Titre: Title:	An Empirical Study of Testing and Release Practices for Machine Learning Software Systems
Auteur: Author:	Moses Openja
Date:	2021
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Openja, M. (2021). An Empirical Study of Testing and Release Practices for Machine Learning Software Systems [Master's thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/9177/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

An Empirical Study of Testing and Release Practices for Machine Learning Software Systems

MOSES OPENJA
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Août 2021

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

An Empirical Study of Testing and Release Practices for Machine Learning Software Systems

présenté par **Moses OPENJA**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Ettore MERLO, président
Foutse KHOMH, membre et directeur de recherche
Heng LI, membre

DEDICATION

Dedicated to
My dear beloved mother, Rose Biywaga (decease),
and My dad John Yuka, My big brother Godfred, my dear siblings,
and not forgetting Adubango Emmanuel and his family.
Thank you for all your endless love,
sacrifices, and support...

ACKNOWLEDGEMENTS

I want to acknowledge all who contributed to the completion of this thesis in one way or another. First and foremost, I wish to express my deep and sincere gratitude to my supervisor, prof. Foutse Khomh, for his tremendous support throughout, guidance, motivation, and patience, and for giving me an excellent atmosphere to conduct my research. His vast knowledge and logical way of thinking helped me throughout this process. Furthermore, his knowledge and warmth made the collaboration highly beneficial for me. It has always been a great pleasure to work with him; the lessons he taught me to go beyond what is written in this thesis and will help me in all aspects of my life.

In a special way, I want to acknowledge prof. Foutse Khomh, prof. Zhen Ming (Jack) Jiang, prof. Ahmed E. Hassan, and prof. Bram Adams for reviewing these works and all their comments, sharing their knowledge and ideas throughout this study.

I want to acknowledge the committee members of my thesis, prof. Ettore Merlo and Prof. Heng Li for taking interest in my work and sacrificing their efforts and time to review my thesis and provide insightful feedback. I also want to acknowledge the member of SWAT Lab, especially those who help in the manual labeling of the different testing strategies, ML properties, the test methods, and release engineering topics presented in this thesis. These include the following SWAT Lab member: Amin Nikanjam, Ph.D., Mehdi Morovati, Ph.D., Dima Gumenyuk, Msc., Mohamed Raed El Aoun, Msc, Mouna Abidi, Ph.D., Gias Uddin, Ph.D., Md Saidur Rahman, Ph.D., and Gabriel Laberge Ph.D.

I want to acknowledge the other professors and technical staff at the Computer Engineering Department at École Polytechnique de Montréal and SWAT Lab to provide a conducive environment for my research career. Finally, I would like to thank the SEMLA committee for organizing different workshops, most of which were directly beneficial in this study.

RÉSUMÉ

Nous assistons à une adoption croissante des algorithmes d'apprentissage automatique et d'apprentissage profond dans de nombreux systèmes logiciels, y compris dans des domaines critiques telque la santé et les transports. D'une part, assurer la qualité logicielle de ces systèmes est encore un défi ouvert pour la communauté des chercheurs, principalement en raison de la nature inductive de l'apprentissage automatique. Mais, d'un autre côté, les équipes d'ingénierie et de mise en production de systèmes intégrant l'intelligence artificielle, sont tenues de fournir continuellement des produits logiciels de haute qualité aux utilisateurs.

Récemment, la communauté de génie logiciel a commencée à adapter plusieurs concepts des tests de logiciels traditionnels, tels que les tests par mutation, afin d'améliorer la fiabilité des systèmes logiciels basés sur l'apprentissage automatique. De plus, pour faciliter le processus de livraison de ces systèmes, les éditeurs de logiciels proposent de nouveaux changements dans leur processus de mise en production qui s'adaptent aux nouvelles technologies telles que le déploiement continu et l'Infrastructure-as-Code. Cependant, les ingénieurs en charge de la mise en production de ces logiciels éprouvent des difficiles d'implémentation de ces technologies et ont recours à des sites Web de questions et réponses tels que StackOverflow pour trouver des réponses. Concernant la qualité des systèmes logiciels basés sur l'apprentissage automatique, il n'est pas clair si les techniques de test proposées dans les travaux de recherche académiques sont adoptées en pratique. De plus, il existe très peu d'information sur les stratégies de test employés par les ingénieurs des systèmes logiciels basés sur l'apprentissage automatique.

Pour combler les lacunes, cette thèse rapporte trois études empiriques: 1) Une étude sur les pratiques de test de systèmes logiciels basés sur l'apprentissage automatique; identifiant, les propriétés testées, les stratégies de test suivies, et leur mise en œuvre tout au long du cycle de développement de composants d'apprentissage automatique. 2) Une étude des types de tests/méthodes de tests utilisés en pratique tout au long des phases de développement de systèmes logiciels basés sur l'apprentissage automatique. 3) Une étude basée sur StackOverflow, permettant d'identifier les défis de la mise en production des systèmes logiciels.

Nous présentons une analyse systématique des différentes stratégies de test (telles que l'approximation Oracle), des propriétés testes dans les composants d'apprentissage automatique (telles que l'exactitude, la robustesse, l'efficacité, le biais et l'équité) et les méthodes de test (par exemple, les tests unitaires, les tests d'intégration, ou les tests exploratoire). Cette étude a été menée via des analyses qualitatives et quantitatives. Plus précisément, nous avons extrait les fichiers de test et les cas de test correspondants de neuf systèmes de basés sur l'apprentissage

automatique, à code source libre, hébergés sur GitHub, et, à la suite d'une procédure de codage ouverte, nous avons examiné manuellement les stratégies de test proposées dans la littérature de recherche existante qui ont été mises en œuvre, ainsi que les propriétés testées, et les différents types de tests/méthodes de test, dans le but de comprendre comment le test est opérationnalisé dans les systèmes logiciels basés sur l'apprentissage automatique. Nos résultats montrent que : 1) L'injection de fautes, les vérifications *Oracle Approximation*, *State transition*, *Value Range analysis* et *Decision & Logical conditional* sont cinq stratégies de test couramment utilisées par les ingénieurs tout au long du cycle de développement. 2) Nous identifions 20 propriétés testées fréquement, notamment *Sécurité & Confidentialité*, *Consistance*, *Correction fonctionnelle*, *Robustesse* et *Importance des fonctionnalités*. 3) Les stratégies de test telles que Fault-injection (c'est-à-dire *Null Reference detection, Module import error*), Oracle Approximation (c'est-à-dire *Absolute Relative Tolerance, Error bounding*), *Negative Test*, *Value Range Analysis* sont utilisées dans différentes activités du cycle de développement des composants d'apprentissage automatique. De même, les propriétés telles que *Consistance*, *Complétude* ou *Efficacité* sont testées fréquement lors de plusieurs activités. De plus, nous examinons comment les stratégies de test et les méthodes de test sont utilisées pour vérifier différentes propriétés dans les projets logiciels étudiés. Concernant les méthodes de test utilisées, nous avons identifié un total de 11 de différents types de tests, dont seulement six sont inclus dans la pyramide des tests pour systèmes basés sur l'apprentissage automatique. En outre, nous examinons comment les méthodes de test sont utilisées dans les différents projets logiciels étudiées et comment les méthodes de test sont utilisées pour vérifier différentes propriétés des composants d'apprentissage automatique.

En ce qui concerne les défis de mise en production, en utilisant des techniques de modélisation de sujets (topic modeling), nous constatons que (i) les développeurs discutent sur un large éventail de sujets d'ingénierie de mise en production. Au total, nous avons identifiés 38 sujets couvrant les six phases de l'ingénierie de mise en production de systèmes logiciels, (ii) les sujets *Conflit de fusion, Branching & Remote Upstream* sont plus populaires, tandis que les sujets *Code review, Web deploy, MobileApp Debugging & Deployment, Continuous Deployment* sont moins populaires mais plus difficiles, (iii) En particulier, le sujet portant sur la "sécurité " est à la fois populaire et difficile selon les données collectées à partir de StackOverflow. De plus, nous avons constaté que le sujet portant sur le test, i.e., *Software Testing* est le plus discuté (pourcentage le plus élevé) au cours de la phase CI/CD de la mise en production. Ce qui pourrais signifier que le test constitue le défi majeur de l'automatisation du processus de mise en production des systèmes logiciels, tant celui des logiciels traditionnels que celui des logiciels basés sur l'apprentissage automatique.

ABSTRACT

We are witnessing an increasing adoption of machine learning (ML) and deep learning (DL) algorithms in many software systems, including safety-critical systems such as health care systems or autonomous driving vehicles. On the one hand, ensuring the software quality of these systems is yet an open challenge for the research community, mainly due to the inductive nature of ML software system. But, on the other hand, the ML and the Release engineering teams are continuously required to deliver high-quality ML software products to the end-user.

Few recent research advances in the quality assurance of ML systems have been adapting different concepts from traditional software testing, such as mutation testing, to help improve the reliability of ML software systems. Also, to assist in the delivery process of these systems, modern ML software companies are proposing new changes in their delivery process that adapt to new technologies such as continuous deployment and Infrastructure-as-Code. However, the ML and release engineers still find these practices challenging and resort to question and answer websites such as StackOverflow to find answers. For the ML software quality, it is unclear if any of the proposed testing techniques from research are adopted in practice. Moreover, there is little empirical evidence about the testing strategies of ML engineers. Software testing and release engineering together are important for the efficient delivery of reliable ML applications.

To fill the gaps, this thesis reports three empirical studies: 1) a study of the ML testing practices in the wild, to identify the ML properties being tested, the testing strategies followed, and their implementation throughout the ML workflow. 2) a study of the types of test/ testing methods used in practice throughout the development phases of an ML software system. 3) instead of studying release engineering for ML applications only, the thesis studies release engineering in the general software domain, as releasing engineering of ML applications still lack sufficient data and lessons learnt from general software applications can provide insights for releasing ML applications.

To study ML testing practices, first, we systematically summarized the different testing strategies (such as Oracle Approximation), the ML test properties (such as Correctness, Robustness, Efficiency, Bias and Fairness), and the testing methods (e.g., Unit Test, Integration, Manual/ Exploratory Test) from the previous literature. Then, we conduct the study using a mixture of qualitative and quantitative analysis. Specifically, we extracted the test files and the corresponding test cases of nine open source ML systems hosted on GitHub, and

following an open coding procedure we manually examined the testing strategies proposed in the existing research literature that were implemented, as well as the tested properties, and the different types of tests/ testing methods with the aim to understand how the tests are being operationized in the delivery of ML based software systems. Our findings show that: 1) *Fault Injection*, *Oracle Approximation*, *State transition*, *Value Range analysis*, and *Decision & Logical conditional* checks are five commonly used testing strategies used by engineers throughout ML workflow. 2) We identify 20 ML properties that engineers frequently test in an ML workflow, including *Security & Privacy*, *Consistency*, *Functional Correctness*, *Robustness* and *Feature importance*. 3) The testing strategies such as Fault-injection (i.e., *Value error, Null Reference, Runtime error and Exception*), Oracle Approximation (i.e., *Absolute Relative Tolerance, Error bounding*), *Negative Test*, *Value Range Analysis* are used across different ML workflow activities. Similarly, the ML properties such as *Consistency*, *Completeness*, or *Efficiency* are being tested across multiple ML workflow activities compared to other ML properties. Moreover, we examine how the testing strategies and testing methods are used to verify different ML properties across the studied ML software systems. For the testing methods used in ML software system, we identified a total of 11 different types of tests, out of which only six are included in the Test Pyramid of ML. Also, we examine how the testing methods are being used across different ML software systems, and how the testing methods are used in verifying different ML properties.

Regarding the release engineering topics, using topic modeling techniques, we find that (i) developers discuss on a broader range of 38 release engineering topics covering all the six phases of modern release engineering, (ii) the topics *Merge Conflict, Branching & Remote Upstream* are more popular, while topics *Code review, Web deployment, MobileApp Debugging & Deployment, Continuous Deployment* are less popular yet more complicated, (iii) particularly, the release engineering topic "security" is both popular and difficult according to data collected from StackOverflow. Moreover, we found that *Software Testing* is the most discussed (highest percentage) during the CI/CD phase of release engineering. This can potentially indicate that *Software Testing* is a general challenge when automating the delivery process of both ML and traditional software.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

ML	Machine Learning
DL	Deep Learning
CD4ML	Continuous Delivery for ML software project
CI	Continuous Integration
CD	Continuous Deployment
AUT	Application Under Test
UI	User Interface
VCSs	Version Control Systems
API	Application Programming Interface
SA	Sensitive Attribute
LDA	Latent Dirichlet allocation
NLP	Natural Language Processing
OA	Oracle Approximation
F-I	Fault Injection
BLOB	Binary Large Object
CNN	Convolution Neural Network

Tol	Tolerance
FST	Finite State Transducer
Q&A	Question and Answer
RQ	Research Question
HTML	Hypertext Markup Language
URL	Uniform Resource Locator
releng	Release Engineering

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

The increasing adoption of Machine Learning (ML) and Deep Learning (DL) in many software systems, including safety-critical software systems (e.g., autonomous driving [3, 4], medical software systems [5]) raises concerns about their reliability and trustworthiness. Ensuring the quality of these software systems is yet an open challenge for the research community. The main reason behind the difficulty to ensure quality in ML software systems is the shift in the development paradigm induced by ML. On the other hand, due to the market pressure, software industries are continuously required to deliver high-quality software products to the end-users faster. Unlike a few years ago, when ML software companies could work for months or even years on a release, many software companies now have only a limited time (e.g., a few weeks, days, or even hours) to ship their latest features to end users [6]. For instance Facebook Mobile app have reduced their release "cycle time" to between two to six weeks, while Facebook web releases new features (1-2 times) a day [7].

A Few research advances in quality assurance on ML systems recently have been adapting different concepts from traditional software testing (i.e., software not using ML), such as evaluating their effectiveness (e.g., mutation testing [8, 9]) and new techniques to verify the security or privacy of the ML system (e.g., Spoofing attacks in autonomous systems [10, 11]), to help improve the reliability of ML based software systems. Little is known about the current practices on testing ML software systems. It is unclear if there are new unique testing techniques that originated from the practice. Also, the ML and release engineers may find it challenging to adapt the different release engineering practices and resort to question and answer websites such as StackOverflow to find answers. Software testing and release engineering together are important for the efficient delivery of reliable ML systems.

To fill the gaps, in this thesis, we perform the empirical studies broken down into two parts: 1) study of the ML testing practices in the wild, to identify the ML properties being tested, the testing strategies follow, and their implementation throughout the ML workflow. 2) instead of studying release engineering for ML systems only, the thesis studies release engineering in the general software domain from the discussion in StackOverflow to understand the modern release engineering topics of interest and their difficulty. Releasing engineering of ML systems still lack sufficient data and therefore the lessons learnt from general software applications can provide insights for releasing ML systems.

1.1 Practice of Testing ML software systems

Here, we perform the first fine-grained empirical study on ML testing practices in the wild, specifically broken down into two empirical studies. 1) Examined the ML properties being tested, and the testing strategies follow, and their implementation throughout the ML workflow. ML testing strategies are important to ensure that test activities meet software quality assurance objectives, and that the best approaches are used to test the behavior of ML systems (e.g., Oracle Approximation [12]) or to evaluate the effectiveness of existing ML test cases (e.g., mutation testing). In the context of ML software system, in addition to ensuring that the program behaves adequately, they also help ensure that important ML properties (*i.e.,* underspecification issues) are not violated during the development and evolution of the system [13]. 2) In the second part of understanding the ML test practices, we examined the different types of tests/ testing methods that are implemented in the ML development phases, and how they are used to verify the various ML properties. Testing method is the classification of various testing activities into categories (black or white-box), each aiming to validate the Application Under Test (AUT) for a defined set of test objective. Like in traditional software systems, the ML engineers may utilize numerous testing methods, such as Unit tests, Integration tests, or Manual tests, throughout the development process [2] to ensure that the ML software system can operate successfully in multiple conditions and across different platforms. Moreover, we examined how the testing strategies, test properties, and testing methods are being used (composition) across different ML software systems and across the ML properties.

1.2 Release Engineering Topics

In this part, we conduct a large-scale empirical study of release engineering related posts on Stack Overflow.

Release engineering deals with all activities in between regular development and delivery of a software product to the end user. Through a series of phases such as code integration from the development branches, build & compilation, package, testing, and signing of the product for release, release engineers transform developers' source code into a product ready for users' consumption [14–16]. Releasing complex traditional or ML software systems with hundreds to thousands of users can be challenging and requires skills that are not always well mastered by developers and engineers. In fact, release engineers must implement continuous delivery and deployment practices and must be knowledgeable about specialised technologies and tools that support activities like continuous integration & source control management,

testing, cloud provisioning, configuration management, application deployment, release orchestration [17]. It is therefore not surprising to see an increase of the prevalence of discussions about release engineering practices and tools on Q&A online developer forums, such as Stack Overflow.

By applying topic modeling [18] to understand the discussion topics of release engineers and identify the most important challenges that they face. We find that (i) developers discuss on a broader range of 38 release engineering topics covering all the six phases of modern release engineering, (ii) the topics *Merge Conflict, Branching & Remote Upstream* are more popular, while topics *Code review, Web deployment, MobileApp Debugging & Deployment, Continuous Deployment* are less popular yet more complicated, (iii) - Particularly, the release engineering topic "security" is both popular and difficult according to data collected from StackOverflow.

1.3 Thesis Overview

In this thesis, we report three empirical studies. First, we perform the fine-grained empirical study on ML testing practices in the wild to identify the ML properties being tested, the testing strategies follow, and their implementation throughout the ML workflow. Secondly, using the projects selected in the first empirical study, we further examine the different testing methods that ML engineers implement during the development process of ML software systems. Third, we conduct a large-scale empirical study of release engineering posts on Stack Overflow and apply topic modeling to understand the discussion topics of release engineers and identify the most important challenges they face.

In the following, we detailed the three empirical studies:

1. *Studying the practice of Testing ML Software Application in the wild:* First, we systematically summarized the different testing strategies (e.g., Oracle Approximation), the ML test properties (e.g., Correctness, Robustness, Efficiency, Bias and Fairness)

   Then, we conducted the study using a mixture of qualitative and quantitative analysis. Specifically, we extracted the test files and the corresponding test cases of nine open source ML systems hosted on GitHub, and following an open coding procedure we manually examined the testing strategies proposed in the existing research literature that were implemented, as well as the tested properties with the aim to understand how the test are being operationized in the delivery of Machine Learning.

   Our findings show that: 1) Fault-injection, *Oracle Approximation*, *State transition*,

Value Range analysis, and *Decision & Logical conditional* checks are five commonly used testing strategies used by engineers throughout ML workflow. 2) We identify 20 ML properties that engineers frequently test in an ML workflow, including *Security & Privacy*, *Consistency*, *Functional Correctness*, *Robustness* and *Feature importance*. 3) The testing strategies such as Fault-injection (i.e., *Null Reference detection, Module import error*), Oracle Approximation (i.e., *Absolute Relative Tolerance, Error bounding*), *Negative Test*, *Value Range Analysis* are used across different ML workflow activities. Similarly, the ML properties such as *Consistency*, *Completeness*, or *Efficiency* are being tested across multiple ML workflow activities compared to other ML properties. Finally, we compare how the testing strategies and ML properties are used across project and then examine how the testing strategies are used to verify different ML properties across the studied ML software systems. We found that at least two (2) testing strategies are used to verify a single ML property (as observed for at least 80% of the identified ML properties).

2. *Studying the Types of Test in ML Development Phases:* In this second empirical study, we used the same dataset collected in the first empirical study above but now focused on understanding the testing methods implemented by ML engineers. We first systematically summarized the different types of test/ testing methods (e.g., Unit, Integration, Manual/ Exploratory Test) from the Test Pyramid for ML software systems introduced by Sato et al. [2]. Then we analysed the types of test used in the studied Ml software systems. We identified a total of 11 different types of tests, out of which only six are included in the Test Pyramid of ML software system. The newly observed types of tests are : *Regression Test*, *Sanity test*, *Periodic Validation and Verification*, *Thread test*, and *Blob test*. This finding suggests that the current Test Pyramid of ML based software system proposed by Sato et al. is incomplete and should be updated. We also observed that the following ML properties are tested at different levels on the Test Pyramid of ML (from the unit level to the system level): *Consistency*, *Completeness*, *Correctness*, *Validity*, and *Data Distribution*.

3. *Studying the Release Engineering Topics using StackOverflow:* By using the Latent Dirichlet allocation (LDA) [18] technique, we group and label StackOverflow posts into 38 topics, from a total of $260,023$ release engineering questions and answers posted in a period of 11 years; i.e., from 2008 to 2019.

   Then, we analyze the popularity of release engineering topics using 3 well-known metrics, and find the most popular topics to be: *Merge Conflict, Branching & Remote Upstream, and Feature Expansion*.

Also, we identified the topics that are challenging for the release engineers using a set of 2 well-known metrics. We find that topics *MobileApp Debug & Deployment, Continuous Deployment and Docker* are among the most difficult, suggesting that novel tools and techniques may be needed to support release engineering teams performing these activities. Also, we found that the topic *Software Testing* has the highest percentage of questions asked in the CI/CD phase of release engineering process, potentially indicate the challenge in the software testing practice.

Moreover we examined the correlation between the popularity and difficulty of release engineering topics and found that topic Security is both popular and difficult, topics *Merge Conflict, Branching & Remote Upstream, Feature Expansion* are among the most popular, while topics *MobileApp Debug & Deployment, Continuous Deployment, and Docker* are among the most difficult.

