NALYZER: AI BASED COMMUNITY-DRIVEN SOURCE CODE ANALYSIS TOOL

Denuwan Himanga Hettiarachchi

(199327M)

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa
Sri Lanka

April 2021

NALYZER: AI BASED COMMUNITY-DRIVEN SOURCE CODE ANALYSIS TOOL

Denuwan Himanga Hettiarachchi

(199327M)

Thesis/Dissertation submitted in partial fulfillment of the requirements for the degree Master Of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

April 2021

DECLARATION OF THE CANDIDATE & SUPERVISOR

I declare that this is my own work and this dissertation does not incorporate without acknowledgment any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgment is made in the text. Also, I hereby grant to the University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other media.

I retain the right to use this content in whole or part in future works (such as articles or books).

Signature …….. Date:

The above candidate has carried out research for the dissertation under my supervision.

Name of Supervisor: Dr. Indika Perera

15-11-2021

Signature …….. Date: ……..

ABSTRACT

Identifying error-prone code snippets and potential vulnerabilities in the early stages of the development process allows reducing the considerable amount of time & the cost of the software project. But the process of ensuring the reliability of software projects has become a significant challenge due to the high complexity & the scalability of modern software projects. Also, the dynamic nature of modern frameworks & programming languages becomes a barrier to consistency. Manual code reviews/automated code analysis tools are obsolete due to time constraints & lack of adaptability for new programming languages & frameworks.

Nalyzer project aims to build a Machine Learning (ML) model to identify error-prone code snippets and potential vulnerabilities in the source code. And introduce a self-sustainable approach to adopt future programming languages & framework changes.

We used Convolutional Neural Network (CNN) deep learning algorithm to build an ML model for classifying buggy & non-buggy code snippets from source code. And introduce a maven customized build plugin to push source code to ML model & get prediction as a step in the Continuous Integration/Continuous Delivery (CI/CD) pipeline. Then the generated Nalyzer analysis result was published on the interactive dashboard inside the project directory. Interactive dashboard facilitated to get feedback from developers to improve ML model accuracy & future adaptations.

We evaluate the ML model in terms of F-measure. The evaluation results demonstrated the compatibility of ML techniques in the source code analysis paradigm with a significant score. And the interactive dashboard makes sure of the self-sustainability of the ML model through a Community-Driven approach.

Nalyzer project proves that the ML approach is an alternative for overcoming the limitations of manual code reviews and automated code analysis tools.

Key Words: Machine Learning (ML), Neural Network, Convolutional Neural Network (CNN), Source Code Analysis

ACKNOWLEDGEMENT

I would like to take this opportunity to express my gratitude to my supervisor Dr.Indika Perera for all guidance & advice provided through this journey.

And also I would like to thank industry experts, who shared their experiences and thoughts during the literature review and background studies. Their contribution adds value to this project to achieve the end goals in a more practical manner.

Also I would like to thank Kirill Eremenko, Hadelin de Ponteves for their great content on Udemy platform. Their amazing tutorials help me to clear my path of implementation.

Finally yet importantly, I want to mention the role of my parents in my journey. They add unconditional support to manage my career & studies in a free mindset.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

1. INTRODUCTION

1.1. Background Context & Motivation

Identifying error-prone code snippets and potential vulnerabilities in the early development stage reduces the considerable amount of time & cost in Software engineering projects. Because the cost of finding and fixing defects increases overtime. (Fig. 1 from Capers Jones, Software Assessments, Benchmarks, and Best Practices book came up with an analysis of cost -per defect analysis [1]). In the modern agile DevOps era, the software industry expects to ship products in a continuous manner within small-time sprints. The help of automated testing & different analysis tools allows software developers to focus their core effort to implementation of the feature. And reduce the fair amount of time of the testing and reviews. But the Gartner report revealed that 99% of respondents in their survey face some kind of challenge with automated testing & identify potential vulnerabilities using advanced tools in agile development. [2]

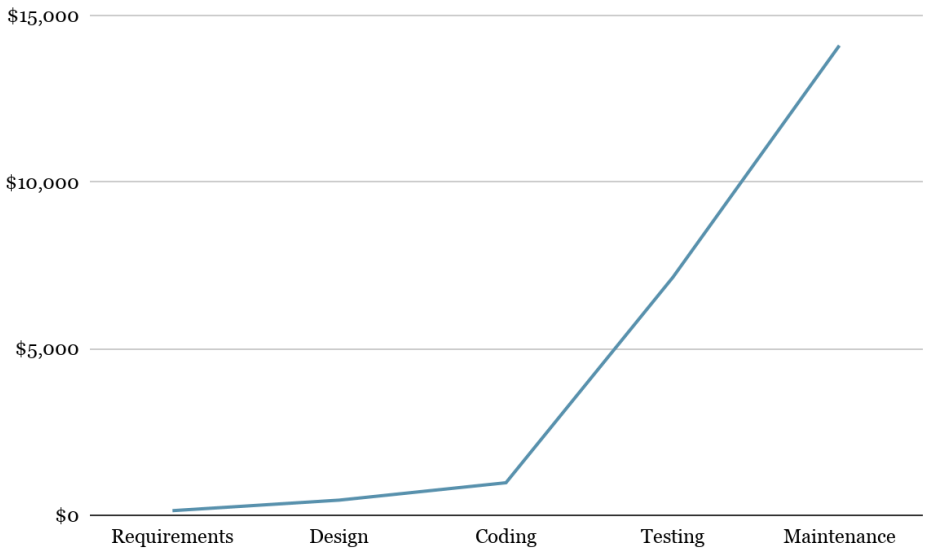

Figure 1: Cost per defect analysis in SDLC

Identifying error-prone code snippets and potential vulnerabilities during manual code reviews, required a considerable amount of time in development timelines and industry expert's contributions. Still, manual code reviews are challenging due to various reasons.

Such manual reviews may have potential human errors, and decisions made during reviews may be based on the reviewer's perspective, and lack of expert resources with the required knowledge.

Process automation is a trending approach in the modern agile development paradigm. Unit testing, Code analysis, Test automation, Deployment activities automated as much as possible in continuous integration and continuous development (CI/CD) pipelines. Automated CI/CD pipelines reduce the work & time in the software development process and allow us to ship products in a small time frame. Building such a tool required expert knowledge and it's a complex process. Also, most of the automated tools required new updates frequently due to programming languages and framework frequent release cadence.

Artificial Intelligence (AI) has been a hype in the modern world recently. Gartner reveals by 2025, 50% of enterprises will leverage intelligent automation tools driven by AI and machine learning.[2] AI-based code analysis for identifying error-prone code snippets and potential vulnerabilities allows an advanced approach to solving challenges in most of the tools used in the industry today. Developer community contributions also improve lots of areas in the software industry. But we don't see a trend to get involved in community contribution to improve the quality of the analysis in a more efficient manner yet.

1.2. Research Problem Definition

Common service-based software development teams include independent software test teams and different reviews before merging code into the master branch to verify the quality of the code. Both product and service-oriented companies move to test automation and robotic process automation (RPA) mechanisms to minimize the release timelines. But training and building an experienced test team is a time-consuming and costly effort. Building a team with domain knowledge & technical knowledge is a more complex and time-consuming task.

As described above, from a code review perspective, doing an ideal code review also depends on the tech lead's experience and time. This research is primarily focused to build a code analysis tool using a Neural Network model (NN) and the contribution developer community.

Since Artificial Intelligence is a new trending alternative in many technology areas, AI-based code analysis approaches are not new. But training an AI-based model and the existence of

a Machine Learning (ML) approach is still questionable due to the dynamic nature of the programming languages & frameworks. Most of the technologies available today will be obsolete in the next couple of years or improving in a rapid manner. Adapting to the dynamic nature of the software industry is another big challenge for AI-based training models.

Time is the most critical aspect of a software project, most of the values depend on time in modern developments. The software product line is a common approach to address timeline limitation in management, reusability is a key aspect of a software product line. The existence of an AI-based solution is still questionable in this reusability approach. Coming up with a universal solution is more practical and also it will avoid an obsolete framework among the software developer community in the future. It will help to build the AI model with a high accuracy rate.

The NALYZER project primarily identified the following limitations in the traditional code analysis realm in software engineering.

1. Lack of technology experts resources
2. New engineer's training and experience issues
3. Time limitations in a project plan and priorities issues
4. Dynamic usage of technologies in projects
5. Rapid changes in technologies in software engineering

In the following section (1.4), we will explain how NALYZER addresses the above issues.

1.3. Research Objectives

NALYZER comes up with a different perspective of the code analysis to identify buggy code & potential vulnerabilities. Researchers focus on building neural network-based solutions to identify the buggy code patterns & potential vulnerabilities code snippets which should be re-considered before merge to the master branch by analyzing JAVA source code to reduce human errors, time & effort.

In order to build neural networks, researchers will use various error-prone code snips and documentation available on online code repositories. Based on the analysis report generated by NALYZER, using the developer's feedback can evolve the neural network for future purposes to adopt newer versions of the programming language/frameworks, in order to get developer community involvement researchers will develop a dashboard based solution.

The following components are the key deliverables of this research project.

1. Neural network model - This allows an analysis of the source code based on the trained data sources.
2. Dashboard - Allows to training the neural network by providing developer's inputs.
3. Maven Library Plugin - Grab the metadata of JAVA source code and push it to the analyzer to analyze the code.
4. Real-Time Analyzer - Provide analysis based on trained neural network findings.

1.4. Scope of the Research

In this research, the research team focused on building a build plugin on top of a maven project management and comprehension tool. Due to time limitations, initial developments focus only on JAVA-based projects.

In this documentation, the research area covers only the JAVA programming language. With the help of this scope finding, the approach can be applied to other standards technologies in the future. For the demonstration purpose and due to a limited amount of time the system will be implemented in a specific technology scope only.

The documentation has been categorized into several categories. Describing the user/audience characteristic of the system. System technical descriptions and functional and non-functional requirements are the categories described in this documentation. This documentation uses a modeling language UML to describe the system architecture. The hardware which needs to operate the system and the user operations required in order to use the system are also described in this documentation.

2. BACKGROUND AND LITERATURE REVIEW

The researcher did a comprehensive analysis of other researchers in the same domain by different institutes. The following perspectives have been covered which are associated with the Machine learning (ML) & source code analysis.

- Unit Testing and Coding in Paradigm

  Under Unit Testing and Coding in Tandem, we cover the best practices and standards we should consider when we build a unit testing framework. Especially we focus on the Java programming language in this scope to use as a primary programming language.

  Our objective is to identify industry standards and best practices introduced by reputed authors and companies. This will allow building an industry-capable standard framework.

- Java-based Unit Testing best practices and guidance

  Under Java Unit Testing best practices and guidance, we primarily focus on standards and practices specified for the Java programming language. Since unit testing is universal programming practice, we need to do prototypes by focusing on a limited scope. We are moving with the Java programming language for prototype development.

- Automated Code Analysis tools and best practices

  Under Automated Code Analysis tools and best practices, we focus on covering existing tools and frameworks available in the industry to identify the drawbacks that will be prevented via replacing artificial intelligence. Studying the existing frameworks and tools will allow understanding rooms for improvements.

- Neural Network-based Code Analysis Systems

  Under Neural Network-based Code Analysis Systems, we focus to understand existing studies and experiences in Neural Network-based code analysis tools and systems. This will allow identifying the preliminary studies and findings in the research area.

The researcher expects to build a system using a source code analysis tool with an interactive dashboard platform aiming at more visualization capabilities along with a neural network engine. Which will provide self-sustainable capabilities to the neural network model through community driven approach.

The above sub-topics are areas which we covered during the background analysis & literature review, our findings are mentioned in the next sub sections.

2.1. A Survey of Unit Testing Practices

In 2006 under the IEEE software, Per Runeson based at Lund University evaluated practices in unit testing paradigm by discussions with focus-focused network software development (SPIN) teams and presented a series of questionnaires to validate results. He also investigated the unit testing method's strengths and weaknesses. [1]

Since the Nalyzer project's primary objective is to come up with an alternative for existing source code analysis and ensure the software product reliability, this survey allows researchers to get an overview of the scope of software analysis, define the key terms & scope with the help of industry experts. This survey covers the groups representing industries in different domains and scales such as Telecom, Transportation, Health care and, banking. Each domain group into another segment as small, medium & large. In a conclusion Per Runeson state, this survey is the focus to build a universal definition of unit testing, also he believes the results summarized after the survey reached the objective at a certain level. One of the key purposes of this consideration is that the literature review is, Nalyzer's approach shouldn't be obsolete in the future due to the incompatibility with industry standards and practices, we aim to thrive together and make sure the community-driven self-sustainable approach doesn't put into a challengeable state in future. Because one of the core outcomes is the Nalyzer self-sustainable neural network module with the help of the developer community. Companies without understanding the scope of the testing task and the limitation will lead to an incomplete testing process and waste a considerable amount of the project life cycle. Also the process we used in SDLC should be transparent to all and if we are unable to provide a clear picture to the team will lead to frustrations across the team. Also, tools such as Nalyzer should define the exact barriers and scopes. Therefore, this survey builds a bridge between industry and the research outcomes.

Many surveys and questionnaires were shared across the developer community in the past couple of years, but Per Runeson's approach is more practical due to the range of participant groups, Runeson segmenting the audience in a well-defined manner by focusing to gather the maximum amount of data. He used the audience in companies in a different domain as well as scales. And the pre-defined mechanism Runeson used to evaluate this finding also takes into account the results structured in a formal format according to Zachman's

framework. Zachman's framework is well known for analyzing information system architectures. Widely using in structure for Enterprise Architecture paradigm. [32]

When we consider famous industry survey, which we use in many studies such as StackOverflow annual survey, GitHub survey does not allow to provide an abstraction of the area they covered, since most of the industry terms and practices have verbal definitions and organization oriented definitions, Runeson's approach is to get a holistic view about unit tests and the role this technology plays in the industry today. Also, Per Runeson focuses to identify the required improvement in the testing domain, the second part of the survey contains a lightweight assignment to get feedback from the focus group. When we look at this survey, it contains a holistic view of unit testing including definition, role, and improvement required in the domain.

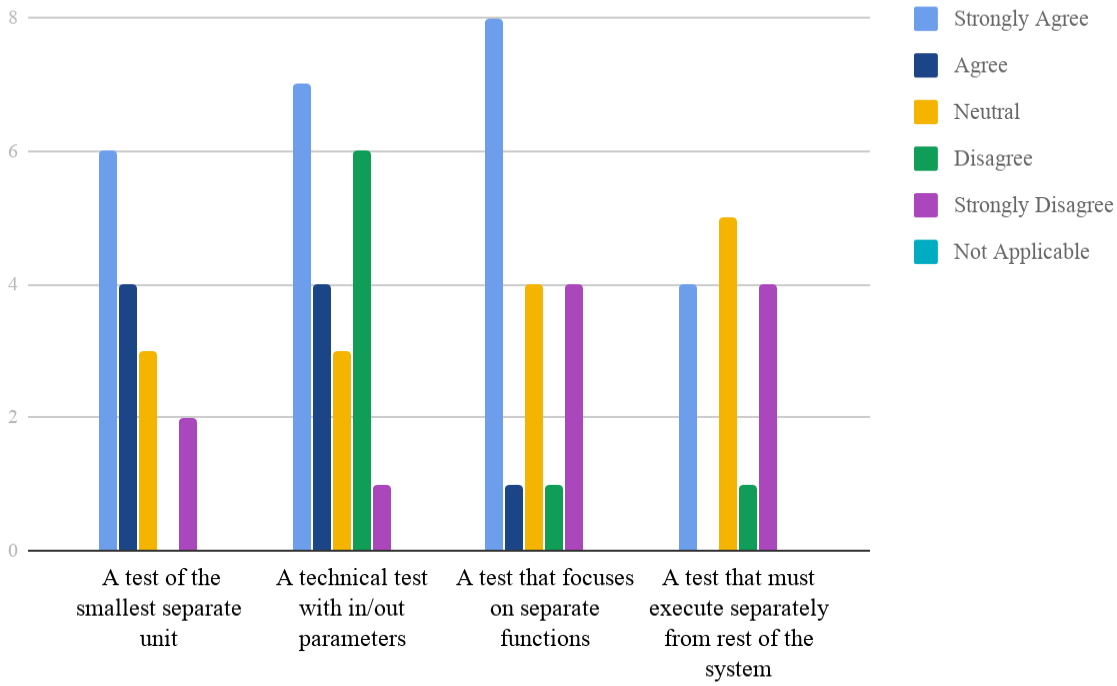

Figure 2: Survey finding of unit testing practices

Serve conducted identified communities discussed following ares primarily,

- The definition of unit testing & scopes.
- Strengths and weaknesses of practices.

- Problems faced during the implementations.

Table 1: Conclusions of the survey (Definitions of Unit Test paradigm)

	Definition	**Strengths**	**Problems**
What?	Test the smallest unit or units	Unit identification Test of surrounding modules	GUI testing Unit identification Test script and harness maintenance Data structures
How?	Structure-based Preferably automated	Test framework	Documentation Framework tailoring Test selection Test metrics
Where?	Solution domain	None found	None found
Who?	By developer	Independent test Competence network	Competency Independence Introduction strategy
When?	Quick feedback	Continuous regression test	Stopping criteria
Why?	Ensure functionality	External requirement (safety) Agile methods	Cost versus value

For each question, respondents chose the best answer from the following scale: strongly agree, agree, neutral, disagree, strongly disagree, or not applicable. And the following figure has revealed the response waves to the first question in terms of unit test definitions "What is unit testing?"

