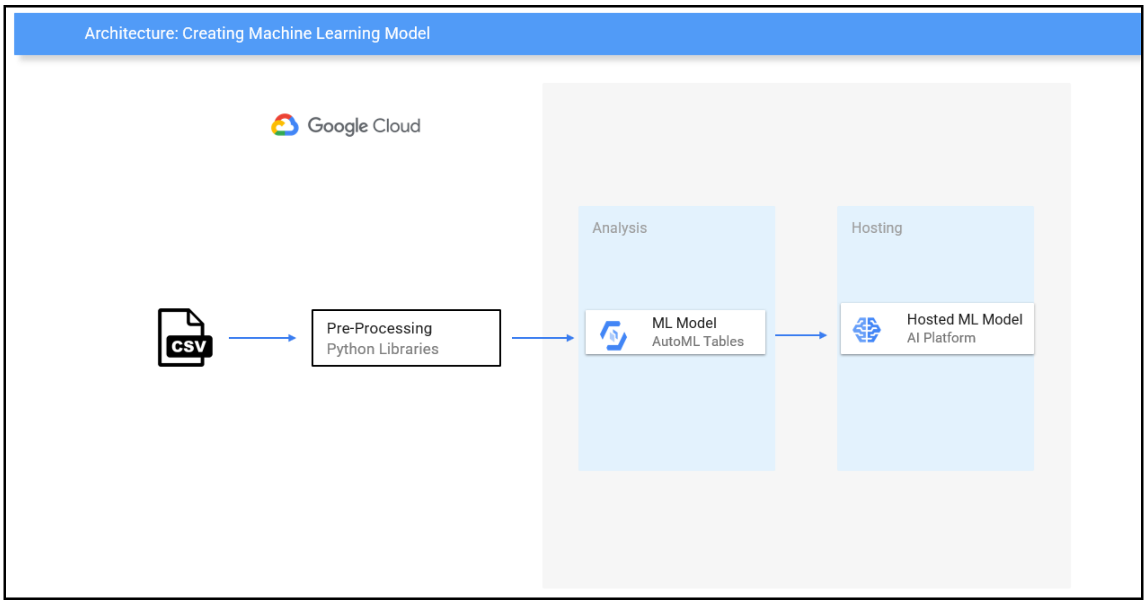
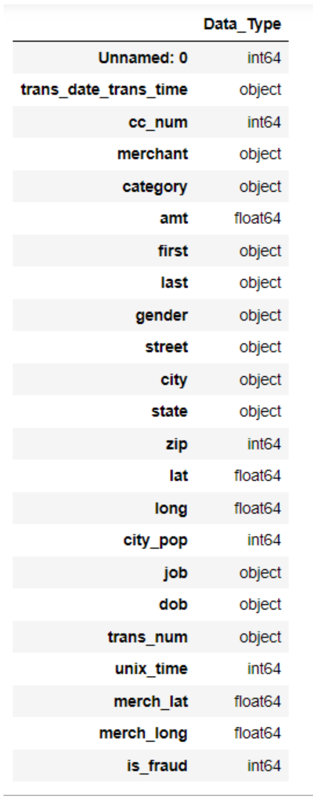


Figure 6 : Machine Learning Model

3.1.1 Dataset

Financial institutions have strict regulations and compliance requirements for credit card transactions and storing their data. Therefore, freely available actual credit card datasets that can be used for the machine learning models are limited. Kaggle is one of the popular websites among the machine learning community with open datasets that can be used to explore and build models. Kaggle is one of the most popular online communities for people interested in machine learning and data science. It provides key services like machine learning competitions, cloud-based workbenches, public dataset platforms, and educational platforms with artificial intelligence related courses. Therefore, credit card fraud detection datasets are searched in Kaggle and sorted according to relevance in this project. In Kaggle, public datasets are searched using *Keywords=credit card fraud detection, Dataset Size=largeORmedium, AND DatasetFileTypes=csv* and sorted the results according to relevancy. Thirty datasets were shown in the result set, and then a few datasets were shortlisted based on the usability score. The usability score is a score generated by Kaggle considering the level of documentation for the dataset. Usability is essential to understanding the dataset. Finally, three datasets were selected which had high usability scores. Then one dataset was dropped from the list as it was not directly related to credit card transition even though the title says so. It had data related to credit card transactions considered when lending the loans. The other dataset was related to credit card transactions, but the data are after Principal Component Analysis (PCA) dimensionality reduction, which has been done to remove sensitive information and protect users. The data columns are named V1, V2, etc. Hence, actual data columns cannot be identified. So, it is not suitable for the purpose as understanding the features is essential in creating a machine learning model and the rule engine. Then the most relevant and usable database, *https://www.kaggle.com/kartik2112/fraud-detection* is selected for the project. This dataset is a simulated credit card transaction database using *Sparkov*, and it contains legitimate and fraud 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 about features is self-explaining. The dataset contains fraudTest.csv and fraudTrain.csv, divided the dataset into two parts. Furthermore, Figure 7 shows the columns present and their data type.



	Data_Type
Unnamed: 0	int64
trans_date_trans_time	object
cc_num	int64
merchant	object
category	object
amt	float64
first	object
last	object
gender	object
street	object
city	object
state	object
zip	int64
lat	float64
long	float64
city_pop	int64
job	object
dob	object
trans_num	object
unix_time	int64
merch_lat	float64
merch_long	float64
is_fraud	int64

Figure 7 : Columns and Data Types

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.

3.1.2 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 [2].

Pre-processing is the first step carried out in model creation, and GCP has several tools that can be used for cleansing and transforming data. *BigQuery ML* supports automatic pre-processing and manual pre-processing [73]. In *BigQuery ML*, a single *SELECT* statement with different *WHERE* clauses will be used to load and process the data for the model. To write the unit tests *SELECT* statement should be executed for each pre-processing step. For example, first, a *SELECT* statement should be performed, and then the following *SELECT* statement should be executed on the previous result. This step execution is complex in *BigQuery* as it uses one *SELECT* statement to load all data, and because of that, it was not used in the project. *Dataprep* is another popular pre-processing and great visualization tool provided by Trifacta. It is an intelligent data service for visually exploring, cleaning, and preparing structured and unstructured data for analysis, reporting, and machine learning. However, *Dataprep* mainly supports UI, and the REST API is not self-explainable. Therefore, it was not easy to use in the project. 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, following cleansing and transformation steps were carried out using the *Python* libraries mentioned above. 1) In the first step raw dataset is scanned to check if there are any missing values in the *TARGET* column, which *is_fraud*, and removed the rows where the *TARGET* is missing. 2) Then, the columns with a high number of missing values are searched, and here search for the columns where half of the rows have missing values and remove them. 3) In the third step, duplicate rows and columns with a single value are explored as they do not add value to machine learning model creation. 4) After that, categorical data columns are transformed into numerical, but this was not used for the fraud detection dataset as *AutoML Table* can handle absolute values. Nevertheless, the tests and implementation are available for this pre-processing step.5) In the fifth step, the missing values are imputed using the strategy mean in *sklearn.impute.SimpleImputer*. It replaces the missing values with the mean of each column. 6) Then, outlier removal has been performed using the Inter Quartile Range mechanism, the most trusted mechanism for removing unusual data points [72]. Inter Quartile Range depicts a measure in descriptive statistics that tells the middle range of the dataset [72]. 7) Finally, the top features are selected 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. Ultimately, the cleaned data was written to a CSV file. Figure 8 shows the automatic pre-processing carried out for the data.

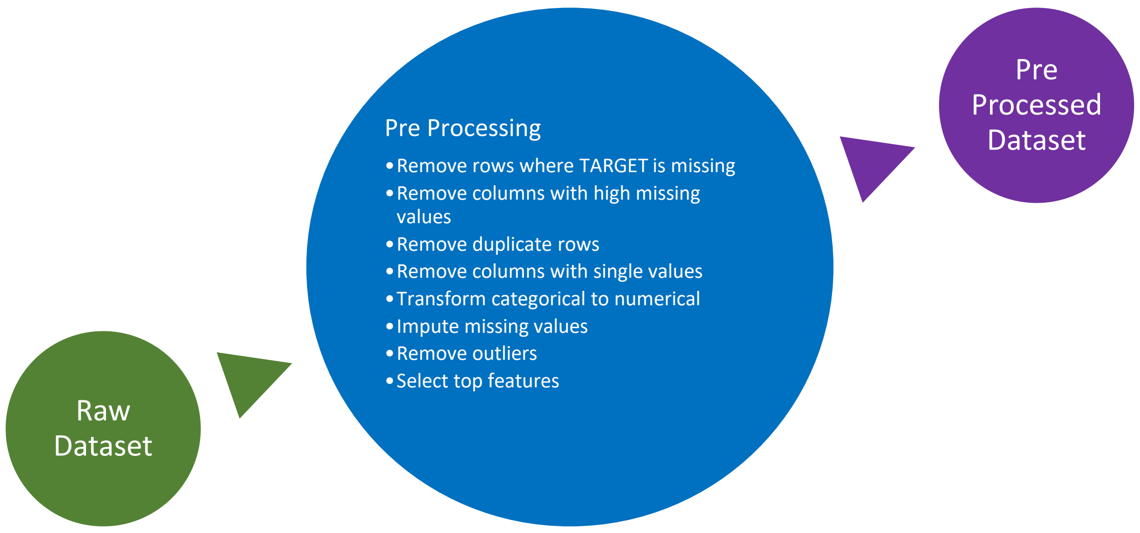
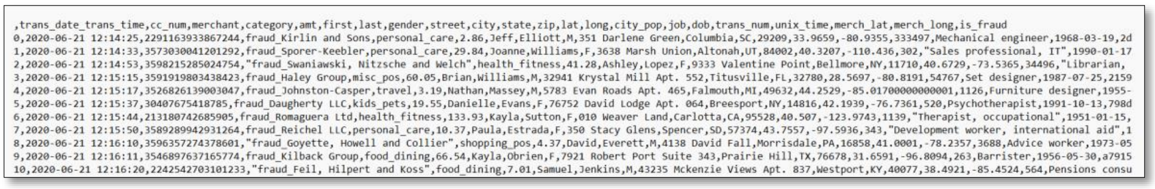

Raw Dataset
Pre Processing
• Remove rows where TARGET is missing
• Remove columns with high missing values
• Remove duplicate rows
• Remove columns with single values
• Transform categorical to numerical
• Impute missing values
• Remove outliers
• Select top features
Pre Processed Dataset

Figure 8 : Pre-Processing Process

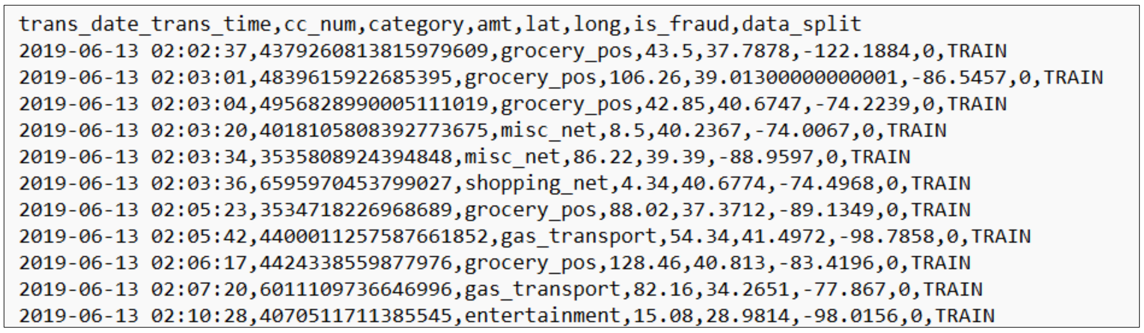
Figure 9 shows a sample of raw data before pre-processing.



```
,trans_date_trans_time,cc_num,merchant,category,amt,first,last,gender,street,city,state,zip,lat,long,city_pop,job,dob,trans_num,unix_time,merch_lat,merch_long,is_fraud
0,2020-06-21 12:14:25,2291163933867244,fraud_Kirlin and Sons,personal_care,2.86,Jeff,Elliott,M,351 Darlene Green,Columbia,SC,29209,33.9659,-80.9355,333497,Mechanical engineer,1968-03-19,2d
1,2020-06-21 12:14:33,3573030041201292,fraud_Sporer-Keebler,personal_care,29.84,Joanne,Williams,F,3638 Marsh Union,Altonah,UT,84002,40.3207,-110.436,302,"Sales professional, IT",1990-01-17
2,2020-06-21 12:14:53,3598215285024754,"fraud_Swaniawski, Nitzsche and Welch",health_fitness,41.28,Ashley,Lopez,F,9333 Valentine Point,Bellmore,NY,11710,40.6729,-73.5365,34496,"Librarian,
3,2020-06-21 12:15:15,3591919803438423,fraud_Haley Group,misc_pos,60.05,Brian,Williams,M,32941 Krystal Mill Apt. 552,Titusville,FL,32780,28.5697,-80.8191,54767,Set designer,1987-07-25,2159
4,2020-06-21 12:15:17,3526826139003047,fraud_Johnston-Casper,travel,3.19,Nathan,Massey,M,5783 Evan Roads Apt. 465,Falmouth,MI,49632,44.2529,-85.0170000000001,1126,Furniture designer,1955-
5,2020-06-21 12:15:37,30407675418785,fraud_Daugherty LLC,kids_pets,19.55,Danielle,Evans,F,76752 David Lodge Apt. 064,Breesport,NY,14816,42.1939,-76.7361,520,Psychotherapist,1991-10-13,798d
6,2020-06-21 12:15:44,213180742685905,fraud_Romaguera Ltd,health_fitness,133.93,Kayla,Sutton,F,010 Weaver Land,Carlotta,CA,95528,40.507,-123.9743,1139,"Therapist, occupational",1951-01-15,
7,2020-06-21 12:15:50,3589289942931264,fraud_Reichel LLC,personal_care,10.37,Paula,Estrada,F,350 Stacy Glens,Spencer,SD,57374,43.7557,-97.5936,343,"Development worker, international aid",1
8,2020-06-21 12:16:10,3596357274378601,"fraud_Goyette, Howell and Collier",shopping_pos,4.37,David,Everett,M,4138 David Fall,Morrisdale,PA,16858,41.0001,-78.2357,3688,Advice worker,1973-05
9,2020-06-21 12:16:11,3546897637165774,fraud_Kilback Group,food_dining,66.54,Kayla,Obrien,F,7921 Robert Port Suite 343,Prairie Hill,TX,76678,31.6591,-96.8094,263,Barrister,1956-05-30,a7915
10,2020-06-21 12:16:20,2242542703101233,"fraud_Feil, Hilpert and Koss",food_dining,7.01,Samuel,Jenkins,M,43235 Mckenzie Views Apt. 837,Westport,KY,40077,38.4921,-85.4524,564,Pensions consu
```

Figure 9 : Raw Data

Figure 10 shows the dataset after pre-processing and after adding the new data column *data_split*.



```
trans_date_trans_time,cc_num,category,amt,lat,long,is_fraud,data_split
2019-06-13 02:02:37,4379260813815979609,grocery_pos,43.5,37.7878,-122.1884,0,TRAIN
2019-06-13 02:03:01,4839615922685395,grocery_pos,106.26,39.01300000000001,-86.5457,0,TRAIN
2019-06-13 02:03:04,4956828990005111019,grocery_pos,42.85,40.6747,-74.2239,0,TRAIN
2019-06-13 02:03:20,4018105808392773675,misc_net,8.5,40.2367,-74.0067,0,TRAIN
2019-06-13 02:03:34,3535808924394848,misc_net,86.22,39.39,-88.9597,0,TRAIN
2019-06-13 02:03:36,6595970453799027,shopping_net,4.34,40.6774,-74.4968,0,TRAIN
2019-06-13 02:05:23,3534718226968689,grocery_pos,88.02,37.3712,-89.1349,0,TRAIN
2019-06-13 02:05:42,4400011257587661852,gas_transport,54.34,41.4972,-98.7858,0,TRAIN
2019-06-13 02:06:17,4424338559877976,grocery_pos,128.46,40.813,-83.4196,0,TRAIN
2019-06-13 02:07:20,6011109736646996,gas_transport,82.16,34.2651,-77.867,0,TRAIN
2019-06-13 02:10:28,4070511711385545,entertainment,15.08,28.9814,-98.0156,0,TRAIN
```

Figure 10 : Pre-Processed Data

3.1.3 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 one of the popular GCP tools that automatically builds and deploys a machine learning model [74]. 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, decreasing the Root Mean Square Error (RMSE), etc., without manual intervention for feature engineering, ensembling, etc. [74]. *BigQuery ML* is also another candidate for model creation, but the *AutoML Tables* are used here as the main focus is on increasing the model quality rather than model experimentation.

In *AutoML Tables*, the dataset is split automatically as 80% for training, 10% for validating, 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 [74]. 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 via *Terraform* to be able to access AutoML Tables. Storage buckets are containers that can hold data, and other GCP services can connect to these containers and access data. *Terraform* is an infrastructure code tool that enables the infrastructure's safe and efficient building and maintenance. 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 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. The created machine learning model will be deployed in an AI platform in this project.