Finally, we grouped the questions asked by release engineers into *How*?, *Why*? and *What*? categories. We found that release engineers frequently ask about *how*? to do things; often seeking clarifications and explanations (i.e., *what*?), and less frequently questioning (i.e., *why*?) certain aspects of release engineering practices, techniques, and tools. The high percentage of questions in the category *How*? suggests that release engineers need support to create working solutions. Our dataset is available at [19]

1.4 Thesis Contributions

This thesis make the following main research contributions:

- To the best of our knowledge, this is the first empirical study that examines the adoption of research advances in software testing of ML software systems by the industries, specifically by examining the testing strategies, the test properties, and the testing methods adopted in the field.
- For the testing strategies, we highlighted nine major categories of the testing strategies used by ML engineers, among which only the *Oracle Approximation* has been empirically studied. We reported eight newly derived testing strategies from practices, which are: *Fault Injection*, *Negative Test*, *State Transition*, *Value Range Analysis*, *Instance and Type Checks*, *Decision and Logical condition*, *Membership Testing*, and *Swarming testing*.
- For the testing properties, we derived a total of 20 different ML properties that the ML engineers commonly test. Among them, only five of the derived ML properties have

been listed by the previous research work (*i.e.,* Correctness, Completeness, Robustness, Efficiency and Bias and Fairness) [20].

- We summarized 11 different types of tests, out of which five are not included in the Test Pyramid of ML software systems. These newly identified test types are: *Regression Test*, *Sanity test*, *Periodic Validation and Verification*, *Thread test*, and *Blob test*.
- We provided a comparison of the testing strategies, the testing properties, and testing methods used across the studied projects to identify common or inconsistent practices in their existing testing practices. We also compared how the testing techniques are used to verify each testing property, across the studied ML software systems.
- We highlighted some challenges and proposed new research directions for the research community. ML practitioners can also leverage our findings to learn about different testing strategies, properties to test, and testing methods that can be implemented to improve the reliability of their next ML software system.
- For the release engineering topics, we reported a total of 38 release engineering topics covering all the six phases of modern release engineering.
- We identified the most prevalent release engineering topics and topics that are challenging for the release engineers and the topics that are both popular and challenging to find answers on StackOverflow. Also, we identified the types of release engineering questions asked by release engineers.
- We highlight a set of implications of the release engineering topics derived in this thesis to the researchers, the practitioners, and the educators.

1.5 Thesis Implication

Overall, the findings presented in this thesis provide the following actionable implications:

ML engineers can use our presented taxonomy to learn about the existing ML testing strategies that they can implement in their ML workflow, especially the most frequently used testing strategies. Our results can also guide them on how to test different ML properties. Also, we recommend that ML and release engineering teams take into consideration the difficult release engineering topics when distributing works between the project team members. Release activities related to *Security, Docker, Virtualization, Code Review and Continuous deployment* are more difficult and they should be assigned to the more experienced team

members. Whereas the tasks related to code integration (with the exception of code review), could be assigned to even a less experienced team members. Moreover, we found that *Software Testing* is the most asked questions (highest percentage) in the category CI/CD of release engineering. This can potentially indicate that *Software Testing* is a challenge when automating the delivery process of ML and traditional software.

Researchers can build on our work to further investigate the effectiveness of different ML testing strategies, and develop quality checking mechanisms for ML software pipelines. We also, encourage researchers to tackle the top 10 difficult releng topics that we have identified, i.e., *MobileApp Debug & Deployment, Code Review, Docker, Continuous Deployment, Virtualization, Web deployment, Build Error Debug, Web UI Testing, Build System Performance, Security and Configuration Management*. The *Security* topic is both difficult and popular. This calls for more attention on releng security challenges. By infecting a build, malicious users could distribute their malware to thousand and even millions of users. Researchers should invest in developing efficient techniques and tools to support release engineers.

Designers of testing tools can develop better tooling support to help ML maintenance team effectively tests for the 20 common ML properties identified throughout ML development lifecycle. They can also provide infrastructure to automate the identified tests on the continuous delivery pipeline of ML systems. Also, novel tools and techniques may be needed to support release engineering teams performing the activities on the most difficult release engineering topics *MobileApp Debug & Deployment, Continuous Deployment and Docker*.

The popularity of release engineering topics *Merge Conflict, Branching & Remote Upstream, and Feature Expansion* suggests that despite the many version control systems and source code management tools that exist, developers still struggle with merge conflicts and branching issues.

1.6 Thesis organization

Chapter 2 provides all background information of the concept used in this thesis, including the testing strategies, ML properties, the Test Pyramid for ML, the ML workflow, and Modern release engineering and corresponding the six major phases.

Chapter 3 details the software engineering research works on ML testing and release engineering. We also noted that only few research works has explored the topics on ML software testing and release engineering.

Chapter 4 presents the first empirical study examining the the adoption of testing strategies,

and ML properties from the literature in the field. In this Chapter we answer our first four proposed research questions.

In Chapter 5, we present the second empirical study examining the the adoption of types of test, in the field, and how the test are used to test for different ML properties identified in Chapter 4. In this Chapter we answer three new proposed research questions.

In Chapter 6, we present the third empirical study on understanding the release engineering topics, the engineers asks in StackOverflow. We answer five more research questions in this Chapter.

In Chapter 7, we summarized the all the studies presented in this thesis. We also highlights our future research directions and conclude the thesis.

CHAPTER 2 BACKGROUND

2.1 Testing practices for ML software system

This subsection covers the background information on testing strategy, ML properties, ML Workflow and the test Pyramid for ML software systems in a continuous delivery pipeline.

2.1.1 Testing Strategies and ML Properties

Testing Strategies are the various techniques or approaches used in ensuring the quality of ML software systems. ML testing strategies are important to ensure that test activities meet software quality assurance objectives, and that the best approaches are used to test the behavior of ML systems (e.g., Oracle Approximation [12]) or to evaluate the effectiveness of existing ML test cases (e.g., mutation testing). In the context of ML software system, in addition to ensuring that the program behaves adequately, they also help ensure that important ML properties (*i.e.,* underspecification issues) are not violated during the development and evolution of the system [13].

ML Properties are requirements that are essential to ensure that the models generalize as expected in deployment scenarios. A violation of such property often results in instability and poor model behavior in practice. Testing these properties is not only important for producing high-quality software systems, but also for regulatory or auditing purposes [21].

2.1.2 Testing in Python and C/C++-based projects:

Here introduces the terminology used throughout this thesis (i.e., test cases, tests, testing strategies, and testing properties/ ML properties) using examples. Python and C/C++ are the main programming languages [22] used in ML models and ML software systems, hence we are focusing on their testing practices.

Python provides a rich support of test packages or frameworks to handle both white-box and black-box testing. Moreover, it is also possible to write any black-box test in Python, even if the AUT is not written in Python. Depending on the types of tests, different test frameworks may be chosen. For example, a basic unit test could be handled by the `unittest` framework (i.e., a build-in test framework into Python standard library), but other test frameworks like `pytest` may work better for higher-level testing such as using test fixtures [23].

Figure 2.1 shows the example code of a test case in Python that uses the build-in unit test

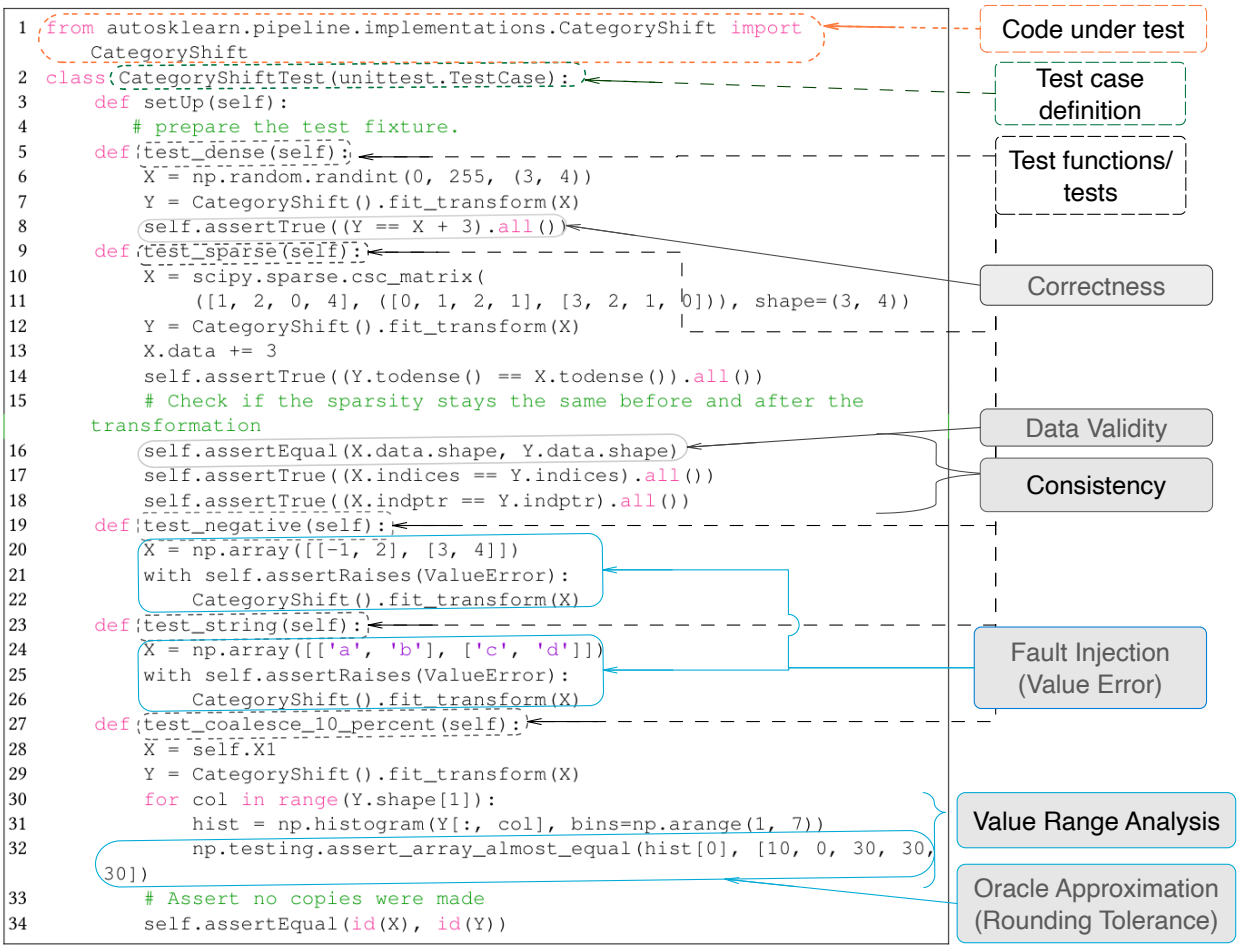


Figure 2.1 The example of a test file in Python that uses `unittest` testing framework. We use the different color codes defined as follows:
The *dotted orange* line is the code under test, *dotted green* line indicate the test case definition, *dotted gray* lines indicate the test functions/ tests, *bold gray* lines indicate the ML properties being tested, and the bold light blue color indicate the testing strategies.

framework (`unittest`). A *test case* is a collection of unit tests that demonstrates that a function works as expected under a broad range of conditions (in which that function may find itself and is expected to work successfully). Usually, a test case considers all possible kinds of input a function under test is expected to receive from users and therefore contains the tests (test functions/ methods) representing each expected input situation. According to Figure 2.1, the name of the test case is `'CategoryShiftTest'` and it contains five (5) test functions (i.e., methods/functions containing the word 'test') such as `'test_dense'`, `'test_sparse()'`, `'test_negative` `'test_string()'`, and `'test_coalesce_10_percent()'`. For simplicity, we will refer to these functions/methods defined inside the test cases containing the word 'test' as *Test functions* or *Tests* inter-

changeably throughout this thesis. A test file may contain one or multiple test cases, for example the following reference [24] contains a test file with three test cases corresponding to three functions. Also, for the situations where the test functions/methods are independent of each other (no class defined), e.g., manual testing or integration testing, such as in [25], each test function is defined a separate test case C/C++ supports multiple unit testing frameworks. The most popular testing frameworks are *Google C++* (gTest) [26] and *Boost.Test* (Boost) [27]. Both frameworks have closely related features that allow the creation of test cases and their organization as test suites that can be registered automatically or manually. Other features supported by both frameworks include the addition of test fixtures for each test case, test suite, or globally for all test cases. Test fixtures allow for a consistent initialization and cleaning of resources during the testing process. They also reduce code duplication between test cases. Other unit testing frameworks used for C/C++ based software systems include CppUTest [28], Unity [29].

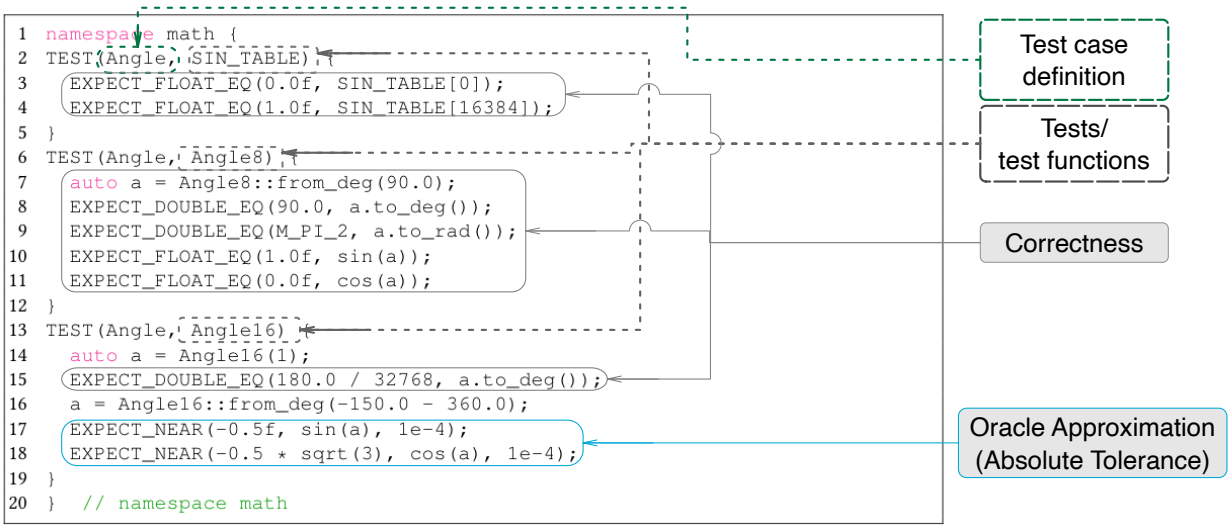


Figure 2.2 The example of test file in C++ that uses `'Google Test'` testing framework. We use the different color codes defined as follows:
The *dotted green* line indicate the test case definition, *dotted gray* lines indicate the test functions/ tests, *bold gray* lines indicate the ML properties being tested, and the bold light blue color indicate the testing strategies.

Figure 2.2 shows an example of test case using the gTest framework. Figure 2.2 creates a test case named `'Angle'` that contains three (3) unit tests (similarly, for simplicity, we will refer to them as *Test functions* or *Tests* throughout this study) `'SIN_TABLE'`, `'Angle8'` and `'Angle16'`. TEST is a predefined macro defined in gTest that helps define the test case.

A *testing strategy* is a technique used to verify that an application or function behaves as

expected. A *test property* is a software requirement (functional and non-functional) that should be meet to ensure that the software product behaves as expected. In the context of ML based applications, ML properties are essential to ensure that the models generalize as expected in deployment scenarios. Figure 2.1 and Figure 2.2 presents an example of test cases implementing different testing strategies for testing different properties. In this work, we identify test properties and testing strategies from such test. In Figure 2.1 and Figure 2.2, we further highlighted the different testing strategies and ML properties being tested using the colour codes *Bold light blue* lines and *Bold gray* lines, respectively. We will describe in details the labeling procedure we followed to derive the different testing strategies and ML properties in Step (5) and Step (6) of our methodology (Section **??**).

2.1.3 Machine Learning workflow

Figure 2.3 highlights the nine activities of a machine learning workflow to build and deploy ML software system adopted from Microsoft [1] which is also similar as other companies' process, such as Google [30] or IBM [31].

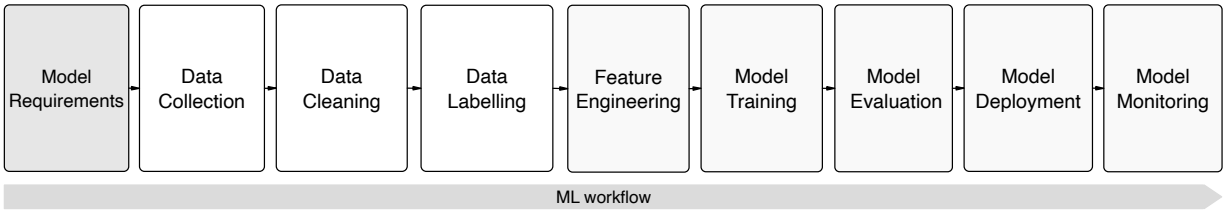


Figure 2.3 An overview of the machine learning workflow adopted from [1]

- *Model Requirements:* When constructing a machine learning model, the primary stage is the models requirements stage. Based on the product and the problem statement, the decision is made on the appropriate types of models for the problem and features to implement with machine learning.
- *Data collection:* This stage follows after the model requirements, where the data from different data sources are identified and gathered. This data may come from batch storage software component such as databases, file storage or generated from external components such as detection components (e.g., camera or LiDAR sensors). Indeed, most ML training would require a considerably large amount of data to train from and make meaningful inferences on the new data, making the data collection challenging.

- *Data cleaning:* Once the data is gathered, the next stage (*Data cleaning*) is to process or clean the data to remove anomalies that are likely to hinder the training phase. Most activities common to data science are performed during this step, such as generating descriptive statistics about the features in the data, the distribution of the number of values per example.

- *Data labeling:* Assigns ground truth labels to each data record, such as assigning labels to a set of images with the objects present in the image. This step is required in most supervised learning techniques to induce a model.

- *Feature engineering:* This stage is where the features are being prepared and validated for training. This stage also include data validation which ensure that properties such as data schema (e.g., features present in the data, expected type of each feature) are correct. This step further ensures the quality of the training data. As a ML platform scales to larger data and runs continuously, there is a strong need for an efficient component that allows a rigorous inspection for data quality.

- *Model Training:* The training stage uses the prepared features on different implementations of algorithms to train ML models. Also, the implemented algorithms are subjected to hyperparameter tuning to get the best performing ML model. Tests such as verifying the validity of model input parameter, or memory leak error detection may be performed during this step. The result from this step is a trained ML model.

- *Model Evaluation:* The trained model is then evaluated on an holdout data set, commonly referred to as validation set. This evaluation stage is required to confirm the model adequacy for deployment; *i.e.,* that it can generalize beyond training data.

- *Model Deployment:* Once the model is evaluated, the ML Model and entire workflow are then exported (i.e., meta-data information of the model are written in a file or data store.), to be reused (imported) in other platforms for scoring and—or making predictions. Developers may test the exported model against the original model's performance for consistency. Moreover, the portability of the model may be tested at this stage, to allow its deployment on different platforms (e.g., a model designed in Python can be utilized inside an Android app).

- *Model Monitoring:* The final stage of an ML workflow is monitoring, to ensure that the ML model is doing what is expected from it in a production environment.

  Behold the standard workflow structure; in most cases, there is a need for shared *configuration and utilities* to allow for integrating these components in a single platform,

ensuring consistency across the workflow. For example, transformations at a serving time may utilize statistics generated by the data analysis component using a shared utility.

Throughout this thesis, we will refer to the different stages of the ML workflow described in Figure 2.3 (including the *configuration and utility*) as ML workflow activities.

2.1.4 testing methods and Test Pyramid for ML software system

Software Testing method is the classification of various testing activities into categories (black or white-box), each aiming to validate the Application Under Test (AUT) for a defined set of test objective. Like in Traditional software systems, the ML engineers may utilize numerous testing methods (such as Unit tests, Integration tests, or Manual tests) throughout the development process [2] to ensure the ML application can operate successfully in multiple conditions and across different platforms.

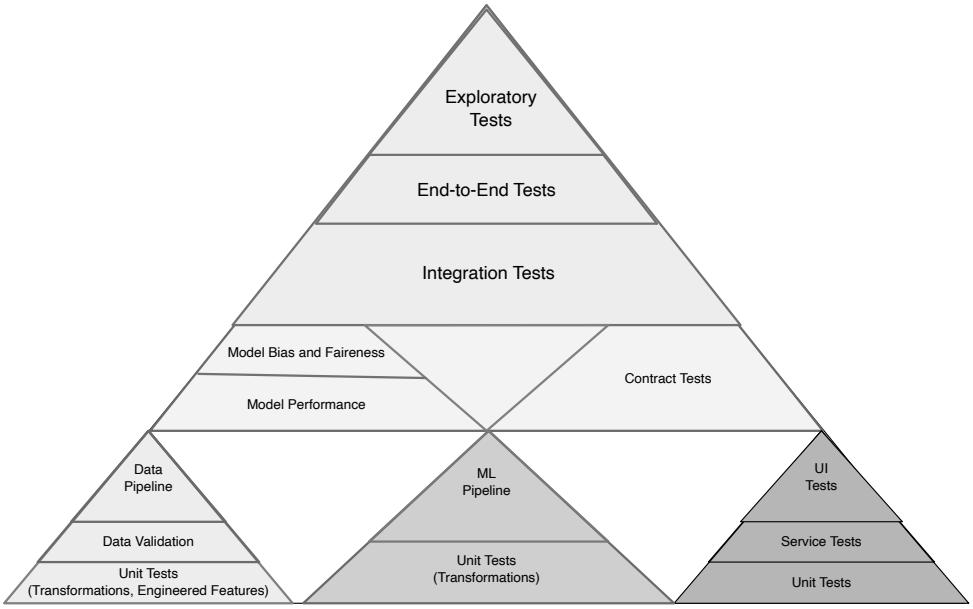


Figure 2.4 The Test Pyramid for Continuous Delivery of Machine Learning based software system proposed in [2]

Tests in machine learning-based projects are applied to the following three high-level components: data, model, and ML code. Each of these components containing multiple functions. Sato et al. [2] proposed a revised version of the initial Test Pyramid introduced by Mike Cohn [32], to help developers of ML software system visualize different layers of testing and

estimate the amount of testing effort or frequency required on each layer (*i.e.,* the higher is the level of a type of test on the Pyramid, the lower should be the number of its instances). This Test Pyramid shown in Figure 2.4 is composed of the following types of test

- *Unit tests:* They are in the first level of testing at the bottom of the test pyramid. They aim to verify the correctness of each functionality once the implementation is complete. We discuss some examples of unit test cases in the following paragraph.