Table 1 summarizes the research results. With the help of survey the authors define what the industry means by "unit testing," as a specification for evaluating the performance of an organization's evaluation, and a turning point of organization improvement of practices. The

result provides a holistic view of the unit testing process in a well-structured manner, one of the key reasons for selecting this survey to Nalyzer literature review is to identify important points to implement the community-driven approach with focusing on a self-sustainable neural network model. Our purpose is to identify how the industry looks at this kind of sub-processes which add value to the project in terms of long term and community perspective of quality of the product over quantity. We can see the role of who should take care of unit testing, and the problems developers use in this kind of practice in day-to-day life. Automation is one of the key aspects to overcome this limitation, but building and maintaining the scripts may directly impact the SDLC process and timelines. Also, another important finding is the independence of automated scripts, since traditional unit testing is a process of developers building automated scripts to test their creation. This may impact the quality of the result, also the end product may depend on the developer's competencies and experiences. We can't expect the same potential from junior developers. The cost vs value is another questionable point of traditional unit testing, the time we focus to build the automation unit test script does not provide direct value in the short term, especially in the agile context industry expect to deliver minimum viable product within a short period. Even reusable unit testing scripts useful for long-term development and maintenance, especially management audiences, don't agree to cost vs. value fact in the unit testing practices.

One of the key limitations of this survey is the limited segmentation of the focus group. Same as any other industry, the software industry also can see the knowledge gap, expertise depends on the working years and different age groups. But Runeson doesn't consider the expertise level or years of experience of contributors for the data collection process. Also, the education level isn't taken into consideration. The knowledge & experience may impact the perspective of this kind of process. Especially sub-process add values to the core process may not identify as important in junior resources. Also, different roles such as project managers do not prioritize the sub-processes which are impacted to project timelines. This limitation of focus group segmentation impacted the end goal of Runeson's survey. Also, that different segmentation feedback will improve the quality of findings and that part itself may become an interesting part of the community perspective analysis. So, this segmentation limitation deviates the final results from reality. If we look at the Nalyzer research perspective, the community-driven approach depends on the individual user's vision and contribution. The way of contributing to society highly impacts this kind of project.

Since this survey mainly focuses on identifying the core definitions and as well as strengths, and problems of unit tests, our intention to identify the capabilities Nalyzer should provide in terms of automated testing. Our solution will mitigate the timeline impacted due to the SDLC sub-process and improve the quality of the testing process. Cost vs Value questioning in survey results, and independence also major problems in traditional automation testing practices. Nalyzer approach is to overcome timeline impact and dependencies which should mitigate to build quality test results during the SDLC processes. Also, we focus to improve the visualization capabilities through the dashboard approach, we believe that will overcome the documentation problems and well-structured metrics allow us to define stop criteria in terms of testing perspective. We refer to the finding of this survey as the primary source of the traditional unit testing cookbook and identify the possibilities to overcome weaknesses, which are Runeson's listed as found in table 1.

2.2. Junit: Unit Testing and Coding In Tandem

In 2005 IEEE Software journal published a journal article about JUnit framework capabilities in modern Java development practices. Panagiotis Louridas at the Department of Management Science and Technology authored this article. Journal authors focused on providing comprehensive guidance about the JUnit framework and its capabilities by comparing competitive products in the current market.

The main reason for Christof Ebert's journal article refers to Nalyzer literature review is to get a deeper understanding of the role of automated tests in software development & the features provided by industry standards tools apart from core functionality. Also, Ebert's journal article provides an exemplary comparison of industry-leading automated test platforms across different programming languages. This comparison includes an overview of features in the industry standards tools & how each tool achieves those features via a different add-on or third-party extensions. Nalyzer research end goal is to become an alternative for existing code analysis tools & traditional test automation, we expect to improve a community-driven approach by referring to existing features already available in standard tools such as Junit, Nunit. The expecting features such as generating reports play a major role in traditional test automation/unit testing frameworks. Plugins such as the JUnitReport Ant task facilitate this type of additional features that deviate from core functionality. But there's a trend across the developer's community to use an embedded tool such as JTest to get comprehensive features that are expecting in the standards automation test domain, instead of referring to another third-party add-on or plugin, Ebert's journal expectation is to highlight these kinds of trends evolving in the modern software development paradigm.

When we need to summarize the functionality more comprehensively, we can find lots of resources on the internet, published by different authors and publications. But Ebert's journal article has a uniquely independent perspective than many other resources available on the internet. Many articles on the internet contain details from a product bias perspective, especially official documentation that does not contain independent comparison and most of the vendors do not reveal the limitations of their product. And when we look at the support tools and third-party plugins which developers use in their day to day life, we need to care

about the perspective of data gathering, which means we should collect data in an open mind and always give priority to collect data from ground level and the high-level management. The reason to consider both ends for gathering data is, the practicality of any tool or process evaluated by the ground level developers and that without their contribution any process or tool does not achieve the highest efficiency in SDLC phases. And high-level management should encourage this kind of process to deviate from the core process but add value in the long term. Because the high-level management is the source fund to the ground level and they also look at the alternatives from a finance perspective. Both ground-level developers and high-level management should balance introducing new tools or processes to the existing system. And the platform which this article published also took it to consider for selecting this journal. Because the source of the resource adds value to the conclusions we listed after studying any journal or article. IEEE is a well-known source for engineering decisions in the industry and academic decisions. Also, the reputation they achieved adds value to the literature review.

The unit testing set can be implemented as three different setups, (a) The way to set up JUnit, (b) Test cases built using JUnit, and (c) Create all test cases in a dynamic way for all test suite. [2]

a)

```
import junit.framework.TestCase;
import junit.framework.Test; import
junit.framework.TestSuite;

public class ComplexTest extends TestCase {

     private Complex a;
     private Complex b;

     protected void setUp() {
        a = new Complex(1, -1);
        b = new Complex(2, 5);
     }
}

public void testComplexEquality() {
     Complex expected = new Complex(1, -1);
     assertEquals(expected, a);
}
```

```
    public void testComplexAddition() {
         Complex expected = new Complex(3, 4);
         assertEquals(expected, a.add(b));
    }

b)  public void testComplexMultiplication() {
         Complex expected = new Complex(1*2 - (-1)*5, 1*5 +
    (-1)*2);
         assertEquals(expected, a.multiply(b));
    }

c)  public static Test suite() {
        return new TestSuite(ComplexTest.class);
    }
```

Figure 3: JUnit test setup & test cases

More broadly, this research provides useful tools and frameworks available in the industry to measure unit testing status. Those are;

1. Clover (www.cenqua.com/clover)
2. Coverage (www.jcoverage.com)
3. GroboUtils (http://groboutils.sourceforge.net)
4. Emma (http://emma.sourceforge.net)
5. NUnit (http://ounit.sourceforge.net/using.html)
6. Jester (http://jester.sourceforge.net)
7. Gcov integration tool (http://gcc.gnu.org).

This research provides a journal that opens a new perspective on Unit testing that is, carefully designed test sets also editing the most effective project documents.

Christof Ebert's journal article provides a comprehensive overview of the unit testing/automated testing concept with examples. In the first section, Ebert is dedicated to summarizing the standards and concepts which we need to focus on when we implement the automation test cases. Steps such as scaffolding, which is the initial variable and object creation, help to maintain a well-structured unit test set, also improve the maintainability of the test pack. And the concept call teardown also helps to finalize the automation on a clean slate. These key concepts improve the maintainability of the unit testing suite, and Ebert's effort to provide a comprehensive view of each concept we should follow in the automation

paradigm. And the second part of the article contains a clear explanation of building the test cases including all the best practices and standards which are required to achieve the maximum throughput from unit testing. Ebert's mentioned three steps we should follow in any test suit; those are;

1. JUnit test setup: Initial variable and object creation
2. JUnit test cases: The assertion process of output
3. Finalizing approach: The teardown process

Also, Ebert's journal article contains a really good section about automation tool legacy and competition, in this section, they thoroughly examine the additional addon and the echo system of Junit. Based on the requirements JUnit provides a complete echo system in which developers can utilize their requirements effectively and efficiently. The open-source culture adds value to the JUnit ecosystem. Few external plugins available in the market are listed below;

1. MockObjects (www.mockobjects.com)
2. JTestCase (http://jtestcase.sourceforge.net)
3. EasyMock (www.easymock.org)
4. JUnitPerf (www.clarkware.com/software/JUnitPerf.html)
5. JUnitDoclet (www.junitdoclet.org)

Each plugging plays an additional, but important role in the unit testing/test automation paradigm. Visualization, data management, object creation, test coverage calculation are the key domains that came up with the Junit ecosystem. This information allows us to open completely different paradigms because the existence of tools depends on various facts, especially the open-source ecosystem adds value to most of the software tools. Our approach should facilitate building separate tools on top of our tool, which will improve the features in the future and allows us to make possibilities to initiate the Nalyzer ecosystem.

Since this journal article mainly focuses on identifying the core practices and as well as functionalities & standards, our intention to identify the capabilities Nalyzer should provide in terms of automated testing. Our solution will mitigate the timeline impacted due to the SDLC sub-process and improve the quality of the testing process. The functionalities &

competition are well explained in the article, and the eco-system also a major plus point in Junit automation testing practices. Nalyzer approach is to overcome timeline impact and dependencies which should mitigate to build quality test results during the SDLC processes. Also, we focus to improve the visualization capabilities through the dashboard approach, we believe that will overcome the documentation problems and well-structured metrics allow us to define stop criteria in terms of testing perspective. We refer to the finding of this journal article as the primary source of the traditional unit testing cookbook and identify the possibilities to survive in the industry without becoming obsolete in near future.

2.3. The Art of Unit Testing (2nd Edition)

Based on the author Roy Osherove's 15-year industry experience, published a 'The Art of Unit Testing' first edition in 2009, with the improvements and additional details 2nd edition came up with more focussing on isolation techniques to mocking objects in programming languages support object-oriented such as Java, .Net.

Since the Nalyzer project's primary objective is to come up with an alternative for existing source code analysis and ensure the software product reliability, this book allows researchers to get an overview of the scope of software automated testing, define the key terms & scope with the help of industry experience. The author Roy Osherove is an industry expert with more than 15 years of working experience in different domains, and he is well known for providing independent consultations and conducting training for major brands in the software industry such as IBM, Microsoft. We can find plenty of good articles on Roy's personal blog ArtOfUnitTesting.com. One of the key reasons to list this book on Nalyzer literature review is to get a holistic idea about the following topics;

1. To know how to build effective and efficient unit test
2. The third-party framework available in the industry for support mocking/analyzing test results
3. Identify the dependency injection practices and theories in the area of unit test paradigm
4. Refactoring practices in traditional projects

Also, this book guides developers to build effectively and efficiently unit testing step by step. The most important aspect of this book is, this is sort of a beginner guide. But within the next couple of chapters, this book guides you to come up with more complicated topics such as mocking objects/build stub, this book references the famous framework like FakeItEasy to teach mocking practices and build well-structured test suites. Most importantly this book explores the era to do testing in legacy applications that have an 'Untestable' component. The Refactability section is focused on that legacy application. And testing in database integrated applications is also a challenge in automated testing. This book covers how to overcome the limitation of the database domain also.

The reason to select the ‘The Art of Unit Testing’ book over many other publications is the comprehensive perspective of content in the automated testing domain. Because this book reveals the areas which most of the publication in this area doesn’t touch. Specifically ‘The process of refactoring’ the way to do the executions in a large legacy project which does not contain any kind of automation testing processes in the initial stage. This is one of the key considerable chapters in the book due to many reasons. Because of the majority of the implementations currently in the software industry in the maintenance stage. Comparatively less amount of projects start from scratch in software development today. And the majority of companies provide support to existing legacy applications that are already developed by completely different companies on different sides of the world. We can see as a current trend Countries like Sri Lanka, India focus to provide Application Support and Manitancy (ASM) as the core project of their companies over developing completely new applications. Since Nalyzer come up as a plugin in CI/CD pipeline, the refactor chapter explaining the real-world picture of the opportunities available in the current market.

The majority of content in this book can be categorized as technical, not just technical but deep-technical. The reason is the above statement is, this book contains 100+ coding samples. The practical examples drive this book into a complete real-world case study. That should be a plus point to studies like a research literature review. But Roy’s perspective is not limited to the technical details. During the flow, he moves to non-technical forms of information and comes up with more detailed explanations. The core purpose of this book is to provide a comprehensive understanding of how to write a unit test, with maintainability, trustworthy & readability. And the flow author maintains a well-defined ‘zero to hero’ aspect with really good examples. According to Roy, his expectation to come up with an IDE independent, language-independent test automation solution with the help of third-party plugins and addons. Another aspect of this book is to introduce the third-party plugins which we can use to improve the test automation experience, each plugging plays an additional, but important role in the unit testing/test automation paradigm. Visualization, data management, object creation, test coverage calculation are the key domains that came up with the Junit ecosystem. This information allows us to open completely different paradigms because the existence of tools depends on various facts, especially the open-source ecosystem adds value to most of the software tools. Our approach should facilitate building separate tools on top

of our tool, which will improve the features in the future and allows us to make possibilities to initiate the Nalyzer ecosystem.

Art of Unit Testing, Program Two leads you to become a champion by writing the very first test POCs using complete test sets that are safe, readable and reliable. Also, the book goes a long way in complex subjects such as comedy and driving, while explaining the use of classification elements (comics) such as Moq, FakeItEasy, and Typemock Isolator. We can do a proper study of test and order patterns, refactor codes in applications, and test how we can test "uncertain" code. Along the way, we will learn about tests related to integration and testing strategies with databases and sub-programs. This book provides a comprehensive overview of improving unit testing skills and achieving a high percentage of unit test availability and industry standards.

In this book following areas are more important;

1. Using stubs to break dependencies
2. Interaction testing using mock objects
3. Isolation (mocking) framework
4. Digging deeper into the isolation framework
5. Working with legacy code
6. Design and testability

The Art of Unit Testing provides end to end guidance to identify the unit test coverage using industry-recommended tools such as TeamCity. This will focus on tracking the progress visible.

All the code snippets and examples in this book are based on the C# programming language, this may be a drawback of this book, since open-source culture plays a considerable role in modern software development, we can’t avoid that contribution in any aspect. And the ecosystem growth over the past couple of years should also be taken into consideration because the majority of software development in the industry depends on open-source platforms. Due to that programming language selection, this book may be limited to a specific audience, and this did not represent the real aspect of the software development community. As we mentioned in 2.3 section the, the open-source ecosystem can’t be avoided

because each plugging plays an additional, but important role in the unit testing/test automation paradigm. Visualization, data management, object creation, test coverage calculation are the key domains that came up with the ecosystem in open source unit testing frameworks. Since they mention C# represents the object-oriented programming audience, such as Java, C++ but the mocking practices, isolation practices may vary from native Java/C++ language.

Since this book's contents mainly focus on identifying the core practices and as well as third-party tools and legacy application implementations, our intention to identify the capabilities Nalyzer should provide in terms of automated testing. Our solution will mitigate the timeline impacted due to the SDLC sub-process and improve the quality of the testing process. The functionalities & competition have been well explained in the book, and the eco-system also a major plus point in automation testing practices. Nalyzer approach is to overcome timeline impact and dependencies which should mitigate to build quality test results during the SDLC processes. Also, we focus to improve the visualization capabilities through the dashboard approach, we believe that will overcome the documentation problems and well-structured metrics allow us to define stop criteria in terms of testing perspective. Also, the possibility to implement test automation practices in traditional legacy code is a considerable aspect of this book. We refer to the finding of this book as one of the primary sources of the traditional unit testing cookbook and identify the possibilities to survive in the industry without becoming obsolete in near future.

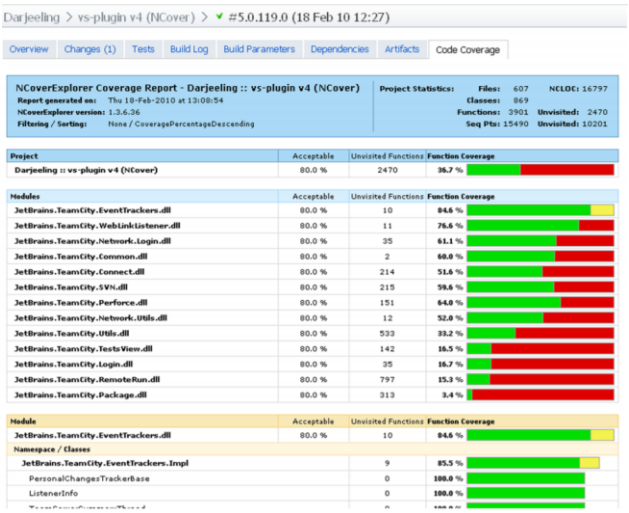

Figure 4: An example of a test-code-coverage report in TeamCity with NCover

2.4. Improving the Bug Tracking System

Microsoft Research Center Redmond, USA and Computer Laboratory, University of Cambridge, UK based 4 researchers published this research paper by focusing on identifying bug reporting in a more informative and descriptive manner to complete resolve bugs quickly.