A *Python* project is created to pre-process data and implement the machine learning model due to the availability of many powerful, easy-to-use machine learning-related python libraries. For the development *PyDev*, *9.2.0 addon*, which is the latest, is installed in Eclipse IDE and utilized. The project was based on the machine learning project structure with src, features, models, visualization, tests, and data [75]. *Numpy, pandas,* and *sklearn* libraries are utilized to clean the data, and the *google-cloud-automl_v1beta1* Python module was used to build and deploy the machine learning model. The source code of this project can be found here, *https://github.com/maneendra/TDD-FraudeDetection-GCP-Project.*

3.1.3.1 TDD in Machine Learning Model

Test-driven development is the methodology used in the project. First, the test is written, and then the relevant code is implemented to pass the test. *Unittest* is used to write the tests, the default library bundled with Python. It is said that Python *unittest* has been inspired by the *Java JUnit* test framework [76]. *Unittest* support important concepts like test fixture – preparation steps to run a test or tests, test case - the smallest unit of testing, test suite – the collection of test cases, and test runner – component which executes the test cases provides the result to the user [76]. *Pytest* is another most popular, widely used unit test framework for *Python*. However, it has to be installed, but less code is required, and it has a compact view [77]. Since the focus of the thesis is TDD, it is essential to understand the test structure and test cases precisely. Therefore, in the project, *unittest* is employed as the tests are easily understandable.

Tests written for pre-processing functions are executed locally in Eclipse IDE. 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 [78]. However, since the *AutoML Tables* do not have the emulation option, tests were executed in the real environment minimizing the costs by selecting the bank notes dataset, a small benchmark classification dataset that fulfills the *AutoML Tab*le's minimum requirements mentioned below [74].

- The dataset must contain the TARGET column
- The dataset must contain at least two columns
- The dataset must contain at least 1000 rows

Figure 11 shows a sample of the banknote’s dataset.

```
variance,skewness,curtosis,entropy,Target
3.6216,8.6661,-2.8073,-0.44699,0
4.5459,8.1674,-2.4586,-1.4621,0
3.866,-2.6383,1.9242,0.10645,0
-3.7503,-13.4586,17.5932,-2.7771,1
-3.5637,-8.3827,12.393,-1.2823,1
-2.5419,-0.65804,2.6842,1.1952,1
```

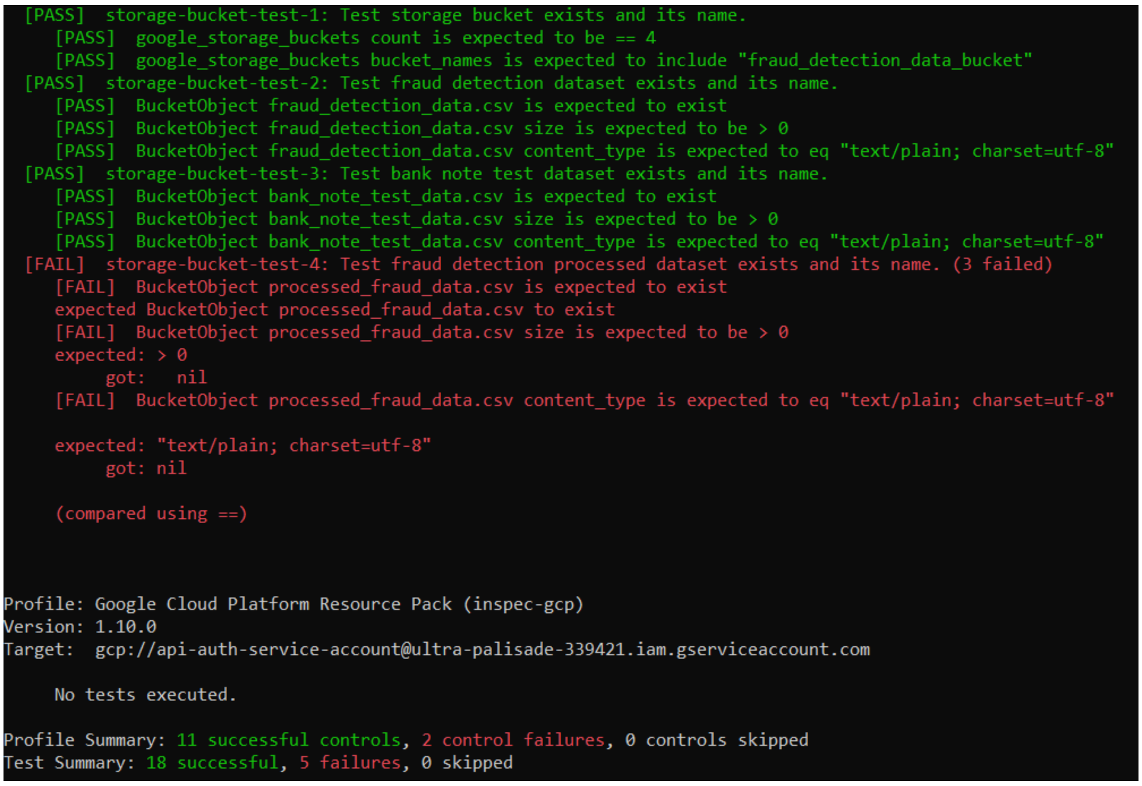
Figure 11 : Banknotes Dataset

Then a classification model is created using the banknote dataset, and then the actual fraud detection classification model is created using pre-processed credit card dataset. 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 them, *Chef InSpec* 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 [79]. Then the *Terraform* scripts are written and executed, and then again, *Chef InSpec* tests are executed to verify that the resource creation is successful. Figure 12 shows two tests written to verify the availability of the storage bucket and the CSV files and their content.

```
# Author - Maneendra Perera

title "Storage Bucket Tests"

gcp_project_id = input("gcp_project_id")


# Verify that the storage bucket exists.

control "storage-bucket-test-1" do

        impact 1.0

        title "Test storage bucket exists and its name."

        describe google_storage_buckets(project: 'ultra-palisade-339421') do

                its('count') { should be == 4}

                its('bucket_names'){ should include "fraud_detection_data_bucket" }

        end

end


# Verify that the dataset exists.

control "storage-bucket-test-4" do

        impact 1.0

        title "Test fraud detection processed dataset exists and its name."

        describe google_storage_bucket_object(bucket: 'fraud_detection_data_bucket',  object: 'processed_fraud_data.csv') do

                it { should exist }

                its('size') { should be > 0 }

                its('content_type') { should eq "text/plain; charset=utf-8" }

        end

end
```

Figure 12 : Storage Bucket Tests

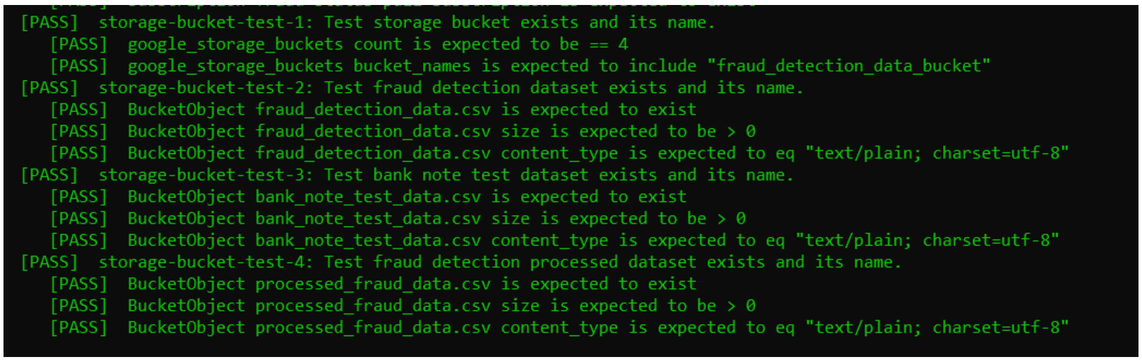
Figure 13 shows the result gained when the *Chef InSpec* tests were executed before running the *Terraform* scripts. Before running the scripts, storage buckets have not been created in the cloud environment, and thus, tests that are written to check the existence of storage buckets fail.



```
  [PASS]  storage-bucket-test-1: Test storage bucket exists and its name.
     [PASS]  google_storage_buckets count is expected to be == 4
     [PASS]  google_storage_buckets bucket_names is expected to include "fraud_detection_data_bucket"
  [PASS]  storage-bucket-test-2: Test fraud detection dataset exists and its name.
     [PASS]  BucketObject fraud_detection_data.csv is expected to exist
     [PASS]  BucketObject fraud_detection_data.csv size is expected to be > 0
     [PASS]  BucketObject fraud_detection_data.csv content_type is expected to eq "text/plain; charset=utf-8"
  [PASS]  storage-bucket-test-3: Test bank note test dataset exists and its name.
     [PASS]  BucketObject bank_note_test_data.csv is expected to exist
     [PASS]  BucketObject bank_note_test_data.csv size is expected to be > 0
     [PASS]  BucketObject bank_note_test_data.csv content_type is expected to eq "text/plain; charset=utf-8"
  [FAIL]  storage-bucket-test-4: Test fraud detection processed dataset exists and its name. (3 failed)
     [FAIL]  BucketObject processed_fraud_data.csv is expected to exist
     expected BucketObject processed_fraud_data.csv to exist
     [FAIL]  BucketObject processed_fraud_data.csv size is expected to be > 0
     expected: > 0
          got:    nil
     [FAIL]  BucketObject processed_fraud_data.csv content_type is expected to eq "text/plain; charset=utf-8"

     expected: "text/plain; charset=utf-8"
          got: nil

     (compared using ==)


Profile: Google Cloud Platform Resource Pack (inspec-gcp)
Version: 1.10.0
Target:  gcp://api-auth-service-account@ultra-palisade-339421.iam.gserviceaccount.com

     No tests executed.

Profile Summary: 11 successful controls, 2 control failures, 0 controls skipped
Test Summary: 18 successful, 5 failures, 0 skipped
```

Figure 13 : Tests Before Terraform Script Executed

Figure 14 shows the *Chef InSpec* test results after running the *Terraform* scripts. Now the tests are passing as required resources are available in the cloud environment.



```
  [PASS]  storage-bucket-test-1: Test storage bucket exists and its name.
     [PASS]  google_storage_buckets count is expected to be == 4
     [PASS]  google_storage_buckets bucket_names is expected to include "fraud_detection_data_bucket"
  [PASS]  storage-bucket-test-2: Test fraud detection dataset exists and its name.
     [PASS]  BucketObject fraud_detection_data.csv is expected to exist
     [PASS]  BucketObject fraud_detection_data.csv size is expected to be > 0
     [PASS]  BucketObject fraud_detection_data.csv content_type is expected to eq "text/plain; charset=utf-8"
  [PASS]  storage-bucket-test-3: Test bank note test dataset exists and its name.
     [PASS]  BucketObject bank_note_test_data.csv is expected to exist
     [PASS]  BucketObject bank_note_test_data.csv size is expected to be > 0
     [PASS]  BucketObject bank_note_test_data.csv content_type is expected to eq "text/plain; charset=utf-8"
  [PASS]  storage-bucket-test-4: Test fraud detection processed dataset exists and its name.
     [PASS]  BucketObject processed_fraud_data.csv is expected to exist
     [PASS]  BucketObject processed_fraud_data.csv size is expected to be > 0
     [PASS]  BucketObject processed_fraud_data.csv content_type is expected to eq "text/plain; charset=utf-8"
```

Figure 14 : Tests After Terraform Script Executed

Terraform is an infrastructure as code tool, and it consists of a three-stage workflow for resource creation. In the first phase, the cloud resource configurations are written to a file, and *Terraform* creates a *terraform.tfstate* file for each configuration. This phase is called the *write* stage, and the next stage *plan*, *Terraform*, reviews the changes that need to be performed on the infrastructure. Finally, actual modifications are performed in the cloud resources in the *apply* stage, and the *Terraform* state file is updated with the latest changes [80]. As described above, *Chef InSpec* tests are executed before and after the *Terraform* scripts. To run controls, *Chef InSpec* connects to the GCP environment based on the configurations provided in *inspec.yml*. Figure 15 depicts the architecture of resource creation.

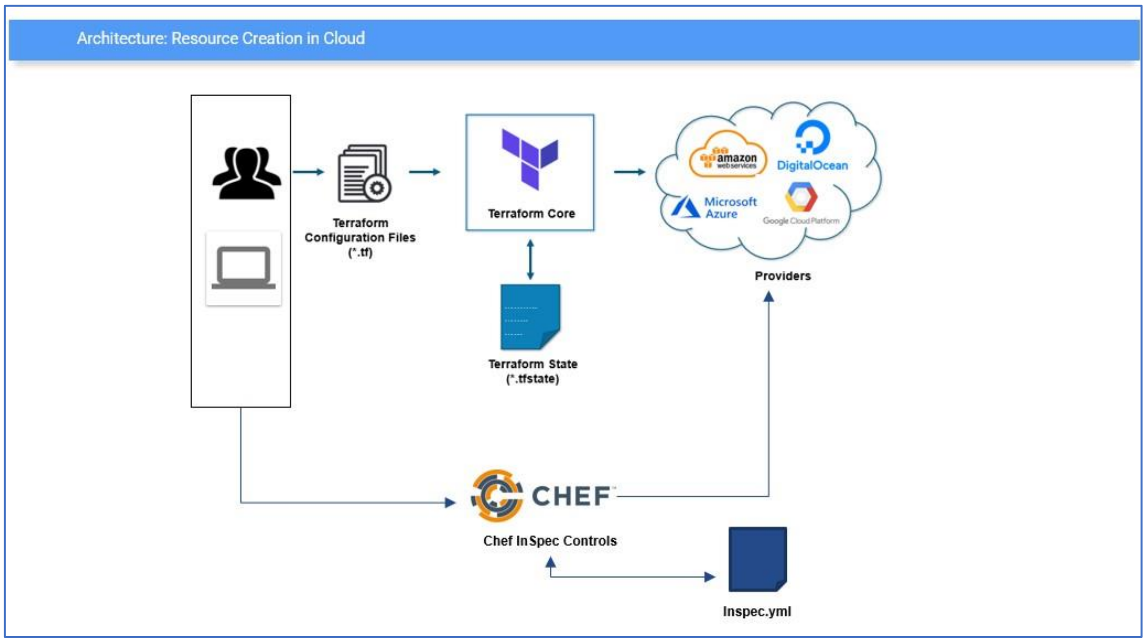


Figure 15 : Resource Creation in Cloud

The TDD process used in the resource creation is shown below in Figure 16.

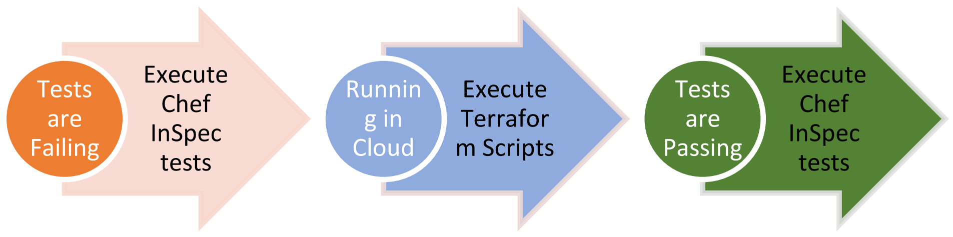

Tests are Failing
Execute Chef InSpec tests
Running in Cloud
Execute Terraform Scripts
Tests are Passing
Execute Chef InSpec tests

Figure 16 : Resource Creation TDD Process

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, used to detect fraud in the fraud detection domain. 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 computer 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 the *Dataflow* will invoke the rule engine application.

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* is installed, and 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 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 below. These five rules are created based on the common factors used in fraud detection models [81].

- Check if the transaction amount exceeds the daily transaction amount (threshold = 500)
- Check if the daily transaction total exceeds the daily transaction total (threshold = 1000)
- Check if the daily transaction count exceeds the daily transaction count (threshold = 5)
- Check if the time difference of the subsequent daily transactions is greater than the minimum time difference (threshold =5 seconds)
- Check if the distance of the subsequent daily transactions is lesser than the maximum distance(threshold=1000km)

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 hosted AL 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 [82]. Also, it provides a more flexible architecture and allows to write unit tests quickly [82]. Figure 17 shows the architectural pattern used in the rule engine. The source code of the project can be found here, *https://github.com/maneendra/gcp-fraud-detection/tree/master*.

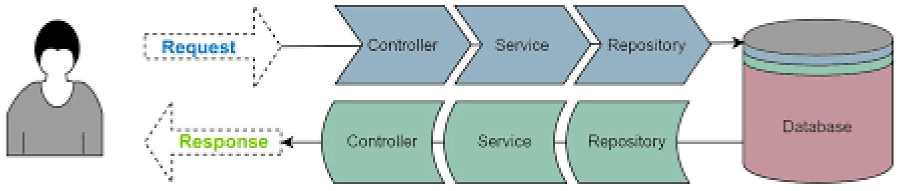


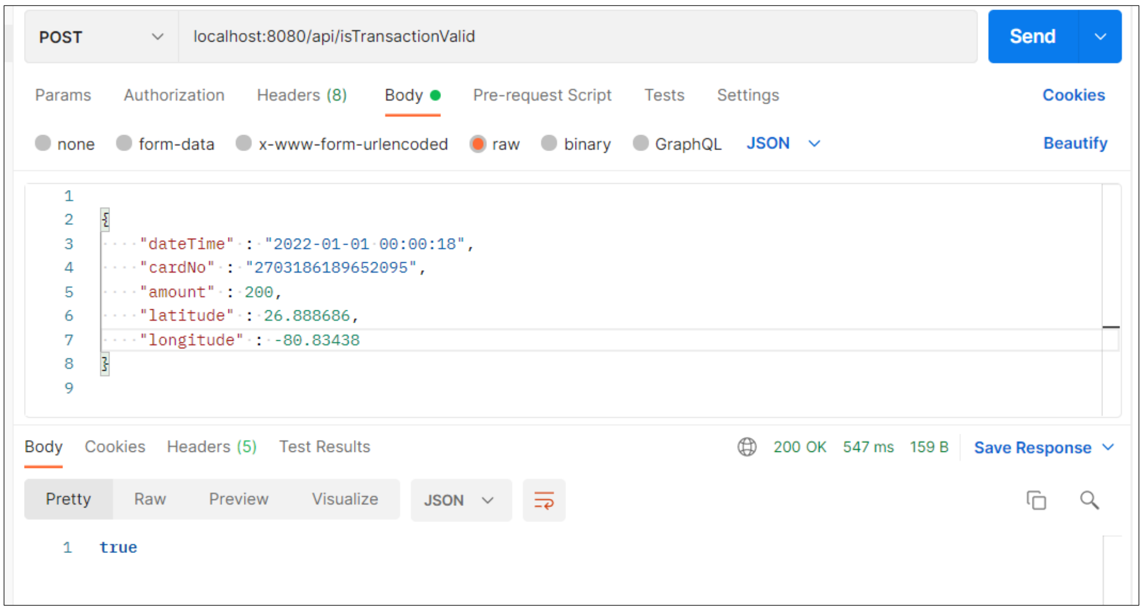
Figure 17 : Controller Service Repository Pattern

Source - https://livebook.manning.com/book/code-like-a-pro-in-c-sharp/chapter-10/v-8/16

3.2.1 TDD in Rule Engine

During the development of the web service, 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. Then the 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 as below.