  The data pipeline takes input data through a series of transformation steps to generate the respective output data. Usually, the input and output data are retrieved or stored in locations such as: a database system, a stream processing, or a file system.

- *Data validation tests:* Ensure that input data values are valid and follow the right format. This is done for example by checking for the null value, checking the shape of the data or ensuring that input data fall within a specific range. Unit tests can also be used to check that numeric features are scaled or normalized properly, that missing values are replaced appropriately, or that features vectors are encoded correctly.

- *Services* are the functional unit of application or business process, which can be reused or repeated by any other application or process (i.e., a collection of services that are part of a single functionality). Service Level Testing includes testing the component for functionality, interoperability, security, and performance. Unit testing ensures each service are tested independently.

- *Model performance tests:* Are used to compare performance metrics (such as loss error, accuracy score, precision and recall, AUC, ROC and confusion matrix) against a specific threshold, to validate the quality of a model before its deployment.

- *User Interface (UI) tests:* Are used to verify the aspects of any software application that an end-user will interact with. This involves testing any visual elements of the software system, verifying that they are functioning according to requirements (in terms of functionality and performance and ensuring they are bug-free).

- *Bias* in the ML model occurs when there is a systematic prejudice in the ML model for some set of feature values resulting from wrongful assumptions made during the training process. Many sources of bias exist, such as sampling bias (i.e., over-represented or under-represented in a training dataset) or label bias (i.e., when the annotation process introduces bias during the creation of the training data). On the other hand, an ML algorithm is fair if its results are not dependent on some variables, such as those

considered sensitive. For example, the characteristics of individuals (such as sexual orientation, gender, disability, ethnicity) should not correlate with the ML model outcome. Model bias and Fairness tests are used to ensure that no bias is introduced from the training data, for example by checking how the model performs on a specific data slice.

- *Contract test:* Is a testing method for checking the interaction between two separate systems, such as two microservices for an API provider and a client. It captures and stores the communication exchanged between each service in a contract, and the stored interaction is used to verify that both parties adhere to it. Compared to other closely related approaches (such as Schema testing, Compatibility testing [33]), in the Contract test, the two systems can be tested independently from each other and the contract is autogenerated using code.

- *Integration tests:* Aim to ensure that small combinations of different units (usually two units) behave as expected and testing that they coherently work together.

- The *end-to-end test:* Is a type black-box testing type that targets the entire software product from start to finish, ensuring that the application behaves as expected. Through simulating real users' scenarios, this type of test checks whether all integrated components can work together as expected.

- *Exploratory tests:* Are tests performed on the fly (i.e., test cases are not created in advance), whereby the test cases are designed and executed simultaneously, and the results observed are used to design the next test.

For simplicity, in this thesis we will use the term test types or testing methods interchangeably to refer to the test described in Figure 2.4.

2.2 Modern Release Engineering

Release engineering is the process of bringing a high-quality software release product from the individual code contributions of developers to the end-user. This involves a myriad of tasks that need to be performed from beginning to finish by an organization's release engineers (i.e., the individuals responsible for developing, operating, maintaining an organization's release infrastructure) [16]. A modern release engineering pipeline consists of six (6) major phases discussed in the following paragraphs.

Integration: This first phase of the release engineering pipeline consists of two main activities: branching and merging is where the source code changes from different developers branches (development branches) are moved to a central project repository (master branch) [34]. The key goal of any software organization is to bring high-quality code changes as fast as possible to the central repository without affecting code quality [35]. Software development teams usually rely on Version Control Systems (VCSs) such as Git or Subversion to store the subsequent revisions of each manually created and maintained file. The project branches are created from some parent branch, such as a master branch, to allow the development team to record a chronological sequence of code changes ("commits") on their individual branch. The changes are made visible to the rest of the team after the merging—for example, code changes related to fixing a bug or a proposed new feature.

Continuous Integration (CI): This phase of release engineering consists of activities building and Testing. CI is the process of continuously polling the VCS for new developers' commits or merges, verifying and compiling them on dedicated build machines, and running the regressions test [36, 37], to ensure that the new code change do not bring any regressions to the existing source code.

Build System: This is the set of build specification files used by the Continous Integration tools (and software engineers) to generate the necessary software deliverables such as binaries, libraries, or packages from the software source code. In addition, some build system further automates many other activities, such as test execution and deployment [38].

Infrastructure-as-Code: The term "infrastructure" (or "environment") refers to either a cloud, a server, container, or virtual machine on which a new software version is deployed for testing or even for production. Previously, the deployment team has to manually configure a server or a virtual machine in line with the deployment demand or whenever a new version of a software deliverable is available. The infrastructure-as-code is the most recent innovation in release engineering [39] where the correct environment based specification is generated automatically using a dedicated programming language such as *Ansible*, *Puppet*, *CFEngine*, *Chef*, and *Salt*.

Deployment: In this phase, the tested deliverables for the new release are staged, waiting to be released [39, 40]. For example, in the context of web applications, deployment would mean pushing ML software systems deliverables across a network to the right directory on a web server. In contrast, deployment of a mobile application may include uploading an application's binary to the app store to be ready for release phase (below).

Release: This final phase of a modern release engineering pipeline makes the deployed

releases available to the end-user [40]. This phase is followed relatively faster once the deployment phase is complete. With the aid of modern release engineering tools available, this step has become as simple as clicking a button to make a new release version visible to the user, for example, releasing a mobile app in the app store. Moreover, some releasing mechanisms allow fine-grained access to software based on the user subscription level. Different system users may access different variants of the software functionality, e.g., a premium user may have early access to the new release before it is visible to the other users or may access a "deluxe" version of the software.

CHAPTER 3 A COMPREHENSIVE REVIEW OF SOFTWARE ENGINEERING STUDIES

3.1 Software Engineering Studies on Testing practices for ML software system

This subsection discuss the related works on testing practices for ML software systems.

Braiek and Khomh [41] described the main sources of faults occurring during the development process of ML based systems, from data preparation to the deployment of models in production. Then, they reviewed testing techniques proposed in the literature to help detect these faults both at the implementation and model levels, describing the context in which they can be applied and their anticipated outcome. Finally, they highlighted gaps in the literature related to testing of ML programs and suggest some future research directions. Zhang et al. [20] presented a comprehensive review of ML testing research, covering testing properties (such as correctness, robustness, and fairness), testing components (i.e., data, learning program, and framework), testing workflows (i.e., test generation and test evaluation), and some application scenarios (e.g., autonomous driving, machine translation). They defined ML testing as "any activity designed to reveal machine learning bugs" and ML bugs as any flaw in a machine learning item that leads to an inconsistency between the existing and the required conditions. They also analyzed research trends, ML testing challenges, and discussed some avenues for future researches. In this thesis, we examine how the testing properties and the strategies discussed in their literature review are adopted in practice. Masuda et al. [42] investigated challenges in ensuring the software quality of ML services available through APIs on the cloud (i.e., ML-as-a-services). They observe that ML-as-a-services products lack rigorous quality evaluations, and that there are no specialized techniques adapted to the specificity of this type of ML software system. Further, they highlighted and discussed problems related to the deployment of ML models in real-world situations. They also discussed about adapting well-known software engineering methodologies to ML applications. Nikanjam A. et al [43] proposed *NeuraLint* for detecting a fault in DL software systems. Their fault detection approach was built on top of meta-modeling and graph transformations to cover up to 23 rules to detect flaws and design issues in the generated models (i.e., instances of the meta-model). They benchmarked their solution on both synthesized and real-world DL programs extracted from Stack Overflow posts and GitHub repositories and found 64 design inefficiencies and flaws with a 100% precision and 70.5%.

Among the ML testing strategies discussed in the literature, *Oracle Approximation* is the strategy that recently received an attention [12]. *Bias and Fairness* are the testing properties

that received an attention and was studied empirically in details [44].

• **Oracle Approximation:** Many reasons may require ML engineers to use Oracle approximations in ML software system. For example, the implementation results in the ML software system may slightly differ in each test run due to randomness. Also, ML engineers may find it hard to define the exact value for a test involving floating-point numbers, thereby allowing a considerable difference between the computed results and the expected outcomes. Nejadgholi et al. [12] examined oracle approximation practices in DL libraries. Specifically, they investigated the prevalence of oracle approximations in the test cases of DL libraries and found that up to 25% of all the assertions use oracle approximations. The identified Oracle Approximation techniques include: *Absolute Relative Tolerance*, *Absolute Tolerance*, *Rounding Tolerance* and *Error Bounding*. Then, they studied the diversity of test oracles and thresholds used in oracle approximations and found that oracles used in oracle approximation are mainly derived through computation. Finally, they examined the intent behind code changes in oracle approximation, and reported some maintenance challenges faced by DL engineers when using oracle approximation.

• **Bias and Fairness:** ML software system in addition to being correct and robust should be fair and without bias. Recently, Galhotra et al. [44] proposed a metric-based testing approach for measuring fairness in machine learning software systems. The approach measures the fraction of inputs for which changing specific characteristics causes the output to change, by relying on causality-based discrimination measures. Using these metrics, automatic test suites are generated to allow for the detection of any form of discrimination in the ML models under test. The test cases modify training data inputs that are related to a sensitive attribute aiming at verifying if the modification causes a change in the outcome. Also, for a given sensitive attribute (SA), a fairness test validates the model under test by altering the value of SA for any input and verifying that the output remains unchanged. Using the proposed fairness test they examined the fairness of 20 real-world ML software systems (12 of these ML software systems were designed with fairness in mind) and observed that even when fairness is a design goal, developers can easily introduce discrimination in software.

3.2 Software Engineering Studies on Release Engineering

This subsection discuss the related works on testing practices for ML software systems.

Generally speaking, release engineering remains one of the least studied areas in software engineering. Here we discuss some relevant release engineering works. We also discuss some works that used topic modeling.

(1) Release Engineering: Adams and McIntosh [16] summarized releng activities into six major phases, and outlined some research direction for the community. Wright and Perry [45] conducted semi-structured interviews with release engineers to understand why release process faults and failures occur and how companies recover from them, and how they can be foretold, and avoided. Castelluccio et al. [46] examined patch uplift operations in rapid release pipelines and formulate recommendations for improving their reliability. Khomh et al. [47] analyzed rapid release practices at Mozilla and found that despite the benefits of speeding up the delivery of new features to users, shorter release cycles can negatively impact software quality. Karvonen et al. [48] performed a systematic literature review to understand the different impacts (both direct and indirect) of agile release engineering (ARE) practices and how the empirical research has studied them. Specifically, they used search keys rapid release, continuous integration, continuous delivery, and continuous deployment to search for the different research works on ARE. They found that ARE practices can create shorter lead times and better communication within and between development teams. In addition, they reported different challenges and drawbacks of ARE in change management, software quality assurance, and stakeholder acceptance. Kerzazi [49] performed an empirical study to determine and compare the main tasks of release and DevOps engineers, globally and across countries, by analyzing the online job postings. They indicated that the automation step is one of the most important activities across the three roles, as articulated in job posting description data. The release engineer role combines the top activities of the DevOps and more traditional build engineer roles.

(2) Topic Modeling: Mehdi et al. [50] conduct a study to understand what big data developers discuss in StackOverflow. They used topic modeling techniques to identify 28 big data topics, and analyzed their popularity and difficulty. Anton et al. [51] used latent Dirichlet allocation (LDA) to analyze posts related to software development in StackOverflow. They compared their relationships and trends over time, to gain insights about the development community. Rosen and Shihab [52] used latent Dirichlet allocation (LDA) to summarized mobile-app related discussions in StackOverflow. They further determined the popular and challenging mobile-related issues, explored issues specific to the platform, and investigated the types of questions (e.g., what, how, or why) that mobile developers ask.

CHAPTER 4 STUDYING THE PRACTICES OF TESTING MACHINE LEARNING SOFTWARE IN THE WILD

4.1 Introduction

The increasing adoption of Machine Learning (ML) and Deep Learning (DL) in many software systems, including safety-critical software systems (e.g., autonomous driving [3, 4], medical software systems [5]) raises concerns about their reliability and trustworthiness. Ensuring the quality of these software systems is yet an open challenge for the research community. The main reason behind the difficulty to ensure quality in ML software systems is the shift in the development paradigm induced by ML. Contrary to traditional software systems, where the engineers have to manually formulate the rules that govern the behavior of the software system as program code, in ML the algorithm automatically formulates the rules from the data. This paradigm makes it difficult to reason about the behavior of software systems with ML components, resulting in software systems that are intrinsically challenging to test and verify. A defect in an ML software system may come from its training data, program code, execution environment, or even third-party frameworks. A Few research advances in quality assurance on ML systems recently have been adapting different concepts from traditional software testing (i.e., software not using ML), such as evaluating their effectiveness (e.g., mutation testing [8,9]) and new techniques to verify the security or privacy of the ML system (e.g., Spoofing attacks in autonomous systems [10,11]), to help improve the reliability of ML based software systems. Little is known about the current practices on testing ML software systems. On the other hand, it is unclear if there are new unique testing techniques originated from the practice.

Therefore, in this chapter, we present the first empirical study that examined testing strategies used by ML engineers throughout the ML workflow. We define testing strategies as the various techniques or approaches used in ensuring the quality of ML systems. ML testing strategies are important to ensure that test activities meet software quality assurance objectives, and that the best approaches are used to test the behavior of ML systems (e.g., Oracle Approximation [12]) or to evaluate the effectiveness of existing ML test cases (e.g., mutation testing). In the context of ML software system, in addition to ensuring that the program behaves adequately, they also help ensure that important ML properties (*i.e.,* underspecification issues) are not violated during the development and evolution of the system [13]. We manually analyse the test code contents of nine ML software systems from six (6) different application domains (including Autonomous driving, ML toolkit, NLP/ Voice recognition,

Intelligent computing and Distributed ML) by following an open coding procedure to derive a taxonomy of the ML testing strategies used by ML engineers. Secondly, we examine the specific ML properties that engineers commonly test throughout the ML workflow. ML properties are requirements that are essential to ensure that the models generalize as expected in deployment scenarios. A violation of such property often results in instability and poor model behavior in practice. Testing these properties is not only important for producing high-quality software systems, but also for regulatory or auditing purposes [21]. Then, we examine the composition (in percentage) of the testing strategies and ML properties across the projects and how the testing strategies are used to verify different ML properties. Specifically, in this chapter, we study the following four research questions:

RQ1 *What are the common testing strategies used in an ML workflow?*

Through a manual analysis of ML test cases following an open coding procedure, we derived a taxonomy of nine major categories of test strategies highly dominated by *Fault Injection (19.22%)* followed by *State Transition (18.59%)*, *Value Range Analysis (17.20%)*, *Oracle Approximation (16.79%)*, and *Swarming testing* (0.53%). Among the identified testing strategies, only the *Oracle Approximation* has been empirically studied. The newly derived testing strategies from practices are: *Fault Injection*, *Negative Test*, *State Transition*, *Value Range Analysis*, *Instance and Type Checks*, *Decision and Logical condition*, *Membership Testing*, and *Swarming testing*.

Overall, we found that a high proportion of testing activities take place during model training (32.68%) followed by feature engineering (13.9%) activities of the ML workflow.

Moreover, the testing strategies are used across at least 50% of different ML workflow activities include Fault Injection (*Value Error, Null Reference, Type Error*), Oracle Approximation (*Absolute Relative Tolerance* and *Error Bounding*), *Decision and Logical Condition*, and *State Transition*. Finally, we examined the testing strategies that are used to test only the ML code in the studied systems, and identified the following six testing strategies (derived from three main categories): Fault Injection (*Value Error, Type Error, and Memory Error*), Oracle Approximation (*Absolute Relative Tolerance, Rounding Tolerance*), and *Membership Testing*.

RQ2 *What are the specific ML properties tested in an ML workflow?*

We manually classified the tested ML properties, following an open coding procedure, and identified 20 commonly tested ML properties in ML workflows. Among these properties, we have: *Functional Correctness*, *Consistency*, *Data Distribution*, *Data relation*,

Efficiency, *Data validity*, *Security*, and *Feature importance*. Some of the identified ML properties such as *Uncertainty*, *Security & Privacy*, *Concurrency*, and *Model Bias and Fairness* are tested in fewer ($\leq 50\%$) specific ML workflow activities. In contrast, the ML properties *Consistency*, *Completeness*, *Correctness*, *Data Validity*, and *Robustness* are tested in most ($\geq 80\%$) ML workflow activities. Notably, we categorized the ML properties and highlighted the ML properties such as *Correctness*, *Robustness*, or *Efficiency* which have been studied in the software engineering literature and the ML properties that are newly identified, such as *Data Distribution*, *Feature Importance*, *Data Uniqueness* and *Timeliness*.

RQ3 *Are there any testing strategies, and ML properties not being tested in some projects?*

We examined whether the ML testing strategies, and ML properties identified in **RQ1**, and **RQ2** are consistently tested across multiple ML software systems. We found that there is a non-uniform use of different testing strategies within and across the studied ML software systems. The testing strategies consistently used in at least 80% of the studied ML software systems include: *Absolute Relative Tolerance (Oracle Approximation)*, *Error bounding (Oracle Approximation)*, *Instance and Type Checks*, *Negative Test*, *State Transition*, *Value Range Analysis*, and *Value Error (Fault Injection)*.

For the ML properties, we found that only about 20% to 30% of the ML test properties such as *Correctness*, *Consistency*, *Completeness*, *Data Distribution*, *Data Validity*, and *Efficiency* are consistently being tested across at least 90% different ML software systems. In contrast, the ML test properties *Bias and Fairness*, *Compatibility and Portability*, *Security and Privacy*, *Data Timeliness* and *Uncertainty* are not tested consistently in most (about 80%) of the studied ML software systems.

RQ4 *Are testing strategies consistently used in verifying different ML properties?*

We examine the percentage of the testing strategies verifying a given ML property across the studied ML software systems. We found that at least two (2) testing strategies are used to verify a single ML property (as observed for at least 80% of the identified ML properties). The most tested ML properties across multiple testing strategies (specific to ML software testing) are: *Completeness*, *Correctness* and *Data Validity*. In particular, the testing strategies commonly used to test for *Data Validity* are: Fault Injection (*Value Error* and *Type Error*), and *Membership Testing*. The testing strategies, *Absolute Relative Tolerance*, and *Rounding Tolerance* are used commonly to test *Completeness* and *Correctness*. Similarly, the most frequently used ML specific testing strategies, which are used to verify the ML properties *Efficiency*, and *Feature Impor-*

tance, are *Memory Error*, and *Absolute Relative Tolerance*.

Overall, our findings provide actionable implications for three groups of audiences: (1) ML engineers can use our presented taxonomy to learn about the existing ML testing strategies that they can implement in their ML workflow, especially the most frequently used testing strategies. Our results can also guide them on how to test different ML properties. (2) Researchers can build on our work to further investigate the effectiveness of different ML testing strategies, and develop quality checking mechanisms for ML software pipelines. (3) Designers of testing tools can develop better tooling support to help ML maintenance team effectively tests for the 20 common identified ML properties throughout ML workflow. In summary, we make the following main contributions:

- To the best of our knowledge, this is the first empirical study that examines the adoption of research advances in software testing of ML software systems by the industries, specifically by examining the testing strategies and the test properties adopted in the field.
- For the testing strategies, we highlighted nine major categories of the testing strategies used by ML engineers, among which only the *Oracle Approximation* has been empirically studied. We reported eight newly derived testing strategies from practices, which are: *Fault Injection*, *Negative Test*, *State Transition*, *Value Range Analysis*, *Instance and Type Checks*, *Decision and Logical condition*, *Membership testing*, and *Swarming testing*.
- For the testing properties, we derived a total of 20 different ML properties that the ML engineers commonly test. Among them, only five of the derived ML properties have been listed by the previous research work (*i.e.,* Correctness, Completeness, Robustness, Efficiency and Bias and Fairness) [20].
- We provided a comparison of the testing strategies, and ML properties tested across the studied ML software systems to identify common or inconsistent practices in their existing testing practices.
- We compared how the testing strategies are used to verify each ML property, across the studied ML software systems.
- We highlighted some challenges and proposed new research directions for the research community. ML practitioners can also leverage our findings to learn about different

testing strategies, and the properties to test to improve the reliability of their next ML software system.

Chapter organization. Section 4.2 describes the eight major steps of our methodology. Section 4.3 presents the results of our analysis, answering our research questions. Section 4.4 discusses the results of our study. Section 4.5 explains the possible threats to the validity of this work, while Section 4.6 summaries this chapter.

4.2 Methodology

This section presents the methodology we followed to conduct this study. We used a sequential mixed-methods [53] comprising of both qualitative and quantitative analysis to answer our proposed research questions **RQ1** through **RQ4**. Figure 4.1 shows an overview of our methodology. In the following we describe each step in details.