As a core outcome of the Nalyzer project, we aim to deliver user interactive dashboards, which play a major role in a community-driven approach. The existence of the Nalyzer project depends on the success of the user interactive dashboard. Because based on the user feedback we aim to improve the accuracy of the neural model, for that we have to encourage users to use the Nalyzer maven plugin as much as possible in developers day to day life, this is sort of a win-win approach because while Nalyzer neural network model improves using developer's feedback, we provide AI-enabled analysis with more user-friendly manner to improve the code quality. The 'Improving the Bug Tracking System' research aims to provide a solution for the limitation of the traditional bug tracking system by introducing a proof of concept tracking system. Therefore, we refer to this research to avoid the common mistakes of traditional bug tracking systems and improve the usability of the Nalyzer user interactive dashboard. The usability and the user-friendliness of a system highly impact the growth of a system. In recent years, cloud technologies have values come to the hype of modern software development due to the impact of easy configuration via dashboard concepts. Even cloud technologies are famous across junior developer communities due to the usability and user-friendliness. Microsoft Azure plays an important role in simplifying complex technologies by improving usability and user-friendliness. So, the main purpose of this literature review is to get the scientific approach to improve usability and the user-friendliness of the Nalyzer user interactive dashboard functionality.

Many researchers and surveys conducted aiming to enhance the usability of bug tracking systems and improve the effectiveness of the defect fixing phase of the SDLC process. In the research 'Modeling bug report quality' by P. Hooimeijer and W. Weimer came up with a conclusion in their study which is bugs reported with descriptive comments can resolve faster than fewer comment bugs. [1] The technical aspect of bug reporting is discussed in many types of research, but this research outcome focuses on distributing the responsibility

from developers to the whole team by aiming to reduce the workload of developer's day to day life. They believe a collaborative approach will provide capabilities to isolate bugs from large projects more easily and effectively. Instead of analyzing the current limitations of traditional bug tracking systems, this research develops a proof of concept implementation to come up with a real-world solution. Practical implementations allow researchers to identify the real-world limitation and gap between the theoretical world and the real world. One of the key reasons to select this research for Nalyzer literature review is this practical approach of research outcome. Practicality over theoretical analysis is allowed to reach the developer community by breaking several barriers such as time constraints.

The Hooimeijer and Weimer observations revealed that the descriptive comment makes an impact on the amount of time to resolve bugs. But the way to build more descriptive comments is a bit tricky. Since analytical capabilities are different from another, we can't expect to maintain the same quality of descriptive bug analysis comment from every team member as same. Researchers aim to get the descriptive analysis from reporters into the same format and structure without depending on human capabilities and perspective. The questions they introduced allow us to get the same structured descriptive analysis and isolate code snippets from large software projects. The key challenge researchers try to overcome is to isolate the component which contains the bugs, three out of seven questions aim to get the proper idea about bug location by isolating component & version of the software product. The solution mainly focuses on providing descriptive information about defects by providing answers to the context-sensitive seven questions. Researchers believe that incomplete information directly impacts the time and efficiency of the bug fixing process and providing structured well-defined information about defects directly improves the time and efficiency of bug fixing timelines consumed by the developers.

As bad designed / sophisticated tracking systems are directly affected by this information exchange it is not helpful over time. This research paper addresses concerns about traditional tracking systems for bed bugs in the industry today and suggests four indicators of improvements to reduce the limitations. As proof of concept (POC), researchers also implemented a POC bug tracking platform, this tracking system collects relevant data from the developers and tracks source code which should need attention or fix. [50]

They work on following mentioned ways to improving the bug tracking platform;

- **Tool-centric**- This approach is introduced to the features provided by the defect tracking platform.
- **Process-centric-** This method to bug tracking systems focus on the administration and monitoring of activities related to bug fixing.
- **User-centric-** To user centric method involve both the QA team and the dev team. QA team can be educated on what relevant information to hand over and how to manage it.
- **Information-centric-** Info. centric method focuses the area on the information being provided by the QA team or the relevant parties

Based on the above suggestions researchers present a type of information-based information collection that collects information from journalists to identify candidate files that need to be edited. In prototype implementation, researchers come up with a questionnaire to track bugs during the QA process in a more descriptive manner. These questions are listed below,

1. Identify the severity of the bug
2. Identify the spec. of operating system
3. Identify the component details (ID)
4. Identify the priority of the bug
5. Identify the version of release
6. What is the QA engineer identification (ID/Name)
7. Identify the affected area.

Based on answers, researchers build a decision tree to get the full picture of the situation. The decision tree shows how questions can be minimized where there is an error. Each method corresponds to a series of questions / answers. An encrypted file is an area with a defect. [47]

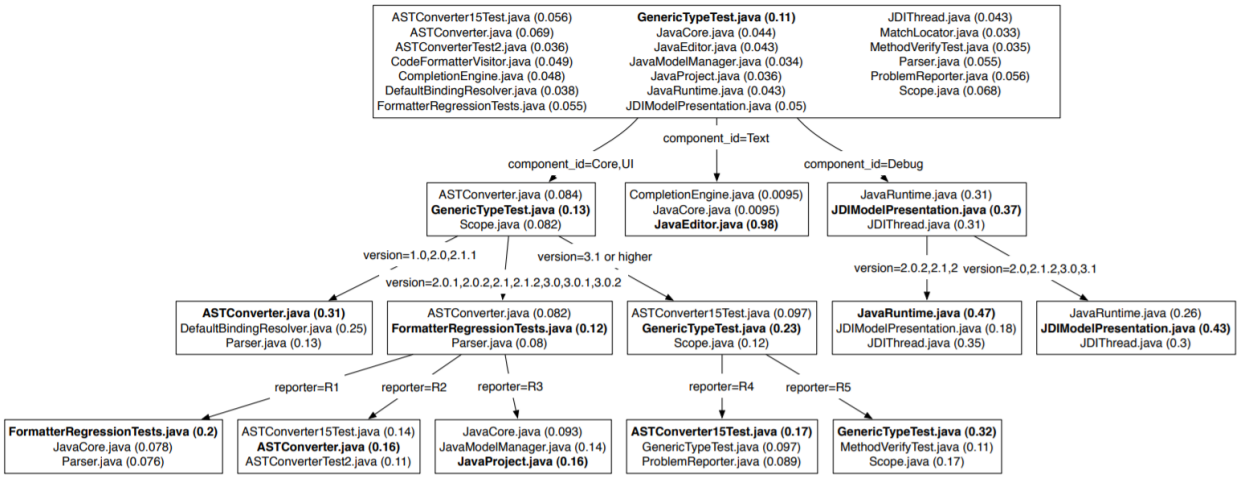

Figure 5: The decision tree used to get the full picture of the situation

This research paper came up with four broad areas, which we should focus on to improve on current bug tracking systems. While using a number of identified improvements from these four areas can be a last resort, this platform instead can choose to specialize, therefore offering different options. Above mentioned approach can be a revolutionary improvement movement of the limitations of current Bug tracking systems where everything provides the same functionality.

One of the major drawbacks of this reporting approach is that the way they evaluate the research outcome, researchers come up with a practical proof of concept implementation but the way of evaluation of their product is not clear in the research content. Specifically, practical implementation in the research should reach the appropriate communities and get feedback from them. This kind of approach allows them to grow the final product and break the practicality barriers. This key limitation of this research is final product reachability to the relevant group for the evaluation purpose. Because like any other industry, the software industry also can see the knowledge gap, expertise depends on the working years and different age groups. The impact of, limitation of, consider the real-world developer community expertise level or years of experience of contributors for the product evaluation process make a considerable impact on the research goal. Also, the education level should be taken into consideration. The knowledge & experience may impact the perspective of this kind of process. Especially sub-process add values to the core process may not identify as

important in junior resources. Also, different roles such as project managers do not prioritize the sub-processes which are impacted to project timelines. This limitation of the evaluation process of research impacted the end goal of the research objectives. This limitation questions the final practicality of the outcome of the research.

Since this research contents mainly focus on identifying the core practices and as well as third-party tools and legacy bug tracking systems, our intention to identify the capabilities Nalyzer should provide in terms of interactive dashboard analysis tracking. Our solution will mitigate the timeline impacted due to the SDLC sub-process and improve the quality of the testing process. The limitations & improvements required to build an efficient bug tracking system are well explained in the research, and the eco-system also a major plus point in automation testing practices. Nalyzer approach is to overcome timeline impact and dependencies which should mitigate to build quality test results during the SDLC processes. Also, we focus to improve the visualization capabilities through the dashboard approach, we believe that will overcome the documentation problems and well-structured metrics allow us to define stop criteria in terms of testing perspective. Also, the proof of concept implementation of bug tracking systems is a considerable aspect of this research. We refer to the finding of this research as one of the primary sources to identify the functionality required for the effective dashboard to reach the developer community by breaking the barriers and identify the possibilities to survive in the industry without becoming obsolete in near future.

2.5. Building Useful Program Analysis Tool Using an Extensible Java compiler

In 2012 Google based Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan addressed the large-scale product company source code maintaining issues through this research paper.

When we segmenting Nalyzer audience from a scale perspective, we can see small, medium, and large scale organizations available in the current industry. This segmentation does not consider the size of projects the organization involves, but the size of organizations. This means we have to consider the behavior of the organizations also. Especially large-scale organizations host multiple small projects in different languages. That will be a challenge to introduce a universal approach to track bugs/any other type of analysis. In this research, Edward Aftandilian and his team's primary focus is to build a customized source code analysis tool for 'Google-Scale' organization. Because they identified in-house tool limitations, such as incompatible with the different language/ uncommon code patterns. Specifically, the dynamic nature of large-scale organizations should be considered in code analysis domain-based projects, when introducing additional steps to the CI/CD pipeline we have to be careful about 'Google-type' scenarios. According to the researchers inside google they work on 20+ changes every 60 seconds and approximately 50% merge happens to the master branch every month. This kind of high-frequency compilation across millions of source code should handle via an extraordinary approach, Edward Aftandilian and the team's approach is to come up with a solution to mitigate the issues when analyzing the different libraries/Programming languages and handle 'Google-scale' CI/CD requirements efficiently and effectively.

We can see most of the studies happen in code analysis paradigm for achieving different goals, specifically in term of identify vulnerable coding patterns, program language specific error patterns, identify insufficient interdependencies between different type of integrations such as database/ESB may available in most of the organization's customized tools, and also research conduct in this same area in the past couple of years. But most of the solution scope is limited to the IDE level, if the project codebase is in that scale, which IDE can handle in a local system on top of its processing power is not practical in 'Google-scale' projects. The uniqueness of this research approach is that they focus on independent the analysis process

from local IDE and cater large scale project source code analysis in the compilation stage in the center CI/CD pipeline. The scale of the code repository and the complexity of implementation add another important object to the research, which should achieve within the scope. That is the performance of the CI/CD pipeline workflow. Performance impact may directly impact the release timeline of a software project and it comes with a cost to the SDLC process. While handling large-scale code-based projects, researchers focus to control the performance of existing release pipelines with different approaches, this is a plus point to Edward Aftandilian and his team's solution. Because we should take into consideration the impact of existing release pipeline performance. The reason is even the solution achieves its core objective, we have barriers to reach the expected audience to the additional workload and time add due to the SDLC process change.

The Edward Aftandilian and team's approach is to utilize the OpenJDK java compiler, which provides capabilities to add customization capabilities into existing compilers using Java Compiler API. The purpose of utilizing the OpenJDK java compiler is to get the maximum advantage of reaching the ground level Java source code without run execution on top of the Java virtual machine layer. This allows improving the performance of the analysis and the compiler level abstract syntax tree generate process also comparatively efficient than involving third-party plugins such as JavaParser. Because the compiler can reach the foundation of java code without additional effort during the compilation stage. The process of analysis is executed on top of third-party analysis tools after extracting the metadata from the abstract syntax tree of the java source code. One of the key important findings of this project is the indexing approach, which they call 'Thindex', to mitigate the challenge of integrating a monolithic large-scale codebase is resolved via the Thindex approach. The algorithm works in the way that, collecting compilation dependencies of project source files. And then do the calculation, for that algorithm refers to the graph traversal using the breadth-first search approach to collect the node and index the files. The Thindex indexing algorithm primarily focuses to enhance the speed of compilation with the help of the IDE index, which will enhance the compilation process effectively and efficiently.

When we get to the big 'Google-Sized' development companies, we re-use their daily development code to get the top product. At Google, in every one minute an average 20+ requirement changes happen, and in every month on average 50% of changes happen to the

main repo. To manage large code repositories, especially in Google scaled companies need customized tools. Most of the customized tools are usually following the ad-hoc build pattern, on top of hard but liable to break easy approaches such as reg. [38] expression and in-house parsers. Dynamic nature of language causes the tools to be obsolete easily. One of the key things is, these ad-hoc tools have limitations to adopt uncommon code patterns. [42]

In this paper the researchers present their experience by re-using the javac compiler APIs in the analysis of a useless system and encouraging the "opening" of production collaborators in expanding frameworks for greater development.

They provide evidence to test how this compiler-based analysis fits into the flow of their work and helps us manage our large Java code.

They report their experience and create source code analysis tools on Google in addition to an external, extensible compiler under open source license. Researchers did analyze three different platforms used in their Java code repo.

1. Strict Java Dependencies - The main focus is to minimize the sizes of the JAR file and QA scope.
2. Error-prone - Define step to code compile stage and automate bug fixes.
3. The index - Introduce a way to minimize indexing burden for large scale project for Java IDE.

In this paper, authors described their experience of implementing customized source code analysis platform on top of the production compiler, using implemented tools to a larger scale code base, and combining the tool into the Google dev build flow. [35] They encouraged developers to prioritize customization capabilities on compilers of future enhancement so that those enhancements could be developed on other engineers.

The limited scope of this research adds drawbacks to the outcome. All the implementations focus on Java programming language and do not provide the approach to extend the scope to the different languages, this may be a drawback of this research, since 'Google-scaled' large companies do not limit to one single programming language, different languages with different capabilities such as typescript, C, GO play a considerable role in modern large scale microservices type software development, we can't avoid that contribution in any aspect.

And the ecosystem growth over the past couple of years should also be taken into consideration because the majority of software development in the industry depends on the different programming languages and different frameworks platforms. Due to that limitation of programming language selection this research may be limited to a specific audience, and this did not represent the real aspect of the large-scale software development scenarios. As we mentioned in above section the, the open-source ecosystem and microservice architecture can't be avoided because each language/framework plays an important role, and bugs or a bottleneck of the implementation will lead to a huge impact on the end-user experience.

Since this research contents mainly focus on identifying the large-scale software development limitations in legacy application implementations, our intention to identify the capabilities Nalyzer should provide in terms of automated testing. Our solution will mitigate the timeline impacted due to the SDLC sub-process and improve the quality of the testing process. The limitation of analyzing large-scale implementation and the way to overcome these limitations are well explained in the research, and the eco-system also a major plus point in automation testing practices. Nalyzer approach is to overcome timeline impact and dependencies which should mitigate to build quality test results during the SDLC processes. Also, we focus to improve the usability and integration capabilities through the maven plugin approach, we believe that will overcome the integration limitations problems and the well-defined format of the maven CI/CD plugin allows us to reach the developer community without barriers. Also, the possibility to implement test automation practices in traditional legacy code is a considerable aspect of this research. We refer to the finding of this research as one of the primary sources of the large-scale product analysis cookbook and identify the possibilities to survive in the industry without becoming obsolete in near future.

2.6. Cute: A Concolic Unit Testing Engine for C

This paper was published in 2005, the Department of Computer Science University of Illinois at Urbana-Champaign based 4 researchers, D.Marinov, K. Sen,, G. Agha published this research paper on ACM SIGSOFT Software Engineering Notes journal, vol. 30.

CUTE: A Concolic Unit Testing Engine for C, the paper focused on providing a solution for unit testing limitations that happens due to pointers-based memory device management during unit testing, this problem occurred in languages like C, which heavily use pointers in device memory management. [46]

When we look at the Nalyzer existence in the industry dependent on the adaptability of different programming languages, since the initial scope primarily focuses on Java programming language, we should check the potential of reach to different programming languages and different developer communities. Because one of our primary objectives is to build universal code analysis tools across all languages and reach a large developer audience as much as possible. But different languages have independent behaviors, when we look at the object-oriented language Java, C# contains unique features at a certain level, but when we look at the languages like C behave at completely different levels. Especially in memory management, we can identify as a key barrier to reach the C developer's community. For that, we have to focus our studies on existing capabilities to overcome the limitation of pointer-based memory management architecture. In this research Koushik Sen and his team focus to identify the possibilities to do a mocking, isolation in C programming language during the unit testing activities. They try to approach the memory graph which contains executable details of the C application and try to mock the variables and method parameters to resolve the limitation of pointer-based memory management.

CUTE research is not just a theoretical approach, it builds a production-ready unit test tool kit with a practical evaluation of the two different projects. To identify the C program application execution paths, this tool has a mechanism to separate the code which is considered for unit testing. Then the application builds the memory map based on the identified path. The build memory map represents the pointer path in a more descriptive manner. Building the memory map is the core functionality of the CUTE project. Once it builds the required memory map, CUTE orchestrates the unit testing execution by feeding

appropriate data to the executable code section. The practical aspect of the CUTE implementation adds value to this research over any other research in the same domain. Also, the way of evaluating the result adds value in terms of the practicality of the implementation. The considered two case studies results show the capabilities of CUTE defect identification with considerable evaluation scores. Also, the functionalities & methods provide a complete experience in a more user-friendly manner. Koushik Sen and his team, introduce JUnit type inbuilt service of methods that can refer to the unit testing activities. The usefulness of the implementation shows the more developer-friendly less work approach, which is not common in programming languages like C. Especially when we deal with internal processing at the operating system level, user-friendliness may decrease frequency due to the complexity of implementation. But CUTE implementation overcomes those types of non-functional features also within the research scope. These kinds of plus points allow Nalyzer researchers to consider this research to our literature review over other researches in the same domain.