After local testing, the code is bundled as a WAR file and deployed in the compute engine. The WAR file is simply a collection of *Java* classes and all necessary resources to be deployed on a server. Before creating the compute engine virtual machine instance in GCP, the *Chef InSpec* test is written for testing the resource creation. Then 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 like machine type. Then in the VM, *Java*, *Tomcat*, and *MYSQL* are installed to be able to run the web service and war file deployed. After deployment, POSTMAN was used to send the requests to the REST API and verify that the deployment was successful. Figure 18 shows sending a request and getting a response from the locally deployed rule engine.



```

{
    "dateTime" : "2022-01-01 00:00:18",
    "cardNo" : "2703186189652095",
    "amount" : 200,
    "latitude" : 26.888686,
    "longitude" : -80.83438
}

```

```
true
```

Figure 18 : Postman Request and Response

3.3 Online Fraud Detection Platform

The final phase of creating the application is developing the online fraud detection platform using GCP products. A separate *Maven* project was created for the purpose. The system's architecture and each microservice are shown in Figure 19. The online fraud detection platform 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 relevant service.

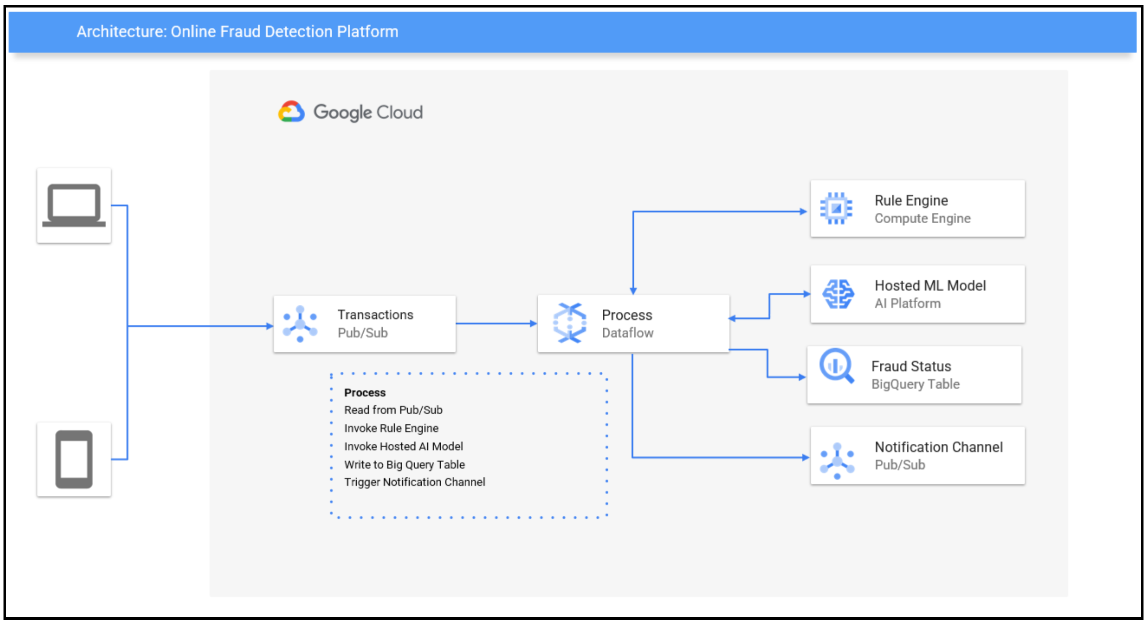


Figure 19 : Online Fraud Detection Platform

The microservice architecture is a variant of Service-Oriented Architecture (SOA), and there, a collection of autonomous services communicate with each other to perform a business function [11]. Different asynchronous communication patterns can be used among services and messaging; the event-driven approach and RESTful are some widely used patterns in software engineering. For example, the online fraud detection platform is planned to use this architecture, and the source code can be found here, *https://github.com/maneendra/Fraud-Big-Data-Application.*

3.3.1 Pub/Sub

In the fraud detection platform, transaction details come from different services like desktop computers, mobile phones, laptops, etc. 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. Figure 20 illustrates the *Pub/Sub* messaging flow of the system.

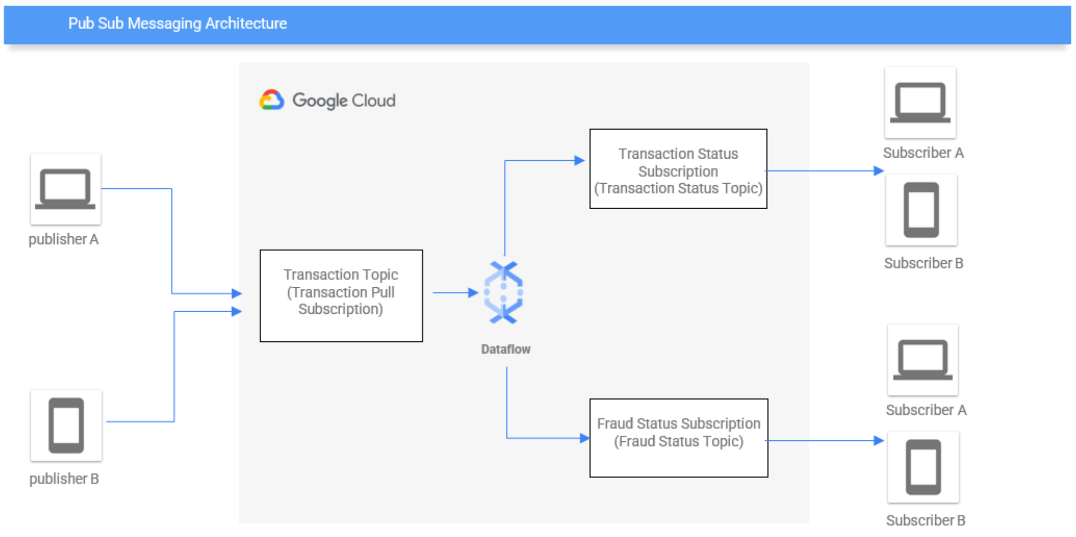
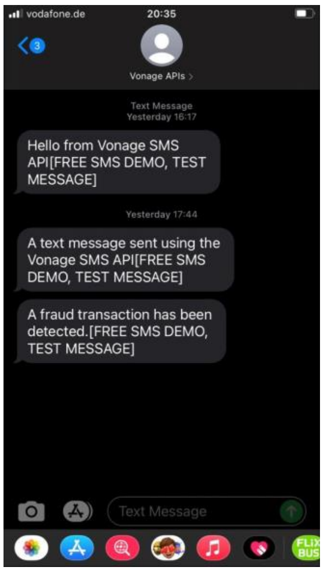

Pub Sub Messaging Architecture
Google Cloud
publisher A
publisher B
Transaction Topic (Transaction Pull Subscription)
Dataflow
Transaction Status Subscription (Transaction Status Topic)
Fraud Status Subscription (Fraud Status Topic)
Subscriber A
Subscriber B
Subscriber A
Subscriber B

Figure 20 : Pub/Sub Messaging Architecture

Here there is a topic called *transaction-topic*, and all the 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, an SMS or a call can be sent to the relevant parties. In this project, a sample cloud function has been written, and *Vonage API* is used to send an SMS when a new message arrives. Figure 21 shows the sample SMS received.



vodafone.de 20:35

Vonage APIs >

Text Message
Yesterday 16:17

Hello from Vonage SMS API[FREE SMS DEMO, TEST MESSAGE]

Yesterday 17:44

A text message sent using the Vonage SMS API[FREE SMS DEMO, TEST MESSAGE]

A fraud transaction has been detected.[FREE SMS DEMO, TEST MESSAGE]

Figure 21 : Sample SMS

3.3.1.1 TDD in Pub/Sub

Prior creating the *Pub/Sub* topics and subscriptions, *Chef InSpec* tests are written to assure the resource exists in the GCP project. For example, Figure 22 shows two tests written to verify the topic and subscription in the cloud.

```
# Author - Maneendra Perera

title "PubSub Tests"

gcp_project_id = input("gcp_project_id")

# Verify the transaction topic exists.

control "pubsub-test-1" do

        impact 1.0

        title "Test pub sub transaction topic exists."

        describe google_pubsub_topic(project: gcp_project_id, name: 'transaction-topic') do

          it { should exist }

        end

end

# Verify the subscription for the transaction topic exists.

control "pubsub-test-2" do

        impact 1.0

        title "Test pub sub subscription exists for the transaction topic."

        describe google_pubsub_subscription(project: gcp_project_id, name: 'transaction-pull-subscription') do

          it { should exist }

        end

end
```

Figure 22 : Pub Sub Chef InSpec Tests

It failed before executing the *Terraform* scripts, but the tests were passed after running the script. Then before writing the *Java* code to publish and subscribe messages, *JUnit* tests are written for every function. Figure 23 shows the JUnit tests written. 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 [74].

```
        /*
         * Test publishing messages to pub sub topic.
         * In the real environment.
         */
        @Test
        public void testPublishingMessagesToPubSubTopic() throws IOException {
                String messageId = pubsub.publishAMessage(false, GCP_PROJECT_ID, PUBUB_TOPIC_ID,
message);
                assertFalse(messageId.isEmpty());
        }


        /*
         * Test subscribing messages from subscription.
         * In the real environment.
         */
        @Test
        public void testSubscribingMessagesFromPubSubPushSubscription() throws IOException {
                System.setOut(new PrintStream(outputStreamCaptor));

                pubsub.publishAMessage(false, GCP_PROJECT_ID, PUBUB_TOPIC_ID, message);
                pubsub.subscribeToPubSubSubscription(false, GCP_PROJECT_ID, PUBSUB_SUBSCRIPTION_ID);
                assertTrue(outputStreamCaptor.toString().contains("{\r\n"
                                + "    \"distance\": 82.8,\r\n"
                                + "    \"amt\": 9.16,\r\n"
                                + "    \"unix_time\": 1608428547,\r\n"
                                + "}"));

                System.setOut(standardOut);

        }


        /*
         * Test publishing messages to pub sub topic.
         * Using the emulator.
         */
        @Test
        //https://cloud.google.com/pubsub/docs/emulator#windows
        public void testPublishingMessagesToPubSubTopicUsingEmulator() throws IOException {
                String messageId = pubsub.publishAMessage(true, GCP_PROJECT_ID, PUBUB_TOPIC_ID,
message);
                assertFalse(messageId.isEmpty());
        }


        /*
         * Test subscribing messages from subscription.
         * Using emulator.
         */
        @Test
        public void testSubscribingMessagesFromPubSubPushSubscriptionUsingEmulator() throws
IOException {
                System.setOut(new PrintStream(outputStreamCaptor));

                pubsub.publishAMessage(true, GCP_PROJECT_ID, PUBUB_TOPIC_ID, message);
                pubsub.subscribeToPubSubSubscription(true, GCP_PROJECT_ID, PUBSUB_SUBSCRIPTION_ID);
                assertTrue(outputStreamCaptor.toString().contains("{\r\n"
                                + "    \"distance\": 82.8,\r\n"
                                + "    \"amt\": 9.16,\r\n"
                                + "    \"unix_time\": 1608428547,\r\n"
                                + "}"));

                System.setOut(standardOut);
        }
```

Figure 23 : Pub/Sub JUnit Tests

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 [83]. Therefore, *Dataflow* is used in the project, and the message received from the *Pub/Sub* is transferred to process it and take necessary actions. Figure 24 shows the streaming pipeline created using Dataflow.

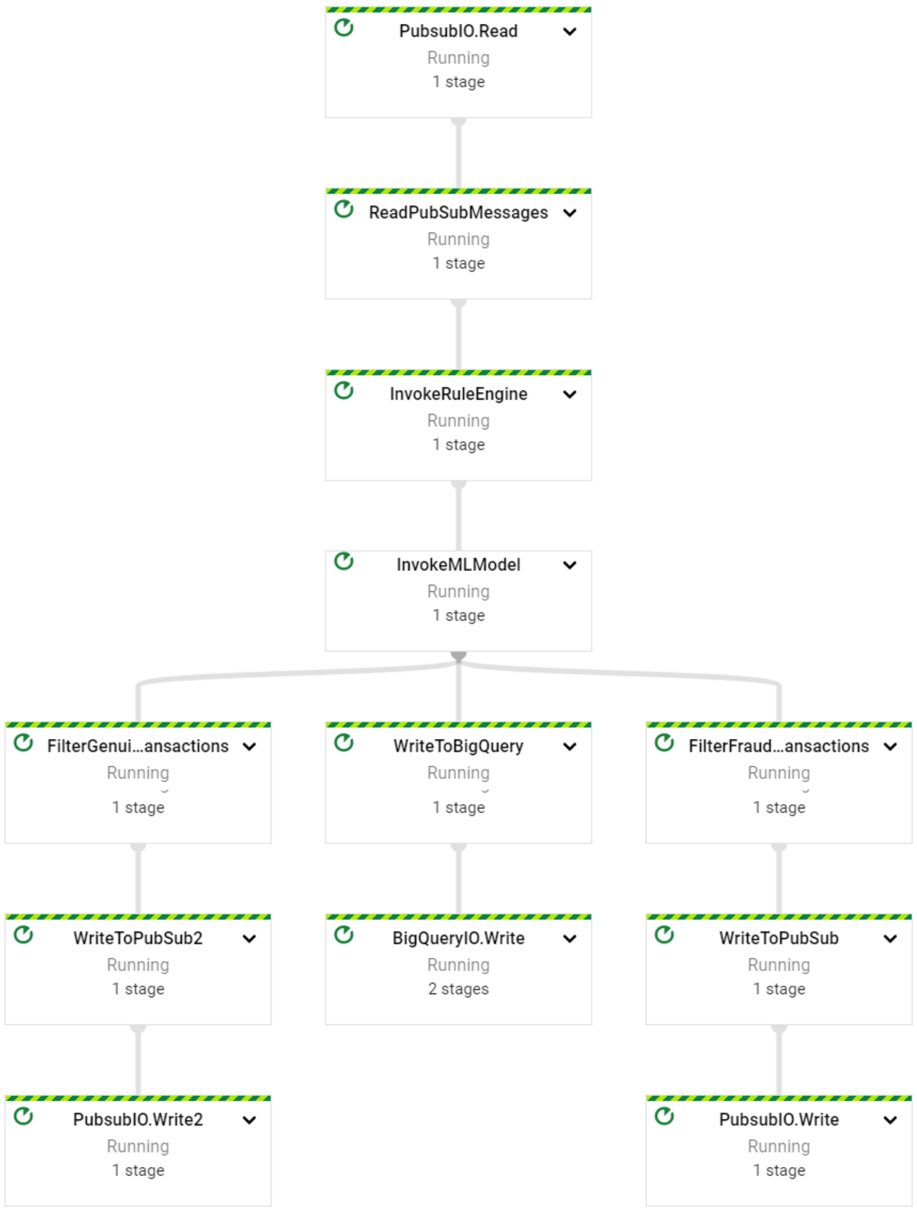


Figure 24 : Dataflow Pipeline

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.

The next step is to retrieve the transaction fraud status 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 machine learning model is invoked, 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 rule engine response are carried to the next step of the job.

The responses retrieved from both models are essential to identify the transaction as fraudulent and legitimate. Furthermore, it is essential 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 sent to the topic called *fraud-status-topic*. This topic has bounded with a cloud function that triggers an SMS when a new message reaches the topic. Then the platforms subscribed to this topic will get an SMS alert about the suspicious activity, and they can decide to abort the transaction without further processing.

In the filtering process, if the transaction is filtered as 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.