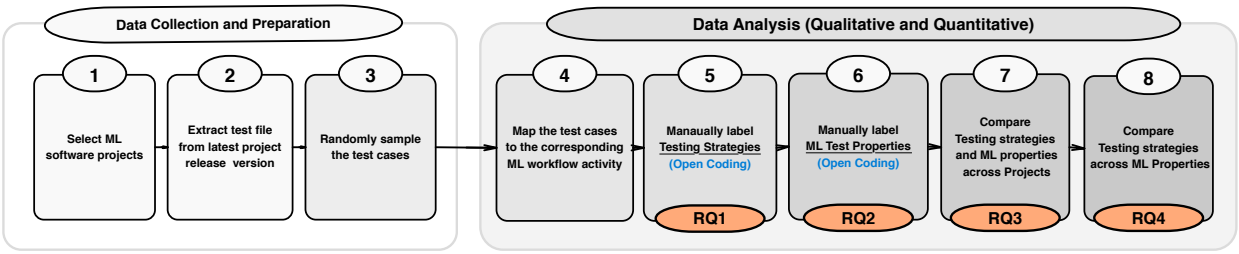


Figure 4.1 An overview of our study design/ methodology

① *Select ML software systems*:

To select ML software systems for our study, we first generated a set of relevant keywords for searching GitHub. We queried GitHub using the keywords, and filtered the resulting set of repositories using the inclusion/ exclusion criteria described bellow. Specifically, we proceed as follows.

(i) *Generate Search Keywords (T_ml)*:

This step aims to identify a rich set of keywords (topics) allowing us to capture a broad range of domains for the ML software systems hosted on GitHub. To generate the search keywords, by using the search API [54] provided by GitHub, we first searched through GitHub topics with the keywords “machine-learning” and “deep-learning”. Topics in GitHub are sets of labels assigned to repositories to allow searching and exploring

through the GitHub repositories basing on the technology, the type, or category. Using these initial keywords returned a set of repositories which we then manually summarized into twelve (12) major tag categories (T_{ml}) by looking at co-occurring tags including *'machine-learning', 'deep-learning', 'deep-neural-network', 'reinforcement-learning, 'artificial-intelligence', 'computer-vision', 'image-processing, 'neural-network, 'image-classification', 'convolutional-neural-networks', 'object-detection' and 'machine-intelligence'.*

(ii) *Extract Machine Learning Repositories Using T_{ml}*: Using the list of keywords obtained in the previous step, we queried the GitHub API [55], searching for repositories that: 1) contain at least one of the keywords in T_{ml} (case insensitive) either in the repository name, descriptions, or project README file; 2) are mainline software systems (i.e., not a forked repository); 3) the README or description is written in English (to allow us to obtain more details about the project); 4) are not archived repositories. This initial search returned a total of 506 unique repositories.

(iii) *Apply inclusion/exclusion criteria*: We applied the following inclusion/exclusion criteria to retain only mature ML software systems for our study: 1) We considered only repositories that had recorded at least 100 commits, 10 GitHub issues or pull requests, 10 contributors/ project engineers, and have been forked at least twice (to reduce the chance of selecting student's class projects). This criteria follows best practices established by previous studies [56–58]. 2) We retained only ML software systems that have been released at least twice (to allow us to later study the evolution of tests in the selected ML software systems) and have been active for at least one year (i.e., the active period is computed as the difference between the date of last update and the creation date of the project). This step removes 412 projects, and thus 94 ML repositories remain. 3) We manually filtered projects to retain only those where tests are written in either Python, C/C++ programming languages, which left us with nine ML software systems.

(iv) *Final List of Machine Learning Repositories*: Table 4.1 provides descriptive statistics about the nine retained ML software systems. These ML software systems are of varying size, of different types (i.e., AutoML frameworks, Toolkits, and ML applications), cover different domains (i.e., Autonomous driving, Intelligent computing, or Distributed ML), and has been stared more than $4,000$ times (the popularity of the selected ML software system [59]).

② ***Extraction of test files from the latest release of software systems*:**

Table 4.1 **List of studied Machine learning software systems and their characteristics.** *Com*: Number of commits, *Stars*: Number of Stars, *Language*: The dominant programming language and their composition, *Samp*: The random sample size of the test cases based on 95% confidence and a 5% confidence interval (refer to following step), *Release*: The release version studied, *Testing Framework*: The unit testing framework

	ML System	Com	Stars	Language	Samp	Release	Domain	Testing Framework	Project Descriptions
1	apollo	17,536	17,650	C/C++(83.7 Python(5.1%	206	v6.0.0	Autonomous driving	gTest, unittest	A high performance, flexible architecture autonomous driving platform for the development, testing, and deployment of Autonomous Vehicles. Apollo contains 28 ML models that are trained based on various deep learning frameworks (e.g., Caffe, Paddle, and PyTorch).
2	tpot	2,368	8,120	Python (99.4%)	141	v0.11.7	Data science	numpy, unittest, pytest	A Tree-based workflow Optimization Tool (Tpot) is a Python-based automated ML tool that uses genetic programming for optimizing machine learning workflows [60–62].
3	ray	7,895	15,700	Python (61.3%), C/C++ (26.7%)	159	ray-1.1.0	Distributed ML	numpy, unittest, pytest, Ray	An open-source framework for building distributed applications. Ray is packaged with a scalable reinforcement learning library, scalable Hyperparameter Tuning, distributed training wrappers, and scalable and programmable Serving
4	nni	2,023	9,500	Python (67.5%)	90	v2.0	ML toolkit	numpy, unittest, pytest	An open-source AutoML toolkit for automate machine learning lifecycle, including feature engineering, neural architecture search, model compression and hyperparameter tuning.
5	autokeras	1,250	7,920	Python (99.3%)	181	1.0.12	ML toolkit	numpy, unittest, pytest	A Python based Automated Machine Learning (AutoML) system based on deep learning, neural architecture search (NAS), which aims to search for the best neural network architecture for the given learning task and dataset [63].
6	auto-sklearn	2,454	5,410	Python (99.6%)	162	v0.12.0	ML toolkit	numpy, unittest	A Python based automated machine learning with scikit-learn from algorithm selection and hyperparameter tuning. It leverages recent advantages in Bayesian optimization, meta-learning and ensemble construction
7	automl	6,24	4,210	Python (99.9%)	100	1.1	ML toolkit	numpy, unittest, pytest	A repository for Google Brain AutoML containing a list of AutoML related models and libraries for building and deploying a custom machine learning models for image, video, text, and tabular data.
8	nupic	6,625	6,187	Python (97.7%)	270	1.0.5	Intelligent computing	numpy, unittest, gTest	Numenta Platform for Intelligent Computing is an implementation of Hierarchical Temporal Memory (HTM), a theory of intelligence based strictly on the neuroscience of the neocortex. The HTM uses time-based continuous learning algorithms to store and recall spatial and temporal patterns.
9	DeepSpeech	3,329	16,165	C/C++ (68.1%), Python (21.4%)	124	v0.9.3	NLP/Voice recognition	Boost, gLog	An embedded (offline, on-device) open source speech recognition engine which can run in real time on devices ranging from a Raspberry Pi 4 to high power GPU servers. At its core, *DeepSpeech* uses a trained recurrent neural network (RNN) consisting of 5 layers of hidden units to ingests speech spectrograms and generates English text transcriptions.

This study aims to understand the testing practices applied in ML software systems by analyzing the test-related code. Hence, in this step, we identified all tests related source files from the latest release of each ML software system as follows: First, we identified the latest release version (R_n) of each selected ML software system (from which we need to extract the tests' related code) using Git tags REST API of the form: `https://api.github.com/repos/{owner}/{repo}/tags` (e.g., `https://api.github.com/repos/numenta/nupic/tags` for Nupic software system). Git releases represent the different snapshots of a project by marking a specific point in time of the repository's history using Git tags [1]. In this study, we considered only the latest stable release version (i.e., not pre-release versions) of each studied ML software system, to ensure that we analyze fully tested versions. Hence, whenever the latest release version R_n of a project is a pre-release (i.e., contain the word 'alpha' on release name indicating alpha release version), we select the previous stable version of the project. We do not consider the alpha release because usually, an alpha release may indicate

[1] `https://git-scm.com/book/en/v2/Git-Basics-Tagging`

that the current version is unstable and may not contain all of the planned features for the final version. The latest release for the studied ML software systems (at the time of this study) are listed in column 'Release' of Table 4.1.

After identifying the tags of the latest release R_n and the corresponding tagged source code in the target projects, we manually downloaded this source code, for further processing locally. We followed prior studies (e.g., [64, 65]) and utilize the naming convention to identify test files. Specifically, we extracted all the files of the local copy of the target projects versions R_n that contain the word 'test' either at the beginning or end of their file names. We used a spreadsheet software to store source code file information such as the file name, file path, and other related meta-data. We further assigned each of the file a sequential unique identifier (F_{id}), starting from the value 1, to help us easily refer to each file later during the manual labeling and the rest of the analysis.

③ ***Randomly sample the test cases*:**

In this step, first, we manually extracted all test cases and their corresponding test functions defined inside the test files while referring to the project documentation. For example, for the test using python unit testing frameworks, the test functions contain the word 'test' appended to the name of the function (for automatic test runner), while tests in google test use the macro TEST() to define the test or test fixtures. Then, for each project, we randomly selected a sample of test cases using a 95% confidence level and a 5% confidence interval. This resulted into a total sample of 1,450 test cases (T_s). The distribution of the sample sizes of test cases obtained for each of the 9 studied projects is shown in column '*Samples*' of Table 4.1.

④ ***Map the test cases to the corresponding ML workflow activity*:**

As part of the goal of this study is to understand how the different testing practices expand across the ML workflow, the first two authors manually mapped all the test cases (T_s) to the corresponding ML workflow activities (e.g., *Data Cleaning*, *Feature Engineering*) illustrated in Figure 2.3. To map a test case to an activity of the ML workflow, the authors used information such as the name of the test file, the incode comments (i.e., comments left inside the test source code describing what the test is about) and official documentation related to the code under test (to get familiar with what the test cases are about). They then assigned the ML workflow activity name (i.e., mapping) to the test case manually. Each decision was discussed between the two authors until a consensus was reached before a label was assigned.

⑤ ***Label the testing strategies implemented to test different ML properties*:**

Tests strategies are the various techniques or approaches used to test an application, in order to ensure that its behavior matches the specification. In this part, we specifically focused on forming the taxonomy of categories and sub-categories of testing strategies used in ML software systems. Four individuals consisting of the first two authors of this study and two additional researchers (a graduate student and a research professional with a PhD degree) was involved in the construction of the taxonomy. All the participants have strong knowledge in machine learning and software testing. Also, the participants used the research literature on testing strategies discussed in Section 3, during the preparation steps and when assigning labels. To generate the taxonomy, first, the documents, each containing randomly sampled test cases selected from the total samples $T_s(1, 450)$ of the test cases, were distributed to the labeling team. Each individual then constructed a taxonomy of categories and sub-categories of testing strategies, following the open coding procedure [66], by analyzing the test file contents in a bottom-up process as described below.

The participants were provided with the cloned of the studied ML software systems release version R_n together with a spreadsheet document containing the name of sampled test cases, the corresponding test file name, F_{id}, and the complete path of the file corresponding to the source code files in R_n. They then read through the test source codes of the target ML software systems and refer to the official documentation to be familiar with code implementation before assigning short sentences as initial labels, to indicate the test strategy. Specifically, our approach to derive the complete set of commonly used test strategies is divided into two steps:

(i) Examine the structure and content of the test code; the algorithm being tested, control statements, logical statements, loop and error-handling techniques, and other run-time options. For instance, the test `'test_coalesce_10_percent'` (line 27 to 34) in Figure 2.1 compares each element of the transformed target data Y using a looping statement and approximates them with the expected value $[10, 0, 30, 30, 30]$.

(ii) Identify and extract the assertions API used within the test cases; to help identify categories and sub-categories of testing techniques based on the extracted assertions as described in the following paragraph.

Test assertions are statements within the test functions through which desired program specifications are checked against actual program behavior. As such, assertions represent the core part of test functions that evaluate the program's internal state. Most unit testing frameworks provide multiple assertion functions to compare different types of actual-vs-expected variables [67]. Assessing which assertions are used the most can

help understand what is being tested. In this step of our analysis, we categorized the commonly used assertions to derive the tests strategies in ML software systems.

First, we identify the test functions that are used in the studied ML software systems to express assertions. For example the test functions `'test_dense()'`, `'test_sparse()'`, `'test_negative()'`, `'test_string()'`, and `'test_coalesce_10_percent()'` in Figure 2.1. We only considered the test functions within the randomly selected samples described in step (3) above. Then, we extracted assertions by finding the usages within the identified test functions. In particular, for every test functions, we manually extracted all the lines representing the assertion statement (e.g., For the example shown Figure 2.2, we would extract the assertions in lines 3, 4 for test function `SIN_TABLE`, lines 8,9,10,11 for test function `'Angle8'`, and lines 15, 17, 18 for test function `Angle16`). In Python, assertions are expressed in two ways: 1) assert keyword, which is then followed by a Boolean expression; and 2) using customized assertion APIs, e.g., internally defined by each Python project, the Python *unittest* built-in functions, and assertion APIs provided by NumPy, i.e., a commonly used library in Python that supports computation on arrays and matrices. The most popular unit testing frameworks for C++ are *Boost.Test* (Boost), and *Google C++* (gTest). Both have similar features; for example, in Boost, engineers can organize test cases into test suites that could be registered automatically or manually. Moreover, they both allow a broader number of checkers/assertions such as Exceptions: Throws/Not throws, Equal, Not Equal, Greater, Less, the Equality checking for collections & bits, explicit Fail/Success, Floating-point numbers comparison, including control of Closeness/Approximation of numbers. To extract the assertions statement, we followed a combined approach of both manual and automated technique that include first reading the official documentation of the target ML software systems and understanding the code structure to initiate regular expression queries such as assert*, EXPECT*, BOOST* or CHECK*/ ACHECK* while continuously examining the query results. The four participants confirmed that all the assertions are extracted and are correctly mapped to the respective test functions. Manual analysis allowed us to identified rarely used assertion APIs that could not be identified using an automatic process. For example, some projects define a custom assertion API instead of using the standard assertion API provided by the testing framework (e.g., `SLOPPY_CHECK_CLOSE` defined in DeepSpeech project instead of using the standard `BOOST_CHECK_CLOSE` provided by the Boost framework, due to reasons such as types matching problem). Our approach to extract the assertion APIs is similar to the following previous work [12]. Leveraging the extracted information, the team proceeded to forming the categories of the related

assertions statements separately before the general group discussion.

Using Figure 2.1 to illustrate the labeling steps described above. We identified testing strategies from such test by proceeding as follows. First, we followed the path to the code under test specified in Line 1, to understand the content of the code being tested. Next, we examine the structure and content of the test code; test data generation, control statement, logical statement, loop, error-handling technique, run-time options, and assertion API choices, following a top-down or bottom-up procedure. For instance, the test `'test_coalesce_10_percent'` (line 27 to 34) compares each element of the transformed target data Y using a looping statement and approximates them with the expected value $[10, 0, 30, 30, 30]$. The assertion API `'assert_array_almost_equal'`, provided by NumPy [68], at line 32 approximates the two values with the default of 7 decimal places [2], an example of *Oracle Approximation* of sub-category *Rounding Tolerance* testing strategy. Note that the approximation assertions API used in line 17 and line 18 of Figure 2.2 test case (i.e., `'EXPECT_NEAR'`) instead approximates the result using the absolute range (i.e., $1e-4$). We referred to this sub-category of *Oracle Approximation* as *Absolute Tolerance*.

Also, the test (i.e., `test_coalesce_10_percent`) at the same time analyses the transformed target Y elements expecting the test to pass within the range of value using the loop statement at line 30. We therefore referred to this kind of testing strategy as *Value Range Analysis*.

Next, the team proceeded to generate a hierarchical taxonomy of test strategies by grouping related labels into categories. The process for grouping is iterative, where they continuously go back and forth between groups of categories to refine the taxonomy. Also, a test function may belong to either one or multiple testing strategies. All conflicts were discussed and resolved by introducing a new person, a practitioner with extensive research experience in machine learning. In Table A.1 and Table A.2 in the appendix, we show the representation of the selected test strategies derived from this analysis. The results of this analysis answers our **RQ1** and are discussed in Section 4.3.1.

⑥ ***Label the ML Properties being tested*:**

ML properties are requirements that should be satisfied to ensure that a model behaves as expected. A violation of such property often results into poorly performing and unstable models. These ML properties are tested throughout the ML workflow to ensure that the generated models meet expected requirements. To categorize ML test properties, we followed

[2] `https://het.as.utexas.edu/HET/Software/Numpy/reference/generated/numpy.testing.assert_almost_equal.html`

the same labeling procedure discussed in step (5), only now focusing on analyzing the test properties and classifying the requirements being tested into functional and non-functional requirements. The labeling team used the test properties reviewed by Zhang et al. [20] (i.e., *Correctness, Model Relevance, Robustness, Security, Efficiency, Fairness, Interpretability, and Privacy*), during the preparation steps and when assigning labels. To derive the labels, the labeling team focused on understanding the code under test based on the following three dimensions:

1. Understanding the problem domain, by analyzing the test input data, the process, and the output, by continuously referring to the official documentation related to the code under test. 2. Examining the algorithm being tested and scrutinizing it for any imprecision. This helped understand how the testing data set are constructed. These two initial steps were important to understand the algorithms being tested. For example, they allowed understanding what kind of inputs the algorithms expect and what different outputs should be expected from the algorithms. 3. Examining the oracle comparison and the run-time options. This step helped to understand the specific point in the test cases at which the engineers verify the test results and how they permute the order of the input data.

Using one or a combination of the above three dimensions, the labeling team members were able to understand the ML properties being tested. Then, leveraging the ISO 25010 Software and Data Quality standard[3] and the ISO/IEC TR 29119-11:2020 Software and systems engineering — Software testing standard (Part 11: Guidelines on the testing of AI-based systems)[4], they assigned initial labels to each identified property.

For instance, in Figure 2.1, to understand what ML properties are being tested, in the test function `'test_dense'` (line 5 to 8), which verifies the correctness of the implemented function `'fit_transform'` by checking that the returned transformed data Y has all the components of input X successfully incremented by 3. The property being tested is *Correctness*. Test `'test_sparse()'`, in addition to functional correctness (line 14) also verifies the consistency of the transformed data (line 16 to 18) against the original data. It also verifies the validity of data format (line 16). We referred to these two ML properties being tested as *Consistency* and *Validity*. Other test scenario related to *Consistency* could be an test oracle executed inside a loop statement without changing the test input or the oracle value.

After this initial labeling step, they grouped the labels into categories to form the final taxonomy of ML test properties. During the grouping step, the team followed an iterative

[3]https://iso25000.com/index.php/en/iso-25000-standards/iso-25010
[4]https://www.iso.org/obp/ui/#iso:std:iso-iec:tr:29119:-11:ed-1:v1:en

process by going back and forth between categories while refining the taxonomy. The team discussed and resolved all the conflicts by introducing a new practitioner with extensive research experience in software engineering and machine learning. The results of this step answers our **RQ2** and are discussed in the Section 4.3.2.

⑦ ***Compare Testing strategies and ML properties across Projects*:**

This step aims to understand how the identified testing strategies, and ML properties, from Step ⑤, and Step ⑥ respectively, are being implemented across the different studied ML software systems. We hope to shed more light on how developers choose testing strategies and implement tests, while working on different ML software systems. Additionally, we aim to understand if there is any ML property which is examined in different ML software systems. To this end, first, we compared the composition of the test strategies and ML properties tested in the studied ML software systems as follows. We computed the percentage of tests cases corresponding to each testing strategies $\{t_1, t_2, t_3, ...t_n\}$ for a given ML software system as:

$$\mathbf{T}(Proportion\ of\ unique\ testing\ strategy) = \frac{t_i * 100}{t_N} \tag{4.1}$$

Where: t_i: Is the total number of test cases corresponding to a single test strategy in a given ML software system, and t_N is the total number of test cases in the given ML software system.

Similarly, we independently compute the proportion of tests aimed at verifying each of the ML test properties in the target ML software system, following the same steps: For each ML test properties $\{p_1, p_2, p_3, ..., p_n\}$ derived in step6, we identified and counted all unique test cases corresponding to each of them in the target ML software system and computed the percentage of test case aimed at verifying each of the ML test properties as:

$$\mathbf{P}(Proportion\ of\ test\,cases\,for\,ML\,test\ property) = \frac{p_i * 100}{p_N} \tag{4.2}$$

Where: p_i: Is the total number of test cases corresponding to a single test property in a given project, and p_N is the total number of test cases verifying the identified ML properties in a given ML software system.

The results of **T** and **M** were used to generate Figure 4.3 and Figure 4.4 respectively. We will discuss the results for this step in Section 4.3.3 answering our **RQ3**.

⑧ ***Compare Testing strategies used across ML Properties*:**

We examined the various testing strategies that are used to verify each of the ML properties (identified in Step (6)) in the ML software systems, as follows: For every tests cases $\{t_{p0}, t_{p1}, t_{p1}, ..., t_{pi},\}$ corresponding to both ML properties and testing strategies in a given ML software system, we computed their percentage proportion as:

$$\textbf{TP}\ (Proportion\ of\ test\ cases\ for\ ML\ Property\ and\ Testing\ Strategies) = \frac{t_{pi} * 100}{t_{pN}} \tag{4.3}$$

Where: t_{pi}: Is the total number of test cases corresponding to a single testing strategy and ML property in a given project, and t_{pN} is the total number of test cases in the given ML software system. The result for **TP** is shown in Table 4.4 and will discuss the results of this step in Section 4.3.3 answering **RQ4**.