The CUTE tool kit provides a 'JUnit' type experience to the users;

1. **CUTE input(x):** This method is used to inject the input into the application.
2. **CUTE input array(p, size):** This method is used to inject the input to the application, and this is specified for creating array type input.
3. **CUTE assume(pred):** This method is C predicate functionality built on top of the inbuilt C function.
4. **CUTE assert(pred).** This functionality is the same as Junit assert function capabilities.

Accessing the hardware level memory component is a bit complicated, and we have to focus on several aspects of the implementation. Otherwise, we will end up with hardware-level faults such as memory leaks. In the practical evaluation, researchers identified the incident such as memory leaks, and few failures in segmentations. During the evaluation process identified memory leak scenarios considered and mitigate the impact of the situation. Overall evaluation plays two different aspects of the research, which will check the impact of implementation and evaluate the results.

In unit testing, the source code was divided into smaller units that were collections of independent functions. The specified unit component can be tested independently of the input parameters for a single input operation. In C, the log function can contain arguments in the cursor, where the conditions that apply to the memory graphs of the unit. The research addressed those limitations inherited as a default unit test with the graphs of memory which is included in C programming language. Previous research work refers to implementing this approach, and in particular, uses such a combination to create experimental inputs to identify all possible possibilities. The current research work uses a method of representation and follow-up issues that follow the performance of an object model with memory graphs such as input stages. In addition, an effective problem resolver suggested further creation of input parameters of the test. As a last step, CUTE, the platform that came with this approach is explained in combination with the finding of using CUTE C-code examples in real-world.
Authors show how CUTE performs tests using a simple example C project. Using the test functionalities testme depicted below the diagram. This operation has an unreachable error given certain installation values. Apparently, in the testme input contains the p and x e arguments values. Nevertheless, the value p is an identifier, so the input includes an access memory graph from to the program pointers. List of cells is a graph in this demonstration.

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int
f(int v) {
  return 2*v + 1;
}