3.3.2.1 TDD in Dataflow

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. Therefore, 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. Therefore, transformations and integrations need thorough testing. *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 [84]. It has a runner called *DirectRunner*, which runs the pipeline locally on a small scale. In the project, *TestPipeline* is created, primarily used for testing transformers. All the data transformations are tested using *PAssert* statements, which can verify the content inside collections. Figure 25 shows two tests written to test Invoking rule engine and machine learning model and verify the outputs.

```
        /*
         * Test the transaction validity via InvokeRuleEngine PTransform.
         * Scenario 1 - transaction is valid
         */
        @Test
        public void testInvokingTheRuleEngineWhenTransactionIsValid() {
                Pipeline pipeline = TestPipeline.create().enableAbandonedNodeEnforcement(false);

                Transaction trans = new Transaction();
                trans.setAmount(9.16);
                trans.setCardNo("123456789");
                trans.setDateTime("2022-01-01 00:00:18");
                trans.setLatitude(82.8);
                trans.setLongitude(82.8);
                List<Transaction> transList = new ArrayList<>();
                transList.add(trans);

                PCollection<Transaction> transInput = pipeline.apply(Create.of(transList));
                PCollection<TransactionStatus> ruleEngineOutput = transInput.apply(new
InvokeRuleEngine());

                TransactionStatus ruleObj = new TransactionStatus();
                ruleObj.setAmount(9.16);
                ruleObj.setCc_number("123456789");
                ruleObj.setLatitude(82.8);
                ruleObj.setLongitude(82.8);
                ruleObj.setMl_fraud_status(0);
                ruleObj.setRule_fraud_status(1);
                ruleObj.setTrans_date_and_time("2022-01-01 00:00:18");

                PAssert.that(ruleEngineOutput).containsInAnyOrder(ruleObj);

                PipelineResult result = pipeline.run();
                System.out.println(result.getState());
                System.err.println(result.metrics());

        }
```

```java
/*
 * Test invoking ML model and transforming result when the prediction is not fraud.
 */
@Test
public void testInvokingMLModelWhenPredictionIsGenuine() {
        Pipeline pipeline = TestPipeline.create().enableAbandonedNodeEnforcement(false);

        TransactionStatus trans = new TransactionStatus();
        trans.setAmount(15.56);
        trans.setCc_number("30263540414123");
        trans.setLatitude(36.841266);
        trans.setLongitude(-111.69076499999998);
        trans.setRule_fraud_status(1);
        trans.setCategory("entertainment");
        trans.setTrans_date_and_time("2020-06-21 12:12:08");

        List<TransactionStatus> transList = new ArrayList<>();
        transList.add(trans);

        PCollection<TransactionStatus> transInput = pipeline.apply(Create.of(transList));
        PCollection<TransactionStatus> mlOutput = transInput.apply(new InvokeMLModel());

        TransactionStatus mlObj = new TransactionStatus();
        mlObj.setAmount(15.56);
        mlObj.setCc_number("30263540414123");
        mlObj.setLatitude(36.841266);
        mlObj.setLongitude(-111.69076499999998);
        mlObj.setMl_fraud_status(0);
        mlObj.setRule_fraud_status(1);
        mlObj.setCategory("entertainment");
        mlObj.setTrans_date_and_time("2020-06-21 12:12:08");

        PAssert.that(mlOutput).containsInAnyOrder(mlObj);

        PipelineResult result = pipeline.run();
        System.out.println(result.getState());
        System.err.println(result.metrics());

}
```

Figure 25 : Dataflow Rule Engine and ML Model Tests

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. It verifies that all the integrations are working without any issue. Figure 26 shows a test written to verify the entire pipeline locally.

```
        /*
         * Test pipeline locally.
         * Scenario 1 - when transaction is fraud.
         *
         * rule engine status = 0(fraud)
         * ml model status = 1(fraud)
         *
         * Filter fraud transactions list has an object.
         * Filter genuine transactions list is empty.
         */
        @Test
        public void testEndToEndPipelineScenario1() {

                messageWithCategory = "{\r\n"
                                + "    \"latitude\": 26.888686,\r\n"
                                + "     \"longitude\": -80.834389,\r\n"
                                + "    \"amount\": 977.01,\r\n"
                                + "    \"dateTime\": \"2020-06-21 01:00:08\",\r\n"
                                + "    \"category\": \"shopping_net\",\r\n"
                                + "     \"cardNo\" : \"3524574586339330\",\r\n"
                                + "     \r\n"
                                + "}";

                Pipeline pipeline = TestPipeline.create().enableAbandonedNodeEnforcement(false);

                byte[] data = messageWithCategory.getBytes();
                PubsubMessage pubsubMessage = new PubsubMessage(data, null);
                List<PubsubMessage> msgList = new ArrayList<>();
                msgList.add(pubsubMessage);

                PCollection<PubsubMessage> pubSubInput = pipeline.apply(Create.of(msgList));
                PCollection<Transaction> pubSubOutput = pubSubInput.apply(new ReadPubSubMessages());

                PCollection<TransactionStatus> ruleEngineOutput = pubSubOutput.apply(new
InvokeRuleEngine());
                PCollection<TransactionStatus> mlModelOutput = ruleEngineOutput.apply(new
InvokeMLModel());

                PCollection<TransactionStatus> filteredFraudTransactions = mlModelOutput.apply(new
FilterFraudTransactions());

                PCollection<TransactionStatus> filteredGenuineTransactions = mlModelOutput.apply(new
FilterGenuineTransactions());

                TransactionStatus trans = new TransactionStatus();
                trans.setAmount(977.01);
                trans.setCc_number("3524574586339330");
                trans.setLatitude(26.888686);
                trans.setLongitude(-80.834389);
                trans.setRule_fraud_status(0);
                trans.setCategory("shopping_net");
                trans.setTrans_date_and_time("2020-06-21 01:00:08");
                trans.setMl_fraud_status(1);

                PAssert.that(filteredFraudTransactions).containsInAnyOrder(trans);
                PAssert.that(filteredGenuineTransactions).empty();

                PipelineResult result = pipeline.run();
                System.out.println(result.getState());
                System.err.println(result.metrics());

        }
```

Figure 26 : Dataflow End to End Tests

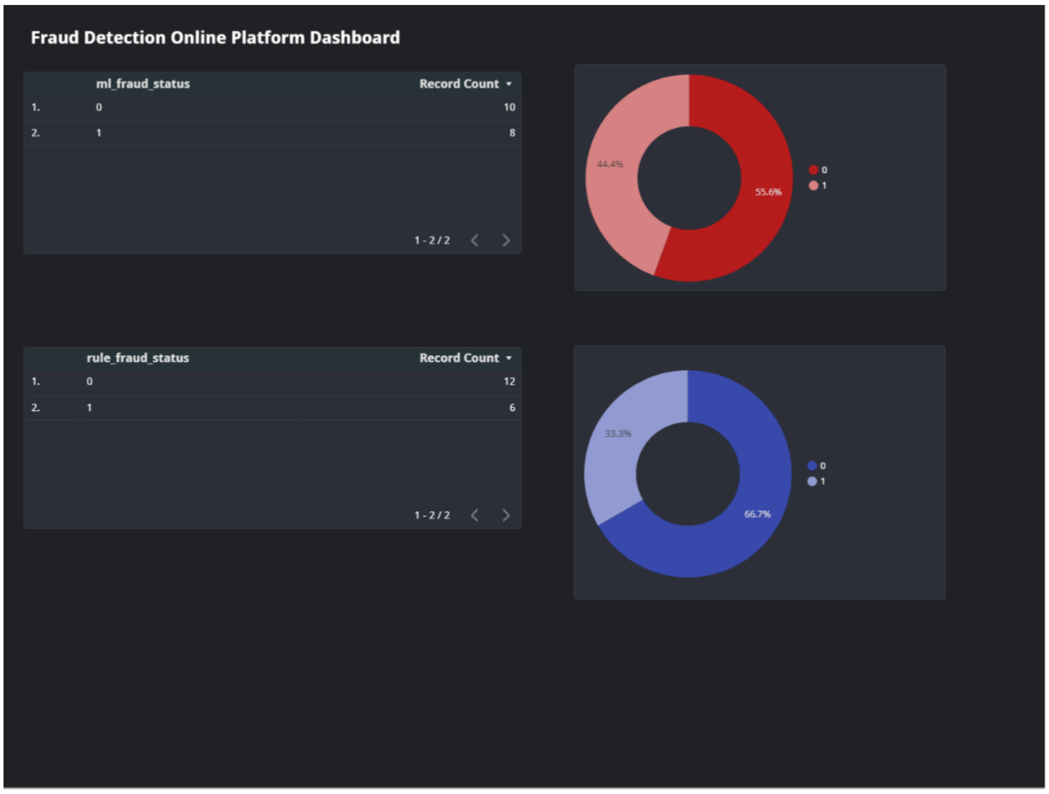
All these above tests are executed locally, and it is necessary to test the entire pipeline in the actual environment. Therefore finally, a test has been written to execute the entire pipeline in the actual GCP project and verify the successful execution via asserting the pipeline result state. Figure 27 shows the test written to verify the pipeline status in the actual production environment.

```
        /*
         * Test pipeline in actual environment.
         */
        @Test
        public void testPipeline() {
                GCPDataflowIntegration dataflow = new GCPDataflowIntegration();
                Pipeline pipeline = dataflow.getPipeline(false, GCP_PROJECT_ID, GCP_PROJECT_REGION);
                PipelineResult result = dataflow.createAndRunPipeline(pipeline, GCP_PROJECT_ID,
TRANSACTION_TOPIC, DATASET_ID, TABLE_ID, FRAUD_STATUS_TOPIC, TRANSACTION_STATUS_TOPIC);
                assertEquals("RUNNING", result.getState().toString());
        }
```

Figure 27 : - Dataflow Production Environment Tests

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 [17]. 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, and Figure 28 shows how it can be used in *Data Studio* to generate graphs. The graphs below depict the true and false response counts received when invoking the machine learning model and rule engine separately hosted in the cloud.



Fraud Detection Online Platform Dashboard

	ml_fraud_status	Record Count ▾
1.	0	10
2.	1	8

1 - 2 / 2

	rule_fraud_status	Record Count ▾
1.	0	12
2.	1	6

1 - 2 / 2

Figure 28 : Fraud Detection Dashboard

3.3.3.1 TDD in Big Query

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. Therefore, 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 assuring 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 below properties.

- BigQueryIO.Write.CreateDisposition.CREATE_NEVER
  - Specifics that tables should not be created
- BigQueryIO.Write.WriteDisposition.WRITE_APPEND
  - Specifics that rows should be appended to an existing table

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. Figure 29 shows a test written for the *BigQuery* table.

```
        /*
         * Test CreateDisposition.CREATE_NEVER during BigQueryIO.Write and there is
         * already a table in Big Query
         */
        @Test
        public void testBigQueryWriteWhenCreateDispositionIsCreateNeverAndTableIsPresent()
                        throws InterruptedException, IOException {
                table = new Table();
                tableRef = new TableReference();
                tableRef.setDatasetId("dataset-id");
                tableRef.setProjectId("project-id");
                tableRef.setTableId("table-id");
                table.setTableReference(tableRef);

                table.setSchema(
                                new TableSchema().setFields(ImmutableList.of(new
TableFieldSchema().setName("name").setType("STRING"),
                                                new
TableFieldSchema().setName("number").setType("INTEGER"))));
                fakeDatasetService.createTable(table);


                TableRow row1 = new TableRow().set("name", "a").set("number", 1);
                TableRow row2 = new TableRow().set("name", "b").set("number", 2);
                TableRow row3 = new TableRow().set("name", "c").set("number", 3);

                List<TableRow> rows = new ArrayList<TableRow>();
                rows.add(row1);
                rows.add(row2);
                rows.add(row3);

                GCPBigQueryIntegration bqIntegration = new GCPBigQueryIntegration();
                WriteResult result = bqIntegration.writeRowsToBigQuery(true, pipeline, rows,
DATASET_ID, TABLE_ID, fakeBqServices);

                PAssert.that(result.getSuccessfulTableLoads())
                                .containsInAnyOrder(new TableDestination("project-id:dataset-
id.table-id", null));

                pipeline.run();

                fakeDatasetService.deleteTable(tableRef);
        }
```

Figure 29 : BigQuery Test

3.3.4 Summary

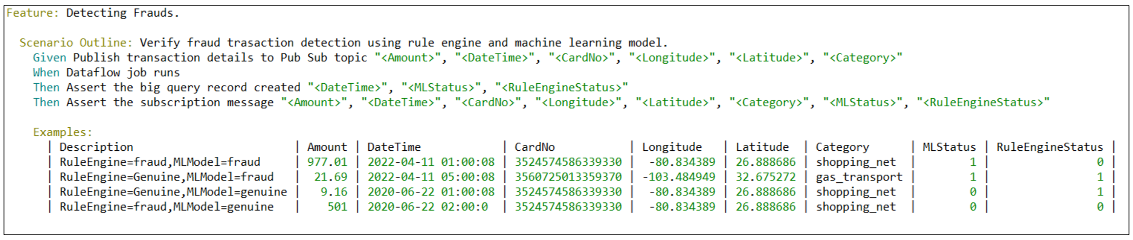
Table 8 is a summary of all the technologies, tools, frameworks used for each microservice and the system.

Microservice	Step	Technologies/Tools/ Libraries/ Frameworks for Implementation	Technologies/Tools/ Libraries/ Frameworks for Testing
Machine Learning Model Creation	Pre-processing	PyDev IDE Sklearn library Pandas library	unittest library InSpec GCP resource

		Numpy library Terraform scripts	pack
	Build machine learning model	Google-cloud-automl_v1beta1 library	unittest library
	Host machine learning model	Google-cloud-automl_v1beta1 library	unittest library
Rule Engine	Implement the rule engine	Spring boot	JUnit Jupiter library, spring-restdocs-mockmvc libraries
	Deploy the rule engine	Terraform	Chef InSpec resource pack
Online Fraud Detection Platform	Pub/Sub	Terraform scripts Pub sub Google-cloud-pubsub library	Chef InSpec resource pack JUnit library Pub-Sub emulator
	Dataflow	Apache beam SDK Java libraries	DirectRunner TestPipeline JUnit library
	Bigquey Table	Google-cloud-bigquery Java library	Beam-runners-direct-java library
Entire System			Cucumber framework

Table 8 : Microservice Summary

After conducting unit, component, and integration testing, system tests that test the entire system functions successfully as a single unit without any failures should be performed. In the project to create system tests and execute them, *Cucumber* is used, which is an open-source behaviour-driven development framework. *Cucumber* allows us to write the tests in a plain English format called *Gherkin*, and the actual code can be written in multiple languages. The tests written in plain English, called *Feature* files, consist of all the test scenarios. In the project, tests are written in *Java* and executed in Eclipse IDE after installing *Cucumber* plugins that allow easy test execution. Here, it is used as *Cucumber* tests are easy to understand, easy to maintain, and easy to execute the same scenario multiple times with different cases. Figure 30 shows the feature file written in the project. Here four test cases are written to verify that fraud detection is accomplished, and the results are written to the *BigQuery* table. Also, it verifies that final transaction status details are published to the correct *Pub/Sub* notification channel.



```
Feature: Detecting Frauds.

  Scenario Outline: Verify fraud trasaction detection using rule engine and machine learning model.
    Given Publish transaction details to Pub Sub topic "<Amount>", "<DateTime>", "<CardNo>", "<Longitude>", "<Latitude>", "<Category>"
    When Dataflow job runs
    Then Assert the big query record created "<DateTime>", "<MLStatus>", "<RuleEngineStatus>"
    Then Assert the subscription message "<Amount>", "<DateTime>", "<CardNo>", "<Longitude>", "<Latitude>", "<Category>", "<MLStatus>", "<RuleEngineStatus>"

    Examples:
      | Description                      | Amount | DateTime            | CardNo           | Longitude   | Latitude  | Category      | MLStatus | RuleEngineStatus |
      | RuleEngine=fraud,MLModel=fraud     | 977.01 | 2022-04-11 01:00:08 | 3524574586339330 |  -80.834389 | 26.888686 | shopping_net  |        1 |                0 |
      | RuleEngine=Genuine,MLModel=fraud   |  21.69 | 2022-04-11 05:00:08 | 3560725013359370 | -103.484949 | 32.675272 | gas_transport |        1 |                1 |
      | RuleEngine=Genuine,MLModel=genuine |   9.16 | 2020-06-22 01:00:08 | 3524574586339330 |  -80.834389 | 26.888686 | shopping_net  |        0 |                1 |
      | RuleEngine=fraud,MLModel=genuine   |    501 | 2020-06-22 02:00:0  | 3524574586339330 |  -80.834389 | 26.888686 | shopping_net  |        0 |                0 |
```

Figure 30 : Cucumber Feature File

4 Evaluation

In the thesis, the project is implemented only using test-driven development. Therefore, a comparison cannot be conducted, and the best possible way to conduct an evaluation is by testing coverage and measuring the code quality. Accordingly, the project is evaluated using the below criteria.

- The functionalities covered by the tests.
- The number of test cases written.
- The code quality aspects like overall quality, reliability, maintainability, code smell, and code coverage.
- The testing types covered in the project.

4.1 Test Cases

The test case is a collection of steps to verify the program's actual output against the expected output. A test case contains a set of inputs and expected outputs, and it is the backbone of the software testing process [85]. The test case is a fundamental concept in software testing, and in coding, tests are written to verify and validate the code automatically. In the project, 100+ test cases are written using the TDD approach, and Table 9 shows the test cases for each microservice. The foundation for writing these test cases is explained below with their respective project.

Microservice	Test Cases
Machine Learning Model	**Pre-processing test cases.** • Test reading an empty file. • Test reading a non-empty file. • Test target column has missing values. • Test removing missing values from the target column. • Test columns with high missing values. • Test dropping columns. • Test finding duplicate rows. • Test removing duplicate rows. • Test finding columns with a single value. • Test removing columns with a single value. • Test finding columns with categorical data. • Test transforming columns with categorical data to numerical. • Test imputing missing values. • Test removing outliers. • Test finding correlations with the target. • Test writing data frame to CSV.

	Model creation test cases. • Test creating Auto ML table client connection. • Test creating a dataset in AutoMl. • Test importing data to the dataset in AutoMl. • Test creating model in AutoMl. • Test retrieving operational status. • Test retrieving model deployment status. • Test retrieving model evaluations. • Test deploying AutoML model. • Test retrieving model deployment status after deploying the model. • Test retrieving online model prediction. • Test un deploying AutoML model. • Test retrieving model deployment status after un deploying the model. **Infrastructure test cases.** • Verify that the storage bucket exists. • Verify that the dataset exists.
Rule Engine	**Infrastructure test cases.** • Check the Spring context is creating the controller. • Test that the injected components are not null. **Controller test cases.** • Check that the endpoint returns success and true. • Check that the endpoint returns false. **Service test cases.** • Test if the transaction amount is valid when the amount is lesser than the threshold. • Test if the transaction amount is valid when the amount is greater than the threshold. • Test if the transaction amount is valid when the amount is equal to the threshold. • Test if the total transaction amount exceeds the daily limit when the total transaction amount in the DB is lesser than the threshold. • Test if the total transaction amount exceeds the daily limit when the total transaction amount in the DB is null. • Test if the total transaction amount exceeds the daily limit when the total transaction amount in the DB is equal to the threshold. • Test if the daily transaction count exceeds when the count in the DB is lesser than the threshold. • Test if the daily transaction count exceeds when the count in the DB is lesser than the threshold.