We share our replication package in [69].

4.3 Results

In this section, we present the results of our analysis, answering the proposed research questions.

4.3.1 RQ1: *What are the testing strategies used across the ML workflow?*

This subsection report the results for testing strategies broken down into two parts: 1. the types of testing strategies and their break downs, and 2. mapping the testing strategies to the ML workflow.

Figure 4.2 presents the nine (9) high-level categories of test strategies found in the studied ML systems (at Step (5) of our analysis methodology). For each strategy, we computed and presented the average proportion of tests ($\overline{T}$) corresponding to each testing strategy over all the studied ML software systems as: $\overline{T}(Average\ percentage\ of\ testing\ strategy) = \frac{1}{N}(T_i)$.

Where N ($N = 9$) is the total number of studied ML projects, and T_i is the percentage of the each test strategy in a single ML software system. The values of $\overline{T}$ is shown in Figure 4.2. For example, the $\overline{T}$ values for *Value error*, *Null Reference Detection*, *State Transition*, *Value Range Analysis* and *Absolute Relative Tolerance* are 5.96%, 4.44%, 18.59, 17.20% and 7.14% respectively, as shown in Figure 4.2.

The example code for each of the testing strategy is highlighted in the column 'Code Example/

assertion API' of Table 4.2. In the following, we describe each category and sub-category of the obtained taxonomy in details, providing examples of test scenarios.

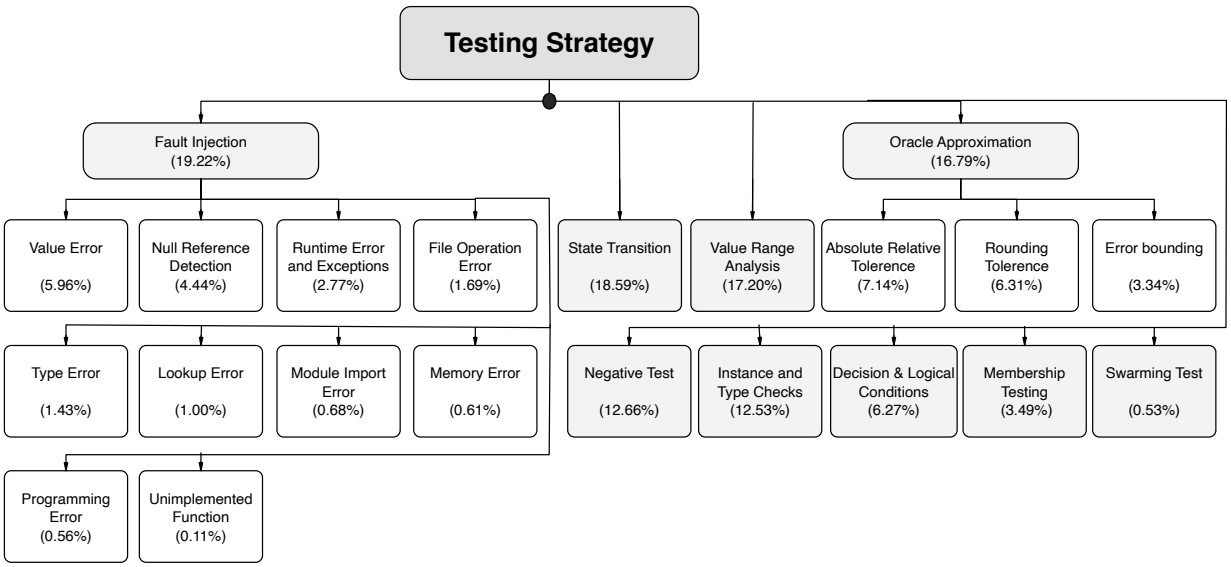


Figure 4.2 Overview of the common testing strategies found in the studied ML Systems. The nine (9) high-level categories of the testing strategies are highlighted in light gray color code, while the sub-categories are shown in white boxes.

• ***Fault Injection Testing:*** The test focuses on identifying an error in the ML systems source codes by making changes on certain source code statements to check if the test cases can find errors in the source code. This test evaluates how the system responds and recovers from error conditions, that can occur in a software application; maintaining a normal flow of execution. As shown in Figure 4.2, our analysis indicates that, on average, 19.22% of the test cases across the studied ML software systems belongs to this category. The goal of *Fault-Injection* testing is to ensure test case quality regarding the robustness of the application such that it should fail the modified source code. According to the Figure 4.2, we uncover ten (10) Fault-injection techniques that ML engineers followed in the studied ML systems, including Value Error, Null-pointer, Type Error, File Operation Error Lookup Error, Module Import Error, Memory Error, Programming Error, Unimplemented Function, Runtime and Exceptions, and Negative tests. In the following, we explain each of these techniques.

```
1 x=range(0,10,2)
2 for i in x:
3     assert i%2 == 0
```

Listing 4.1 original code

```
1 x=range(1,10,2)
2 for i in x:
3     assert i%2 == 0
```

Listing 4.2 modified line 1

(i) *Value Error*: In the test, the modification is made by introducing the inappropriate value to a given function. Example of value error is shown in Listing 4.1 and Listing 4.2. Considering the code under test in Listing 4.1 that expects the input list x to be all even numbers; the corresponding mutant was created in Listing 4.2 by modifying x starting value 0 replaced with 1 (i.e., $x = range(1, 10, 2)$). This modification (mutated source code) will make the assertion (line 3) to fail due to Value error 1 introduced in the input x. The *Value Error* Fault injection tests dominates (highest percentage) in the model training related test cases (38.67%) as shown in Table 4.2.

(ii) *Null Reference*: This error targets the operations on a specific object considered as null or calling for some method on the null object (i.e., a pointer or reference trying to access an invalid object). Usually, NULL is used (in most high-level language, e.g., Java, C/C++) to indicate a problem in the program that the ML engineers need to be aware of. In Python, the singleton None is used to represent the NULL object. As indicated in Table 4.2, the example of assertion API for testing for the None reference in Python include: `assertIsNone(indices)` (unittest), `assert statement is None` (pytest) or checking that a given object is not NULL reference: `assertIsNotNone(x)`. The highest percentage of *Fault-Injection* tests related to *Null Reference* is observed during the model training (31.14%), shown in Table 4.2.

(iii) *Type Error*: Unlike the *Value Error*, the target program under test is instead injected with an inappropriate type to invoke the *Type error*, indicating that the operation on the attempted object is not supported. The test of this category dominates with an average of 1.26% of the test cases in the studied ML software systems, according to our analysis in Figure 4.2. An example of this Fault-Injection includes passing a wrong argument to a function (like a transformation function) expecting a matrix array but receiving unsupported data of type dictionary or set, resulting in a *Type Error*. On the other hand, passing arguments with the wrong value (e.g., a one-dimensional array when expecting a two-dimensional array) results in a *Value Error*. According to Table 4.2, the highest percentage of *Type error* (Fault-Injection) is observed during *Data Cleaning* (50.83%) activity of the ML workflow.

(iv) *Memory Error*: This fault detection strategy targets memory usage related errors that can affect application stability and correctness, such as memory access errors (occur when a read/ write instruction references unallocated or deallocated memory), memory leak (the allocated memory is not released), and memory corruptions. Other examples of memory errors in C/C++ broadly classified as *Heap Memory Errors* and *Stack Memory Errors* [70] include missing allocation (freeing memory which has been freed already), mismatched allocation/ deallocation, cross stack access (a thread try to access stack memory of a different thread), uninitialized memory access. Python adopts the memory management architecture similar to the C language (i.e., `malloc()` function). In rare cases, Python may raise `OutofMemoryError` when the interpreter entirely runs out of memory, allowing it to be catch; enabling a recovery from it. The ML engineers, therefore, may catch out of memory errors within their script (as `except MemoryError` or `assertRaises(MemoryError)`); however, in some cases, MemoryError will result in an unrecoverable crash. Moreover, Python provides a cross-platform library called psutil [5] (process and system utilities) that allows developers to retrieve the information on running processes and system utilization (memory, disks, CPU, network). Solutions that depends on the psutil module, such as a memory-profiler (the example code is shown in Table 4.2) have been proposed to help monitor the memory usage of a process and perform a line-by-line analysis of memory consumption for python programs.

(v) *Programming Error:* This category represents a specific fault injection technique used to identify the incorrect operation of the program or code under test. Specifically, we used this category for errors such as 1) code logic errors that are due to mistakes in the program statements' sequence, such as using the wrong formula or function. The errors of this category may compile successfully but instead produces inaccurate results. 2) Syntax error: this error is a mistake in the programming language rules or a violation of the rules governing the language structure. Moreover, this category includes checking for assertion errors to indicate that something in the code under test should never have happened. This fault injection strategy allows devising test cases to ensure that the program's state, inputs, and outputs are correct.

(vi) *File Operation Error:* This category of fault injection is used to detect issues with files operations, e.g., when reading or creating files. Some of the issues tested during file operations include: invalid filename, invalid directory name specification, permission right for creating files, disk error, end of file error, or file schema error (i.e., problems

[5] `https://pypi.org/project/psutil/`

with the file structure, order, or it content, e.g., an invalid character preventing reading the file). This category of fault injection is mostly used (high percentage) in test cases related to Data collection and storage activity (41.83%) followed by utility and configuration files (21.97%), according to our results presented in Table 4.2.

(vii) *Lookup error:* This category concerns faults due to an invalid or incorrect index or key specification in a given sequence or a dictionary.

(viii) *Module Import error:* This category concerns the fault raised when the code under the test can not successfully import the specified module, typically due to a problem, such as an invalid or incorrect path.

(ix) *Negative testing:* This test category focuses on how the application under test can gracefully handle invalid inputs. Note that we presented *Negative test* as an independent category (although it belongs to the Fault-Injection testing strategy) due to the high percentage (12.66%) of tests of this category, as shown in Figure 4.2. *Negative test* checks whether the software application behaves as expected with a negative or unwanted data inputs. An example code of negative assertion test is shown in Table A.2a from the Appendix. The purpose of negative testing is to ensure that the application does not crash and continues normally when given invalid data inputs. In negative testing, exceptions are expected, which indicates that the application handles improper data input behavior correctly. Moreover, negative test cases can detect more defects in the software application compared to positive tests [71]. This testing strategy dominates in four major ML workflow activities, i.e., Model Training (22.28%), Feature Engineering (18.85%), Data collection (15.7%), Deployment (11.9%), according to our results presented in Table 4.2.

(x) *Runtime error and Exceptions*: This category presents the error detection and exception-handling during program execution that does not fall in any other *Fault Injection* testing categories presented. An example of this category is testing for the timeout error(such as socket timeout when a network client hanged while trying to make a request to a server). In `Tpot`, a prediction or a model training function is set to throw a Runtime error whenever some required parameters for pipeline optimization was not specified, before calling a training or prediction function.

Table 4.2 show the composition of the test for each of the testing strategies in the different ML workflow activities, calculated as follows: For each testing strategy, we identified and counted the number of unique test cases corresponding to each of the ML workflow activities

Table 4.2 **The list of common Testing Strategies and their percentage composition across the ML Workflow.** The highlighted cells in yellow indicate the highest value of the percentage test proportional T_c for each testing strategies. Due to space constrians, the following terms are abbreviated:
Data: Data Collection, *Clean*: Data Cleaning, *Label*: Data Labelling, *Feat*: Feature Engineering, *Train*: Model training related activities including model fit, prediction, hyper-parameter tuning, *Eval*: Model Evaluation and Post Processing, *Deploy*: Model Deployment activities including Model inspection, model update, pickling and pipeline export, *Moni*: Monitoring including Model Monitoring and Inspection, *Config*: Share configurations and Utility file or frameworks used across ML workflow activities

	Testing Strategy	Data	Clean	Label	Feat	Train	Eval	Deploy	Moni	Config	Code Example/ assertion api	ML specific
Fault-Injection	Value Error	6.4	24.2	5.89	7.8	38.67	7.45	5.25	3.02	1.31	`# raises on invalid range` `scaler=MinMaxScaler(feature_range=(2,1))` `assertRaises(ValueError,scaler.fit,data)`	✓
	Null Reference	20.63	5.65	5.2	8.74	31.14	10.01	1.56	5.65	11.42	`assert tpot_obj._memory is None` `assert_equal(tpot_obj._fitted_imputer, None)` `assertIsNone(fx.getNextRecordDict())` `EXPECT_EQ(nullptr, obstacle_ptr103);`	
	Type Error	7.21	50.83	0	7.15	8.65	0	21.3	4.86	0	`x = np.array([[['unknown']]])` `with pytest.raises(TypeError) as info:` `  x = input_node.transform(x) #unsupported input x` `EXPECT_FALSE(JsonUtil::GetString(json_obj, "int", value))// Value is not string`	✓
	Memory Error	57.83	0	8.46	0	22.29	0	11.42	0	0	`def test_memory:` `  device = "CUDA_VISIBLE_DEVICES=0 "` `  script = "setsid python -m memory_profiler benchmark_test.py "` `  for name in LIBSVM_DATA:` `  command = device + script + "--pipeline_name " + pipeline_name + " --name " + name + " --object " + test_object + " >" +log_name + " 2>&1 &"` `  os.system(command)`	✓
	Module Import Error	0	0	0	6.06	22.22	0	32.33	0	39.4		
	Lookup Error	15.56	4.1	32.3	18.07	12.46	0	0	0	17.5	`def testGetOutOfBounds:` `  assertRaises(IndexError, patternMachine.get, *args)`	
	File Operation Error	41.83	0	0	10.63	8.04	0	17.52	0	21.97	`BOOST_CHECK_THROW(f.ReadLine('\0'), util::EndOfFileException);`	
	Programming Error	0	0	7.21	39.38	19.23	21.89	0	12.3	0	`# Test that a float radius raises an assertion error` `with self.assertRaises(AssertionError):` `  encoder.encode((coordinate, float(radius)))`	
	UnImplemented Function	0	0	0	0	100	0	0	0	0		
	Negative Test	15.7	7.9	5.64	18.34	22.28	5.44	11.9	3.2	9.6	`assertTrue(a != b)` `EXPECT_FALSE(edges_[3] < edges_[4])`	
	Runtime error and Exception	3.16	6.23	1.86	0	18.85	15.45	23.86	11.57	20.39	`pool.submit(lambda a, v: a.f.remote(v), 0)` `with pytest.raises(TimeoutError):` `  pool.get_next_unordered(timeout=0.1)`	
Oracle Approximation [12]	Absolute Relative Tolerence	3.11	2.28	8.52	24.43	43.05	16.8	1.82	0	0	`assertAllClose(keras_class_out[i - 3], legacy_class_out[i], rtol=1e-4, atol=1e-4)`	✓
	Error Bounding	4.62	7.45	1.54	6.66	45.31	15.59	3.3	6.6	8.94	`EXPECT_LE(box.area(), min_area + 1e-5)`	
	Rounding Tolerance	0	4.37	17.17	16.93	43.7	12.03	0.64	5.16	0	`assertAlmostEqual(result['actualValues'][0], 34.7, places=5)`	✓
	Decision & Logical Condition	11.28	4.29	1.64	22.01	36.05	3.7	4.18	4.22	12.63	`# Assert that pick_two_individuals_ eligible_for_crossover` `# picks the correct pair of nodes` `# to perform crossover with` `def test_pick_two_individuals_ eligible_for_crossover:` `  assert ((str(pick1) == str(ind1) and str(pick2) == str(ind2)) or str(pick1) == str(ind2) and str(pick2) == str(ind1))` `assertTrue(all(map(lambda x: isinstance(x, int_type if is_int else float_type), results)))`	
[72]	State Transition	11.16	5.5	4.36	16.5	27.12	11.02	9.49	7.67	7.18	`EXPECT_FALSE(environment_features.has_ego_lane())` `environment_features.SetEgoLane("1-1", 1.0)` `EXPECT_TRUE(environment_features.has_ego_lane())`	
	Instance and Type Checks	5.02	16.04	12.57	22.03	27.94	8.94	1.59	2.38	3.49	`assertIsInstance(transform, np.ndarray)`	
[73, 74]	Value Range Analysis	9.84	13.31	0.54	22.08	30.41	11.5	2.13	4.16	6.03	`results = [c for c in clock_range]` `assertTrue(all(map(lambda x: expected_min <= x <= expected_max, results)))` `assertTrue(0 <= b1 and b1 < 1000)`	
	Membership testing	0	6.81	6.18	23.55	42.38	1.94	14.64	4.48	0	`def testNeighborsORadius(self):` `  neighbors = self.encoder._neighbors(np.array([100, 200, 300]), 0).tolist()` `  self.assertIn([100, 200, 300], neighbors)`	✓
[75]	Swarming Test	0	0	23.08	7.69	53.85	7.19	0.50	7.69	0	Used in random testing to improve test cases' diversity to enhance test coverage and fault detection. The example test is found in the link [6]	
	AVERAGE	**10.67**	**7.83**	**7.11**	**13.9**	**32.68**	**7.45**	**8.14**	**4.15**	**8.15**		

presented in Figure 2.3 for every ML software system in which the test exists at least once. Next, we computed the percentage of tests for each testing strategies $\{t_{c1}, t_{c2}, t_{c3}, ...t_{cn}\}$ in a single ML software system, as follows:

$$\mathbf{T_{ci}}(Composition\ of\ testing\ strategy\ in\ a\ ML\ workflow\ activity) = \frac{t_{ci} * 100}{t_{cN}} \tag{4.4}$$

Where t_{cN}: is the total number of unique test cases belonging to the ML workflow activity in a single ML software system. The results of $\mathbf{T_{ci}}$ are shown in Table 4.2 representing the proportion of testing strategies for each ML workflow activity. For example, the test *Absolute Relative Tolerance* dominates (i.e., highest percentage proportion) within the ML model training activity with $\mathbf{T_{ci}} = 43.05\%$ compared to its percentage proportion in other ML workflow activities, shown in Table 4.2.

• ***Oracle Approximation***

Here the output is allowed to accept a value within a specified ranges, unlike the equality checks commonly used in traditional software testing. As can be seen in Figure 4.2, 18.18% of the test cases in the studied ML systems, used at least one oracle approximation. We further grouped the oracle approximation into three main categories adopted from the related work [12].

(i) *Absolute and Relative Tolerance*: this test expresses the range of accepted oracles by using absolute and/or relative thresholds. Whether the assertions using these APIs pass or not depends on the result of $assert\ abs(res - oracle) < aVal + rVal * abs(oracle)$, where the variable res is the result from the code under test, $aVal$ is absolute tolerance, and $rVal$ is relative tolerance. Some assertions API approximate the code under test by considering only the absolute range/ tolerance instead of using the combination of absolute and relative tolerance, represented as: $assert\ abs(res - oracle) < aVal$. We, therefore, use the category name *Absolute and Relative Tolerance* to refer to both types of Oracle Approximation throughout this chapter.

(ii) *Rounding Tolerance*: This oracle approximation uses a significant decimal digit to round between the resulting value of code under test and its closeness with the expected described as: $assert\ abs(oracle - res) < 1.5 * 10 * *(-dp)$. The variables res is value of the code under test, $oracle$ is the expected oracle, and dp variable is used to specify the number of significant digits in res.

(iii) *Error Bounding*: In this category of Oracle Approximation, the first step is to calculate the difference between the two values (i.e., the code under test and expected

value) instead of directly comparing variables *res* and *oracle*, then asserts whether the difference (i.e., error) is smaller than a threshold as follows: $assert\ error < threshold$.

We highlight assertion APIs that are most commonly used for expressing Oracle Approximations in our studied ML software systems, in Table A.2b from the Appendix.

• ***State Transition*:** this testing strategy aims to analyze the behavior of an application through changes in input conditions that cause state changes or output changes in the code under test [72]. This category is the second most dominating with an average of 18.24% of the test cases in the studied ML systems according to our results from Figure 4.2. Also, *State Transition* testing strategy is observed in all the activities of ML workflow, according to Table 4.2.

• ***Value Range Analysis:*** This testing strategy similar to an interval analysis [73, 74] is a static analysis technique used to infer the set of values that a variable may take at a given point during the execution of a program. This category of testing strategy includes array and loop bounds checking, array-based data dependence testing, pointer analysis, feasible path analysis, and loop timing analysis, among others presented in the appendix' Table A.1. This category is the third most (17.20%) commonly used testing strategy by ML engineers according to our results from Figure 4.2. Table 4.2 further shows that the *Value Range analysis* testing strategy is used in all the activities of ML workflow, with the highest usage happening during model training activity.

• ***Instance and Type Checks:*** This test strategy aims to verify if the object argument under test is an instance or a subclass, or if the type of the argument matches with the specified class argument. An example of this category includes verifying if a transformation function returned a data of valid type. The tests of this testing strategy was observed in all activities of ML workflow in the order Model training (27.94%), Feature engineering (22.03%), Data cleaning (16.04%), and Data Labeling (12.57%) related test cases.

• ***Membership testing:*** This testing strategy checks if a specific value is contained within a collection of items or sequences (i.e., a list, a set, a dictionary, or a tuple). Usually, the time complexity when checking for a specific value depends on the type of the target collection (or data type) [76, 77]. In Appendix, Table A.2, we highlighted the example of the *Membership Testing*, including the assertion API in Python that uses the build-in Python-based membership testing operator called `'in'` [76]. We observed tests corresponding to *Membership Testing* during the Model training (42.38%), Feature Engineering (23.55%), Deployment (14.64%) and activities of ML workflow. Also, we label this testing strategy as ML specific because we didn't observe the test cases corresponding to this testing strategy in the non-ML

source code (i.e., not in Configuration and utility related test source code) of the studied ML software systems.