int
testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          ERROR;
  return 0;
}
```

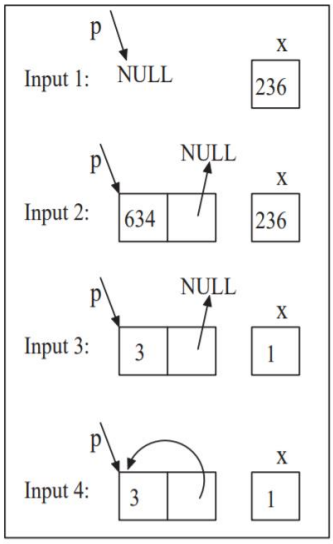

Figure 6: CUTE generates input against C code

This research paper came up with two real-world studies of C-source code testing using CUTE implementation. Preliminary case studies include C code for the use of implementing

this platform. The second one, of the percentage of two previously unknown bugs (infinite loop and coding error) in SGLIB, a famous library in C refer as an industry trading tool. They reported SGLIB errors to SGLIB engineers who corrected them in the next release. Finally, the benefits of using CUTE in addition to traditional testing methods designed are listed below. [32]

1. Pointer in arithmetic operations & casting operations
2. Functions with integration issues
3. Approximating values against different scenarios
4. Functions identified as Black-box
5. Lazy initialization

The process of evaluation contains two case studies, which is one of the key limitations of this research. Because evaluation should cover the majority of possible aspects in the real world. But the CUTE project is limited to 2 real-world scenarios and those scenarios are also not selected from the independent source. That would be another questionable decision of the evaluation process. Because the projects considered for evaluation should reflect the real-world scenarios as much as possible, at least 10 projects from different vendors/ public repositories should be used for the evaluation in terms of maintaining the independence of evaluation results. Also, the domain of the project should take into account, when we consider a programming language we can see different projects available in different domains, such as networking, distributed systems, standalone applications, web-based applications. Even for all applications built on top of the same programming language, the different domain programming practices may contain completely different practices, naming conventions, and standards. Because of that, when research outcome evaluate against a software project we have to strictly follow the following rules;

1. Identify the project in a different domain (Distributed computing, Database)
2. Identify the project maintained by different vendors (IDM, Oracle)
3. Focus on both open-source/different licensing versions
4. The scalability of the project should be taken into consideration

Since this research contents mainly focus on identifying the limitation of mocking and isolating the component of C programming language-based implementations, our intention to identify the capabilities Nalyzer should provide in terms of automated testing in different programming languages. Our solution will mitigate the timeline impacted due to the SDLC

sub-process and improve the quality of the testing process. The limitation of analyzing programming languages that maintain different memory management and the way to overcome these limitations is well explained in the research, and the eco-system also a major plus point in automation testing practices. Nalyzer approach is to overcome timeline impact and dependencies which should mitigate to build quality test results during the SDLC processes. Also, we focus to improve the usability and integration capabilities through the maven plugin approach, we believe that will overcome the integration limitations problems and the well-defined format of the maven CI/CD plugin allows us to reach the developer community without barriers. Also, the possibility to implement test automation practices in traditional legacy code is a considerable aspect of this research. We refer to the finding of this research as one of the primary sources of the large-scale product analysis cookbook and identify the possibilities to survive in the industry without becoming obsolete in near future.

2.7. A Comparison Of Bug-finding Tools for Java

In the 2004 University of Maryland, College Park-based 3 researchers Nick Rutar, Christian B. Almazan, Jeffrey S. Foster published this research at the 15th International Symposium on Software Reliability Engineering (ISSRE'04), IEEE Computer Society Press.

Nalyzer project aims to overcome the limitations of the traditional code analysis tool we have to refer to the studies focus to identify tradeoff in traditional static code analysis tools in a more broad perspective. In modern software development, automated static code analysis is a fundamental step in CI/CD pipelines. Different tools and frameworks are used to get an analysis report in most of the industry projects as well as in day-to-day experiments. But in most scenarios developers and project management choose the tool/framework without doing scientific research. Since we intend to identify the limitations of traditional static code analysis tools and provide solutions to overcome the bottleneck, we refer to this research to identify the fats from more scientific and analytical perspectives. Nick Rutar and his team consider different industry standards tools such as PMD, FindBugs, JLint, and ESC/Java 2, to their studies, those tools are well-reputed in the industry for the past couple of years. Many industry leaders such as IBM, Oracle prefer those tools in their projects. The most important point is the limitation of the real analysis against those tools, still, the industry prefers one of the tools available on the market based on their personal opinion. The majority of the tools are in open source licenses, the decision does not impact the financial aspect of the project. Due to that reason, personal perspective brings choice to the table.

When we need to summarize the functionality & limitations more comprehensively, we can find lots of resources on the internet, published by different authors and publications. But Nick Rutar and his team's research has a uniquely independent perspective than many other resources available on the internet. Many research/articles on the internet contain details from a product bias perspective, especially official documentation that does not contain independent comparison and most of the vendors do not reveal the limitations of their product. And when we look at the support tools and third-party plugins which developers use in their day to day life, we need to care about the perspective of data gathering, which means we should collect data in an open mind and always give priority to collect data from ground level and the high-level management. The reason to consider both ends for gathering

data is, the practicality of any tool or process evaluated by the ground level developers and that without their contribution any process or tool does not achieve the highest efficiency in SDLC phases. And high-level management should encourage this kind of process to deviate from the core process but add value in the long term. Because the high-level management is the source fund to the ground level and they also look at the alternatives from a finance perspective. Both ground-level developers and high-level management should balance introducing new tools or processes to the existing system. And the platform in which this article was published is also considered for selecting this journal. Because the source of the resource adds value to the conclusions we listed after studying any journal or article. IEEE is a well-known source for engineering decisions in the industry and academic decisions. Also, the reputation they achieved adds value to the literature review.

In this research, they have analyzed five different code analysis tools. And their studies focus on identifying the overlapping scenarios across the different tools, during the study they notice most of the warnings listed by the tools distinct. And the comprehensive tool they implement using all five application's final results, allows developers to pay attention to the classes which contain comparatively high distinct warnings. And the annotation mechanism they introduced also helps developers to manage the result in different situations and the meta-tool also has the capabilities to take actions based on the customizable annotated tool. Such a mechanism improves the usability of the tools, we also prioritize the key objectives of the project and do not build barriers on the path of SDLC in any case. The study is comparing the analysis results, which are generated from five different tools. As a future work they record the finding, also the analysis results comparison provide bord analysis capabilities to this domain.

Errors are difficult to find and fix in software products. Over the last few years, different solutions were introduced for identifying bugs by analyzing source code or statistical source code (compilation stage) as automated tools. Different strategies in the industry have different limitations, but the value added to the productivity level of these instrument tradeoffs is not understood in the relevant study. In this paper, the researchers looked at five bug detection tools in the industry, most notably FindBugs, Bandera, PMD, JLint, ESC/Java 2 in various Java applications. [27] Using the tools mentioned above, the researchers were able to review these tool reports and warnings after analysis. Their test results showed that

no tool results strictly follow other tool results, and indeed tool results often detect non-specific bugs. Researchers discuss how and how the majority of these tools are based, and they propose how specific strategies and strategies affect the release of the final results of the tool. Eventually, they came up with a meta-tool to provide analysis collaboratively specified to warn about method and class file by looking at source code.

By about 2004, a number of tools were developed to automatically detect bugs in software products, based on methodologies like synthetic pattern matching, analysis flow of data, model testing, proven theorems, type system and checking model. Majority of these approaches test similar types of editing errors, to date, with direct comparisons between them. Authors of this research execute comprehensive analysis of Java bug detection solutions for different tasks. [25]

To produce results in this paper, researchers developed a pack of scripts which cover and guide the analysis result of different industry tools. Cooperatively, developed scripts constitute the first version of the bug detection tool mentioned above. Developed Meta tools give capabilities to the developer to evaluate the effect on all solutions in the same format and to determine which categories, methods, and lines generate alerts. [44]

Table 2: Finding of analysis tools

Bug Category	**Example**	ESC/Java	FindBugs	JLint	PMD
General	Null dereference	√*	√*	√*	√
Concurrency	Possible deadlock	√*	√	√*	√
Exceptions	Possible unexpected exception	√*			
Array	Length may be less than zero	√		√*	
Mathematics	Division by zero	√*		√	
Conditional, loop	Unreachable code due to constant guard		√		√*
String	Checking equality using == or !=		√	√*	√
Object overriding	Equal objects must have equal hashcodes		√*	√*	√*
I/O stream	Stream not closed on all paths		√*		
Unused or duplicate statement	Unused local variable		√		√*
Design	Should be a static inner class		√*		
Unnecessary statement	Unnecessary return statement				√*

√ - tool checks for bugs in this category * - tool checks for this specific example

Table 3: Each tool generated warnings

Name	NCSS (Lines)	Class Files	Time (min:sec.csec)				Warning Count			
			ESC/Java	FindBugs	JLint	PMD	ESC/Java	FindBugs	JLint	PMD
Azureus 2.0.7	35,549	1053	211:09.00	01:26.14	00:06.87	19:39.00	5474	360	1584	1371
Art of Illusion 1.7	55,249	676	361:56.00	02:55.02	00:06.32	20:03.00	12813	481	1637	1992
Tomcat 5.019	34,425	290	90:25.00	01:03.62	00:08.71	14:28.00	1241	245	3247	1236
JBoss 3.2.3	8,354	274	84:01.00	00:17.56	00:03.12	09:11.00	1539	79	317	153
Megamek 0.29	37,255	270	23:39.00	02:27.21	00:06.25	11:12.00	6402	223	4353	536

In this paper, the researchers aim to do a comparison of analysis generated by different source code analysis applications. As a future work, the team is interested in collecting more information about real errors in the system, which will allow researchers to better identify false negatives and false positives. Gathered analysis data can be utilized to identify how those approaches accurately identify errors in its measurements. Investigators may also examine. Using the metrics suggested by this paper can use to identify the result of two separate tools.

The limited scope of this research adds drawbacks to the outcome. All the implementations focus on Java programming language and do not provide the approach to extend the scope to the different languages, this may be a drawback of this research, since the majority of the companies does not limit to one single programming language, different languages with different capabilities such as typescript, C, GO play a considerable role in modern large scale micro services type software development, we can't avoid that contribution in any aspect. And the ecosystem growth over the past couple of years should also be taken into consideration because the majority of software development in the industry depends on the different programming languages and different frameworks platforms. Due to that limitation of programming language selection this research may be limited to a specific audience, and this did not represent the real aspect of the large-scale software development scenarios. As we mentioned in above section the, the open-source ecosystem and micro service architecture can't be avoided because each language/framework plays an important role, and bugs or a bottleneck of the implementation will lead to a huge impact on the end-user experience.

Since this research contents mainly focus on identifying the tradeoff of java static analysis tools, our intention to identify the capabilities Nalyzer should provide in terms of automated

testing. Our solution will mitigate the timeline impacted due to the SDLC sub-process and improve the quality of the testing process. The limitation of using the non-AI-based static analysis tool and the way to overcome these limitations are well explained in the research, and the eco-system also a major plus point in automation testing practices. Nalyzer approach is to overcome timeline impact and dependencies which should mitigate to build quality test results during the SDLC processes. Also, we focus to improve the usability and integration capabilities through the maven plugin approach, we believe that will overcome the integration limitations problems and the well-defined format of the maven CI/CD plugin allows us to reach the developer community without barriers. Also, the possibility to implement test automation practices in traditional legacy code is a considerable aspect of this research. We refer to the finding of this research as one of the primary sources of the large-scale product analysis cookbook and identify the possibilities to survive in the industry without becoming obsolete in near future.

2.8. Pragmatic Unit Testing In Java 8 with JUnit

This book was published in 2015, J. Langr, A. Hunt, D. Thomas, and S. Davidson Pfalzer are the authors of Pragmatic unit testing in Java 8 with JUnit with a focus to provide complete guidance in the Unit testing paradigm. [26]

This book suggests holistic guidelines to implement industry standards unit test coverage in Java 8 software projects. For NALYZER implementation, we focus on chapter 5: FIRST Properties of Good Tests.

Since the Nalyzer project's primary objective is to come up with an alternative for existing source code analysis and ensure the software product reliability, this book allows researchers to get an overview of the scope of software automated testing, define the key terms & scope with the help of industry experience. The author Jeff Langr is an industry expert with over three decades experience of working experience in different domains, and he is well known for providing independent consultations and conducting training for major brands in the software industry such as IBM, Microsoft. Recently Jeff is working at Outpace Systems, Inc. One of the key reasons to list this book on Nalyzer literature review is to get a holistic idea about the following topics;

1. The efficient way to build your unit tests.
2. The way to maintain a clean systemic approach to maintain the unit test.
3. The way to test uncomfortable stuff.
4. The key points we need to focus on unit test writing.
5. How to achieve the highest benefit of unit test.
6. And the example is one of the key important parts of this book

Also, this book guides developers to build effectively and efficiently unit testing step by step. The most important aspect of this book is, this is sort of a beginner guide. But within the next couple of chapters, this book guides you to come up with more complicated topics such as mocking objects/build stub, this book references the famous framework like FakeItEasy to teach mocking practices and build well-structured test suites. Most importantly this book explores the era to do testing in legacy applications that have an 'Untestable' component. The 'How to achieve the highest benefit of unit test' section is

focused on that legacy application. And testing in database integrated applications is also a challenge in automated testing. This book covers how to overcome the limitation of the database domain also.

The reason to select the 'Pragmatic Unit Testing in Java 8 with JUnit' book over many other publications is the comprehensive perspective of content in the automated testing domain. Because this book reveals the areas which most of the publication in this area doesn't touch? Specifically 'How to achieve the highest benefit of unit tests the way to do the executions in a large untestable project which does not contain any kind of automation testing processes in the initial stage. This is one of the key considerable chapters in the book due to many reasons. Because of the majority of the implementations currently in the software industry in the maintenance stage. Comparatively less amount of projects start from scratch in software development today. And the majority of companies provide support to existing legacy applications that are already developed by completely different companies on different sides of the world. We can see as a current trend Countries like Sri Lanka, India focus to provide Application Support and Maintains (ASM) as the core project of their companies over developing completely new applications. Since Nalyzer come up as a plugin in CI/CD pipeline, the refactor chapter explaining the real-world picture of the opportunities available in the current market.

The majority of content in this book can be categorized as technical, not just technical but deep-technical. The reason is the above statement is, this book contains 100+ coding samples. The practical examples drive this book into a complete real-world case study. That should be a plus point to studies like a research literature review. But Roy's perspective is not limited to the technical details. During the flow, he moves to non-technical forms of information and comes up with more detailed explanations. The core purpose of this book is to provide a comprehensive understanding of how to write well-defined with maintainability, trustworthy & readability. And the flow author maintains a well-defined 'zero to hero' aspect with really good examples. According to Roy, he expects to come up with an IDE independent, language-independent test automation solution with the help of third-party plugins and addons. Another aspect of this book is to introduce the third-party plugins which we can use to improve the test automation experience, each plugging plays an additional, but important role in the unit testing/test automation paradigm. Visualization, data management,

object creation, test coverage calculation are the key domains that came up with the Junit ecosystem. This information allows us to open completely different paradigms because the existence of tools depends on various facts, especially the open-source ecosystem that adds value to most of the software tools. Our approach should facilitate building separate tools on top of our tool, which will improve the features in the future and allows us to make possibilities to initiate the Nalyzer ecosystem.

In this book, the FIRST approach has proposed as a protocol to avoid the many pitfalls that unit inspectors often enter through the FIRST principles of unit testing:

- Fast

  The distinction between fast and slow testing is a challenge - as Justice Potter Stewart put it, "I know when I see it." The rapid unit test works perfectly in the source code and takes a few milliseconds to perform. Slow unit testing is compatible with the source code which should handle external external requirements such as DB, network calls, IO performance those tasks take a long time.

- Isolated

  Good unit tests scope on a small chunk of source code to verify. That’s in line with the definition of the small unit. The more source code that your test interacts with, directly or indirectly, the more things are likely to go awry.

- Repeatable

  Tests don’t come without a little air — you’re the one who gets the design, which means they’re completely under your control. You have the power to set up test terms, which also means you don't need a crystal ball to know what the test result should be. Part of your job in the design of the test, then, is to provide a guarantee that explains what the outcome should be each time the test is performed.

- Self-validating

  Tests aren't proper tests when they assert that things went as expected. Developers automate unit tests to save their time, not take more of implementation time. Manually verifying the unit test results is a considerable

time-consuming process that can also introduce unnecessary risk—it's easy to get dozy and gloss over important signs when you pore over the voluminous output that pseudo tests can produce.

- Timely

  Unit testing is a good practice. With so many good habits that engineers have not yet fully established, such as brushing your daily hygiene habits, it is easy to write more and make excuses as to why you can skip tasks "at once." Your dentist may love your support for his or her practice, but you will hate the time it takes to remove the built-up tartar.

As a summary of the chapters, the book concludes that writing test units requires a lot of investment in startup time. While the tests can pay off as an investment, every test engineer writes a lot of code that you have to keep. Be aware of that investment by ensuring that unit testing maintains high quality. Use the FIRST summary to remind you of the features of the quality test. All the code snippets and examples in this book are based on Java programming language, this may be a drawback of this book, since micro services culture plays a considerable role in modern software development, and we can't avoid that contribution in any aspect. And the ecosystem growth over the past couple of years should also be taken into consideration because the majority of software development in the industry is moving from monolithic to micro services platforms. Due to that programming language selection, this book may be limited to a specific audience, and this did not represent the real aspect of the software development community. As we mentioned in 2.3 section the, the micro services ecosystem can't be avoided because each plugging plays an additional, but important role in the unit testing/test automation paradigm. Visualization, data management, object creation, test coverage calculation are the key domains that came up with the ecosystem in open source unit testing frameworks. Since they mention Java represents the object-oriented programming audience, such as Java, C++ but the mocking practices, isolation practices may vary from native C#/C++ language.

Since this book's contents mainly focus on identifying the core practices and as well as third-party tools and legacy application implementations, our intention to identify the capabilities Nalyzer should provide in terms of automated testing. Our solution will mitigate the timeline

impacted due to the SDLC sub-process and improve the quality of the testing process. The functionalities & competition are well explained in the book, and the eco-system also a major plus point in automation testing practices. Nalyzer approach is to overcome timeline impact and dependencies which should mitigate to build quality test results during the SDLC processes. Also, we focus to improve the visualization capabilities through the dashboard approach, we believe that will overcome the documentation problems and well-structured metrics allow us to define stop criteria in terms of testing perspective. Also, the possibility to implement test automation practices in traditional legacy code is a considerable aspect of this book. We refer to the finding of this book as one of the primary sources of the traditional unit testing cookbook and identify the possibilities to survive in the industry without becoming obsolete in near future.

2.9. BP Neural Network-based Effective Fault Localization

In 2009 Department of Computer Science, the University of Texas, based on two researchers W. Eric Wong and Yu Qi published this research in the International Journal of Software Engineering and Know Nalyzer project aims to overcome the limitations of traditional source code analysis tools through a machine learning approach. When we look at the trends to use machine learning practices in the code analysis paradigm is a mature topic, a considerable amount of research available in the industry to aim to get the help of machine learning processes to the source code analysis. To identify the trends in the modern software development process our literature review focus to analyse the solutions currently available on the market and study the theoretical aspect of the current findings. We have one of the core objectives of this literature review is, identify the real reason that ML technologies still do no trend in the practical level software development process. Even most of the industries get the help of machine learning techniques, but the most closed industry to machine learning still does not get the 100% benefits of ML techniques. We have a couple of theoretical approaches and studies, but still, we can't see the ground-level usage in day-to-day life in the software industry. Therefore, we select this research and try to figure out the real reason ML solutions still do not reach the practical world, and what is a way to develop sustainable ML solutions without becoming obsolete in near future. Also in the Nalyzer project scope, we did not focus on analyzing different algorithms available in the Machine Learning domain. Also, the next couple of literature reviews we use as a source to choose the Nalyzer neural network model algorithm. Based on the achievement of different algorithms in the code analysis domain we intend to get maximum beneficial algorithm by comparing the next couple of research literature reviews.

One of the key findings of this paper is that the algorithm selection and the techniques used to overcome the typical neural network issues such as over-fitting problems. This research allows to pre-identify the problems that may occur during the neural network implementation as well as the process of data collection and analysis methods. Athe also, the way W. Eric Wong and his team evaluated the efficiency of the algorithm and explained its internal process is one of the key outcomes of the research. Specifically projects in the same domain beneficial for referring W. Eric Wong and his team's approach to evaluate the finding in the neural networks paradigm. This study contains a comprehensive comparison across BPNN,

TBest, and TWorst approaches in a more descriptive manner and the range of projects W. Eric Wong and his team have the potential to replicate the real-world scenarios happening in the modern software development paradigm. The alignment of data representation also a plus point in this research over other research in the studies. Specifically research on the neural networks domain we need to represent the finding in a format where balancing the mathematical aspect of the solution. Also, the evaluation of the result should be in a more comprehensive perspective, and the source of evaluation should represent the real-world aspect with the appropriate case studies.

Another major plus point of this W. Eric Wong and his team's approach is the way they refer to projects in different languages. Programming language independence is a major aspect of the universal solution. Most of the research/journal articles are limited to one single programming language, which is impacted to get the overall view of the opportunities available in the code analysis domain. A wide range of case studies provides a real-world aspect of the situation and allows other researchers to do their studies on top of their findings. As an evaluation of the algorithm, researchers mention 3 reasons for selecting a backpropagation neural network for this studies;

1. Back Propagation neural networks show the capabilities in different areas within the software development paradigm.
2. The capabilities to predict complex nonlinear functions is one of the key reasons for selecting a backpropagation neural network. As an example: the BP network use in simulating the relationship between success or failure scenarios.
3. The backpropagation algorithm refers to the supervised learning algorithm. Therefore, a neural network is more suitable for identifying defects in a large codebase.

In this study, the researchers faced the limitations of actual error detection using back-to-back network (BP) implantation, machine learning model used successfully in software cost forecasting, risk analysis, and industry reliability measurement, to help developers successfully identify system errors. Because in manual debugging, a local error identifies the source code line number and the program error column number. TO identify these bugs via the ad-hoc method or on top of developer's oriented approaches has considerable time constraints.

Researchers found the following advantages over the neural network over other models;

1. Due to their expressiveness, models have capabilities to learn complex combinations.
2. They are very strong in sound training details. Because weights on the connection data are distributed, comparatively less errors in the training data have less effect on the model.
3. Models have capabilities to capture time-variant models.

BP neural network architecture is shown in the following figure.

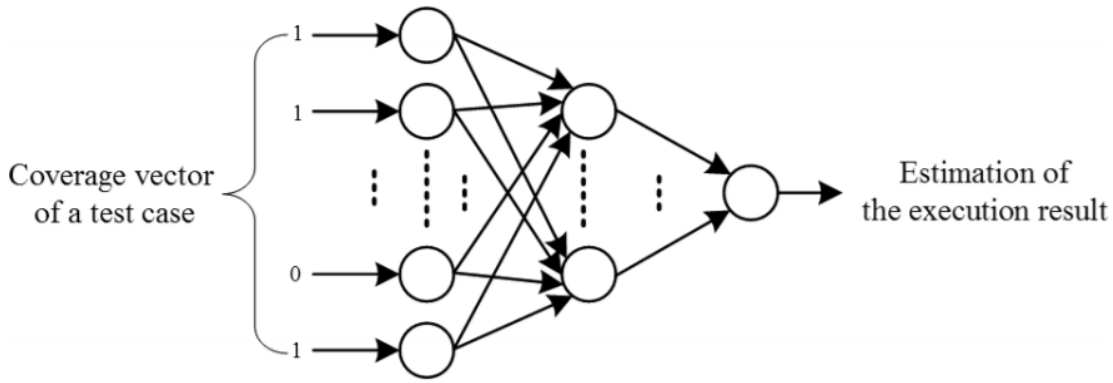

Figure 7: The BP neural network used in the research method

Four case studies using grep and gzip, Siemens suite and Unix suite were performed to illustrate the efficiency and capabilities of selecting a back-to-back distribution of the neural network process in error detection and location. Such efficiency and effectiveness are measured by a scoring method called EXAM based on the percentage of usable statements in the source code to be tested until the first bug-infested statement is reached.

The four case studies using a variety of programs (UNIX suite, Siemens suite, grep and gzip) are designed to illustrate the compatibility and efficiency of BPNN in local error identification. The results of these studies show that the back-end network implementation method is more efficient and compatible than Tarantula (one of the famous defect identification approaches refer BPNN type inputs) the reason is the first one tests the code smaller than the last before the first error is found in the source code. However, their conclusion may not be considered in the dispute process without further testing. Perform,

extra case studies in applications in different domains should be used in the future work of the researcher.

Overall this research doesn't contain major weaknesses, as mentioned in previous paragraphs, the flow of arranging the finding is well structured and descriptive. Researchers in the same domain should use it as a reference to structure the neural network research findings in their studies. The structure of the data arranged in the following way,

1. Summary of the referencing software project; include a small description of the project and the test case summary.
2. The comparison I: Analyze to identify the effectiveness by comparing the three different approaches.
3. Comparison II: Comparison in findings against the total number of executions statement.

As we mentioned earlier this arrangement provides an extensive picture of findings and the final result.

But still, the approach has weaknesses due to the selection of projects, because the comparison should focus on the different coding patterns, naming conventions, and the coding standards maintained by different organizations. The three projects do not reflect the real-world code patterns and practices. For the Nalyzer model evaluation section, we try to address this weakness of the evaluation process by looking at the impact of W. Eric Wong's presentation of findings.

Since this research contents mainly focus on identifying the possibilities to use ML approaches in the code analyzing domain, our intention to identify the capabilities Nalyzer should provide in terms of ML technologies. Our solution will mitigate the timeline impacted due to the SDLC sub-process and improve the quality of the testing process. The limitation of using the non-AI-based static analysis tool and the way to overcome these limitations are well explained in the research, and the eco-system also a major plus point in automation testing practices. Nalyzer approach is to overcome timeline impact and dependencies which should mitigate to build quality test results during the SDLC processes. Also, we focus to improve the usability and integration capabilities through the maven plugin approach, we

believe that will overcome the integration limitations problems and the well-defined format of the maven CI/CD plugin allows us to reach the developer community without barriers. Also, the possibility to implement test automation practices in traditional legacy code is a considerable aspect of this research. We refer to the finding of this research as one of the primary sources of the large-scale product analysis cookbook and identify the possibilities to survive in the industry without becoming obsolete in near future.

3. METHODOLOGY

To overcome the limitation of manual code reviews and traditional automated code analysis tools, Nalyzer approach is to build Neural Network Model (NN) model to identify the buggy code patterns & vulnerable code snippiest and facilitate the analysis as a stage in the continuous integration/continuous delivery (CI/CD) pipeline. The method section's subtopic arrangement dedicated to discussing gathering training data, building the model & building the analysis report components deeply.

For this implementation, we primarily focus on the Java programming language which follows the 6 monthly release cadence since September 2017. Also, we use one of the leading static code analysis tools in Java programming called PMD for gathering training data. To parse the source code we used a java parser in the Nalyzer development kit & wrap the whole implementation inside maven customize plugging. In section II will discuss the methodology of the Nalyzer data gathering approach and tools.

To build a Neural Network Model (NN) we use the Tensorflow 2.0 platform which is run on top of NVIDIA GPU instances, this approach allows us to maximize the efficiency of training the model. Section III will explain the methodology of the Nalyzer building ANN model and the hardware arrangement.

As a stage of the CI/CD pipeline, the Nalyzer analysis report can be generated into the source code locations. The interactive dashboard provides a holistic overview of buggy code patterns and vulnerabilities in the source code of the project. Also, the developer community can be involved to train the model via the analysis dashboard by providing feedback & adding sample code patterns. Developer community involvement will improve the quality of training data and it will make sure to survive Nalyzer analysis tools in the industry without being obsolete due to the frequently released cadence of modern programming languages. In section, IV will discuss the methodology of the Nalyzer maven plugin implementation and dashboard approach.

Figure 8 depicting the proposed system from a technical perspective.

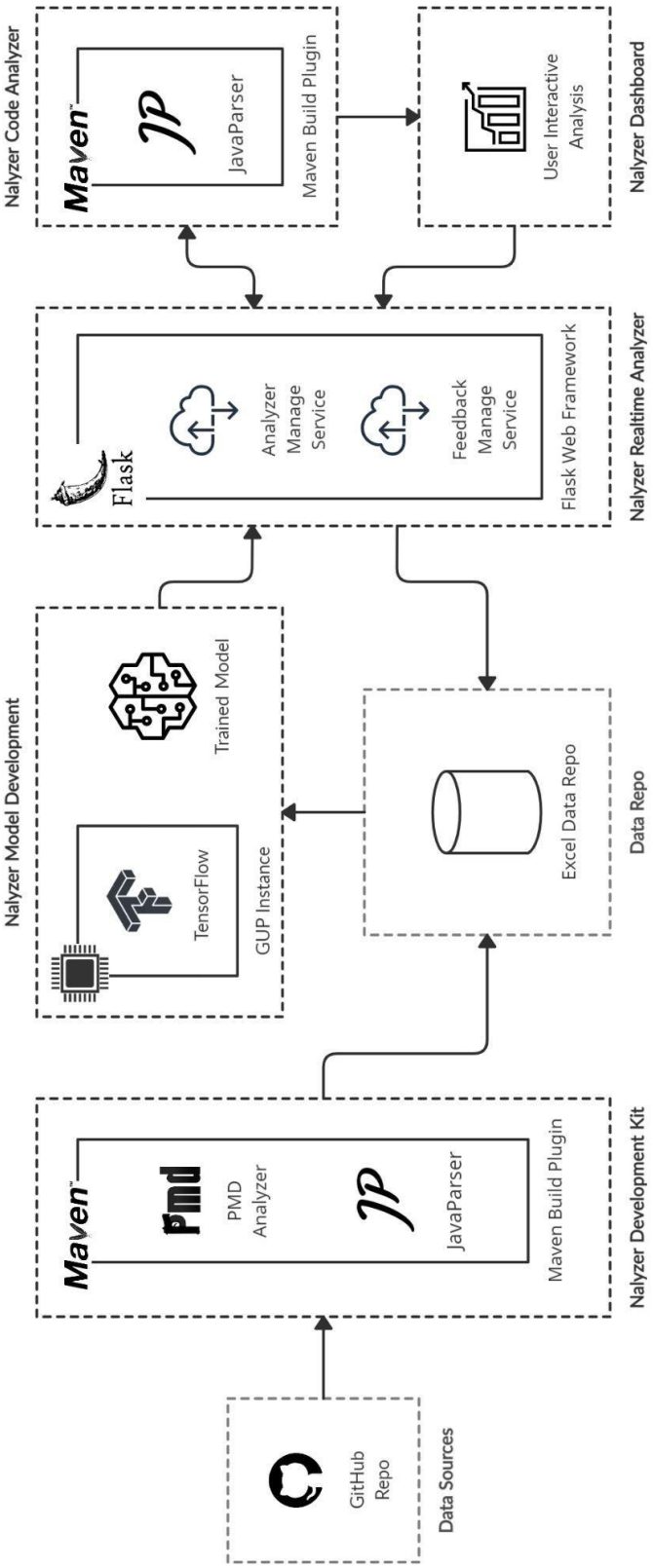

Figure 8: High-Level Architecture Diagram

High-level system architecture contains 5 different independent components, which is identified as an independent subdomain of the solution. Each component is responsible for a unique task. Component-wise development improves the maintainability and improvements focusing to develop compatible source code analysis maven plugins for different programming languages and train different models for different versions of programming languages and frameworks in the future. High-level sub-components in the solution are,

1. Nalyzer Development Kit
2. Nalyzer Model Development
3. Nalyzer Real-Time Analyser
4. Nalyzer Code Analyzer
5. Nalyzer Dashboard

Dynamic nature of programming languages and frameworks, obsolete the training models within the small time window, Also compatibility of source code metadata analyzer also considered as a high dynamic component in the system, due to frequent releases in the software industry. Each attribute in sub-components will be described in the following sections.

3.1. Data Gathering

One of the key challenges in the Nalyzer project is data gathering. Because the Neural Network Model (NN) requires huge amounts of data to function properly. After finding different approaches and implementing different Proof of Concepts (POC) projects, researchers decide to build customized Maven plugging by wrapping a customized PMD static code analysis tool that has the capability to identify twelve (12) different buggy code patterns. Also, this maven plugin has the capability to extract the Abstract Syntax Tree of java source code and convert those syntaxes into machine learning-oriented numeric values using a hash function.

We call this Maven plugging ‘Nalyzer-Development-Kit’ and we will make the source code of this tool publicly available for researchers who are interested in doing researches in the same domain. (Github - https://github.com/Denuwanhh/nalyzer)

3.1.1. Customized maven plugin

Maven is one of the leading build automation tools used primarily for Java projects. For Nalyzer-Development-Kit researchers goal to implement a customized maven goal, which can execute during the project compile stage and generate training data and store.

Maven build automation tool provides customization capabilities & build customized goal using ‘maven-plugin-annotations’ library. Nalyzer-Development-Kit (NalyzerDK) maven plugin requires two non-mandatory parameters which help to generate training data set in a different segmentation. Also, these options help to handle the imbalance of the training data set. Because identified buggy code patterns are comparatively less to the non-buggy code patterns extracted during the maven plugin execution. Using these options researchers can manage the data set by fine-tuning the extraction scope.

The following command can be used to generate test data using NalyzerDK and table (4) listed NalyzerDK options.

Table 4: Maven command for run NalyzerDK