- Test if the daily transaction count exceeds when the count in the DB exceeds the threshold.
- Test if the subsequent transaction time difference is less than 5 seconds when there are no transactions in the DB.
- Test if the subsequent transaction time difference is less than 5 seconds when there are transactions.
- Test if the subsequent transaction time difference is less than 5 seconds when there are transactions.
- Test calculating the distance in kilometers for longitude and latitude.
- Test calculating the distance in kilometers when longitude and latitude are the same.
- Test subsequent transaction locations are inside the distance threshold when there are no today transactions.
- Test subsequent transaction locations are inside the distance threshold when there are transactions.
- Test subsequent transaction locations are inside the distance threshold when there are transactions.
- Test transaction is valid when all the rules are matched.
- Test transaction is valid when one rule is mismatched. (amount > 500)
- Test transaction is valid when one rule is mismatched. (daily limit > 1000)
- Test transaction is valid when one rule is mismatched. (daily transaction count > 5)
- Test transaction is valid when one rule is mismatched. (subsequent transaction time < 5s)
- Test transaction is valid when one rule is mismatched. (subsequent transaction distance > 1000km)

Repository test cases.

- Test the empty repository.
- Test saving the transaction to the database.
- Find today's transactions' total amount when there are no today transactions.
- Find today's transactions' total amount when there are only today's transactions.
- Find today's transactions total amount when there are multiple days of transactions.
- Find the number of transactions that happened today when there are no today transactions.
- Find the number of transactions that happened today when there are only today's transactions.
- Find the number of transactions that happened today when there are multiple days of transactions.
- Find the transaction time of today’s last transaction when there are no transactions in the database.
- Find the transaction time of today’s last transaction when there are only today’s transactions in the database.

	• Find the transaction time of today’s last transaction when there are multiple transactions in the database. • Find the longitude and latitude of today’s last transaction when there are no transactions in the database. • Find the longitude and latitude of today’s last transaction when there are only today’s transactions in the database. • Find the longitude and latitude of today’s last transaction when there are multiple days of transactions in the database. **Infrastructure test cases.** • Verify that the rule engine virtual machine exists.
Pub-Sub	• Test publishing messages to the topic. • Test subscribing messages from the subscription. • Test publishing messages to the topic using the emulator. • Test subscribing messages from subscription using emulator.
Dataflow	• Test creating the pipeline. **Data transforming test cases.** • Test transforming pub sub-message to transaction object. • Test the transaction validity via InvokeRuleEngine PTransform. When the transaction is valid • Test the transaction validity via InvokeRuleEngine PTransform. When the transaction is not valid. • Test invoking ML model and transforming result when the prediction is not a fraud. • Test invoking ML model and transforming result when the prediction is fraud. • Test transforming TransactionStatus object to Big Query table row. • Test transforming TransactionStatus object to JSON. • Test filtering fraud transactions. When rule engine status = 1(genuine), ml model status = 1(fraud) • Test filtering fraud transactions. When rule engine status = 1(genuine), ml model status = 0(genuine) • Test filtering fraud transactions. When rule engine status = 0(fraud), ml model status = 1(fraud) • Test filtering fraud transactions. When rule engine status = 0(fraud), ml model status = 0(genuine) • Test filtering genuine transactions. When rule engine status = 1(genuine), ml model status = 1(fraud) • Test filtering genuine transactions. When rule engine status = 1(genuine), ml model status = 0(genuine) • Test filtering genuine transactions. When rule engine status = 0(fraud), ml model status = 1(fraud) • Test filtering genuine transactions. When rule engine status =

	0(fraud), ml model status = 0(genuine) **Pipeline test cases.** • Test pipeline in the actual environment. • Test pipeline locally. When the transaction is fraud. rule engine status = 0(fraud), ml model status = 1(fraud) • Test pipeline locally. When the transaction is fraud. rule engine status = 0(fraud), ml model status = 0(genuine) • Test pipeline locally. When the transaction is genuine. rule engine status = 1(genuine), ml model status = 0(genuine) • Test pipeline locally. When the transaction is fraud. rule engine status = 1(genuine), ml model status = 1(fraud)
Big Query Table	• Test the exception when using CreateDisposition.CREATE_NEVER during BigQueryIO.Write and there is no table in the Big Query • Test CreateDisposition.CREATE_NEVER during BigQueryIO.Write and there is already a table in Big Query • Test CreateDisposition.WRITE_APPEND during BigQueryIO.Write and there is already a table in Big Query **Infrastructure test cases.** • Verify fraud detection dataset exists. • Verify fraud detection dataset name. • Verify fraud detection table exists. • Verify transaction fraud data table exists.
Online fraud detection platform.	• Verify fraud transaction detection using the rule engine and machine learning model. When RuleEngine=fraud,MLModel=fraud. • Verify fraud transaction detection using the rule engine and machine learning model. When RuleEngine=Genuine,MLModel=fraud. • Verify fraud transaction detection using the rule engine and machine learning model. When RuleEngine=Genuine,MLModel=genuine. • Verify fraud transaction detection using the rule engine and machine learning model. When RuleEngine=fraud,MLModel=genuine.

Table 9 : Test Cases Table

4.1.1 Machine Learning Model

Test everything is one of the best practices in machine learning. Tests should be written to verify data quality and completeness, data errors and consistency, feature quality, code error, model quality, model scalability, and feature consistency [2]. Furthermore, Kirk [86] talks about the risks in machine learning applications and ways to mitigate using TDD. The four risks associated with the machine learning applications are unstable data, underfitting, overfitting, and unpredictable future, which can be reduced through seam testing, cross-validation, benchmark testing, and tracking precision-recall described in Table 10.

Testing Type	Description
Seam Testing	This is a testing concept primarily introduced when interacting with legacy code. For example, seams are integration points and in legacy code. In a legacy application, the internal structure of the code is unknown, yet a prediction can be made of what will be the given output for specific inputs. This is black box testing and can be used for machine learning applications as they behave similarly to legacy applications.
Cross Validation	Splitting data into training and validation sets is called cross-validation. Then the training data set is used to build the machine learning model, and the validation set to validate that the model is functioning as expected.
Benchmark Testing	The simplicity of a machine learning model is expected, and it is essential to avoid overfitting, which is memorizing data. Training time is one of the measures that can be used to measure the simplicity of a model, and there a benchmark value can be set to identify model training is getting faster or slower. Setting a benchmark value and testing against that is called benchmark testing.
Tracking Precision and Recall	Precision is the true positive rate, and it is a measure of machine learning model quality. The recall is the proportion of true positives to true positives and false negatives.

Table 10 : Testing Types in Machine Learning

In this project, some of the tests like overfitting and underfitting are not covered due to technical limitations, but most of the tests mentioned above are covered as below. data quality and completeness - using tests for missing values, using tests to find outliers

- data errors and consistency, seam testing - using tests for duplicate rows, using tests to find categorical values, using tests to validate predictions for known inputs
- feature quality - using tests to find single value columns, using tests to find correlations with the target
- model quality, unpredictable future - using tests to evaluate model quality

4.1.2 Online Fraud Detection Model

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 [87].

4.1.3 Rule Engine

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.

Figure 31 shows the number of tests written for each microservice. One hundred twenty-eight tests are written, and it is evident that tests are equally distributed among the three main components.

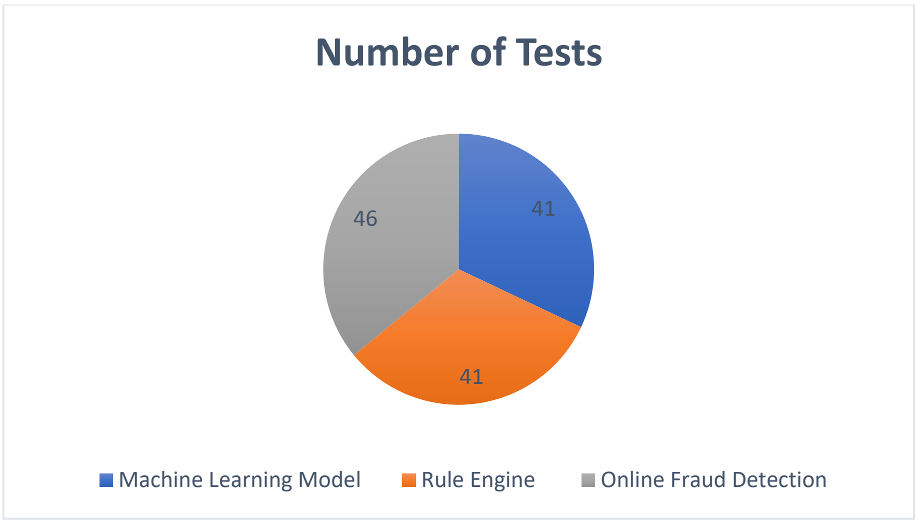


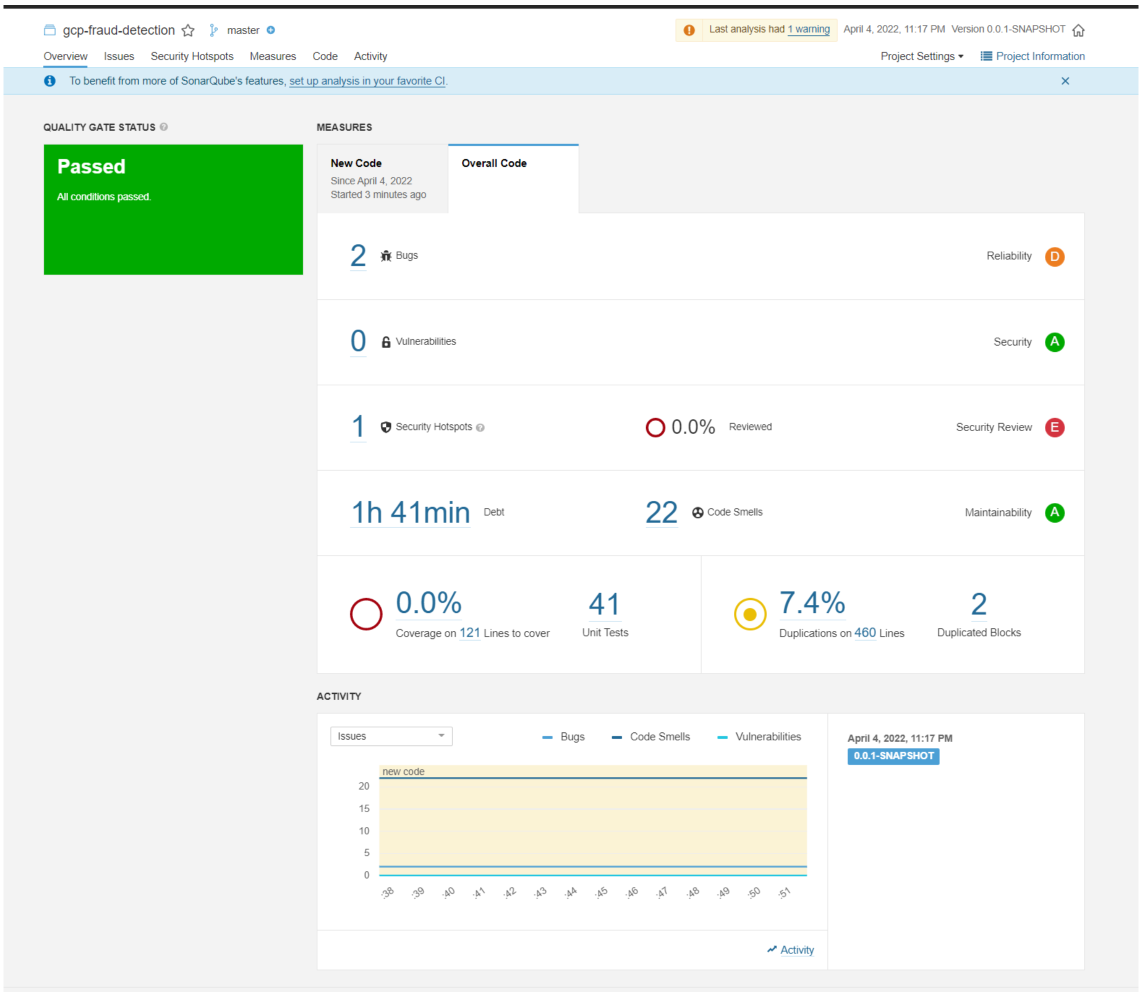
Figure 31 : Number of Tests

4.2 Code Quality

Code quality is finding whether the code is good or bad. There are various metrics in code quality, and different projects use different metrics based on their context. For example, in TDD, overall quality, maintainability [59][63], number of bugs [24][27], reliability, and code coverage by tests [49][59] 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. *SonarQube* has a free community version, and it can be installed locally to inspect the local project's code. It supports analysing both *Java* and *Python* source code used in the project. Therefore, it was well suited for this context. Furthermore, *SonarQube* has thousands of static code analysis tools supporting code quality and code security[88] and uses metrics like 1) complexity – complexity(cyclomatic complexity), cognitive complexity, 2) duplications – duplicated blocks, duplicated lines, 3)issues – new issues, issues, 3) maintainability – code smells, maintainability rating(technical debt ration), technical debt, 4) quality gate – quality gate status, quality gate details, 5) reliability – bugs, reliability rating, 6) security– vulnerabilities, security rating, 7) size – classes, comments, functions, 8) tests – condition coverage, line coverage. Following are the analysis retrieved from the tool. They are the initial code quality of the projects without fixing any issues highlighted in the analysis.

4.2.1 Rule Engine

According to the analysis, QUALITY GATE STATUS=PASSED assures that the rule engine is production ready. Furthermore, the analysis report reveals that the maintainability and security of the project are at the highest level. In the report, two issues are visible related to asserting dissimilar types. However, they 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. However, 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. Figure 32 shows the analysis report obtained for the project.



gcp-fraud-detection
master
Last analysis had 1 warning
April 4, 2022, 11:17 PM Version 0.0.1-SNAPSHOT
Overview
Issues
Security Hotspots
Measures
Code
Activity
Project Settings
Project Information
To benefit from more of SonarQube's features, set up analysis in your favorite CI.

QUALITY GATE STATUS

Passed

All conditions passed.

MEASURES

New Code: Since April 4, 2022 — Started 3 minutes ago

Overall Code

2 Bugs — Reliability D

0 Vulnerabilities — Security A

1 Security Hotspots — 0.0% Reviewed — Security Review E

1h 41min Debt — 22 Code Smells — Maintainability A

0.0% Coverage on 121 Lines to cover — 41 Unit Tests

7.4% Duplications on 460 Lines — 2 Duplicated Blocks

ACTIVITY

Figure 32 : Rule Engine Code Analysis

4.2.2 Machine Learning Model

The machine learning model is created using Python, and SonarQube does not directly support the code coverage and analysis. Therefore, the *Coverage Tool* is incorporated to generate the code coverage as suggested by the Sonar community [88]. Code *Coverage Tool* generates a coverage XML which records the code coverage by tests, and then it was 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.

Furthermore, reliability, security, and maintainability rating is the highest level at level A according to the analysis. The analysis report shows code coverage as 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 shows as 485 in the report. When excluding the files that tests are not essential, code coverage increased to 78.9%, which is good coverage at the first attempt. Also, it is very close to the code coverage rating A level which is 80%. Figure 33 depicts the analysis report of the project.