• ***Decision and Logical Condition*:** This testing strategy uses conditional expressions to test the possible outcomes of a program's decisions and ensure that the different points of entry or subroutine of the program are verified at least once. The test strategy helps to validate the branches in the program under test, making sure that no branch leads to an abnormal behavior of the software application.

• ***Swarming testing:*** This testing strategy is performed to improve test cases' diversity to enhance test coverage and fault detection during random testing [75]. Swarming testing does not follow the general practice of potentially including all features in every test case. Instead, a large "swarm" of randomly created configurations, each containing only specific features (omitting some features), is used, with the configurations accepting similar resources. We observed Swarm tests only in the `Nupic` project [7] and most of the test (high percentage) are done for model training related activity as shown in Table 4.2.

We derived a taxonomy of nine (9) major categories of test strategies that ML engineers use to find software bugs. These strategies are: *Fault-injection and Negative Tests, Oracle Approximation, State transition, Value Range Analysis, Instance and Type Checks, and Decision & Logical conditional check*s. Among these testing strategies only *Oracle Approximation* has been studied in the previous work [12]. The eight newly identified testing strategies not empirically studied by the previous works are: *Membership Testing*, *Fault Injection* (e.g., *Value Error*, *Type Error*, *Memory Error*), *Swarming testing*, *Value Range Analysis*, *State Transition*, *Instance and Type Checks*.

On average, the most dominating test strategy is *Fault Injection (19.22%)* followed by *State Transition (18.59%)*, *Value Range Analysis (17.20%)*, and *Oracle Approximation (16.79%)*.

Six (6) of the identified test strategies i.e., *Fault Injection* (*Value Error*, *Type Error*, and *Memory Error*), *Oracle Approximation* (*Absolute Relative Tolerance*, and *Rounding Tolerance*), and *Membership Testing* were observed only in test cases specific to ML source-code (ML specific test strategies).

Table 4.2 shows the composition of each testing strategy broken down in different activities of the ML workflow. The represented average percentage values of unique test cases in the respective ML workflow activities across the studied ML software systems computed as $\mathbf{T_{ci}}$

[7] `https://github.com/numenta/nupic/tree/master/tests/swarming`

in Equation 4.4. Overall we observe that there is a non-negligible proportion of testing strategies along the ML workflow. The majority of testing is happening during ML training (32.68%), followed by the Feature engineering (13.9%) activity. In Table 4.2, we further highlighted the table cells where the testing strategies demonstrate the highest percentage value of test cases in the ML workflow activity. For example, *Type Error* Fault-Injection test cases dominates during the Data cleaning activity (50.83%) followed by Deployment activity (21.3%) of the ML workflow. In contrast, *File Operation Error* test cases (such as file schema, read/ write operation test cases) are highly dominating (41.83%) in the Data collection and storage related code. The dominating test strategies during ML training and data preparation phases include Fault Injection (i.e., *Lookup error*, *Value Error*, *Null Reference detection*, *Unimplemented Function*, *Programming Error* and *Memory Error*), oracle approximation (*Error-bounding*, and *Absolute Relative Tolerance*), *Decision & Logical conditional test* and *sub component checks*.

ML engineers employ different testing strategies during the creation of ML models. A high proportion of testing activities happens during model training (32.68%) followed by feature engineering (13.9%) activities. Testing strategies such as Fault Injection (*Value Error, Null Reference, Type Error*), Oracle Approximation (*Absolute Relative Tolerance* and *Error Bounding*), *Decision and Logical Condition*, and *State Transition* are among the most common testing strategies used across at least 50% of different ML workflow activities.

4.3.2 RQ2: *What are the specific ML properties tested in a ML workflow?*

This subsection details the results of ML properties (derived in Step (6) of our methodology) broken down into: 1. identification of ML properties, and 2. mapping to ML properties to the ML workflow.

Table 4.3 presents the main ML properties identified during our data analysis, classified as both high-level and low-level functional and non-functional requirements. Table 4.3 also shows how the properties are tested throughout the different phases of the ML workflow. We also provide examples of test scenarios to highlight how the ML properties were tested. Note that, the ML properties presented are not strictly independent of each other, when given the features that facilitate their measurement. Yet, their violations result in different manifestations of the behaviors of an ML system.

- ***Consistency***: A specification is consistent to the extent that its provisions do not

Table 4.3 **The list of common ML Tests Properties and their percentage composition across the ML Workflow.** *Data*: Data Collection, *Clean*: Data Cleaning, *Label*: Data Labelling, *Feat*: Feature Engineering, *Train*: Model training related activities including model fit, prediction, hyper-parameter tuning, *Eval*: Model Evaluation and Post Processing, *Deploy*: Model Deployment activities including Model inspection, model update, pickling and pipeline export, *Moni*: Monitoring including Model Monitoring and Inspection, *Config*: Share configurations and Utility file or frameworks used across ML workflow activities, *C&R*: Category and Related works

	C&R	Test Properties	Data	Clean	Label	Feat	Train	Eval	Deploy	Moni	Config	Example of test scenarios and/or related dimension from studied systems separated by commas
1	Correctness [20, 78, 79]	Consistency	5.07	10.8	9.84	22.78	32.46	4.6	5.69	3.22	5.55	The representations of the same information across multiple datasets, multiple prediction, read/write, data migration, the consistent result across programming languages (compatibility).
2		Completeness	8.01	10.16	6.84	20.46	27.89	7.88	9.21	2.33	7.22	Functional test, data input/ output sets, range of tolerance, tests across data dimensions and data ordering
3		Correctness	6.42	9.42	9.2	20.93	34.53	7.25	5.25	2.36	4.64	Functional correctness (input and expected output), accuracy & precision, consistency & completeness
4		Data Validity	10.2	13.88	8.27	17.89	30.32	5.03	5.63	2.52	6.27	Data syntax conformance (i.e., format, range, type)
5		Data Migration Loss & Corruption	30.35	2.53	3.19	32.98	2.53	0	6.84	4.31	17.26	Data channel buffer, copying data, data segmentation and data consistency
6	[20, 80]	Robustness	0	0	9.75	22.84	53.5	6.61	4.4	0	2.9	Robustness in feature processing, classification algorithm, exporting invalid pipeline, read/write configuration dictionary, robustness given conditions valid input, invalid input and random input data, noise robustness in HTM spatial pooler (Nupic project)
7	[20, 44, 81]	Bias & Fairness	0	0	0	0	53.79	46.21	0	0	0	Model evaluation, mask pruning, probability thresholds (remove unlikely predictions likelihood thresholds), classification fairness
8		Scalability	16.15	8.07	1.9	6.64	53.06	0	3.18	9.3	1.69	Hyper-parameter search, model training efficiency, batch processing, Deep neural network morphism, parallel computing and distributed machine learning.
9	[82, 83]	Compatibility & Portability	12.97	0	8.64	42.25	12.24	0	7.32	3.35	13.23	On-device machine learning inference, Backwards compatibility test, environment variables and different computing environment, identical functionality of python and C++ implementation of same algorithm.
10	[20]	Efficiency	32.04	2.25	1.92	2.64	24.89	2.21	6.75	17.2	10.11	Performance, Training efficiency, data storage, Time behavior, API calls, computing resource usage, Neural network slimming & level pruning
11		Data Uniqueness	29.6	13.84	15.5	7.18	24.44	4.29	0	0	5.15	Only new records accepted, item measured against itself or its counterpart, scenario model prediction (e.g., estimate the lane change), sensor fusion
12		Timeliness	28.08	8.02	0	19.22	15.2	24.06	0	5.41	0	The time difference between data captured and the event being captured, Object detection, catch speed, extraction of point cloud feature
13		Data Relation	5.89	7.98	22.1	40.31	22.02	0	0	0	1.7	Attribute relation File Format (ARFF), Feature meta-learning and feature set selection, performing crossover of two parent nodes to generate new offspring in genetic algorithms, data encoding like scalar encoder and geospatial coordinate (e.g checking closeness) in Nupic project, data segmentation
14		Uncertainty	9.99	0	0	9.99	80.01	0	0	0	0	Evidence theory or Dempster–Shafer theory (DST), information filter
15		Concurrency & Parallelism	0	0	0	0	57.04	0	28.26	0	14.7	Producer/consumer queue (pcqueue), concurrent queue, thread pool, parallel training and parallel prediction
16		Anomaly	2.92	2.15	0	1.63	58.04	27.42	0	0	7.83	Training anomaly (i.e., unreliable training behaviors), prediction & classification region, spike frequency, distribution estimation & anomaly likelihoods
17		Feature Importance	2.7	0	89.52	2.12	4.26	1.4	0	0	0	Data balancing, features selection and classification error rate, neural network layers Transformation, Features importance estimation for Neural Network Pruning [84]
18		Data Distribution	10.27	7.92	11.7	22.56	34.1	5.85	1.08	4.14	2.39	Data analysis, transformed data output, prediction distribution, data completeness and anomaly, data distribution (skew, normal distribution) and data distribution overlap
19	Security [20]	Data Integration & Integrity	4.77	0	0	71.38	0	23.85	0	0	0	Data aggregation, data and feature fusion, efficient object detection, spatial dimension reduction
20		Security & Privacy	27.4	72.6	0	0	0	0	0	0	0	Data visitation, data migration/ transmission, secure matrix, data encapsulation
		AVERAGE	**12.09**	**8.24**	**9.44**	**18.43**	**30.65**	**8.1**	**5.23**	**2.63**	**5.18**	

conflict with each other or with governing specifications or objectives [78]. Therefore, some properties that influence the consistency are non-interferential, boundedness, reversibility, and liveness, i.e., (1) every operation that negatively affects the behaviour of other operations should be identified, (2) resources should have a finite capacity, (3) the operation should retain its initial state again, and (4) every operation is enabled in execution. As shown in Table 4.3, tests for consistency are spread across all major phases of the ML workflow; from data collection, storage, preparation to ML deployment as observed in our analysis. Further, we highlighted in Table 4.3 the areas where consistency was observed in the ML test cases, including the model prediction, read and write operations, and data migration.

In Listing 4.3 we illustrate three test case scenarios related to testing for consistency during data preparation and model training. They are extracted from the studied ML software systems (`Autokeras`, `Auto-sklearn` and `Nupic`). The first scenario (i.e., function `test_consistency_1()`) checks for consistency during prediction, where similar code was executed two times without any modification, and the results are expected to remain consistent. In the second test scenario (i.e., function `test_pca_default()`), the transformation function was executed ten (10) times, and the two results are compared against each other for consistency. Finally, in the third test scenario, read/ write tests the code for encoding data using adaptive scaler encoder in Nupic. A proto encoder was written into a temporary file and then read back into a new proto. The resulting proto was then used to encode the new data, which was then compared with the default encoder to verify consistency.

- ***Completeness***: The completeness of a specification concerns the extent to which all its parts are present and each part is fully developed [78]. We considered a test to be about completeness if at least one of the following criteria was satisfied by the test during our manual labelling: 1) For a program code with multiple functions, there is at least one test for each function of the program. 2) For a program accepting a different set of inputs, the tests are structured as: different classes of input/ output data were classified and tested independently, at least once, to evaluate the program's accuracy. 3) Testing for tolerance: the test cases were derived based on the range of permissible values of a variable to verify that the program understands all of these values and does not accept any other values. 4) Test cases checking that all intended data component are available or that there is no missing data or component.

- ***Correctness***: Correctness means the quality of being able to meet the expected need, satisfactorily [79]. Thereby, having a consistent and complete set of scenarios con-

```
# consistency in prediction
def test_consistency_1:
    for i in range(10):
        assert expected_score == result
# consistency in feature preprocessing
def test_pca_default(self):
   t = []
   for i in range(2):
     transform, original = _test_prep(PCA)
     t.append(transform)
   np.assert_allclose(t[-1],t[-2],rtol=1e-4)
# consistency in data preprocessing
def testReadWrite(self):
  originalValue = self._l.encode(1)
  p1 = AdaptiveScalarEncoderProto.new_message()
  self._l.write(p1)
  # write proto and read back into new proto
  with tempfile.TemporaryFile() as f:
    proto1.write(f)
    f.seek(0)
    p2 = AdaptiveScalarEncoderProto.read(f)
  encoder = AdaptiveScalarEncoder.read(p2)
  # ensure the encodings match on new value
  r1 = self._l.encode(7)
  r2 = encoder.encode(7)
  self.assertTrue(numpy.array_equal(r1, r2))
```

Listing 4.3 Tests examples for consistency extracted from *Autokeras* and *Nupic* ML project

tributes to requirements specification correctness. An example of functional correctness and completeness test (extracted from `apollo` software system) is shown in Listing 4.4 where the return results of the polynomial function were checked against its specification of the form $f(x) = 1+2*x^2+3*x^3+5*x^5$ (i.e., all the four different input values $x = \{0, 2, 3, 5\}$ to the function should output the correct values $\{1.0, 2.0, 3.0, 5.0\}$, or inputting invalid values $x = \{1, 4\}$ should return correct default value 0.0). Also, reversing the input order should return the correct results values of x.

```
TEST(BaseTest, polynomial_test) {
// f(x) = 1 + 2 * x^2 + 3 * x^3 + 5 * x^5
Polynomial poly;
poly[0] = 1.0;
poly[2] = 2.0;
poly[3] = 3.0;
poly[5] = 5.0;
EXPECT_NEAR(poly[0], 1.0, 1e-8);
//unknown input 1
EXPECT_NEAR(poly[1], 0.0, 1e-8);
EXPECT_NEAR(poly[2], 2.0, 1e-8);
EXPECT_NEAR(poly[3], 3.0, 1e-8);
//unknown input 4
EXPECT_NEAR(poly[4], 0.0, 1e-8);
EXPECT_NEAR(poly[5], 5.0, 1e-8);
EXPECT_NEAR(poly(0.0), 1.0, 1e-6);//reverse
EXPECT_NEAR(poly(1.0), 11.0, 1e-6);//reverse
}
```

Listing 4.4 Tests examples for correctness and completeness

- ***Data Validity***: This data quality dimension measures the errors in the data in terms of how the syntax of the data (i.e., type, format, range) conforms to its definition

or the business rule. The related dimension to data validity includes data accuracy, completeness, and consistency. Therefore, to assign this label we considered if a test is designed such that for a given dataset, metadata, or documentation; it checks for the allowed types (e.g., string, integer, floating-point, etc.), the format (e.g., shape or length, number of digits, etc.), or the range (e.g., minimum, maximum or contained within a set of allowable values). Examples of this category include verifying the conformance of data shape or type after the transformation process as follows:

```
assertEqual(transformation.shape[0], original.shape[0])
```

- ***Robustness***: Is a validation process consisting of subjecting the system under test to particular input streams to check if it satisfies some robustness specifications. The input streams may be either a valid input, and invalid input or random input streams. Therefore, a robust system should maintain its performance even when there is a change in input stream or introduction of noise [85]. As shown in Table 4.3, this property is tested throughout the major phases of the machine learning workflow. We further highlighted some of the test case scenarios where robustness testing was observed in the studied ML software systems, such as during feature preprocessing, training of classification algorithms (with robustness testing input datasets), export of the workflow after successfully training and validation, and checking for the stability of model performance on addition of noise to input data of the Hierarchical temporal memory (HTM) spatial pooler algorithm.

- ***Compatibility and Portability***: Compatibility is the ability of two or more components or systems to perform their respective functions within a shared environment. In contrast, Portability concerns the ease of moving components or systems between environments (software or hardware environments). As shown in Table 4.3, we observed the tests related to *Compatibility and Portability* mainly during the model training, deployment, monitoring activities, and in the configuration and utility related test files. The example of a compatibility test includes checking for identical functionality of the same algorithm implemented in different programming languages (e.g., python and C++) by running similar tests side by side, iteratively with random input data and monitoring the consistency of the outcome (in `Nupic` software system). Similarly, we observed Backward compatibility [83] tests performed by checking that users can effectively use either the legacy code implementation or the related updated version shown in Listing 4.5. Testing for the model's portability was observed during model exports; by ensuring that the exported model is consumed from different platforms in

the ML-based production system (e.g., freezing a model designed in Python and general framework into a portable format to be utilized inside mobile apps).

```
def test_object_id_backward_compatibility(ray_start_shared_local_modes)
    :
    # We've renamed Python's `ObjectID` to `ObjectRef`, and added a
    type
    # alias for backward compatibility.
    # This test is to make sure legacy code can still use `ObjectID`.
    # TODO(hchen): once we completely remove Python's `ObjectID`,
    # this test can be removed as well.

    # Check that these 2 types are the same.
    assert ray.ObjectID == ray.ObjectRef
    object_ref = ray.put(1)
    # Check that users can use either type in `isinstance`
    assert isinstance(object_ref, ray.ObjectID)
    assert isinstance(object_ref, ray.ObjectRef)
```

Listing 4.5 Tests examples for backwards compatibility

- ***Bias and Fairness*:** A bias can be defined as a systematic error introduced into sampling or testing of data by choosing or promoting one outcome over others. Bias in machine learning is one of the biggest challenges face by industries today. There is a wide range of different factors that can lead to bias in ML software system, broadly categorized as: selection bias arising from the recruitment of the study subjects or differing rates of study participation such as measurement error, information bias, and the subjects' cultural background, age, or socioeconomic situation [86] among others. An unfair algorithm is one which decisions are skewed toward a specific group of people; on the one hand, a model is considered fair if errors are shared similarly across protected groups.

  While the mitigation of model or data bias and fairness are done in various ways throughout the machine learning life-cycle such as by addressing the problem of over-fitting/ under-fitting and feature selection (e.g., removing the sensitive attributes like gender, sex, race from dataset to achieve fairness). We identified tests about bias or fairness in the studied ML software system during ML model training, and model evaluation activities of ML workflow as shown in Table 4.3. An example of such tests was observed in the code pruning filters and weights when compressing Convolution Neural Network (CNN) model [87] in the `nni` software system.

- ***Data Uniqueness*:** This ML property measures unnecessary duplication in or across the ML software system within a particular field, record, or data set [88], by discrete measures of repeatable data items within or comparison with the counterpart in different data set that complies with the exact business rules or information specifications.

An example of a Data uniqueness test case is shown in Listing 4.6. The test ensures that there is no duplicate entry into the object table by first checking if the object already exists before adding the object with a similar identity to the object table.

```
def testInvalidObjectTableAdd(self):
    # Check that Redis returns an error when RAY.OBJECT_TABLE_ADD
    # adds an object ID that is already present.
    self.redis.execute_command("RAY.OBJECT_TABLE_ADD", "object_id1", 1,
    "hash1", "manager_id1")
    response = self.redis.execute_command("RAY.OBJECT_TABLE_LOOKUP", "
    object_id1")
    self.assertEqual(set(response), {b"manager_id1"})
    with self.assertRaises(redis.ResponseError):
        self.redis.execute_command("RAY.OBJECT_TABLE_ADD","object_id1",
    1, "hash2", "manager_id2")
```

Listing 4.6 Tests examples for data uniqueness

- ***Data Timeliness***: This property measures the degree to which the information/data is up-to-date and made available within the acceptable timeline, time frame, or duration. The main dimensions for measuring the Data timeliness proposed in the literature are currency, volatility, and Timeliness [88–91]. Listing 4.7 illustrate the example of test scenario that checks for the outdated object detected using Rada sensor extracted from Apollo ML system.

```
TEST(ContiRadarIDExpansionSkipOutdatedObjectsTest,
 skip_outdated_objects_test) {
ContiRadarIDExpansion id_expansion;
ContiRadar raw_obstacles;
auto *sensor_header = raw_obstacles.mutable_header();
sensor_header->set_timestamp_sec(0.0);
sensor_header->set_radar_timestamp(0.0 * 1e9);
ContiRadarObs *radar_obs = raw_obstacles.add_contiobs();
radar_obs->set_meas_state(static_cast<int>(ContiMeasState::CONTI_NEW)
 );
auto *header = radar_obs->mutable_header();
header->set_timestamp_sec(0.0);
header->set_radar_timestamp(0.0 * 1e9);
id_expansion.SkipOutdatedObjects(&raw_obstacles);
EXPECT_EQ(raw_obstacles.contiobs_size(), 1);

sensor_header->set_timestamp_sec(0.7);
sensor_header->set_radar_timestamp(0.7 * 1e9);
ContiRadarObs *radar_obs2 = raw_obstacles.add_contiobs();
radar_obs2->set_obstacle_id(0);
radar_obs2->set_meas_state(static_cast<int>(ContiMeasState::CONTI_NEW
 ))
auto *header2 = radar_obs->mutable_header();
header2->set_timestamp_sec(0.7);
header2->set_radar_timestamp(0.7 * 1e9);
id_expansion.SkipOutdatedObjects(&raw_obstacles);
EXPECT_EQ(raw_obstacles.contiobs_size(), 1);
```

Listing 4.7 Tests examples for Timeliness

- ***Feature importance***: This ML property measures the weight or score of the input features during feature selection of a predictive model, indicating the relative importance of the individual features when making a prediction. This ML property help to reduces the number of input features while providing more insights about the data set

by indicating which features are most relevant or least relevant to the target or the model in general. Indeed, there are various techniques and models for measuring the feature importance, such as using model coefficients or permutation testing [92]. We identify the test related to feature importance, such as during structural pruning of neural network that uses the first-order Taylor expansions [93] to estimate the contribution of a neuron (filter) to the final loss, and iteratively removed those with smaller scores [84].