```
mvn nalyzerdk:generate-data -D<Parameter>=<Option>
Ex: mvn nalyzerdk:generate-data -Dscope=all -
Dhomedir=P
```

Table 5: NalyzerDK options

Parameter	**Default Option**	**Option**	**Description**
scope	all	all	Store all the buggy and non-buggy code patterns identified during the analysis

		bugs	Store all the buggy code patterns identified during the analysis
		non-bugs	Store all the non-buggy code patterns identified during the analysis
homedir	G	G	Store identified both buggy and non-buggy code patterns into a common project independent CSV file location. File Location: Users Home Directory\nalyzer\data-repo.csv
		P	Store identified buggy and non-buggy code patterns separately in a project-dependent CSV file location. File Location: Project Directory\target\nalyzer\<buggy/non-buggy>/<id>.csv

3.1.2. Customized PWD plugin

PMD one of the leading is a static source code analyzer in the industry. PMD analyzer is generally used to identify coding best practices, code style issues, design issues, and so forth. For Nalyzer-Development-Kit mainly concerned about four major violations category from PMD Java build-in rules. Those categories are;

1. Error-Prone: Rules to consider about constructs that are either broken implementation, extremely confusing code snippet, or code patterns prone to runtime errors.
2. Multithreading: Rules that consider issues when occurring multiple threads of execution.
3. Performance: Rules that consider suboptimal code.

4. Security: Rules that consider potential security vulnerabilities.

In order to creep the research scope, researchers decide to pick 12 frequent violations identified in Java-based projects. That approach helped to improve the quality of training data. Those rules are listed below.

1. Use Concurrent Hash Map
2. Avoid Catching Throwable
3. Do Not Use Threads
4. Too Few Branches For A Switch Statement
5. Append Character With Char
6. Consecutive Appends Should Reuse
7. Avoid Instantiating Objects In Loops
8. Null Assignment
9. Avoid Using Short Type
10. Avoid Literals In If Condition
11. Avoid Using Volatile
12. Redundant Field Initialize

3.1.3. Java Parser

To store identified buggy/non-buggy code patterns in a structured machine learning-oriented format is another key function in the Nalyzer-Development-Kit. To get the structured source code, Nalyzer uses an open-source Java source code parser project called JavaParser to generate Abstract Syntax Tree (AST) of the java class file.

Before generating the AST of the source file, NalyzerDK removes all the comments on the source code. Then generate the AST. The NalyzerDK data storing process store the type of syntax against the syntax such as ClassOrInterfaceType, SimpleName, BinaryExpr. Also, NalyzerDK consumes the syntax location data, line no & collum no for extract the code snippet. Table (6) represents the 1D array format against a simple Java code Snippet.

Once the PWD library identifies a violation on the source code, using JavaParser execute BreadthFirst traversal on AST and extract the sub-tree which includes buggy-code snippets. Then create a 1D array concatenating Syntax type & actual syntax.

The following example explains the process of syntax extraction.

Table 6: Syntax extraction summary (Process of AST to 1D array)

Code Snippet	**1D Array**	
	Index	**Value**
(randomNum = getId()) == 3	[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]	BinaryExpr (randomNum = getId()) == 3 EnclosedExpr (randomNum = getId()) IntegerLiteralExpr 3 AssignExpr randomNum = getId() NameExpr randomNum

	[10] [11] [12] [13] [14] [15]	MethodCallExpr getId() SimpleName randomNum SimpleName getId

3.1.4. Syntax Tokenizer Mechanism

Since the Neural Network model requires inputs as numerical vectors, the extracted string formatted syntax vectors can't be sent to a Neural network model directly. To overcome this limitation. Once the Nalyzer-Development-Kit generates the 1D <Syntax Type> <Syntax> array, the final step is to convert String values to machine learning-oriented numerical vectors, for that researcher did not use standard python libraries due to the limitation of regenerating the same token for the different data set. NalyzerDK implemented its own tokenizer method aiming to generate the same token regardless of the data set.

For that NalyzerDK overrides the Java hashcode method and builds a mechanism to represent each String value in JavaParser 1D array in between 0 and 2. Also, a Neural network model requires the same length or input array in each data set, to maintain that researchers append 0 value to remain array values to make its lengths consistent.

3.1.5. Summary of the Training Data

Nalyzer-Development-Kit allows feasibility to generate training data by executing customized maven goals inside any maven builder-supported Java project. To generate test data researchers identify 5 open-source java project repositories available on GitHub. Those 5 projects clone from The Apache Software Foundation’s official GitHub repository (https://github.com/apache). The key reason for choosing Apache Software Foundation’s project repository is, researchers believe that this approach will allow simulating real developer

community coding standards, practices and patterns. Cause Apache is the world's largest open-source Java library platform in the world.[16] More the 39,000 code contributors & 460,000 people involved in Apache communities. Apache repository represents the real-world reflection on the Java developer community. Real-world code snippets improve the quality of training data.

Researchers did different data segmentation. NalyzerDK extracts buggy code snippets from source code parsed into 1D array explained as section (3.1.3). Also Once excluding all the buggy code snippets the remaining code snippets are separated as non-buggy code and stored separately. The summary of identified violation code snippets against violation rule in each code repository explained in table (6).

Table 7: Identified violations in each repo

Repository / **Violation Rule**	**apache /cxf**	**apache /dubbo**	**apache /flink**	**apache/ pdfbox**	**apache/ pulsar**
Use Concurrent Hash Map	1217	1110	930	816	1327
Avoid Catching Throwable	1083	745	945	897	678
Do Not Use Threads	910	892	1200	983	798
Too Few Branches For A Switch Statement	696	543	789	567	698
Append Character With Char	589	456	786	456	567
Consecutive Appends Should Reuse	572	520	683	495	680
Avoid Instantiating Objects In Loops	515	765	593	690	538
Null Assignment	491	365	598	465	438
Avoid Using Short Type	486	296	332	396	236
Avoid Literals In If Condition	441	409	395	294	486
Avoid Using Volatile	352	285	401	386	355
Redundant Field Initializer	250	360	230	263	307

3.2. Build Neural Network Model

To build Neural Network Model (NN), researchers did a background study to identify the neural network type used for implementation. For that researchers list down all the pros and cons of different neural network algorithms. (Ex. Recurrent Neural Networks (RNN)/ Convolution Neural Networks (CNN)/ Artificial Neural Networks (ANN)). Section 3.2.1 explained the process of neural network type selection.

Section 3.2.2 explained the hardware architecture researchers used to build the neural network. The hardware arrangement aims to accelerate both the model building process and real-time analysis. This section primarily focuses on the model building process.

Also one of the key design decisions is libraries and the programming language used to build the model. Section 3.2.3 explained all the design constraints and considerations researchers follow for select Tensorflow 2.0 and Python.

3.2.1. Convolution Neural Networks (CNN)

Convolutional Neural Networks (CNNs) are well-known neural networks for predicting data that are formatted in matrix type (grid) data structure [17], such as 1D grid formatted time-series data, 2D grid/3D grid formatted image data.

CNNs have been illustrated successfully in many deep learning domains, including image classification [18], natural language processing (NLP) [20], and speech recognition[19].

To build neural network models in this research, the Nalyzer neural network model gets the advantage of efficient feature generation from software source code files. CNNs Sparse Connectivity and Shared Weights are the key characteristics beneficial for source code analysis domain compared with traditional other neural network algorithms (Ex. Artificial Neural Networks (ANN)/ Recurrent Neural Networks (RNN)) which can benefit the Nalyzer code analyzer model in capturing the structural behavior of a given class file.

Sparse Connectivity: Sparse network is obtained by implementing the smaller size of the kernel comparatively network input input vector. The connections between the input layer and the output layer/hidden layer reduces due to this implementation. To maximize sparsity we can utilize the Inter-channel and intra-channel lay-off.

Furthermore, the process of output requires not many calculations and comparatively less memory to the neural network for storing the weights. Fig: 9 illustrates the architectural difference between dense connectivity and sparse connectivity.

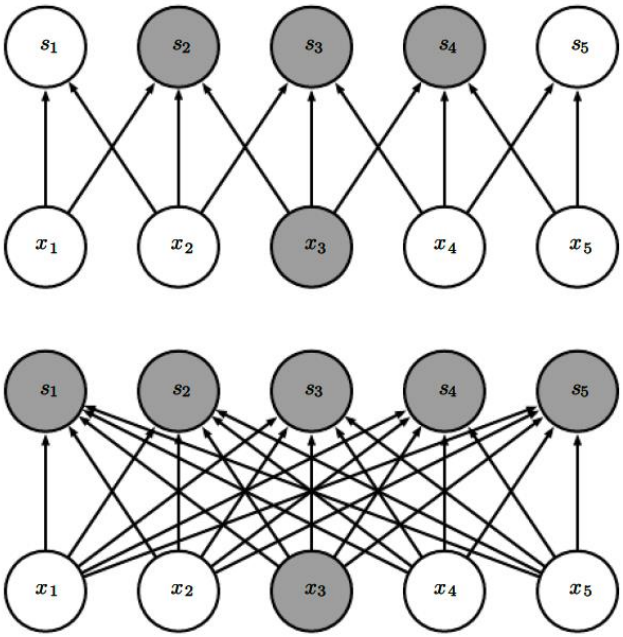

Figure 9: Difference between dense connectivity and sparse connectivity

Shared Weight: The transition layers in CNN contain neurons and each layer contains several filters. Those filters are pre-defined parameters. The width and height of each filter correspond to one unit within the receptive field of the layer area. The action/input maps for each layer created by the convolutional layer can serve as the installation in the next layer.

Regardless of location in the vector we can identify the features with the way of replicating filters in CNN. In addition, by reducing the number of free parameters allows us to increase efficiency of learning significantly.

Figure 10 illustrates the process within the CNN filter system, using a 2x2 filter for 3x3 input, in this case re-using the same four filters assigned to the filter for all inputs. Alternatively, each filter consumes its own range of inputs (Each terrain of the image can have a separate filter), providing 16 weights total, or a dense layer with four layers that have 36 weights total. Sharing weights significantly reduces the number of weights to train, allowing capabilities to train the deepest structures.

Equation 1: Process within the CNN filter system

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

$$F = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$$

$$\beta = [w_{11}, w_{12}, w_{21}, w_{22}]$$

$$F * X = \begin{bmatrix} \beta \cdot [x_{11}, x_{12}, x_{21}, x_{22}] & \beta \cdot [x_{12}, x_{13}, x_{22}, x_{23}] \\ \beta \cdot [x_{21}, x_{22}, x_{31}, x_{32}] & \beta \cdot [x_{22}, x_{23}, x_{32}, x_{33}] \end{bmatrix}$$

3.2.2. Implementation of CNN

To implement a Convolutional Neural network, researchers follow a few design constraints based on the Java programming language and training data set.

Table 8: Structure of neural network

Layer	**Output Shape**	**No of Parameters**
Embedding layer	(None, 12, 32)	3200
Conv 1D layer	(None, 12, 32)	3104
Max Pooling 1D layer	(None, 6, 32)	0
Conv 1D layer	(None, 6, 32)	3104
Max Pooling 1D layer	(None, 3, 32)	0
Flatten layer	(None, 96)	0
Dense layer	(None, 128)	12416
Dense layer	(None, 1)	129

3.2.3. Hardware Implementation

Deep learning frameworks available today such as Tensorflow, Pytorch (and by inheriting any machine learning (ML) libraries which works together with them, e.g. Keras) achieve significant improvement in the training of deep learning when implemented on the platform is built on top of Graphics Processing Unit (GPU) support compared with using a CPU processing power. But, in order to build compatibility with GPU available for the above frameworks should fulfill following concerns, the GPU architecture must be supported with the CUDA toolkit and additionally required supporting libraries such as GPU-Accelerated Library. (for example: cuDNN)

To ensure gain maximum efficiency for the build neural network model, researchers build Cuda enabled TensorFlow deep learning environment. Since CUDA compatibility is limited to Nvidia GPUs. For this hardware, the setup uses the NVIDIA GeForce MX130 GPU adapter, and for that hardware implementation, following prerequisite software required.

1. Microsoft Visual Studio
2. The NVIDIA CUDA Toolkit
3. NVIDIA cuDNN
4. Python
5. Tensorflow (with GPU support)

3.2.4. Access to Build Model

When it comes to web service implementation on Python, there are two frameworks that are widely used in the industry: Django and Flask. Django is comparatively more mature, and a little bit more popular in Dev society. Flask has its new industry. From the ground up, Flask architecture primarily focuses on simplicity and scalability. Compared with Django counterparts, Flask web applications are lightweight.

For the access build model via the dashboard, the Nalyzer project includes a restful web service built on top of the Flask framework. Nalyzer contains two restful API endpoints, those are POST methods to access the build model and submit feedback.

3.3. The Nalyzer Analysis Approach

The way to get the benefit of the build model & the existence of this approach depends on the practices & mechanism followed in Nalyzer analysis implementation. Ultimately researcher's goal is to build a universal neural network model to identify buggy & vulnerable code snippets on source code by involving millions of experts in Dev communities across the world.

Since the Nalyzer analysis step is the interface of this project to the outside world and, researchers approach should make sure the usability & user-friendliness of the final step. In order to achieve this, this implementation provides the capability to integrate 'Nalyzer analyses into the usual CI/CD flow as a maven build step. And the analysis report came up with an interactive dashboard where developers can interact easily without additional effort. This effortless feedback gathering approach will allow for a rich community-driven experience.

This section is dedicated to explaining design constraints and confederations researchers followed in the Nalyzer analysis maven plugin.

3.3.1. Nalyzer Maven plugin

As mentioned in section (3.1.1) Maven is one of the leading build automation tools used primarily for Java projects. For the Nalyzer source code analyzer researcher's goal to implement a customized maven goal, which can execute during the project compile stage and generate training data and store. This approach will help to the improved usability of this project and maven build is a familiar step to the majority of developers in today's CI/CD practices.

Maven build automation tool provides customization capabilities & build customized goal using 'maven-plugin-annotations' library. Nalyzer source code analyzer maven plugin can simply execute using the following command.

Table 9: Command use for run Nalyzer Maven plugin

```
mvn nalyzer:analyze
```

The execution contains several steps to generate an analysis report on the project \target directory.

As the first step, the Nalyzer generates an Abstract syntax tree (AST) of given source code using JavaParser; same as the Nalyzer-Development-Kit, for analyzer, also uses an open-source Java source code parser project called JavaParser to generate Abstract Syntax Tree (AST) of the Java class file. Before generating the AST from the Java class file, NalyzerDK removes all the comments on the source code. Then generate the AST.

Once generating the AST, Nalyzer strategy travels through AST using BreadthFirst traversal and identifies each node containing 6 or more than 6 child nodes and extra code snippets for feed to build the model.

This source code extraction process also extracts the type of syntax against the syntax such as ClassOrInterfaceType, SimpleName, BinaryExpr. Also, Nalyzer analysis consumes the syntax location data, line no & column no for extracting the code snippet.

Since the Neural Network model requires inputs as numerical vectors, the extracted string formatted syntax vectors cannot be consumed as output directly to a neural network model. To overcome this limitation, Nalyzer maven plugin also refers to the Nalyzer-Development-Kit method explained in section (3.1.4) Nalyzer also converts the 1D <Syntax Type> <Syntax> array String values into machine learning-oriented numerical vectors, for that researcher did not use standard python libraries due to the limitation of regenerating the same token for the different data set. So Nalyzer overrides the Java hashcode method and builds a mechanism to represent each String value in JavaParser 1D array in between 0 and 2.

Extracted code combination push to Nalyzer web service and get JSON formatted analysis summary.

Table 10: JSON formated Nalyzer result


```
[{"artifactID": "poc-sample-project",
    "rootList": [
        {
            "fileName": "Application.java",
            "nodeGroupList": [
                {
                    "endColumn": 40,
                    "endLine": 10,
                    "nodeList": [
```