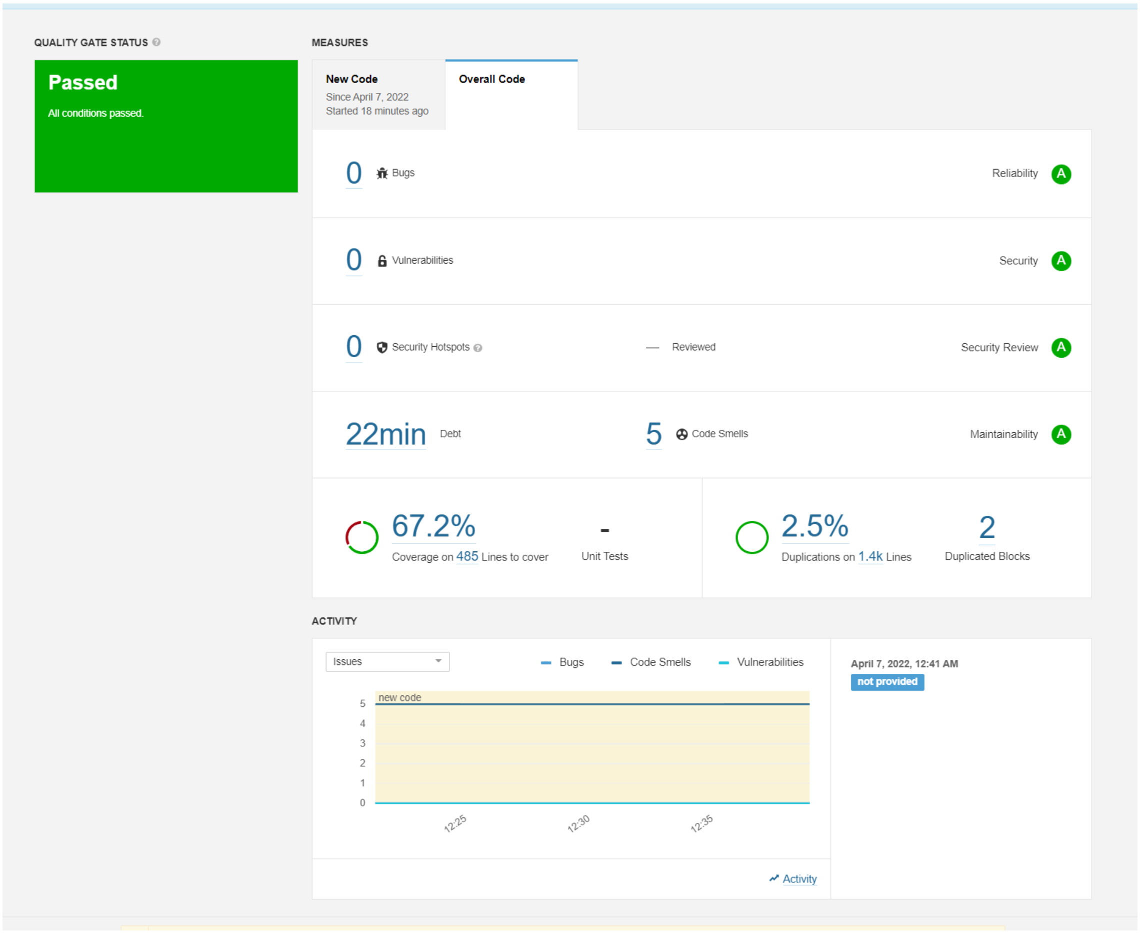
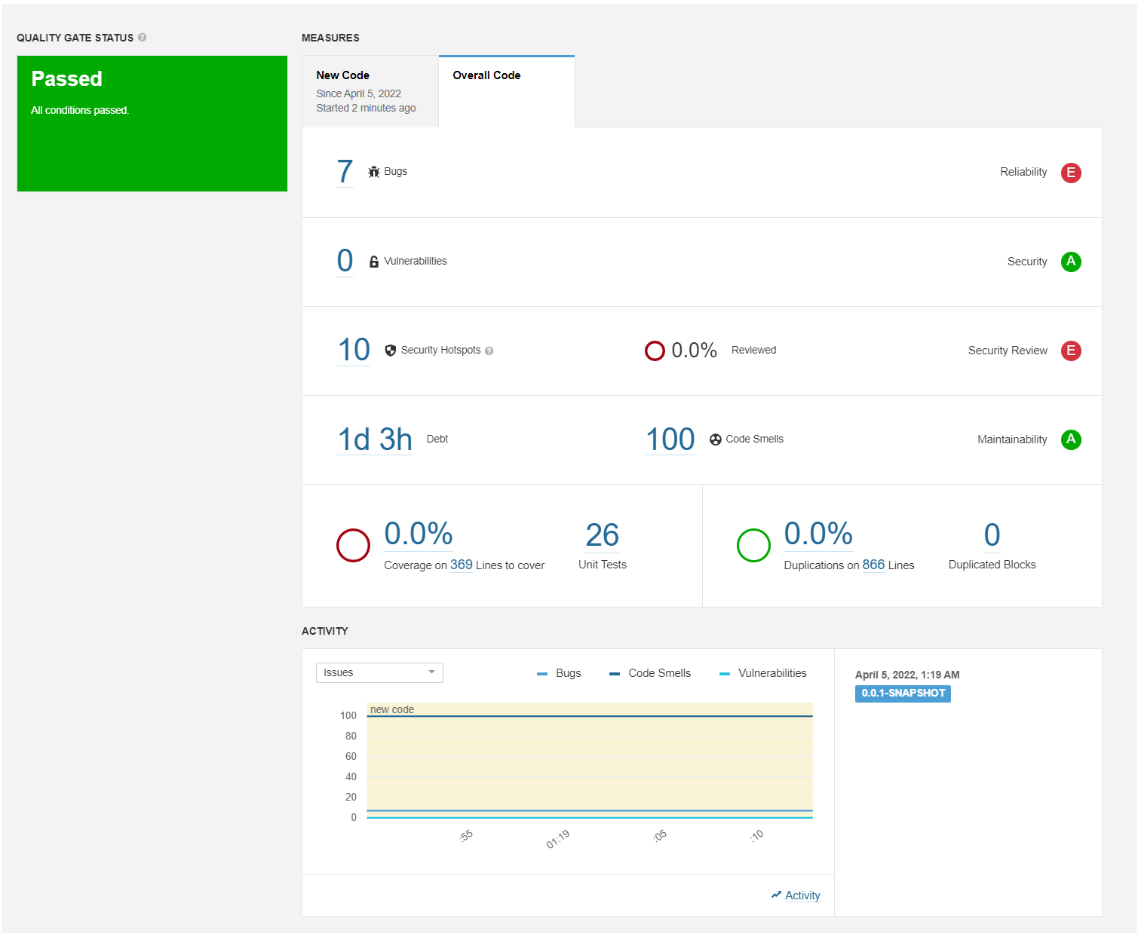


Figure 33 : ML Model Code Analysis

4.2.3 Online Fraud Detection Platform

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. Code coverage tool measures the security aspects, and the analysis shows ten security hotspots. When checking the detailed analysis, all of them are generated because of *the 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 counted picked up by the coverage tool. Three hundred sixty-nine lines need to be covered in the code coverage, but when checked, those lines related to *PTransform* and *DoFn* classes are covered via *TestPipeline* tests. They are not identified via the code coverage tools due to how it is handled in *Apache Beam SDK*. Therefore, considering the above fact, the precise lines to be covered are less than 100, which can be achieved quickly. Figure 34 shows the analysis report of the online platform.



QUALITY GATE STATUS

Passed

All conditions passed.

MEASURES

New Code
Since April 5, 2022
Started 2 minutes ago

Overall Code

7 Bugs — Reliability E

0 Vulnerabilities — Security A

10 Security Hotspots — 0.0% Reviewed — Security Review E

1d 3h Debt — 100 Code Smells — Maintainability A

0.0% Coverage on 369 Lines to cover — 26 Unit Tests

0.0% Duplications on 866 Lines — 0 Duplicated Blocks

ACTIVITY

Issues

Bugs — Code Smells — Vulnerabilities

April 5, 2022, 1:19 AM
0.0.1-SNAPSHOT

Activity

Figure 34 : Fraud Detection Platform Code Analysis

4.3 Best Practices

The main research question addressed in this thesis is best practices in applying TDD. Unfortunately, TDD does not provide hard-coded rules or strict guidelines on its application. However, in the literature review, multiple best practices are identified, derived from the experience of applying TDD in academic or industrial projects. Following are the best practices discovered in the literature [40][41][52][54][55], and they are divided into three main categories people, software, and process. First, the best practices related to the development team and supervisors are categorized as people, and better human resource handling is vital for a successful project. Second, best practices related to tools, frameworks used in TDD applications, and code implementation guidelines are categorized as software. Third, TDD is a process, and finally, essential aspects of the process are related to the process.

4.3.1 People

Awareness of TDD benefits for the management [41], team discussions about TDD benefits and implementations [55] and having a TDD promoting team leader [41] are not applicable in this context as a single person develops this. However, the first point is that gaining the required skill set to adopt TDD and continue TDD is essential for the project's success [40][41]. Most of the projects are developed using the traditional test last approach, and all the academic and industrial projects I have been involved in used the last test approach. Therefore, adopting TDD was difficult because of not having prior experience, necessary skills, and not having the correct mindset. So before starting the project lot of articles were read, and small sample codes were written to practice TDD: Therefore, it is essential to encourage developers and provide necessary resources like training and materials to learn to apply TDD. Furthermore, necessary skills should be gained before starting the project, as adopting TDD from the beginning is required.

4.3.2 Software

Readable and meaningful names improve the readability, understandability, and maintainability of functional code [41]. Big data applications are generally complex and communicate with multiple microservices to fulfil the functionalities. For example, the streaming dataflow pipeline integrates four microservices to identify fraudulent transactions in the fraud detection application. Furthermore, they use different communication mechanisms like notifications, RESTful services, and events, making the application complex. So, it is essential to read and understand the code when an issue arises or when doing a feature modification or improvement to do the modification quickly and accurately. Here, having readable and meaningful names in functions can make the life of a developer easy. For example, all the tests and functional methods in the fraud detection application are given readable meaningful names. Also, method comments for all the methods describing the test logic or actual business logic are added. So, it improves the maintainability of the projects as the methods give a clear explanation of the implementation. Keeping production code simple [54] is another best practice derived from the literature. In this project, both production code and tests are kept as simple as possible and small as possible. Having simple small methods increases the readability and understandability and decreases the bugs created because of the less complex testable units.

Using OOP concepts and design patterns is another best practice mentioned in multiple papers [41][52], producing high-quality code with low coupling and high cohesion. In this project, OOP concepts and design patterns are used when needed. For example, when implementing the rule engine, the controller service repository pattern is used as having different layers helps to handle relevant functionality in relevant layers. Also, it improves the code's maintenance, decreases the duplicates, and decreases the programming errors since relevant functions are centralized. Furthermore, OOP concepts *abstraction*, *encapsulation*, *inheritance*, and *polymorphism* are used in rule engine and online fraud detection application projects. Those two projects were developed using java and spring boot, making it easier to apply OOP concepts. It eventually helped for the better maintainability and high-reliability code.

Using automation tools and code coverage tools are the other two best practices that are mentioned in multiple papers [41][54][55], which improve the productivity of the projects as well as the reliability of the project by decreasing bugs. The *Cucumber* automation framework was used to write and execute system tests in the project. The *Cucumber* framework is an efficient testing tool that focuses on the end-users. It helps all stakeholders in the project understand the business logic by using plain English *Gherkin* feature files. Furthermore, it increases the reusability of the test methods because of the way the codes are written, and also, the setup and execution of tests are straightforward. It is also helpful when running the same test with different inputs as it uses *Scenario Outline* and *Example* pattern in the feature files. Code coverage tools are essential to identify how much code has been covered by tests. The code coverage increases the system's reliability as most of the codes are tested. *SonarQube* is used to evaluate the project, and the scanner provided the code coverage results for the two java projects. A separate tool called *Coverage Tool* is used to measure the coverage of the *PHP* project. Both tools helped identifying the code coverage and the project's reliability, maintainability, and security vulnerabilities, as the *SonarQube* analysis had covered all the quality aspects of the project.

4.3.3 Process

A good process will help a project to succeed. Several papers [41][40][52][54][55] have listed process-related best practices like creating a proper test plan at the beginning, starting TDD at the beginning of the project, and fully adopting TDD in the project and follow TDD practices strictly. All three best practices are followed in this project, and TDD was started initially. Furthermore, TDD was adopted in all the microservices, and the TDD process was followed rigidly. First, the test list is created for each microservice, then the tests are written, and then the actual implementation code is written. During the review, multiple best practices are discovered. They are 1) preparing a test list for each feature[54], 2) writing tests to validate end-user requirements[52], 3) group tests according to fixtures [52], 4) spending sufficient time in creating tests as those are equally important as the production code[52], 5) write small and simple tests, mainly with a single assertion[52][54], 6) watch for test features and work on a single test at a time[54].

According to the suggestion, a test list is created for each microservice in the project. The test list covered all the test scenarios that should cover in tests. It helped cover all the essential test scenarios without forgetting, and it acts as the blueprint of the component of how the system should behave internally and externally. Also, system tests are written using *Cucumber* to validate end-user requirements, as suggested above. Furthermore, small, simple tests contain a single assertion because it is easy to identify the testable logic, easy to fix failures, and easy to refactor. Equal time is spent on writing tests and functional code, and tests are grouped according to the feature. In the literature, it is suggested to group tests based on the fixture. However, here there are no different testing environments. Therefore, tests are grouped by feature. It helped to identify all feature tests quickly. Then, only those feature tests can be executed without running all the tests when a test failure occurs. In the literature, it is suggested to run all the tests after fixing a single test failure [54], but this is not suitable for applications like big data applications because of the high costs. Big data applications are complex application that connects multiple microservices. Also, mostly, they are communicating with third-party APIs. So, if all the tests are running multiple times because of a single test failure, it will waste time and resources. In this project, all the microservices are hosted in the GCP environment. Most tests are executed in the actual environment because they do not have separate testing environments. In the GCP environment, costs are calculated based on the requests made, and tests have also incurred a cost. Therefore, to minimize the cost, it should not execute all the tests for a single test failure but only the tests related to that feature when an issue occurs in a feature test.

Tests and actual business logic are equally important in a big data application. There are some derived best practices for production code. First, the production code should be refactored [54] for better performance, readability, or maintainability, but it should be done when all the tests are green. In this project, tests are written then the production code is implemented. Once the test is passed, the production code is refactored if necessary. Then again, the test is executed to assure that the test is not failing. Also, it is essential to refactor the test code [52] for better quality, and test execution time should be considered [54] when running the tests. In this project, all the services are hosted in the GCP environment, and multiple microservices communicate with each other to get the final results. Therefore, taking a few seconds to run a test was not inevitable. Finally, TDD benefits should be measured for better visibility [41] using tools that have been done using *SonarQube* in the project.

Apart from the best practices found in the literature, there are practices that we can follow in developing TDD applications experienced in the project. When selecting services to develop big data applications, attention should be given to choosing the service that supports TDD. In this project, different services like *BigQuery ML* and *Dataprep* are tried but later dropped as they are not supporting TDD because they do not have a well-defined API, or the process that it follows is not suitable for TDD. Therefore, for better application of TDD, research should be carried out to find and evaluate relevant tools, frameworks, or services and then select the best appropriate tools for the development. After selecting the services, it is essential to have a detailed application design, like the microservices in the applications and the communication flow between them. It will help to identify the test scenarios quickly. Mainly it helps write component tests, integration tests, and system tests. Then a test list should be created for each service so that the necessary tests are not forgotten. Also, most of the tests in the project are executed in the real production environment as the GCP services do not offer well-defined test environments. Therefore, benchmarked datasets should be used to ensure the business logic is correct and to run the tests with less cost and less time. Here a benchmarked dataset is used to test the functionalities of the *AutoML* Table because it minimizes the costs and the execution time. In this project, the fraud detection domain is used, and fraud detection accuracy should be determined at the highest level. Otherwise, the monetary and reputation loss will be high, and therefore, sufficient tests should be added to ensure the fraud identification-related business logic correctness. A rule engine is implemented to identify the transaction's validity against widely used fraud rules, and thorough testing has been done in the rule engine by adding tests for each implementation layer and the outcome. Big data applications are complex and communicate with different services to get the final result. Different microservices use different languages and different data types, and the chances of errors are high during integration. Hence, more attention should be given to writing integration or component tests, and different scenarios should be tested as much as possible to increase the reliability. Thus, multiple tests are written for the *Dataflow* pipeline in the projects, which integrates all the services. Table 11 shows a summary of all the best practices.

Best Practice Type	**Description**
People	Gain the required skills to adopt and continue TDD
Software	Use readable test names which describe the tested functionality and write comments describing the test Use meaningful names in functional code and include method comments for each method Keep the production code as simple as possible Use design patterns for the production code for low coupling and high cohesion

	Use OOP concepts when possible Use automation tools and techniques Use code coverage tools Pay more attention in selecting tools and services and select the best TDD supported tools or services
Process	Start TDD at the beginning of the project Use TDD in every microservice Follow TDD practices strictly Avoid partial adoption of TDD Prepare a test list for each microservice Write tests to validate end-user requirements Group tests according to feature Spend sufficient time in creating tests as tests are equally important as the production code Write small and simple tests, with a single assertion Watch for test features and work on a single test at a time Refactor the production code Refactor the production code only when all the tests are green Run only required tests after a single test failure Refactor the test code Measure TDD benefits Write sufficient tests to ensure the validity of the big data domain used in the project Pay more attention to writing integration tests for microservices Use benchmark datasets for tests when necessary

Table 11 : Big Data Application Best Practices

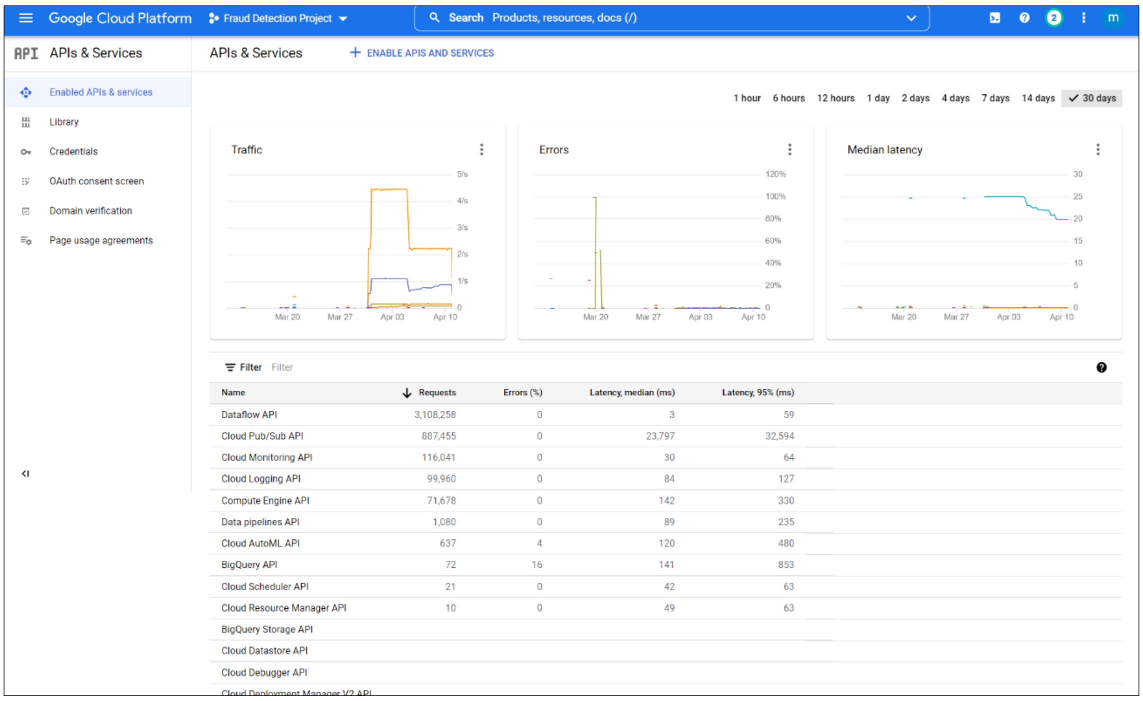
4.4 Testing Types

In the paper "*Exploring the Applicability of Test Driven Development in the Big Data Domain*," Staegemann et al. [8] have designed a TDD testing approach that follows microservice-based architecture that facilitates all four testing types and Table 12 shows how it has been practiced in the project.