ML Engineers test at least (20) different ML properties in ML workflow. These ML properties include: Functional Correctness, Consistency, Data Distribution, Data relation, Efficiency, Data validity, Security, and Feature importance. Among these ML test properties, only six (i.e., *Correctness*, *Bias & Fairness*, *Compatibility*, *Efficiency* and Security) has been studied in the previous works. Also, we could not identify test cases for the ML property *Interpretability* and *Model Reliability* mentioned in the previous work [20].

Comparing the tested properties within the ML workflow, we found that some of the identified ML properties, i.e., *Uncertainty*, *Security & Privacy*, *Concurrency & Parallelism*, and *Model Bias & Fairness* are tested in less than half ($\leq 50\%$) of ML workflow activities. In contrast, the ML properties *Consistency*, *Completeness*, *Correctness*, *Data Validity*, *Efficiency*, and *Data Distribution* are tested in a large majority ($\geq 80\%$) of ML workflow activities.

4.3.3 RQ3: *Are testing strategies and ML properties used consistently across different projects?*

In the first two research questions, we highlighted the testing strategies, and the ML properties that ML engineers test throughout the ML workflow. This section will examine the composition (in percentage) of the testing strategies and ML properties across the studied ML software projects.

Figure 4.3 presents a visual comparison of test cases distributions for the identified testing strategies, across the studied ML software projects. Our analysis shows a non-uniform proportional use of testing strategies, i.e., there is a general high deviation between the most dominating test strategies and the least dominating test strategies, within the studied ML software projects. As shown in Figure 4.3, testing strategies such as *Absolute Relative Tolerance (Oracle Approximation), Error bounding (Oracle Approximation)*, *Instance and Type*

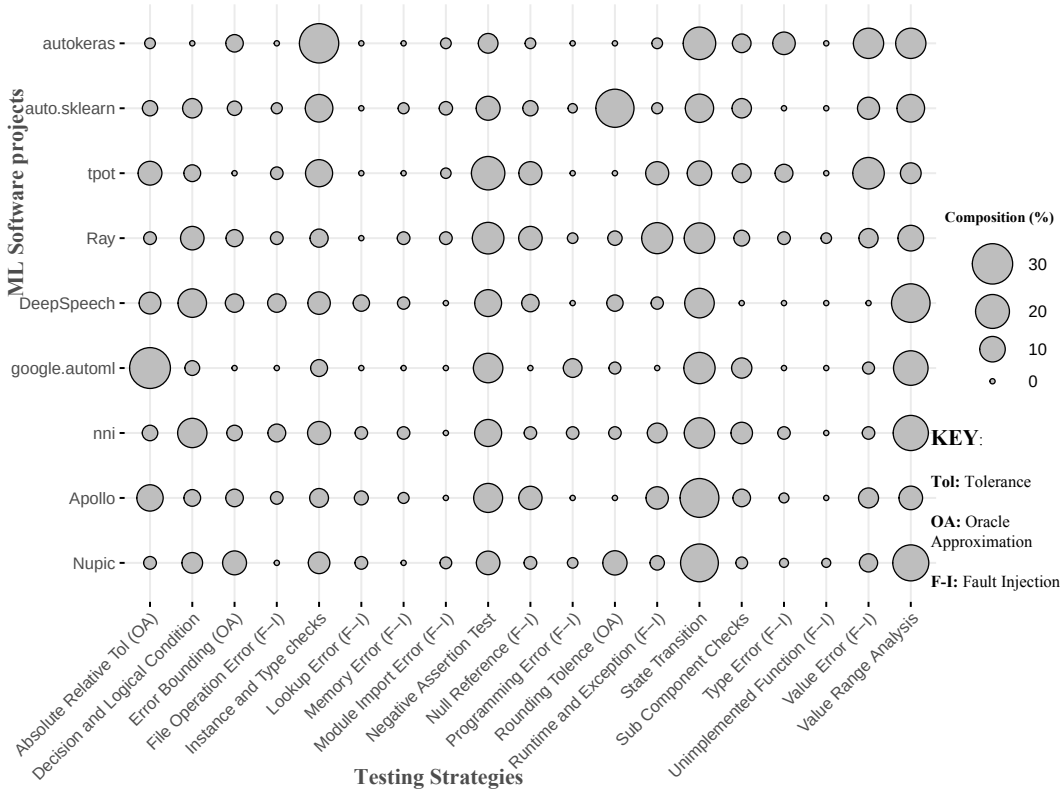


Figure 4.3 The composition of the testing strategies across the studied ML software projects. We used the three (3) Keys: ***Tol*** for Tolerance, ***OA*** for Oracle Approximation, and ***F-I*** for Fault Injection testing.

Checks, *Negative Test*, *State Transition*, *Value Range Analysis*, and *Value Error (Fault Injection)* are among the most commonly used testing strategies; these testing strategies are consistently used in at least 80% of the studied ML software projects. For example, the proportion of the *State Transition* tests ranges from 9% to 27% of all the test cases across the studied ML software project. The proportion of *Value Range Analysis* tests ranges from 5.8% to 27% and the proportion of *Absolute Relative Tolerance* tests ranges from 1% to 30% across the studied ML software projects. In contrast, testing strategies such as *Rounding Tolerance (Oracle approximation)* and Fault Injection (i.e., *File Operation error*, *Lookup Error,* Memory Error, *Module Import Error* and *Type Error*) are not observed in some of the studied ML software projects, suggesting that test strategies are currently being used inconsistently in the field.

There is a non-uniform use of different testing strategies within and across the studied ML software projects. The testing strategies consistently used in at least 80% of the studied ML software projects are: *Absolute Relative Tolerance (Oracle Approximation), Error bounding (Oracle Approximation)* , *Instance and Type Checks*, *Negative Test*, *State Transition*, *Value Range Analysis*, *Decision & Logical Condition*, *Sub Component Checks*, and *Value Error (Fault Injection)*.

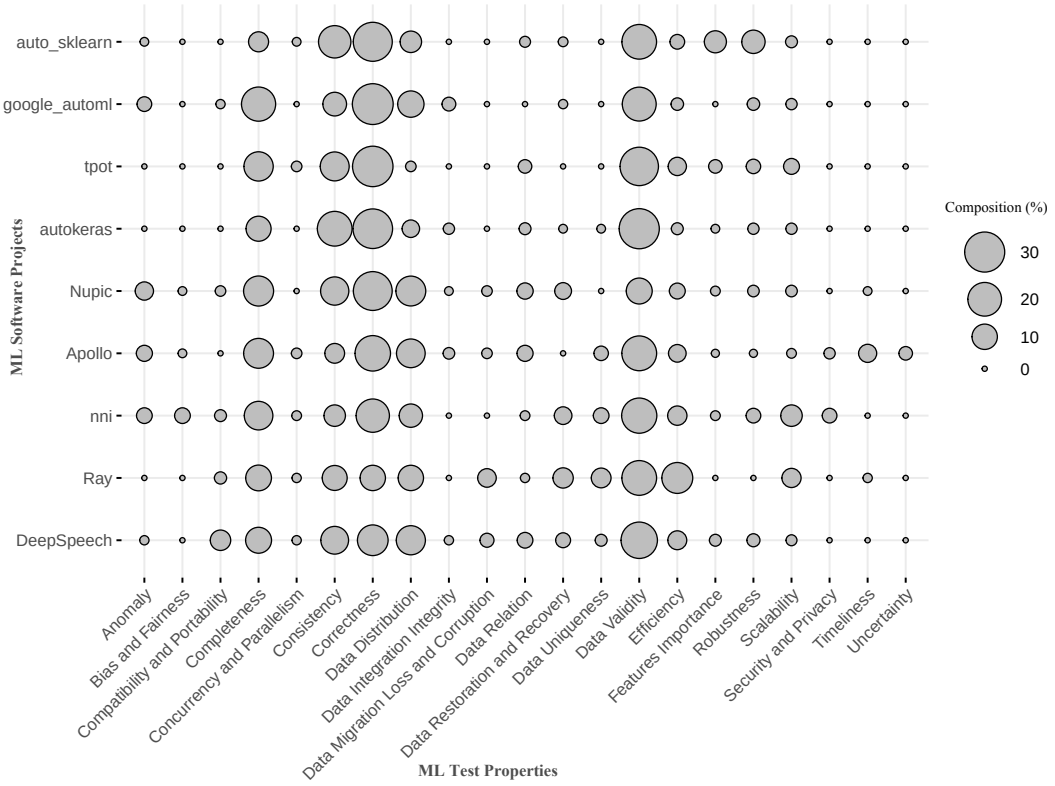


Figure 4.4 The composition of ML properties across the studied ML software projects

Figure 4.4 compares the ML properties being tested across the studied ML software projects. According to Figure 4.4, only about 20% to 30% of the ML properties such as *Correctness*, *Consistency*, *Completeness*, *Data Distribution*, *Data Validity*, and *Efficiency* are consistently being tested across at least 90% of the studied ML software projects. In contrast, we find that the ML properties *Bias and Fairness*, *Compatibility and Portability*, *Security and Privacy*, *Data Timeliness* and *Uncertainty* are not tested consistently in about 80% of the studied ML software projects; according to Figure 4.4. For instance, the test about *Security & Privacy* was only found in only two of the studied ML software projects `apollo` and `microsoft/nni` with only 0.9% and 2.2% of the test cases respectively. In the `apollo` project, we

observed test cases about *Security & Privacy* that are checking the data access management to protect the privacy and confidentially of data storage and ensuring that only the data allocated to a given channel is fetched by a user. In `microsoft/nni`, *Security & Privacy* test cases include a protocol test that verifies the correctness of the encryption function and ensures that the data is consistent between the two communication endpoints. Usually, data sets are encrypted to protect against potential interceptions of communications by a third party during a data transfer, which could violate the confidentiality and the privacy of the data.

> Only about 20% to 30% of the ML properties such as *Correctness*, *Consistency*, *Completeness*, *Data Distribution*, *Data Validity*, and *Efficiency* are consistently tested across at least 90% of the studied ML software projects. In contrast, the ML properties *Bias and Fairness*, *Compatibility and Portability*, *Security and Privacy*, *Data Timeliness* and *Uncertainty* are not tested consistently in about 80% of the studied ML software projects.

4.3.4 RQ4: *How are testing strategies used in verifying different ML Properties?*

In this part of our results, we discuss how the identified ML properties are tested by different ML testing strategies across the studied ML software systems.

Table 4.4 show the percentage number of tests corresponding to both testing strategies and ML properties (**TP**) in the studied ML software systems (computed in Equation 4.3 presented in Step8 of our methodology). For example, the 13.38% of tests in `Tpot` (P_2) software system corresponding to *Value Error* testing strategy are consequently used to verify the ML property *Data Validity*. In comparison, only 0.7% of the tests corresponding to *Value Error* testing strategy in `Tpot` software system are used to verify the ML property *Completeness*. Table 4.4 can also be used to identify different testing strategies that are used to verify each of the ML properties in the studied software systems. For example, at least 70% of the different testing strategies that we identified are used to verify the ML property *Consistency* according to the first row of Table 4.4. In contrast, the only testing strategies used to verify the ML property *Bias and Fairness* are *Decision and Logical Condition* and *Instance and Type Checks*. Similarly, only testing *Decision and Logical Condition* strategy is used to test the ML property *Compatibility and Portability*.

The highlighted columns (in light gray) in Table 4.4 correspond to the seven (7) testing strategies specific to ML software systems. Further, for each of the six (6) ML specific

Table 4.4 **The percentage number of testing strategies aiming to verify the ML test properties in the ML software project.** *The highlighted values (in yellow) are the highest percentage of testing strategies (only for ML specific, refer to column 'ML specific' in Table 4.2) across the ML properties for each projects. For example, in about (55%) projects (i.e., Apollo, tpot, Ray, Autokeras, and Nupic)* the testing strategy *Value Error* has the highest percentage of test verifying the ML property *Data Validity.*

Due to space constraints, we use the following abbreviations for various testing strategies: *Value*: Value Error, *Type*:Type Error, *Runtime*: Runtime Error and Exception, *Memory*: Memory Error, *Module*: Module Import Error, *Lookup*: Lookup Error, *AbsoluteRT*: Absolute Relative Tolerance, *EB*: Error Bounding, *RoundingT*: Rounding Tolerance, *Logical*: Decision and Logical Condition, *State*: State Transition, *ValueRange*: Value Range Analysis, *Member*: Membership Testing

P_1: Apollo, P_2: Tpot, P_3: Ray, P_4: Nni, P_5: Autokeras, P_6: Auto-sklearn, P_7: Automl, P_8: Nupic, P_9: DeepSpeech

			Fault Injection					Oracle Approximation						
	Cat	ML Properties	Value	Type	Runtime	Memory	Lookup	AbsoluteRT	EB	RoundingT	Logical	State	ValueRange	Member
1	Correctness	Consistency	P_5 : 0.55%, P_6 : 1.22%	P_3 : 1.66%	–	–	P_9 : 0.81%	P_1 : 1.85%, P_2 : 2.82%, P_7 : 7.0%, P_9 : 0.81%	P_6 : 0.37%	P_6 : 12.2%, P_8 : 2.2%	P_3 : 0.62%, P_6 : 1.83%, P_8 : 0.37%, P_9 : 0.81%	P_1 : 0.93%, P_2 : 0.7%, P_5 : 1.1%, P_6 : 3.05%, P_7 : 3.0%, P_8 : 2.2%, P_3 : 4.03%	P_5 : 1.1%, P_6 : 1.83%, P_7 : 5.0%, P_8 : 1.47%	P_1 : 0.46%, P_5 : 1.1%, P_6 : 0.61%
2		Completeness	P_1 : 1.39%, P_2 : 0.7%, P_4 : 1.11%, P_6 : 1.22%, P_8 : 0.73%	P_2 : 0.7%	P_1 : 0.46%, P_2 : 0.7%, P_4 : 1.11%	–	P_8 : 0.73%	P_1 : 5.56%, P_2 : 2.11%, P_3 : 1.25%, P_7 : 8.0%	P_1 : 0.46%	P_3 : 0.62%, P_4 : 1.11%, P_6 : 2.44%, P_8 : 1.1%	P_1 : 0.46%, P_8 : 0.73%	P_1 : 4.17%, P_2 : 1.41%, P_3 : 2.5%, P_5 : 0.55%, P_6 : 6.0%, P_8 : 1.47%, P_9 : 0.81%	P_1 : 0.46%, P_2 : 0.7%, P_3 : 0.62%, P_4 : 3.33%, P_5 : 2.76%, P_6 : 0.61%, P_7 : 5.0%, P_8 : 4.4%, P_9 : 2.42%	P_1 : 0.46%, P_5 : 0.61%, P_7 : 1.0%
3		Correctness	P_1 : 1.39%, P_2 : 4.93%, P_8 : 0.37%	P_2 : 2.11%	P_2 : 0.7%, P_6 : 1.0%, P_8 : 0.37%	–	–	P_1 : 2.31%, P_2 : 4.93%, P_3 : 0.62%, P_6 : 0.61%, P_7 : 13.0%, P_8 : 0.37%, P_9 : 3.23%	P_3 : 0.62%, P_6 : 0.61%	P_6 : 14.63% P_8 : 0.73%	P_1 : 0.93%, P_2 : 0.7%, P_6 : 1.22%, P_7 : 1.0%, P_9 : 0.81%	P_1 : 3.24%, P_2 : 1.41%, P_3 : 1.25%, P_5 : 2.21%, P_6 : 3.66%, P_8 : 2.56%, P_9 : 4.03%	P_1 : 1.39%, P_2 : 2.82%, P_3 : 2.76%, P_6 : 7.93%, P_7 : 3.0%, P_8 : 5.49%, P_9 : 2.42%	P_2 : 2.11%, P_3 : 1.1%, P_6 : 1.22%, P_7 : 1.0%, P_8 : 0.37%
4		Data Validity	P_1 : 3.7%, P_2 : 13.38%, P_3 : 1.88%, P_5 : 6.08%, P_6 : 1.22%, P_8 : 2.93%	P_2 : 1.41%, P_4 : 1.11%, P_5 : 1.66%, P_8 : 0.37%	P_1 : 0.93%, P_2 : 2.11%, P_3 : 2.0%, P_8 : 1.1%	P_1 : 0.93%	P_1 : 1.85%, P_8 : 1.1%	P_1 : 1.39%, P_6 : 0.61%, P_7 : 1.0%, P_9 : 0.81%	P_1 : 0.46%, P_3 : 0.62%	P_3 : 0.62%, P_6 : 3.05%, P_9 : 0.81%	P_1 : 0.93%, P_2 : 0.7%, P_3 : 0.62%, P_5 : 0.37%, P_9 : 0.81%	P_1 : 4.63%, P_2 : 2.82%, P_3 : 0.62%, P_4 : 1.11%, P_6 : 4.88%, P_7 : 6.0%, P_8 : 1.83%, P_9 : 2.42%	P_1 : 0.93%, P_2 : 0.7%, P_3 : 1.25%, P_4 : 1.11%, P_5 : 0.55%, P_6 : 2.44%, P_7 : 3.0%, P_8 : 1.47%, P_9 : 9.68%	P_1 : 1.39%, P_2 : 0.7%, P_4 : 1.11%, P_5 : 0.61%, P_7 : 3.0%
5		Data Migration Loss & Corruption	–	–	–	–	–	P_1 : 0.46%	–	–	P_3 : 0.62%	P_3 : 1.25%	P_8 : 0.37%	–
6		Robustness	P_6 : 0.61%	P_2 : 0.7%	P_2 : 0.7%	–	–	P_7 : 1.0%	–	P_6 : 2.44%	P_2 : 0.7%	P_2 : 0.7%	P_1 : 0.46%, P_6 : 1.0%	–
7		Bias and Fairness	–	–	–	–	–	–	–	–	P_4 : 1.11%	–	–	–
8		Scalability	–	–	P_5 : 0.62%, P_4 : 1.11%	–	–	P_7 : 1.0%	–	–	P_4 : 1.11%	P_2 : 0.7%, P_4 : 1.11%, P_5 : 0.55%, P_6 : 1.22%, P_8 : 0.37%, P_9 : 0.81%	P_3 : 0.62%	P_8 : 0.61%
9		Compatibility & Portability	–	–	–	–	–	–	–	–	P_3 : 0.62%	–	–	–
10		Efficiency	P_2 : 2.11%	–	P_2 : 0.7%, P_3 : 1.88%, P_4 : 1.11%, P_5 : 0.55%	P_5 : 0.62%, P_8 : 0.61%	P_1 : 0.46%	P_1 : 0.46%	P_3 : 1.25%, P_4 : 1.11%, P_6 : 0.61%, P_8 : 0.37%	–	P_1 : 0.46%, P_4 : 3.33%	P_3 : 0.62%, P_6 : 0.61%, P_8 : 0.37%	P_3 : 0.62%, P_4 : 1.11%	–
11		Data Uniqueness	P_3 : 0.62%	–	–	–	P_9 : 2.42%	–	–	–	–	P_1 : 0.46%, P_3 : 0.62%	P_3 : 0.62%	–
12		Timeliness	–	–	P_3 : 0.62%	–	–	–	P_1 : 0.46%	–	P_3 : 0.62%	P_1 : 0.93%	–	–
13		Data Relation	–	–	–	–	–	P_3 : 0.62%	–	P_8 : 0.73%, P_9 : 0.81%	P_9 : 0.81%,	P_3 : 0.62%, P_8 : 0.73%	P_8 : 0.73%	–
14		Concurrency & Parallelism	P_2 : 0.7%	–	P_3 : 0.62%	–	–	–	–	–	–	P_1 : 0.93%	P_9 : 0.81%,	–
15		Anomaly	P_1 : 0.46%, P_8 : 2.2%	–	P_3 : 0.62%	–	–	P_7 : 4.0%, P_8 : 0.37%	P_1 : 1.85%, P_4 : 1.11%, P_8 : 2.2%	–	–	P_1 : 6.02%, P_8 : 2.56%	P_1 : 0.93%, P_4 : 3.33%, P_8 : 1.47%	–
16		Feature Importance	–	–	–	–	–	P_6 : 1.83%	–	P_6 : 6.1%	–	P_1 : 0.46%, P_6 : 0.61%, P_8 : 0.37%, P_9 : 0.81%	P_4 : 1.11%	–
17		Data Distribution	P_1 : 0.46%, P_5 : 1.1%, P_3 : 0.62%	–	P_1 : 0.46%, P_8 : 0.37%	–	–	P_1 : 0.46%, P_7 : 8.0%	P_8 : 0.37%, P_9 : 0.81%	P_6 : 1.22%, P_8 : 0.73%, P_9 : 0.81%	P_1 : 0.46%, P_3 : 0.62%, P_9 : 0.81%	P_1 : 3.24%, P_7 : 1.0%, P_8 : 1.83%, P_9 : 1.61%	P_1 : 0.93%, P_3 : 0.62%, P_4 : 4.44%, P_5 : 1.1%, P_6 : 2.0%, P_8 : 1.47%, P_9 : 2.42%	P_4 : 1.11%

testing strategies, we highlighted (in yellow) the tests verifying specific ML properties with highest percentage values in the ML software systems. For example, in Table 4.4, we observe that the highest percentage of tests (in the five ML software systems : `Apollo`, `Tpot`, `Ray`, `autokeras`, and `Nupic`) correspond to the *Value Error* testing strategy verifying the ML property *Data Validity*. Also, the highest percentages of tests in the `Tpot` project, correspond to the testing strategies *Type Error*, *Absolute Relative Tolerance* and *Membership testing*, which are aimed at verifying the ML property *Correctness*, as shown in row 3 of Table 4.4.