```
                        {
                            "node": 1.9043584
                        },
                        {
                            "node": 1.3001704
                        },
                        {
                            "node": 1.1198153
                        },
                        {
                            "node": 1.1560708
                        },
                        {
                            "node": 0.57506806
                        }
                    ],
                    "startColumn": 1,
                    "startLine": 1,
                    "violationIndex": 0.08506527543067932
                }
            ]
        }
    ]
}]
```


3.3.2. Interactive Dashboard

Section 3.3.1 generated JSON doesn't readable and unable to interact in a user-friendly way, to overcome this Nalyzer project include an interactive dashboard which populates data by consuming restful web service responses, dashboard primarily display the cumulative code severity index and provide a comprehensive overview of each and every code combination available on the project. Figure 11 illustrates the screenshot of the dashboard's main view.

Also, the community-driven approach implemented via the feedback mechanism, users can provide the feedback for each result and those feedback stores in a central data repository, in the production level researchers plan to build in predefined time-frequency. Researchers expecting. With Dev community feedbacks Nalyzer central data repository grows rapidly and alos, the accuracy of the neural network model will increase based on that in the future.

Figure 12 illustrates the feedback windows available in the Nalyzer dashboard.

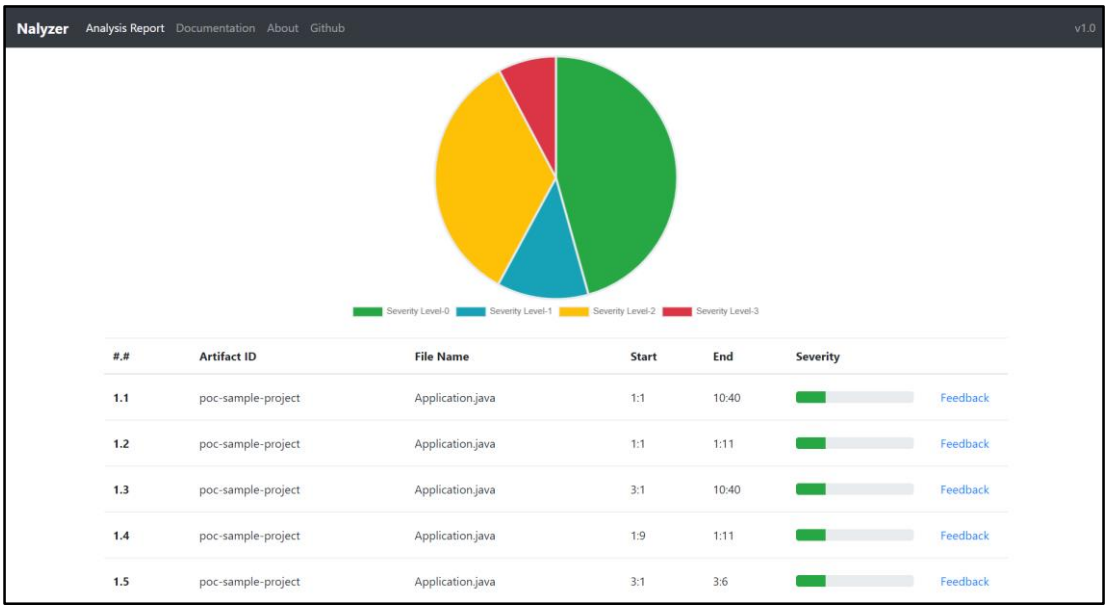

Figure 10: Nalyzer user interactive dashboard

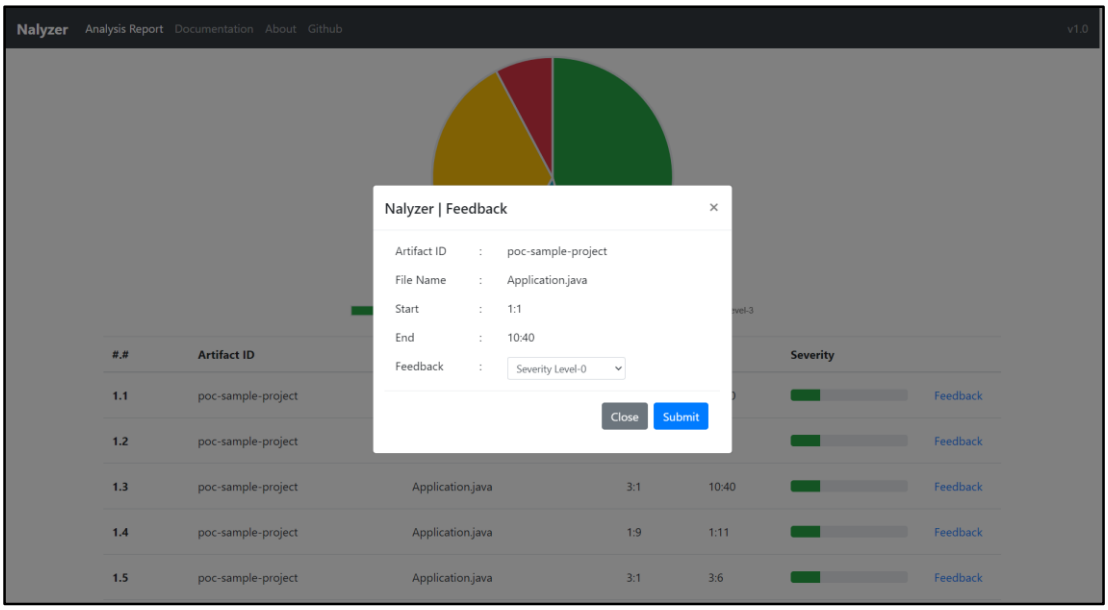

Figure 11: Nalyzer feedback window

4. RESULTS & DISCUSSION

In this section, we evaluate the accuracy of the Nalyzer model using a standard statistical approach, since different mechanisms available to measure a model's accuracy on a dataset, as a first step we identified the most suitable model for evaluating the Nalyzer model. Section (4.1) explains different approaches available in the current deep learning paradigm and the suitability to apply the F-score method to measure the accuracy of the Nalyzer model.

All experiments were run on a Windows hosted PC with 6GB memory allocated NVIDIA GeForce MX130 GPU. For this experiment we used the Nalyzer-Development-Kit to generate data, also average results were reported after running the evaluation process five to ten times on the same setup.

4.1. Machine Learning Performance Measure Mechanisms

Identifying the right metric is a bit challenging while measuring deep learning models. Different metrics are implemented to measure deep-learning models in different frameworks, in this section we summarize metrics mostly used for different categories.

Table 11: ML model evaluation metrics

Category	**Metrics**
Classification Metrics	Area under the ROC Curve, Accuracy, Precision, Recall, F1-score, Receiver Operating Characteristic Curve(ROC),
Regression Metrics	Mean Squared Error (MSE), Mean Absolute Error (MAE)
Ranking Metric	Mean Reciprocal Rank (MRR), Discounted Cumulative Gain (DCG), Normalized Discounted Cumulative Gain (NDCG)

Nalyzer machine learning model categorized under classification problem since this scope we primarily identify whether particular source code snippet is buggy or non-buggy. Classification Accuracy can be simply calculated using the total number of correct predictions divided from the total number of predictions. But classification

accuracy is not suitable for many scenarios when distribution of classes, which we used is imbalanced. In most of the cases, there's a possibility to get a high accuracy rate for the most frequent class. In order to overcome this limitation, one popular metric used in the ML paradigm which calculates using precision and recall values is called F1-score, simply F1-score defined as the harmonic mean of precision value and recall value of evaluation data.

To evaluate the Nalyzer machine learning model, we decided to use the F1 Score method.

4.2. Precision/Recall/F-Measure & Confusion Matrix

In this section, we explain the tools and techniques used to calculate the F1 Score. The roadmap will allow getting an overall idea of measuring a model's accuracy on a dataset.

Table 12: ML model evaluation matrix & terms

Term	**Definition**
Confusion Matrix	One of the key theories in the calculation of the performance of classification problems is prediction of the confusion matrix (Also known as error matrix). Confusion Matrix represents the statistics form of the model predictions vs. the actual classes. Predicted class instance represented by in each row of the confusion matrix and actual class instances represented by in each column.
Precision	The degree of the code snippets classified correctly as buggy-code as a portion of correctly predicted buggy-code & invalidly predicted buggy-code, which are truly non-buggy. *Precision = True Positive / (True Positive + False Positive)*

Recall	The degree of the number of code snippets classified correctly as buggy-code as a portion of correctly predicted buggy-code & invalidly predicted non-buggy-code, which are truly buggy. *Recall = True Positive / (True Positive + False Negative)*

F1-score, define as a harmonic mean of precision value and recall value:

Equation 2: Define F1-score

*F1-score = 2 * Precision * Recall / (Precision + Recall)*

In general, precision and recall has a trade-off. As an example, a model classifying all the trainable code snippets as buggy code, this scenario will get a high number value as recall value as one (1) but a low number value as precision (<<1). Thus, F-measure can be described as a synthesized calculation of precision value and recall value that get 0-1 range value. If the evaluation result is a higher F-measure that means the accuracy of the classification model achieves better performance.

4.3. Nalyzer Model Evaluation

To generate an evaluation data set, we refer to another large 10 open-source java projects which is known for different domains such as distributed database, Online Analytical Processing (OLAP), big data analysis in the Apache, Spring-Projects, Oracle and AWS GitHub code repositories. Each repository contains different sub-projects and approximately more than 1000 Java classes in each repo. Using the Nalyzer-Development-Kit (NalyzerDK) we build the code repositories and store PMD analytical data in 5 different files to generate a generic data set. With the help of NalyzerDK segmentation options, we build a generic data repo to use in the F-measure evaluation. The main reason to use completely independent GitHub repositories and projects in different domains is to verify the exact real-world

scenario & evaluate the impact of over fitting and make sure to generalize on unseen data. The overview of project repositories we used can be summarized like this;

1. **Apache/hbase:** (Domain: BigData)
   A distributed big data store, which is known as a Hadoop database primarily used for hosting large scale data sets. [40]
2. **Apache/incubator-pinot:** (Domain: OLAP)
   This project is in real time distributed OLAP data storage, aim to solve OLAP processing with low latency. [41]
3. **Apache/hudi:** (Domain: Data Streaming/DB)
   This project aims to absorb & manage storage of large analytical datasets over traditional large batch processing. [42]
4. **Apache/ignite:** (Domain: DB)
   This project is distributed databases aimed to achieve high performance. [43]
5. **Apache/zookeeper:** (Domain: Distributed System)
   This project is a centralized directory for maintaining service configuration in distributed computing domain. [44]
6. **Spring-projects/spring-data-rest:** (Domain: Web services)
   This project aims to facilitate a configurable approach to build a simple RestFull web service for doing complete CRUD operations. [45]
7. **Spring-projects/spring-ws:** (Domain: Web services)
   The aim of this project is to introduce a document-driven approach to build RestFull web service. [46]
8. **Oracle/helidon:** (Domain: Web services/Micro service)
   Helidon facilitates the capabilities to build microservices using the set of Java libraries pack. [47]
9. **Oracle/opengrok:** (Domain: Source code management)
   This project allows us to maintain large project documentation against code repository. [48]
10. **AWS/aws-sdk-java:** (Domain: Cloud Computing)
    This project allows Java developers to build web applications on top of AWS architecture. [49]

Table 13 contains the test data overall summary, we used in the next calculations.

Table 13: Overall Evaluation Data Extraction Summary

Repository	Repository summary		PMD analysis summary	
	No of contributors	**No of Java file scanned**	**No of buggy-code patterns***	**No of non-buggy-code patterns**
Apache/hbase	340+	4361	2267	> 1 Million
Apache/incubator-pinot	160+	2425	1957	> 1 Million
Apache/hudi	145+	1259	1281	812279
Apache/ignite	250+	11014	2062	> 1 Million
Apache/zookeeper	140+	887	667	243984
Spring-projects/spring-data-rest	60+	425	40	110710
Spring-projects/spring-ws	35+	828	132	212656
Oracle/helidon	55+	3848	846	356157
Oracle/opengrok	95+	824	829	329211
AWS/aws-sdk-java	170+	87027	320	> 1 Million

*NalyzerDK considers only 12 violation rules, (explained in the section 3.2.1) to identify buggy-code patterns.

The evaluation process drives on two paths, once we extract buggy and non-buggy code snippets using NalyzerDK, we build generic test-data repo containing around 1500 buggy and non-buggy code patterns from all projects. Also, we were trying to understand the impact of overfitting and make sure to generalize on unseen data via the second path. For that, we evaluate data using 100 code snippets from each code repositories independently and check whether there's an impact on overfitting. Since model-trained data repositories know different domains such as networking, remote procedure call (RPC), stream processing. Our analysis focuses on identifying if there

are any dependencies with train data and model is specific to the domain, coding practices, naming conventions, or coding standards.

4.3.1. Build Confusion Matrix

In order to build a Confusion Matrix for each project repositories, we pick over 150 code snippets randomly and evaluate the Nalyzer neural network model. Each code snippet set contains ⅓ of buggy-code patterns identified from the PMD analyzer.

Table 14: Apache/hbase - Confusion matrix

		Actual Class	
		Buggy Code	**Non-Buggy Code**
Predicted Class	**Buggy Code**	43	2
	Non-Buggy Code	7	98

Table 15: Apache/incubator-pinot - Confusion matrix

		Actual Class	
		Buggy Code	**Non-Buggy Code**
Predicted Class	**Buggy Code**	48	4
	Non-Buggy Code	2	96

Table 16: Apache/hudi - Confusion matrix

		Actual Class	
		Buggy Code	**Non-Buggy Code**
Predicted Class	**Buggy Code**	38	1
	Non-Buggy Code	19	99

Table 17: Apache/ignite - Confusion matrix

		Actual Class	
		Buggy Code	**Non-Buggy Code**
Predicted Class	**Buggy Code**	45	2
	Non-Buggy Code	5	98

Table 18: Apache/zookeeper - Confusion matrix

		Actual Class	
		Buggy Code	**Non-Buggy Code**
Predicted Class	**Buggy Code**	48	0
	Non-Buggy Code	2	100

Table 19: Spring-projects/spring-data-rest - Confusion matrix

		Actual Class	
		Buggy Code	**Non-Buggy Code**
Predicted Class	**Buggy Code**	38	0
	Non-Buggy Code	12	100

Table 20: Spring-projects/spring-ws - Confusion matrix

		Actual Class	
		Buggy Code	**Non-Buggy Code**
Predicted Class	**Buggy Code**	49	0
	Non-Buggy Code	1	100

Table 21: Oracle/helidon - Confusion matrix

		Actual Class	
		Buggy Code	**Non-Buggy Code**
Predicted Class	**Buggy Code**	35	1
	Non-Buggy Code	15	99

Table 22: Oracle/opengrok - Confusion matrix

		Actual Class	
		Buggy Code	**Non-Buggy Code**
Predicted Class	**Buggy Code**	46	1
	Non-Buggy Code	4	99

Table 23: AWS/aws-sdk-java- Confusion matrix

		Actual Class	
		Buggy Code	**Non-Buggy Code**
Predicted Class	**Buggy Code**	41	0
	Non-Buggy Code	9	100

4.3.2. Precision Calculation

In this section, we calculate the precision value in cumulative perspective, by summarizing the all confusion matrices we build in section 1.1.1. In this section, our intention to identify the overall portion of buggy-code classification is actually correct.

Equation 3: Precision Calculation

Precision	***= True Positive / (True Positive + False Positive)***
Precision	*= 431/(431+11) = 0.975*

4.3.3. Recall Calculation

In this section, we calculate the Recall value in the cumulative perspective, by summarizing the all confusion matrices we build in section 1.1.1. In this section, our intention to identify the overall portion of actual buggy codes is identified correctly from the Nalyzer neural network model.

Equation 4: Recall Calculation

Recall	***= True Positive / (True Positive + False Negative)***
Recall Buggy	*= 431/(431+76) = 0.85*

4.3.4. F1-Score Calculation

In this section, we calculate the F1-Score value in cumulative perspective, by summarizing the all confusion matrices we build in section 1.1.1. The discussion section will deeply discuss the impact of overfitting and neural model compatibility to adapt to the generic data set without specific to the domain, coding practices, naming conventions, or coding standards. Overall 1500 code snippets are considered for calculating Precision. ⅓ of data-set contains buggy code snippets.

Equation 5: F1-Score Calculation

F1-score	***= 2 * Precision * Recall / (Precision + Recall)***
F1-score	*= 2 * 0.975 * 0.85 / (0.975 + 0.85)*
	= 0.90

4.4. Discussion

Overall training data containing more than 80000 buggy and non-buggy data set, and with the help of NalyzerDK researchers able to maintain 100% balance data for each class. And in order to avoid the impact on the evaluation result by memorized training data, we use a completely independent GitHub repositories to do the model evaluation. Also, during the evaluation process Researchers are tried to mitigate the unnecessary dependencies by using NalyzerDK data segmentation options. In the evaluation, we used 1500 data sets extracted from the 10 different GitHub code repository, and as a summary 500 (see Table 13) code snippets contain buggy-code patterns identified via PMD code analyzer. Our approach is to evaluate the potential of identifying those code-snippets throughout the Nalyzer neural network model.

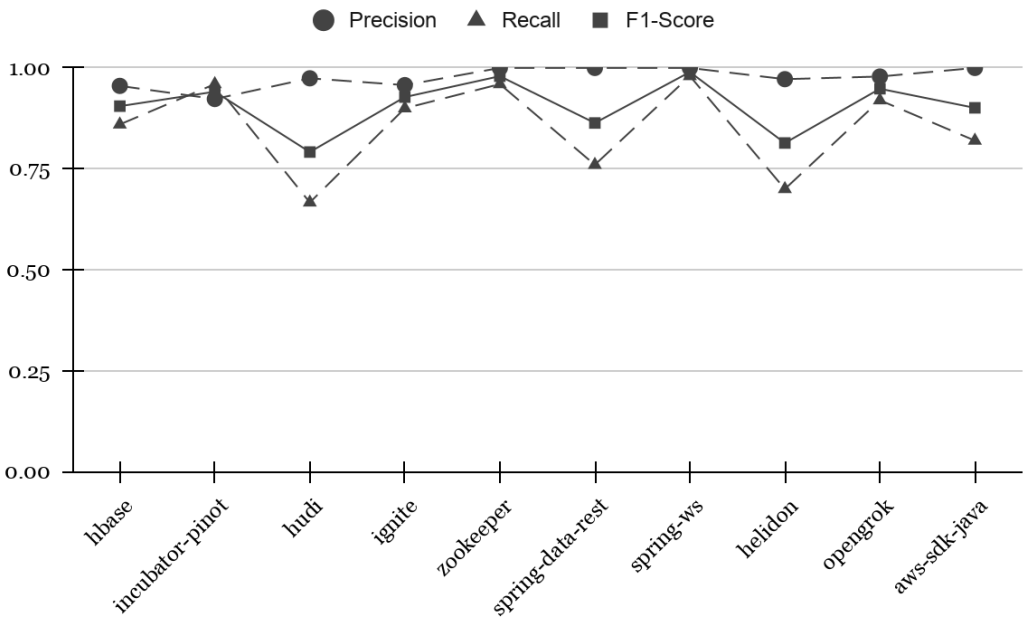

Figure 12: Overall evaluation result

In deep learning classification models the highest possible value can achieve an F-score is 1.0, this will point out the perfect precision and the recall, and the lowest possible value is 0. Nalyzer code analysis model F-score values close to 1 but still we have room for improvements. Since we maintain a 100% balanced data set by segmenting NalyzerDK generated data, still F1 Buggy < F1 Non-Buggy score. The reality is we can see the precision and recall tradeoff in every scenario in a neural

network. In simple terms, if we increase the precision score rate too high, the recall value ends up by a decrease drastically, and vice versa.

One of the key objectives of this evaluation is to identify if there are any dependencies with train data and model is specific to the domain, coding practices, naming conventions, or coding standards. Fig. 13 depicting the precision, recall, and F1 scores deviation against each code repositories. Code snippets in Apache/hudi, Spring-projects/spring-data-rest, Oracle/helidon show deviation from the overall result, but still F1 Score remain in > 0.75 range. The training data we used for the train Nalyzer model is extracted from the Apache GitHub repository, but for the evaluation process, we consider different code repositories maintained by different vendors. Also the projects we referred for extract training data and projects used for extract evaluation data on different domains such as distributed computing, big data analysis. We don't identify the over fitting situation in Nalyzer neural network model, also as an expected model is suitable for evaluating more generalized code repositories.