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 in the machine learning model creation project. 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 project, and these are covered using the collection of unit tests. In other projects, 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 each 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. The research paper mentioned above has stressed 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 like Figure 35.
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.

Table 12 : Testing Types in Big Data Application

Figure 35 summarizes all the APIs enabled in the project and their statistics. In addition, GCP provides different monitoring reports for each service, for example, Job Metrics for *Dataflow*, utilization reports for compute engine and topic, and subscription details reports for *Pub/Sub* service, which can be used for performance monitoring.



Name	Requests	Errors (%)	Latency, median (ms)	Latency, 95% (ms)
Dataflow API	3,108,258	0	3	59
Cloud Pub/Sub API	887,455	0	23,797	32,594
Cloud Monitoring API	116,041	0	30	64
Cloud Logging API	99,960	0	84	127
Compute Engine API	71,678	0	142	330
Data pipelines API	1,080	0	89	235
Cloud AutoML API	637	4	120	480
BigQuery API	72	16	141	853
Cloud Scheduler API	21	0	42	63
Cloud Resource Manager API	10	0	49	63
BigQuery Storage API				
Cloud Datastore API				
Cloud Debugger API				
Cloud Deployment Manager V2 API				

Figure 35 : API Metrics

4.5 Challenges and Limitations

Most software projects and big data applications are developed using the traditional test-last approach and using TDD is in the area is a novel approach. Unfortunately, not many projects have used it since it is a new approach, and limited literature is available in the domain. Therefore, there is not much support that can be gained for the TDD, and the following are the main challenges faced in this project.

Adopting and continuing TDD

TDD is a new approach, and all the projects I have engaged in are implemented using the traditional test-last approach. Therefore, the experience of applying TDD was challenging. Shifting the mind toward the opposite process needed a bit of time. Also, the continuation of TDD needs high discipline.

Technical competency

Developers with little TDD experience require specific competencies to begin and continue the TDD process. Therefore, considerable time has been spent reading the literature about the application of TDD and writing tests. Also, small sample tests are written to acquire practical experience with TDD before starting the actual big data application.

Tool limitations

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. Therefore, tools should be evaluated before using them in big data applications, and the most suitable tools should be used for the purpose.

Lack of test environment

GCP provides limited support to test apps locally, and the provided emulators are also in the beta stage [74]. Therefore, 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. Therefore, not having a test environment to test the big data application is a limitation that needs to be resolved in the future.

Code coverage tools

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 were marked as not covered in the report. However, those functions are covered using multiple tests code coverage tool was not 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.

Different test libraries

Online fraud detection platform consists of different microservices. They are developed using different languages and using different frameworks. Therefore, 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. Therefore, the developers should have the competencies to handle all these libraries.

After a single test failure, executing all the tests is one of the process-related best practices revealed in the literature. However, it was not feasible due to high incurred costs, and therefore, an alternation was conducted for the process by executing only relevant feature tests after a test failure. So, these limitations should be handled in a maximum beneficial way to the project.

5 Conclusion

The main objective of this thesis is to explore the feasibility of applying TDD in implementing big data applications and identify the best practices that can be incorporated during the application. In the paper "*Exploring the Applicability of Test Driven Development in the Big Data Domain*," Staegemann et al. [8] proposed a microservice-based test-driven approach, which was adopted in the thesis. Current literature says TDD improves the internal and external quality when evaluating the project against the traditional test last approach. Therefore, the fraud detection domain has been selected to explore the feasibility as minimizing false positives and improving the quality are essential in this domain. In fraud detection applications, multiple methodologies are used to determine fraud transactions. Hence, both algorithmic and rule-based methodologies are used in this project for better accuracy and reliability.

In the second step, a detailed literature review is conducted on TDD, its process, limitations and challenges, best practices in applying TDD, tools used during the application of TDD, and finally, the effects of TDD on three factors internal quality, external quality, and productivity. In every research, the TDD process is described as writing tests firsts and then implementing the actual functional code. When it comes to the TDD process, literature has slightly different steps. However, the most precise steps are to write tests, watch for test failures, write code, watch for test passes, refactor the code, and watch for tests again to verify that refactoring has not introduced any regression issues. This six-step process is described by Bissi et al. [45], and it is the most precise and reliable set of steps found in the literature which checks for regression failures after each code refactoring. This process is followed in the project, but only the relevant tests are executed to verify that there are no new issues after refactoring, as test-related costs need to be minimized. According to the literature, limitations and challenges are categorized into four main types, people, software, productivity, and process.

During the application of TDD in the big data domain, some people-related limitations like lack of knowledge, experience, and competencies in applying TDD and difficulty shifting to the TDD mindset were evident because of not having prior TDD experience. Therefore, it is crucial to provide necessary resources to uplift the TDD skills before starting the TDD in projects to make them successful. Apart from the limitations discovered in the literature, several challenges were faced during the design and implementation. One of the main challenges was that some of the tools were unsuitable for TDD. Traditional tests are executed locally or in a specific environment. However, some of the GCP services used in the project did not have a testing platform. Tests needed to be executed in the real environment, which was challenging due to costs, and repeating the same tests makes the actual production environment unstable. Multiple best practices are found and categorized in the literature as people, software, and process related. During the project, most of the best practices are followed. Some of them are 1) gaining the necessary skills to start and continue TDD in the project, 2) using readable names in tests and functional code, 3) keeping production code simple, 4) using design patterns and OOP concepts during the implementation, and 5) using automation code coverage tools. Multiple process-related best practices are found and listed in the literature, and almost all of them are used in the current project as they are practical. Applying them made the project a smooth continuation of TDD. Apart from the best practices found in the literature, multiple other best practices are discovered during the application, like 1) paying more attention to selecting tools and services and selecting the best TDD supported tools or services, 2) using TDD in every microservice, 3) group tests according to the feature, 4) run only required tests after a single test failure, 5) write sufficient tests to ensure the validity of the big data domain used in the project, 6) pay more attention to writing integration tests for microservices and 7) use benchmark datasets for tests when necessary. The literature found that *JUnit* is the most used testing library to write the tests. However, in this project, *JUnit* was not solely sufficient for writing tests because of the complexity of the big data application and having multiple microservices. The last topic of the literature review was TDD effects. However, measuring the TDD benefits using metrics as the literature did was not feasible because of not having two projects, one with TDD and one with the test last approach. However, the benefits are measured using *SonarQube* code coverage tools for reliability, maintainability, and security.

After the detailed literature review, the actual design and implementation of the big data application were carried out, and three projects were created for the machine learning model creation, rule engine creation, and finally, the online fraud detection platform. The machine learning model was created using *Python*, and the other two projects used *Java* as microservices can use different languages. TDD was adopted from the start and continued till the end of all these projects. For writing the tests, different libraries like *unittest*, *InSpec GCP*, *JUnit Jupiter* library, *spring-rest docs-mockmvc* libraries, *Beam-runners-direct-java* library used as *JUnit* was not sufficient to write tests for all the microservices. In the proposed microservice-based test-driven approach [8], testing procedures are explained for each testing level. All four testing levels, method, subcomponent, component, and system are covered via tests in the project. Therefore, it is promising that TDD can be used in microservice-based big data applications.

Finally, an evaluation has been carried out to verify the effectiveness of TDD using the aspects number of tests, functionalities covered by the tests, code quality, and testing types covered for each project. When looking at the number of tests, 100+ have been written for all three projects. They are equally distributed among the three projects. When considering the functionalities covered, all the essential aspects of the project are covered using the tests. For example, the most critical functionalities in the machine learning model projects are pre-processing and model creation. Tests cover these two aspects, and apart from that, resource creation is also covered via infrastructure tests. The heart of the rule engine is a set of conditions, and they are thoroughly covered via tests. The online fraud detection platform is the hub that connects all microservices. Therefore, it is crucial to cover all the different microservices' data transformations and integration points. When analysing the tests, it was noticeable that all transformations are integrations are well covered. Code quality is the other aspect that projects were evaluated. According to the analysis report created by *SonarQube*, all three projects are production-ready at the first attempt without making any modifications to the code. Therefore, it is evident that TDD produced high-quality code on the first attempt. Also, according to the results maintainability of the three projects was at the highest level. However, according to the analysis results, rule engine and online fraud detection platform projects displayed some reliability and security issues.

Nevertheless, when analyzed one by one, they were found not critical. Furthermore, those issues could have been fixed quickly as they are easy to fix. Therefore, TDD produced highly reliable, maintainable, and more secure code at the first attempt for all three projects, which show a high-quality code.

Moreover, code coverage was also checked for three projects. A significant limitation was discovered when analysing the code coverage report: the code coverage tool used in the project could not identify all the code covered by the tests. Therefore, in the future, attention should be given to selecting an appropriate code coverage tool capable of covering complex big data application code.

Finally, considering all the above points, it is evident that the TDD process is promising for developing big data applications. Furthermore, the proposed microservice-based test-driven approach can be used successfully to fulfil the requirement. Similarly, the TDD produced high-quality code at the first attempt, which can be further improved by using the most convenient services which support TDD for the implementation and using improved code coverage tools which can capture big data-related code.

5.1 Future Work

In the thesis, the feasibility of applying TDD in big data applications using a microservice-based test-driven approach is discovered and found successful. However, some improvements can be made to enhance the TDD process further.

- Evaluate the project by TDD and TLD

In this project, aspects like the number of test cases written for each project, functionalities covered by the tests, and code quality aspects like reliability, maintainability, security, and code coverage are used for the evaluation. However, it is better if the same fraud detection application can be developed using the approaches TDD and TLD by two groups and then evaluate these projects by comparing them. After implementing the two projects, the effects of TDD on the aspects like class design, internal quality, external quality, and productivity can be measured. Metrics like code complexity, branch coverage, time to generate production code, defect rate, and test coverage can be utilized. Then the values can be compared to see how it has been varied when using TDD and TLD. This approach is more reliable when measuring the TDD benefits rather than evaluating a single project.

- Find the most suitable TDD supported tools.

Lack of support from tools to follow the TDD process was one of the challenges discovered during the implementation. For example, some of the GCP services are more sophisticated when used through UI, but it was not easy to access via API. In addition, some of the services are implemented to make it difficult to write unit tests. Therefore, before deciding to adopt TDD in the project, it is essential to identify the most suitable services to support TDD and start the project. Furthermore, tests are executed in the real production environment when using some of the services, as the services do not provide a testing environment or an emulator. Running all tests in the production environment is costly, and repeat execution makes the environment messy. Therefore, it is crucial to create a list of more suitable services for TDD and provide the necessary facilities to execute tests so that they will be beneficial for future projects.

- Find the most suitable coverage tools.

One of the evaluation methods carried out in this project was running the *SonarQube* and *Code Coverage* tools and checking for the code coverage via tests. However, when analysing the report generated for the online fraud detection platform project, it was noticeable that some of the classes written for transforming elements are recognized as not covered even though they are covered via tests. Therefore, it was observable that code coverage tools have limitations in capturing code coverage for complex systems like big data applications. So, it is crucial to find the most suitable code coverage tools and other quality measurement tools that can analyse complex big data applications for better analysis results and metrics. Therefore, it is beneficial to research tools well suited for big data applications and make a list available for future references.

- Create a benchmark test list for big data applications.

This project explores the feasibility of applying TDD in big data applications, and three projects are implemented. A machine learning model is implemented in one of the projects, and all the tests related to pre-processing and machine learning model development are covered. A rule engine is implemented in the second project to filter valid and fraudulent transactions. There are tests written to verify each condition and covered both positive and negative scenarios. In the third project, an online fraud detection platform is created, and tests are written to verify all transformations and integration points. So, there are essential scenarios in all three projects that need to be covered via tests, which makes the projects more reliable. Therefore, if we can create a benchmark test list for big data applications, it can serve as the basis for writing tests in TDD. Those tests can make the big data application more reliable and explainable at a minimum. Benchmark test lists can be created for each type of application, and below are some examples.

- Test list for machine learning models
  - Pre-processing related tests
  - Verify missing values
  - Verify duplicate rows
  - Verify removing unwanted columns like identification numbers, sequence numbers
  - Verify transforming categorical values to numerical
- Machine learning model related tests
  - Verify accuracy
  - Verify overfitting
  - Verify underfitting
- Test list for big data applications
  - Streaming related tests
    - Verifying transforming elements
    - Verify integration points
    - Verify input sources
    - Verify outputs

- Cover more tests

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. Figure 36 shows a piece of sample log.

```
{
  "insertId": "18orziefjliahm",
  "jsonPayload": {
    "@type":
"type.googleapis.com/google.cloud.automl.master.TablesModelStruc
ture",
    "modelParameters": [
      {
        "hyperparameters": {
          "Enable L1": "False",
          "Skip connections type": "concat",
          "Enable L2": "False",
          "Hidden layer size": 16,
          "Enable embedding L1": "False",
          "Enable layerNorm": "False",
          "Enable embedding L2": "False",
          "Dropout rate": 0.5,
          "Normalize numerical column": "True",
          "Number of cross layers": "3",
          "Number of hidden layers": "4",
          "Embedding numerical embedding": "True",
          "Enable batchNorm": "True",
          "Model type": "nn"
        }
      },
……….
      {
        "hyperparameters": {
          "Center Bias": "False",
          "Number of trees": 150,
          "Max tree depth": 6,
          "Model type": "GBDT",
          "Tree L1 regularization": 1,
          "Tree complexity": 3,
          "Tree L2 regularization": 1
        }
      }
    ]
  },
  "resource": {
    "type": "cloudml_job",
    "labels": {
      "job_id": "TBL6696804267388305408",
      "region": "eu",
      "project_id": "ultra-palisade-339421"
    }
  },
  "timestamp": "2022-04-06T22:24:53.958476887Z",
  "severity": "INFO",
  "labels": {
    "log_type": "automl_tables"
  },
  "logName": "projects/ultra-palisade-
339421/logs/automl.googleapis.com%2Fmodel",
  "receiveTimestamp": "2022-04-06T22:24:53.958476887Z"
}
```

Figure 36 : AutoML Log

- Apply TDD in other domains

In this thesis, 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.

Literature

1. Staegemann, D., Volk, M., Lautenschläger, E., Pohl, M., Abdallah, M., & Turowski, K. (2021). *Applying Test Driven Development in the Big Data Domain – Lessons From the Literature*. 511–516. https://doi.org/10.1109/ICIT52682.2021.9491728
2. Nauck, D. (2019). *Test Driven Machine Learning*. QCon, 24th April 2019, London.
3. Handoko, B., Mulyawan, A., Tanuwijaya, J., Tanciady, F., & Vionita, N. (2020). *Big Data in Auditing for the Future of Data Driven Fraud Detection*. https://doi.org/10.35940/ijitee.B7568.019320
4. Rodríguez-Mazahua, L., Rodríguez-Enríquez, C.-A., Sánchez-Cervantes, J. L., Cervantes, J., García-Alcaraz, J. L., & Alor-Hernández, G. (2016). A general perspective of Big Data: applications, tools, challenges and trends. *The Journal of Supercomputing*, *72*(8), 3073–3113. https://doi.org/10.1007/s11227-015-1501-1
5. Richhariya, P., & Singh, P. (2012). A Survey on Financial Fraud Detection Methodologies. *International Journal of Computer Applications*, *45*, 15–22.
6. Perez-Palacin, D., Ridene, Y., & Merseguer, J. (2017). Quality Assessment in DevOps: Automated Analysis of a Tax Fraud Detection System. *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, 133–138. https://doi.org/10.1145/3053600.3053632
7. Sukkarieh, J., & Kamal, J. (2009). *Towards agile and test-driven development in NLP applications*. 42–44. https://doi.org/10.3115/1621947.1621954
8. Staegemann, D., Volk, M., Jamous, N., & Turowski, K. (2020, October). *Exploring the Applicability of Test Driven Development in the Big Data Domain*.
9. Karac, I., & Turhan, B. (2018). What Do We (Really) Know about Test-Driven Development? *IEEE Software*, *35*(4), 81–85. https://doi.org/10.1109/MS.2018.2801554
10. Awoyemi, J., Adetunmbi, A., & Oluwadare, S. (2017). *Credit card fraud detection using machine learning techniques: A comparative analysis*. 1–9. https://doi.org/10.1109/ICCNI.2017.8123782
11. Google Cloud. 2022. *What Is Microservices Architecture? | Google Cloud*. [online] Available at: <https://cloud.google.com/learn/what-is-microservices-architecture#:~:text=Microservices%20architecture%20(often%20shortened%20to,its%20own%20realm%20of%20responsibility.> [Accessed 1 May 2022].
12. Investopedia. 2022. *How Cloud Computing Works*. [online] Available at: <https://www.investopedia.com/terms/c/cloud-computing.asp> [Accessed 26 April 2022].
13. Google Cloud. 2022. *Cloud Computing Services | Google Cloud*. [online] Available at: <https://cloud.google.com/> [Accessed 26 April 2022].
14. Hoffmann, L. F. S., Vasconcelos, L. E. G. de, Lamas, E., Cunha, A. M. da, & Dias, L. A. V. (2014). Applying Acceptance Test Driven Development to a Problem Based Learning Academic Real-Time System. *2014 11th International Conference on Information Technology: New Generations*, 3–8. https://doi.org/10.1109/ITNG.2014.63
15. Richhariya, P., & Singh, P. (2012). A Survey on Financial Fraud Detection Methodologies. *International Journal of Computer Applications*, *45*, 15–22.
16. Vesta.io. 2022. *How to Avoid False Positives With Credit Card Fraud Detection*. [online] Available at: <https://www.vesta.io/blog/false-positives-credit-card-fraud-detection#:~:text=A%20false%20positive%20means%20a,customer%20purchases%20create%20many%20problems> [Accessed 14 April 2022].
17. Wedge, R., Kanter, J. M., Veeramachaneni, K., Rubio, S. M., & Perez, S. I. (2019). Solving the False Positives Problem in Fraud Prediction Using Automated Feature Engineering. In U. Brefeld, E. Curry, E. Daly, B. MacNamee, A. Marascu, F. Pinelli,

M. Berlingerio, & N. Hurley (Eds.), *Machine Learning and Knowledge Discovery in Databases* (pp. 372–388). Springer International Publishing.

18. Janzen, D., & Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. *Computer*, *38*(9), 43–50. https://doi.org/10.1109/MC.2005.314
19. microservices.io. 2022. *What are microservices?*. [online] Available at: <https://microservices.io/> [Accessed 26 April 2022].
20. Siddiqui, S., & Khan, T. A. (2019). Test Patterns for Cloud Applications. *IEEE Access*, *7*, 147060–147080. https://doi.org/10.1109/ACCESS.2019.2946315
21. Hammond, S., & Umphress, D. (2012). Test driven development: The state of the practice. *Proceedings of the Annual Southeast Conference*. https://doi.org/10.1145/2184512.2184550
22. Sas.com. 2022. *Big Data: What it is and why it matters*. [online] Available at: <https://www.sas.com/en_us/insights/big-data/what-is-big-data.html> [Accessed 26 April 2022].
23. Gupta, A., & Jalote, P. (2007). An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development. *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 285–294. https://doi.org/10.1109/ESEM.2007.41
24. Khanam, Z., & Ahsan, M. N. (2017). Evaluating the effectiveness of test driven development: Advantages and pitfalls. *International Journal of Applied Engineering Research*, *12*, 7705–7716.
25. Pombo, N., & Martins, C. (2021). Test Driven Development in Action: Case Study of a Cross-Platform Web Application. *IEEE EUROCON 2021 - 19th International Conference on Smart Technologies*, 352–356. https://doi.org/10.1109/EUROCON52738.2021.9535554
26. Munck, A., & Madsen, J. (2017). Test-driven modeling and development of cloud-enabled cyber-physical smart systems. *2017 Annual IEEE International Systems Conference (SysCon)*, 1–8. https://doi.org/10.1109/SYSCON.2017.7934714
27. Borle, N. C., Feghhi, M., Stroulia, E., Greiner, R., & Hindle, A. (2018). Analyzing the effects of test driven development in GitHub. *Empirical Software Engineering*, *23*(4), 1931–1958. https://doi.org/10.1007/s10664-017-9576-3
28. Tosun, A., Ahmed, M., Turhan, B., & Juristo, N. (2018). On the Effectiveness of Unit Tests in Test-Driven Development. *Proceedings of the 2018 International Conference on Software and System Process*, 113–122. https://doi.org/10.1145/3202710.3203153
29. Aderaldo, C. M., Mendonça, N. C., Pahl, C., & Jamshidi, P. (2017). Benchmark Requirements for Microservices Architecture Research. *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, 8–13. https://doi.org/10.1109/ECASE.2017.4
30. Dogša, T., & Batič, D. (2011). The effectiveness of test-driven development: an industrial case study. *Software Quality Journal*, *19*(4), 643–661. https://doi.org/10.1007/s11219-011-9130-2
31. Mitrović, D., Ivanovic, M., & Budimac, Z. (2015). *Test-driven development of web and enterprise agents*. 1–5. https://doi.org/10.1145/2801081.2801112
32. Guerra, E. (2014). Designing a Framework with Test-Driven Development: A Journey. *Software, IEEE*, *31*, 9–14. https://doi.org/10.1109/MS.2014.3
33. Nanthaamornphong, A., & Carver, J. C. (2017). Test-Driven Development in scientific software: a survey. *Software Quality Journal*, *25*(2), 343–372. https://doi.org/10.1007/s11219-015-9292-4
34. Hayes, J. H., Dekhtyar, A., & Janzen, D. (2009). Towards traceable test-driven development. *2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, 26–30. https://doi.org/10.1109/TEFSE.2009.5069579

35. Cuadrado, J. S. (2020). Towards Interactive, Test-driven Development of Model Transformations. *J. Object Technol.*, *19*(3), 1–3.
36. Agarwal, N., & Deep, P. (2014). Obtaining better software product by using test first programming technique. *2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)*, 742–747. https://doi.org/10.1109/CONFLUENCE.2014.6949233
37. Hametner, R., Winkler, D., Östreicher, T., Surnic, N., & Biffl, S. (2010). *Selecting UML models for test-driven development along the automation systems engineering process*. 1–4. https://doi.org/10.1109/ETFA.2010.5641258
38. Latorre, R. (2014). A successful application of a Test-Driven Development strategy in the industrial environment. *Empirical Software Engineering*, *19*, 753–773. https://doi.org/10.1007/s10664-013-9281-9
39. Shull, F., Melnik, G., Turhan, B., Layman, L., Diep, M., & Erdogmus, H. (2010). What Do We Know about Test-Driven Development? *IEEE Software*, *27*(6), 16–19. https://doi.org/10.1109/MS.2010.152
40. Maximilien, E. M., & Williams, L. (2003). Assessing test-driven development at IBM. *25th International Conference on Software Engineering, 2003. Proceedings.*, 564–569. https://doi.org/10.1109/ICSE.2003.1201238
41. Buchan, J., Li, L., & MacDonell, S. G. (2011). Causal Factors, Benefits and Challenges of Test-Driven Development: Practitioner Perceptions. *2011 18th Asia-Pacific Software Engineering Conference*, 405–413. https://doi.org/10.1109/APSEC.2011.44
42. Aniche, M., & Gerosa, M. A. (2010). Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers. *ICSTW 2010 - 3rd International Conference on Software Testing, Verification, and Validation Workshops*, 469–478. https://doi.org/10.1109/ICSTW.2010.16
43. Mäkinen, S., Münch, J. (2014). Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies. In S. and B. J. Winkler Dietmar and Biffl (Ed.), *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering* (pp. 155–169). Springer International Publishing.
44. Fucci, D. (2014). Understanding the Dynamics of Test-Driven Development. *Companion Proceedings of the 36th International Conference on Software Engineering*, 690–693. https://doi.org/10.1145/2591062.2591086
45. Bissi, W., Serra Seca Neto, A. G., & Emer, M. C. F. P. (2016). The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology*, *74*, 45–54. https://doi.org/https://doi.org/10.1016/j.infsof.2016.02.004
46. Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., & 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), 597–614. https://doi.org/10.1109/TSE.2016.2616877
47. Fucci, D., Turhan, B., & Oivo, M. (2014, November). *On the Effects of Programming and Testing Skills on External Quality and Productivity in a Test-Driven Development Context*. https://doi.org/10.1145/2745802.2745826
48. Marchenko, A., Abrahamsson, P., & Ihme, T. (2009). Long-Term Effects of Test-Driven Development A Case Study. In P. Abrahamsson, M. Marchesi, & F. Maurer (Eds.), *Agile Processes in Software Engineering and Extreme Programming* (pp. 13–22). Springer Berlin Heidelberg.
49. Causevic, A., Punnekkat, S., & Sundmark, D. (2012, November). Quality of Testing in Test Driven Development. *Proceedings - 2012 8th International Conference on the Quality of Information and Communications Technology, QUATIC 2012*. https://doi.org/10.1109/QUATIC.2012.49

50. Kollanus, S. (2011). Critical Issues on Test-Driven Development. In M. and B. M. T. and V. G. Caivano Danilo and Oivo (Ed.), *Product-Focused Software Process Improvement* (pp. 322–336). Springer Berlin Heidelberg.
51. Cauevic, A., Punnekkat, S., & Sundmark, D. (2013). TDDHQ: Achieving Higher Quality Testing in Test Driven Development. *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, 33–36. https://doi.org/10.1109/SEAA.2013.47
52. Fraser, S., Astels, D., Beck, K., Boehm, B., Mcgregor, J., Newkirk, J., & Poole, C. (2003). *Discipline and practices of TDD: (test driven development).* 268–270. https://doi.org/10.1145/949344.949407
53. Kawadkar, H. (2014). Test driven design. *2014 International Conference on Contemporary Computing and Informatics (IC3I)*, 236–237. https://doi.org/10.1109/IC3I.2014.7019671
54. Tacker, T. (2017). Best Practices for Test Driven Development.
55. Rendell, A. (2008). Effective and Pragmatic Test Driven Development. *Agile 2008 Conference*, 298–303. https://doi.org/10.1109/Agile.2008.45
56. Missiroli, M., Russo, D., & Ciancarini, P. (2017). Teaching Test-First Programming: Assessment and Solutions. *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, *1*, 420–425. https://doi.org/10.1109/COMPSAC.2017.229
57. Shelton, W., Li, N., Ammann, P., & Offutt, J. (2012). *Adding Criteria-Based Tests to Test Driven Development*. https://doi.org/10.1109/ICST.2012.191
58. Fucci, D., Turhan, B., Juristo, N., Dieste, O., Tosun-Misirli, A., & Oivo, M. (2015). Towards an operationalization of test-driven development skills: An industrial empirical study. *Information and Software Technology*, *68*, 82–97. https://doi.org/https://doi.org/10.1016/j.infsof.2015.08.004
59. 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), 169–184. https://doi.org/https://doi.org/10.1016/j.infsof.2009.08.007
60. Müller, M., & Hagner, O. (2002). Experiment about test-first programming. *IEE Proceedings - Software*, *149*, 131–136. https://doi.org/10.1049/ip-sen:20020540
61. Vu, J. H., Frojd, N., Shenkel-Therolf, C., & Janzen, D. S. (2009). Evaluating Test-Driven Development in an Industry-Sponsored Capstone Project. *2009 Sixth International Conference on Information Technology: New Generations*, 229–234. https://doi.org/10.1109/ITNG.2009.11
62. Aniche, M., & Gerosa, M. A. (2015). Does test-driven development improve class design? A qualitative study on developers' perceptions. *Journal of the Brazilian Computer Society*, *21*(1), 15. https://doi.org/10.1186/s13173-015-0034-z
63. Shrivastava, D. P. (2012). *Unit Test Case Design Metrics in Test Driven Development*.
64. Kollanus, S. (2010). Test-Driven Development - Still a Promising Approach? *2010 Seventh International Conference on the Quality of Information and Communications Technology*, 403–408. https://doi.org/10.1109/QUATIC.2010.73
65. Huang, L., & Holcombe, M. (2009). Empirical investigation towards the effectiveness of Test First programming. *Information & Software Technology*, *51*, 182–194. https://doi.org/10.1016/j.infsof.2008.03.007
66. S P, M., Saini, A., Ahmed, S., & Sarkar, S. (2019). Credit Card Fraud Detection using Machine Learning and Data Science. *International Journal of Engineering Research And*, *08*. https://doi.org/10.17577/IJERTV8IS090031
67. Sakharova, I. (2012). Payment card fraud: Challenges and solutions. *2012 IEEE International Conference on Intelligence and Security Informatics*, 227–234. https://doi.org/10.1109/ISI.2012.6284315

68. West, J., Bhattacharya, M., & Islam, R. (2015). Intelligent Financial Fraud Detection Practices: An Investigation. In J. Tian, J. Jing, & M. Srivatsa (Eds.), *International Conference on Security and Privacy in Communication Networks* (pp. 186–203). Springer International Publishing.
69. Mohammed, R. A., Wong, K.-W., Shiratuddin, M. F., & Wang, X. (2018). Scalable Machine Learning Techniques for Highly Imbalanced Credit Card Fraud Detection: A Comparative Study. In X. Geng & B.-H. Kang (Eds.), *PRICAI 2018: Trends in Artificial Intelligence* (pp. 237–246). Springer International Publishing.
70. Rita, A., & Nweke, H. C. (2013). IT-Driven Approaches to Fraud Detection and Control in Financial Institutions. *West African Journal of Industrial and Academic Research*, *8*, 35–42.
71. Salazar, A., Safont, G., & Vergara, L. (2014). Surrogate techniques for testing fraud detection algorithms in credit card operations. *2014 International Carnahan Conference on Security Technology (ICCST)*, 1–6. https://doi.org/10.1109/CCST.2014.6986987
72. Base, K. and range, H., 2022. *How to find the interquartile range*. [online] Scribbr. Available at: <https://www.scribbr.com/statistics/interquartile-range.> [Accessed 27 April 2022].
73. Google Cloud. 2022. *Feature preprocessing overview | BigQuery ML | Google Cloud*. [online] Available at: <https://cloud.google.com/bigquery-ml/docs/reference/standard-sql/bigqueryml-syntax-preprocess-overview> [Accessed 20 January 2022].
74. Google Cloud. 2022. *AutoML Tables documentation | Google Cloud*. [online] Available at: <https://cloud.google.com/automl-tables/docs> [Accessed 13 February 2022].
75. Surya, G., 2022. *Folder Structure for Machine Learning Projects*. [online] Available at: <https://medium.com/analytics-vidhya/folder-structure-for-machine-learning-projects-a7e451a8caaa> [Accessed 15 January 2022].
76. Docs.python.org. 2022. *25.3. unittest — Unit testing framework — Python 2.7.18 documentation*. [online] Available at: <https://docs.python.org/2/library/unittest.html> [Accessed 15 January 2022].
77. Python Pool. 2022. *Python Unittest Vs Pytest: Choose the Best*. [online] Available at: <https://www.pythonpool.com/python-unittest-vs-pytest/> [Accessed 15 January 2022].
78. Google Cloud. 2022. *Testing apps locally with the emulator | Cloud Pub/Sub Documentation | Google Cloud*. [online] Available at: <https://cloud.google.com/pubsub/docs/emulator> [Accessed 1 March 2022].
79. GitHub. 2022. *GitHub - inspec/inspec-gcp: InSpec GCP (Google Cloud Platform) Resource Pack*. [online] Available at: <https://github.com/inspec/inspec-gcp> [Accessed 1 February 2022].
80. Terraform by HashiCorp. 2022. *What is Terraform | Terraform by HashiCorp*. [online] Available at: <https://www.terraform.io/intro> [Accessed 27 April 2022].
81. Fin.plaid.com. 2022. Fin - Algorithmic and rules-based fraud models. [online] Available at: <https://fin.plaid.com/articles/algorithmic-and-rules-based-fraud-models/> [Accessed 12 April 2022].
82. Exception Not Found. 2022. *The Repository-Service Pattern with DI and ASP.NET Core*. [online] Available at: <https://exceptionnotfound.net/the-repository-service-pattern-with-dependency-injection-and-asp-net-core/> [Accessed 26 February 2022].
83. Google Cloud. 2022. *Dataflow documentation | Google Cloud*. [online] Available at: <https://cloud.google.com/dataflow/docs> [Accessed 1 March 2022].
84. Beam.apache.org. 2022. *Test Your Pipeline*. [online] Available at: <https://beam.apache.org/documentation/pipelines/test-your-pipeline/> [Accessed 5 March 2022].

85. Almog, D. and Heart, T. (2009). What Is a Test Case? Revisiting the Software Test Case Concept. In N. and C. G. J. and R. M. R. and S. K. and M. R. O'Connor Rory V. and Baddoo (Ed.), *Software Process Improvement* (pp. 13–31). Springer Berlin Heidelberg.
86. Kirk, M. (2014). *Thoughtful Machine Learning*.
87. Beam.apache.org. 2022. *Test Your Pipeline*. [online] Available at: <https://beam.apache.org/documentation/pipelines/test-your-pipeline/> [Accessed 6 February 2022].
88. Sonar Community. 2022. *[Coverage & Test Data] Generate Reports for Apex, C/C++, Objective-C, Go, JS/TS and Python*. [online] Available at: <https://community.sonarsource.com/t/coverage-test-data-generate-reports-for-apex-c-c-objective-c-go-js-ts-and-python/9687> [Accessed 20 March 2022].
89. Unkelos-Shpigel, N., & Hadar, I. (2018). Test First, Code Later: Educating for Test Driven Development. In R. Matulevičius & R. Dijkman (Eds.), *Advanced Information Systems Engineering Workshops* (pp. 186–192). Springer International Publishing.
90. Mou, D., & Ratiu, D. (2012). Binding requirements and component architecture by using model-based test-driven development. *2012 First IEEE International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks)*, 27–30. https://doi.org/10.1109/TwinPeaks.2012.6344557
91. Paikan, A., Traversaro, S., Nori, F., & Natale, L. (2015). A Generic Testing Framework for Test Driven Development of Robotic Systems. *MESAS*.
92. Hussein, S., Stephan, J., & Maria, K. (2017). Testing Test-Driven Development.

Declaration of Independence

I hereby declare that I have written this thesis independently and have not used any sources or resources other than those specified.

Magdeburg, 03/05/2022 Maneendra Perera