In the background study and literature review, we identified some research done studies to prove the capability of ML approaches in the code analysis domain. Since the structured keyword analysis and the potential of static code analysis via ML model is one of the common paradigms of ML-based research. But in Nalyzer we introduce a self-sustainable approach to adopt changes in the future. Nalyzer project just not limited to theoretically prove the applicability of ML in the code analysis domain but come up with a practical implementation of the useability of ML-based code analysis without impacting to day to day SDLC processes. Also, the Nalyzer neural network model builds on top of the CNN algorithm. Since CNN is well known for identifying the complex features of a given data set and we found studies that are focus to identify the relevance of CNN approaches over other deep learning algorithms such as ANN or RNN. And the surveys we refer to during the literature review, we identified limitations to adopt additional processes during day to day SDLC process and how most of the solutions are obsolete within a couple of years. Usually, developers prefer to get consolidate view of the situation within few minutes via dashboards, most of the DevOps practices move to dashboards over traditional command-line processes. Technology like Kubernetes/Azure popular

across the modern development process due to the usability & simplicity. The interactive dashboard came up with a solution to those problems & limitations.

Since this research scope is limited to one programming language & 12 types of error patterns, but during model evaluation, we saw even with this scope Nalyzer neural network model achieved a high F1-score (Cumulative F1 Score - 0.90). Model evaluation results prove that the machine learning approach is a great alternative for overcoming traditional manual code reviews and automation code analysis tools.

In this scope, Nalyzer neural model can identify buggy or non-buggy code snippets only, in the future we will improve the Nalyzer neural network model to identify different bug classes within the buggy code snippets. To provide recommendations against the analysis findings, we identify as a future work to introduce different classes against the buggy patterns that will improve the usability of the system. For that, the Nalyzer project should contain different network models over programming languages and change the neural network model to handle different buggy classes that need to implement in the future. Also, to coordinate the requests against the different input, the Nalyzer real-time analyzer need to be improve.

To make the Nalyzer code analysis model more generalizable, we have identified a few windows to improve in the future, we will conduct more experiments on different projects in different languages & frameworks.

5. SUMMARY AND CONCLUSIONS

In this research one of the primary objectives is to prove that the machine learning approach is a great alternative for overcoming traditional manual code reviews and automation code analysis tools. Since artificial intelligence is hype in the modern world, the software industry should build more tools & techniques to get the benefit of Machine learning-based approaches to reduce project timelines & efforts.

In this research scope, we did not focus on evaluating the efficiency & effectiveness over different ML algorithms such as ANN or RNN. In the literature & background study, one of the main focuses is identifying the suitable algorithm to build a neural network model based on the previous studies done in this research area. [21] [22] Since CNN is well known for identifying the complex features of a given data set, we also refer to the same approach and basic neural network architecture to build the Nalyzer neural network model.

Nalyzer-Development-Kit (NalyzerDK) also, one of the most important outcomes of the research. NalyzerDK builds for generating the training data for the neural network model, this subcomponent of the Nalyzer project becomes an advanced tool for generating training data by segmenting the source code repositories. We will make the source code of this tool publicly available for researchers who are interested in doing researches in the same domain. (Github - https://github.com/Denuwanhh/nalyzer)

In the result and discussion section, we discussed the potential of identifying the pre-trained error-prone code snippets & vulnerabilities using the NalyzerDK neural network model. As identifying the model, during the literature review, we identify the standard model evaluation method available today. [23] F1-score is one of the highly recommended mechanisms to calculate model performance by overcoming the limitation of the traditional accuracy method. The GitHub repositories public code repository we used to evaluate the Nalyzer model represents the real-world coding patterns with 10000+ commits from 1000+ contributors around the world. And > 75% of Java code contained in the each code repos. More than 75% buggy-code & non-buggy-code F1-score prove that the ML approaches & the CNN algorithm capabilities in the domain of source code analysis.

Also, the software industry has strong community involvement. Nalyzer approach depends on this key characteristic of software engineering. We should utilize community involvements & build platforms other than online forums to facilitate the transfer of

knowledge from the industry experts. Nalyzer project aims to build a universal neural network model to identify buggy code patterns & security vulnerabilities. For that, our approach is a community-driven platform to collect data from the developer community and a self-sustainable platform by reducing the maintainability & modifiability efforts of traditional automation tools facing today due to the frequent releases in programming languages & frameworks. [25]

We primarily used 5 large Java code repositories for generating the training data. Since we limit the PMD analyzer rule for 12 violation rules, the extracted buggy code pattern amount was shown as 40000+ & using NalyzerDK we were able to generate the balance non-buggy code patterns. Both buggy & non-buggy classes contain 80000+ data in the Nalyzer data repository. When we look at the developer community, which is the segmentation we aim to address in this research shows the potential of adopting Nalyzer type solution to their day-to-day life processes. And the sub-cultures in the industry, like open-source, online-forum show the potential of adapting an interactive dashboard approach to attract the community's contribution. [37] The amount of data we used to train the Nalyzer neural network and the F1-score we achieved prove that a community-driven approach will accelerate the self-sustainability against the new framework & programming language of the neural model in the future.

But still, we have room for improvement, we need to make sure to manage the community involvement, for that we will implement a feedback ranking mechanism with the help of third-party platforms such as GitHub/Stackoverflow. [34] We will rank users based on their reputation achieved on those platforms. User ranking functionality will automatically cleanse the data and improve the quality of the training data. Also, to make the Nalyzer model more generalizable, in the future, we will extend our scope to C#, Javascript, Python by gathering different code snippets in different language projects. And we should extend the Nalyzer scope to the different frameworks such as Spring Boot, React Native. By expanding to different programming languages & frameworks allows us to get more contribution from the developer community. In order to achieve high model accuracy, we believe we need to train different models in different languages & enhance Nalyzer real-time analyzer to coordinate input to navigate to the appropriate neural network model. Also, in this scope, Nalyzer neural model can identify buggy or non-buggy code snippets only, in the future we will improve the Nalyzer neural network model to identify different bug

classes within the buggy code snippets. Identifying different buggy classes allows attracting junior developer community to the Nalyzer project and get their contribution to training the neural network model. Since this research scope limited to one neural network model, in the future, we have to build different models within the core model to cater different data sets. Overall, when we look at the future improvement we need to manage user feedback and expand Nalyzer scope to different programming languages & frameworks in the future is recommended.

Since this research scope is limited to one programming language, 12 types of error patterns & less than 100,000 data set scope, we prove that the capabilities to adapt real-world code analysis during model evaluation, we saw even with this scope Nalyzer neural network model achieved a high F1-score (cumulative F1 score - 0.90). Model evaluation results prove that the machine learning approach is a great alternative for overcoming traditional manual code reviews and automation code analysis tools. Furthermore, the Nalyzer project proves that the ML approach is an alternative for overcoming the limitations of manual code reviews and automated code analysis tools.

REFERENCES

[1] C. Jones, Software assessments, benchmarks, and best practices. Boston, Mass.: Addison-Wesley, 2000.

[2] "Gartner Magic Quadrant for Software Test Automation 2019", Tricentis, 2020. [Online]. Available:https://www.tricentis.com/resources/gartner-magic-quadrant-software-test-automation/. [Accessed: 29- Mar- 2020].

[3] P. Runeson, "A survey of unit testing practices", *IEEE Software*, vol. 23, no. 4, pp. 22-29, 2006. Available: 10.1109/ms.2006.91.

[4] P. Louridas, "JUnit: Unit Testing and Coding in Tandem", *IEEE Software*, vol. 22, no. 4, pp. 12-15, 2005. Available: 10.1109/ms.2005.100.

[5] R. Osherove, The art of unit testing, 2nd ed. Greenwich, CT.: Manning, 2014.

[6] Rafael Lotufo, Leonardo Passos, Krzysztof Czarnecki, "Towards improving bug tracking systems with game mechanisms", *Mining Software Repositories (MSR) 2012 9th IEEE Working Conference on*, pp. 2-11, 2012.

[7] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building useful program analysis tools using an extensible java compiler. In Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'12). 14--23.

[8] K. Sen, D. Marinov and G. Agha, "CUTE", ACM SIGSOFT Software Engineering Notes, vol. 30, no. 5, p. 263, 2005. Available: 10.1145/1095430.1081750.

[9] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug-finding tools for Java. In Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04), pages 245–256. IEEE Computer Society Press, Nov. 2004

[10] J. Langr, A. Hunt, D. Thomas and S. Davidson Pfalzer, Pragmatic unit testing in Java 8 with JUnit, 1st ed. The Pragmatic Programmers, LLC., 2015.

[11] W. WONG and Y. QI, "BP NEURAL NETWORK-BASED EFFECTIVE FAULT LOCALIZATION", International Journal of Software Engineering and Knowledge Engineering, vol. 19, no. 04, pp. 573-597, 2009. Available: 10.1142/s021819400900426x.

[12] "Models and layers | TensorFlow.js", TensorFlow, 2020. [Online]. Available: https://www.tensorflow.org/js/guide/models_and_layers. [Accessed: 02- Apr- 2020].

[13] "Introduction to TensorFlow", TensorFlow, 2020. [Online]. Available: https://www.tensorflow.org/learn. [Accessed: 19- Jul- 2020].

[14] K. Team, "Keras: the Python deep learning API", Keras.io, 2020. [Online]. Available: https://keras.io/. [Accessed: 19- Jul- 2020].

[15] "What are microservices?", microservices.io, 2020. [Online]. Available: https://microservices.io/. [Accessed: 19- Jul- 2020].

[16] "Foundation Project", Apache.org, 2021. [Online]. Available: https://www.apache.org/foundation/. [Accessed: 26- Mar- 2021].

[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS'12: Proc. of the Advances in Neural Information Processing Systems*, 2012

[19] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, and G. Penn, "Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition," in ICASSP'12: *Proc. of the International Conference on Acoustics, Speech and Signal Processing*, 2012.

[20] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient based learning applied to document recognition," Proceedings of the IEEE, vol. 86, no. 11, 1998

[21] X. Huo, M. Li, and Z.H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code," *in Proceedings of IJCAI'2016*.

[22] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[23] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," in *OOPSLA'16: Proc. of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016.

[24] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *ASE'16: Proc. of the International Conference on Automated Software Engineering*, 2016.

[25] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," *arXiv preprint arXiv:1409.5718*, 2014.

[26] X. Gu, H. Zhang, D. Zhang, and S. Kim, “Deep api learning,” in *FSE’16: Proc. of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[27] C. Liu, X. Chen, E. C. Shin, M. Chen, and D. Song, “Latent attention for if-then program synthesis,” in *NIPS’16: Proc. of the Advances in Neural Information Processing Systems*, 2016.

[28] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, “Deepcoder: Learning to write programs,” *arXiv preprint arX-iv:1611.01989*, 2016.

[29] C. Shu and H. Zhang, “Neural programming by example,” in *AAAI’17: Proc. of the AAAI Conference on Artificial Intelligence*, 2017.

[30] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, “Deep learning for just-in-time defect prediction,” in *QRS’15: Proc. of the International Conference on Software Quality, Reliability and Security*, 2015.

[31] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” in *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2004, pp. 132–136.

[32] J.A. Zachman, “A Framework for Information Systems Architecture,” I*BM Systems J.*, vol.26, no. 3, 1987, pp. 276–292.

[33] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O’Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, “Moving fast with software verification,” in *NASA Formal Methods Symposium*. Springer, 2015, pp. 3–11.

[34] "What is reputation? How do I earn (and lose) it? - Help Center", *Stack Overflow*, 2021. [Online]. Available: https://stackoverflow.com/help/whats-reputation. [Accessed: 07- Apr- 2021].

[35] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” *ACM Computing Surveys (CSUR),* vol. 51, no. 4, p. 81, 2018.

[36] M. Weiser, “Programmers use slices when debugging,” *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, Jul. 1982.

[37] I. Vessey, “Expertise in debugging computer programs,” *International Journal of Man-Machine Studies: A Process Analysis*, vol. 23, no. 5,pp. 459–494, 1985.

[38] W. E. Wong, V. Debroy, and B. Choi, “A family of code coverage-based heuristics for effective fault localization,” Journal of Systems and Software, vol. 83, no. 2, pp. 188–208, February 2010.

[39] W. E. Wong and V. Debroy, A Survey on Software Fault Localization Department of Computer Science, University of Texas, Dallas, Technical Report UTDCS-45-09, November 2009.

[40] "Apache HBase – Apache HBase™ Home", *Hbase.apache.org*, 2021. [Online]. Available: https://hbase.apache.org/. [Accessed: 08- Apr- 2021].

[41] "Apache Pinot™ (Incubating): Realtime distributed OLAP datastore | Apache Pinot™ (Incubating)", *Pinot.incubator.apache.org*, 2021. [Online]. Available: https://pinot.incubator.apache.org/. [Accessed: 07- Apr- 2021].

[42] "Welcome to Apache Hudi !", 2021. [Online]. Available: https://hudi.apache.org/. [Accessed: 07- Apr- 2021].

[43] "Distributed Database - Apache Ignite®", *Ignite.apache.org*, 2021. [Online]. Available: https://ignite.apache.org/. [Accessed: 09- Apr- 2021].

[44] "Apache ZooKeeper", *Zookeeper.apache.org*, 2021. [Online]. Available: https://zookeeper.apache.org/. [Accessed: 07- Apr- 2021].

[45] "spring-projects/spring-data-rest", *GitHub*, 2021. [Online]. Available: https://github.com/spring-projects/spring-data-rest. [Accessed: 09- Apr- 2021].

[46] "spring-projects/spring-ws", *GitHub*, 2021. [Online]. Available: https://github.com/spring-projects/spring-ws. [Accessed: 07- Apr- 2021].

[47] "Helidon Project", *Helidon.io*, 2021. [Online]. Available: https://helidon.io/. [Accessed: 07- Apr- 2021].

[48] "Switching from Oracle OpenGrok to Sourcegraph - Sourcegraph docs", *Docs.sourcegraph.com*, 2021. [Online]. Available: https://docs.sourcegraph.com/code_search/how-to/opengrok. [Accessed: 09- Apr- 2021].

[49] "Developer Guide - AWS SDK for Java 1.x - AWS SDK for Java", *Docs.aws.amazon.com*, 2021. [Online]. Available: https://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/welcome.html. [Accessed: 07- Apr- 2021].

[50] Y. S. Su and C. Y. Huang, “Neural-network-based approaches for software reliability estimation using dynamic weighted combinational models,” *Journal of Systems and Software*, vol. 80, no. 4, pp. 606–615, April 2007.