In general, more than one testing strategies are used to verify a single ML property, as observed in at least 80% of the identified ML properties.
Among the testing strategies specific to ML software systems: 1. The most tested ML properties (by multiple testing strategies specific to ML software testing) are *Completeness*, *Correctness* and *Data Validity*. 2. The testing strategies commonly used to test for *Data Validity* are : Fault Injection (*Value Error*, and *Type Error*), and *Membership testing*. 3. The testing strategies *Absolute Relative Tolerance*, and *Rounding Tolerance* are used commonly to test ML properties *Completeness* and *Correctness*. 4. Finally, the ML specific testing strategies (commonly used to verify the ML properties *Efficiency*, and *Feature Importance*) are : *Memory Error*, and *Absolute Relative Tolerance* respectively.

4.4 Discussion and Implications

In **RQ1**, we investigated the ML testing strategies used in practice and derived a taxonomy of nine (9) major categories shown in Figure 4.2. Table 4.2 maps the testing strategies to the ML workflow activities where they were implemented. Overall, we observed that the majority of testing activities happens during model training and data oriented activities of ML workflow. This is not surprising since the majority of under-specification issues are related to poor data quality and inadequate model configurations and tuning. We have identified 20 ML properties (see Table 4.3) that were tested during the development process of the studied ML-based software systems. Among these 20 properties, only *Consistency, Completeness, Correctness, Data Validity*, *Efficiency* and *Data Distribution* are more frequently tested by ML engineers across the ML workflow. Critical ML properties such as *Uncertainty, Security & Privacy, Concurrency*, and *Model Bias and Fairness* were tested in less than half of the studied projects and the ML workflow activities. This result suggests that testing for these critical properties is not yet mainstream. This situation could be attributed to a lack of

automated testing tool and–or efficient techniques targeting these specific ML properties. There may also be a lack of awareness in the software development community about the importance of testing for these ML properties.

In Figure 4.3, and Figure 4.4, we compared the testing strategies, and ML properties across our studied ML software systems, with the aim to identify potential differences and–or inconsistencies in the application of testing techniques in the field. Our results reveal that only few testing strategies are consistently used in different ML software systems, i.e., *Absolute Relative Tolerance (Oracle Approximation), Error bounding (Oracle Approximation)* , *Instance and Type Checks*, *Negative Test*, *State Transition*, *Value Range Analysis*, *Decision & Logical Condition*, *Membership Testing*, and *Value Error (Fault Injection)*. For the ML properties, we found only about 20% to 30% of the ML properties such as *Correctness*, *Consistency*, *Completeness*, *Data Distribution*, *Data Validity*, and *Efficiency* are consistently tested across at least in 90% of the studied ML software systems. The ML properties *Bias and Fairness*, *Compatibility and Portability*, *Security and Privacy*, *Data Timeliness* and *Uncertainty* are not consistently tested in about 80% of the studied ML software systems.

In Table 4.4 we found that, at least two (2) testing strategies are used to verify a single ML property, as observed in at least 80% of the identified ML properties. Also, we found that the commonly (highest percentage) used testing strategies specific to ML software are aimed at verifying the ML properties: *Completeness*, *Correctness* and *Data Validity*. These strategies are: *Value Error*, *Type Error*, *Absolute Relative Tolerance*, *Rounding Tolerance* and *Membership Testing*. Table 4.4 also shows that the testing strategies commonly used to test for ML property *Data Validity* are: Fault Injection (*Value Error*, *Type Error*), and *Membership Testing*. The testing strategies *Absolute Relative Tolerance*, and *Rounding Tolerance* are used commonly to verify ML properties *Completeness* and *Correctness*. Similarly, the ML specific testing strategies commonly used to verify the ML properties *Efficiency*, and *Feature Importance* are: *Memory Error*, and *Absolute Relative Tolerance*.

As this study reveals the testing practices of ML engineers, many open questions remain:

1. The effectiveness of the identified test strategies is still unclear. Therefore, further studies need to examine the efficiency of these testing strategies at detecting bugs in ML software systems, especially the strategies that are most frequently used by ML engineering teams. Understanding the efficiency of these testing practices is important to help engineering teams select the most adequate testing strategies for their ML software systems.

2. It is also important to understand how the studied tests are maintained and evolved

through out the ML software development life-cycle. Our analysis highlighted the proportion of tests in the studied projects and the ML workflow activities where the tests are being implemented. However, it is not clear how the tests evolve during the ML software development life-cycle. We still don't know when ML engineers first introduced the studied tests and how their testing strategies may have changed overtime.

3. Our analysis showed that some critical ML properties such as *Security and Privacy*, *Data Uniqueness*, *Timeliness* or *Scalability* are not consistently tested across different ML software systems. This may be an indication that the topics in these areas have not been fully explored. Few recent works [10,11,94] have started exploring Security & Privacy in ML software systems, to help prevent attacks on autonomous systems (e.g., in `Apollo` software system), such as the LiDAR spoofing attacks [10,11]. *Scalability* is another big concern for any real-world ML software system. ML software system should scale during model training to learn on large datasets (usually gigabytes or terabytes) (for example, training a deep learning model from an online image corpus). On the other hand, an ML software system in a production environment is usually required to give the predictions in a few milliseconds from the crunch of new data. To satisfy these constraints, researchers and ML engineers are opting for building more and more larger ML models, using more computation power and adding more training data [95,96], in an effort to reach a scalable ML. In addition, other concepts such as distributed Machine learning [97] are being adapted by ML engineers (e.g., in ML software systems such as the `Ray`) to allow scaling to larger data input sizes, improving performance as well as increasing the model accuracy. The `Nupic` software system introduces an algorithm-based approach to solve performance and scalability problems in deep learning using neuroscience principles through sparsity [98].

From the analysis presented in this thesis, we make the following suggestions to researchers and ML engineers.

• ***To Researchers:*** Our study provides the first empirical study of ML testing practices, highlighting the testing strategies used in ML workflows and the specific ML properties tested by ML engineers. Future works can build on our finding of 20 common ML properties, to develop novel testing techniques and better tool support to help ML engineers test for these identified ML properties. We also invite more studies on the evaluation of the effectiveness of the identified ML testing strategies in future. The end goal is to standardize and automate testing strategies and properties for ML software systems, similar as to the traditional software systems.

• ***To ML engineers:*** We recommend that ML engineers use our presented taxonomy, to learn about the existing ML testing strategies, and implement them in their ML workflow, especially the most used testing strategies such as *Absolute Relative Tolerance (Oracle Approximation), Error bounding (Oracle Approximation)*, *Instance and Type Checks*, *Negative Test*, *State Transition*, *Value Range Analysis*, *Decision & Logical Condition*, *Membership Testing*, and *Value Error (Fault Injection)*. We also encourage ML maintenance teams to test for our identified ML properties in their ML software systems in order to ensure their systems' trustworthiness. They can use our results summarized in Table 4.4 to know what testing strategies they can use to test for the specific ML property. For instance, they can test for ML property *Data Validity* using at least two testing strategies like Fault Injection (*Value Error*, and *Type Error*) testing strategies, similarly test for the ML properties *Correctness* and *Completeness*, they can use the Oracle Approximation (*Absolute Relative Tolerance, and Rounding Tolerance*), *Value Range Analysis*, among others.

• ***To Tool Designers:*** They can develop better tooling support to help ML maintenance team effectively tests for the 20 common identified ML properties throughout ML development life-cycle.

4.5 Threats to Validity

In this section, we discuss the threats that could affect the validity of our results.

Internal Validity threats concern our selection of subject systems, and analysis method. We have selected ML-based software systems where tests are written in either Python, C/C++ programming languages. To extract the relevant test cases, test function, and the assertions, we followed an interactive process. First, analyzing the code base of all the test files while referring to the official documentation of the studied projects. This step was performed manually by researchers with extensive ML expertise. Yet, it is still possible that we may have missed some test cases and–or assertions that are rarely used in ML-based projects and hence harder to recognize. However, we believe that this threat should have a minor impact on our analysis and the results presented in this study.

External Validity threats concern the possibility to generalize our results. First, we studied only nine open-source ML-based software systems hosted in GitHub. Although these projects are selected from different domains such as AutoML frameworks, ML applications, and autonomous systems, it is still possible that we may have missed some important aspects of the testing practices of ML-based software systems. Also, the selected projects do not cover all domains of ML-based software systems. In the future, we plan to expand our study to

cover more ML-based software systems to further validate our results. Second, the focus of this study is ML-based software systems programmed in either Python and or C/C++. Therefore our findings may not generalize to ML-based software systems written in other programming languages. We also plan to expand our study in the future to include other programming languages, such as the Java or GO. Also, when analyzing the different testing strategies in ML systems, we limited our analysis to fewer cases, such as categorizing the assertions and error-handling techniques, while mainly relying on the documentation and the relevant literature. Generally, identifying all the different test strategies would require more manual analysis efforts and a complete understanding of the relevant implementation of the studied systems. We plan to expand the scope of this study to cover all these different aspects in the future. We also plan to conduct some qualitative studies involving the original developers of the studied systems.

Reliability validity threats concern the possibility of replicating this study. Every result obtained through empirical studies is threatened by potential bias from data sets. To mitigate these threats we chose to conduct manual analysis in this study, leverage up to five different participants with extensive ML expertise. We also provide in the thesis, all the necessary details required to replicate our study. The source code repositories of the studied projects are publicly available to obtain the same data. In addition, we provide a replication package in [69], containing the list of the studied ML software systems and their source codes for the selected project's versions, and the data containing the analysis for each of our four research questions (both in raw and processed form).

Construct to validity: We based our categorization on the international standards like the ISO 25010 Software and Data Quality standard[8] and the ISO/IEC TR 29119-11:2020 Software and systems engineering — Software testing standard (Part 11: Guidelines on the testing of AI-based systems)[9], while we categorize and track with the hope of standardize things, and how to count and measure say one test, or one strategy, or ML property and double check with the peers.

4.6 Summary

This Chapter presents the first fine-grained empirical study of ML testing practices. Specifically, we answer four research questions. **First,** we examine different ML testing strategies implemented in ML workflows deployed in the field. We derived a total of nine (9) main categories of ML testing strategies used in the ML workflow and 19 sub-categories, out of

[8]https://iso25000.com/index.php/en/iso-25000-standards/iso-25010
[9]https://www.iso.org/obp/ui/#iso:std:iso-iec:tr:29119:-11:ed-1:v1:en

which seven testing strategies are ML specific. We find than on average, most testing is associated with Model Training (32.68%), followed by Feature engineering activities of the ML workflow. **Second,** we studied the specific ML properties that are tested in a ML workflow and identified 20 commonly tested ML properties that ML engineers commonly test. We find that some of the identified ML properties, i.e., *Uncertainty*, *Security & Privacy*, *Concurrency*, and *Model Bias and Fairness* are tested in less than half ($\leq 50\%$) of ML workflow activities. In contrast, the ML properties *Consistency*, *Completeness*, *Correctness*, *Data Validity*, *Robustness* and *Data Distribution* are tested in a large majority ($\geq 80\%$) of ML workflow activities. **Third,** we compared the testing strategies, and ML properties across our studied ML software systems, to identify any potential differences and–or inconsistencies in the application of testing techniques in the field. We showed that, there is a non-uniform use of different testing strategies and ML properties within and across the studied ML software systems. Moreover, we found that, at least 80% of the studied ML projects consistently use the testing strategies: *Absolute Relative Tolerance (Oracle Approximation), Error bounding (Oracle Approximation)*, *Instance and Type Checks*, *Negative Test*, *State Transition*, *Value Range Analysis*, *Decision & Logical Condition*, *Membership Testing*, and *Value Error (Fault Injection)*. For the ML properties, we found only about 20% to 30% of the ML properties such as *Correctness*, *Consistency*, *Completeness*, *Data Distribution*, *Data Validity*, and *Efficiency* are consistently tested across at least in 90% of the studied ML software systems. **Finally,** by examining the testing strategies used across the ML properties, we observed that at least two (2) testing strategies are used to verify a single ML property, as observed in at least 80% of the identified ML properties.

We encourage researchers, to build on our finding of 20 common ML properties to develop novel testing techniques and better tool support to help ML engineers tests for these identified properties. We also invite more studies on the evaluation of the effectiveness of the identified ML testing strategies in future. We recommend that ML engineers use our presented taxonomy, to learn about the existing ML testing strategies, and implement them in their ML workflow, especially the most used testing strategies such as *Absolute Relative Tolerance (Oracle Approximation), Error bounding (Oracle Approximation)*, *Instance and Type Checks*, *Negative Test*, *State Transition*, *Value Range Analysis*, *Decision & Logical Condition*, *Membership Testing*, and *Value Error (Fault Injection)*. We also encourage ML maintenance teams to test for our identified ML properties in their projects in order to ensure their systems' trustworthiness.

CHAPTER 5 STUDYING THE TYPES OF TEST/ TESTING METHODS in AN ML WORKFLOW

5.1 Introduction

In Chapter 4, we discussed the different testing strategies and the ML properties that ML engineers test throughout the ML workflow activities. This part of our study expands on the analysis reported in Chapter 4, now empirically investigating how the ML engineers implement different types of tests or testing methods during the development and delivery process of ML software systems. Software Testing method is the classification of various testing activities into categories (black or white-box), each aiming to validate the System Under Test (SUT) for a defined set of test objective. Earlier in the background Chapter 2 in Figure 2.4, we introduced and discussed the various stages of test, such as Unit tests, Integration tests, or Manual tests during the delivery process of ML software system proposed in [2]. We now aim to understand how ML engineers operationalize the tests described in the test Pyramid from Figure 2.4 during the development of their ML software system. Specifically in this Chapter, we study the following three research questions:

RQ1 *What are the test types and methods used in an ML workflow?*

We manually examined different types of tests used in the studied ML software systems and classified them using the Test Pyramid of delivery process for Machine Learning based software system introduced by Sato et al. [2]. We identified a total of 11 different types of tests, out of which only six are included in the Test Pyramid of ML software system. The newly observed types of tests are : *Regression Test*, *Sanity test*, *Periodic Validation and Verification*, *Thread test*, and *Blob test*. This finding suggests that the current Test Pyramid of ML based software system proposed by Sato et al. is incomplete and should be updated. Also, the test type *Contract Test* described in this Test Pyramid were not implemented by any of our studied projects; suggesting potential gaps in the current ML testing practices in the field. For each type of test, we identified the testing strategies implemented by engineers to operationalize the test and created a mapping. This mapping can serve as a guideline for ML engineers seeking to implement different types of tests in their ML software delivery pipeline.

RQ2 *Are testing methods used consistently across projects?*

We examined whether the testing methods identified in **RQ1** is consistently tested across multiple ML software systems. We found that the composition of the testing

methods varies across the studied ML software systems (ranging from zero to 16%), except for *Unit Test* which takes the highest percentage of tests ranging between 68% to 91%.

RQ3 *How are ML properties being tested along different testing levels (testing methods)?*

We examined ML properties tested using different testing methods across the ML software systems and observed that the following ML properties are tested at different levels on the Test Pyramid of ML (from the unit level to the system level): *Consistency*, *Completeness*, *Correctness*, *Validity*, and *Data Distribution*.

In summary, we make the following main contributions:

- This study expand the findings presented in Chapter 4, by further taking the initial step to empirically study the testing methods adopted in the field.
- We summarized 11 different types of tests, out of which five are not included in the Test Pyramid of ML software systems. These newly identified test types are: *Regression Test*, *Sanity test*, *Periodic Validation and Verification*, *Thread test*, and *Blob test*.
- We provided a comparison of the testing methods across different ML software systems. Also, we provided comparison on how ML properties are tested at different testing levels (testing methods).
- We highlighted some challenges and proposed new research directions for the research community. ML practitioners can also leverage our findings to learn about different testing methods that can be implemented to improve the reliability of their next ML software system.

Chapter organization. Section 5.2 describes the three major steps of our methodology. Section 5.3 presents the results of our analysis, answering our research questions. Section 5.4 discusses the results of our study. Finally, in Section 5.5 we provide the summary this chapter.

5.2 Methodology

The main focus of this study is to understanding the different testing methods that ML engineers implement in the testing phases for their ML software system. In the following, we describe the three major steps we followed to study the testing methods used by ML engineers.

① ***Categorization of testing types/ methods in the Test Pyramid for ML*:**

This step used the selected ML software systems and the test files extracted in Chapter 4 of this thesis (refer to Step ① until Step ④ of methodology Section 4.2 in Chapter 4). We now focused on identifying the types of test the ML engineers implements throughout the development of ML software system.

Software Testing types/testing methods are testing activities (black-box or white-box) that aim to validate an System Under Test (SUT) for a defined set of test objectives. Following the open coding procedure, the authors classified the different testing activities used in the studied ML projects as follows: For every sampled test file contained in the spreadsheet, the authors first read through the corresponding source code while using the official documentation as reference, to familiarize themselves with what the test activities are about. Specifically, the authors focused on the following three main dimensions: 1) understanding the overall goal and achievement of the test file when executed (test objective), 2) the different tests strategies used, as described in Section 4.3.1 (of Chapter 4) and 3) the test deliverables. Note that, in most cases, the test types such as integration tests, unit tests, regression tests, swarming are already labeled (for some ML software systems e.g., `Nupic`, `autokeras`) by the developers via folder's and file's names (i.e., the meaning of folders/files names revealing the type of tests they contain; in these cases, the authors would directly use the name as the label. Also, in the cases where the test type is not specified in the test folder name, the labeling team examined the tests following the three dimensions described above, to ensure that the correct labels are assigned to the tests. For example, a test containing multiple print statements instead of assertion would most likely be assigned a label as manual/static test. All disagreements that occurred during the labeling process were discussed and resolved before the final labels were assigned. A practitioner with extensive research experience in software testing was involved in the discussions whenever there was a disagreement. A consensus was reach on all the labels. The results of this analysis answers our **RQ1** and are discussed in Section 5.3.1.

② ***Compare the testing methods across the ML software systems*:**

This step aims to understand how the identified types of tests/ testing methods from Step ① is being implemented across the different studied ML software systems. We hope to shed more light on how developers choose different types of test/ testing methods during the development phases of ML software systems.

We computed the proportion of tests corresponding to each of the identified types of test (testing methods) in the target ML software system, following the same steps (i.e., similar to Step ⑦ of Section 4.2 for Chapter 4): For each testing methods $\{m_1, m_2, m_3, ..., m_n\}$ derived

in Step (1), we identified and counted all unique test cases corresponding to each of them in the target ML software system and computed the percentage of test cases implemented in each of the testing methods as:

$$\mathbf{M}(Proportion\ of\ test\ cases\ in\ each\ test\ method) = \frac{m_i * 100}{m_N} \tag{5.1}$$

Where: m_i: Is the total number of test cases corresponding to a single test method in a given project, and m_N is the total number of test cases for all the identified testing methods in a given ML software system.

The results **M** was used to generate Figure 5.1. We will discuss the results of this step in Section 5.3.2 answering **RQ2**.

(3) ***Compare the testing methods across the ML properties*:**

This step aims aim to understand if there is any ML property which is examined using different types of tests.

We computed the testing methods used to verify a given ML properties in the ML software systems, as follows: For every tests cases $\{m_{p0}, m_{p1}, m_{p1}, ..., m_{pi}, \}$ corresponding to both ML properties and testing method in a given ML software system, we computed their percentage proportion as:

$$\mathbf{MP}\ (Proportion\ of\ test\ cases\ for\ ML\ Property\ and\ test\ method) = \frac{m_{pi} * 100}{m_{pN}} \tag{5.2}$$

Where: m_{pi}: Is the total number of test cases corresponding to a single test method and ML property in a given project, and m_{pN} is the total number of test cases in the given ML software system.

The results of this step (**MP** is shown in Table 5.2. We will discuss the results of this step in Section 5.3.3 answering **RQ3**.

5.3 Results

In this section, we present the results of our analysis, answering the three proposed research questions.

5.3.1 RQ1: *What are the software testing methods used in an ML workflow?*

We have presented the different testing strategies and the ML properties that ML engineers test in ML workflows. Earlier in Figure 2.4 we discussed the various stages of test during the delivery process of ML software system proposed in [2]. In this research question, we aim to understand how ML engineers operationalize the tests described in the test Pyramid from Figure 2.4 during the development of their ML software system. Notably, we examine the test types/ methods used across the ML workflow in the studied ML software systems.

Table 5.1 show the list of 11 test types/methods used by the ML engineers of the studied ML software systems; derived in Step 7 of our methodology. The presentation of the tests in Table 5.1 follows the layers of the Test Pyramid; with *Unit test* at the bottom (i.e., last row) and *Experimental test* at the top (i.e., first row). We highlighted rows corresponding to tests that we found in the studied ML software systems but which are not mentioned in the Test Pyramid of ML software system proposed by Sato et al. [2]. The newly observed types of tests are: *Regression testing*, *Sanity testing*, *Periodic Validation and Verification*, *Thread testing*, and *Blob testing*. In the following, we describe each of these newly observed tests in more details.

- *Regression Testing:* This type of tests are used to verify that a previous modification or code change has not adversely affected existing features. The test involves the partial or full selection and execution of already executed test cases to ensure that existing functionalities still work as expected after a change. We observed Regression testing in only three (3) of the studied ML projects (33%) (i.e., `DeepSpeech`, `ray` and `nupic` software systems) with an average of 1.43% of the total test cases in the target ML software systems. In the `nupic` project, a regression benchmark test [1] is executed every time a change is made to ensure that changes don't degrade prediction accuracy, through a set of standard experiments with thresholds for the prediction metrics. For example, limiting the permutations number may cause this test to fail if it results in lower accuracy.

- *Sanity Testing:* This test is part of regression testing. It aims to check the stability of new functionality or code changes in an existing build, to ensure for example that bugs are fixed and that no new issues are introduced as a result of these changes. The Sanity test focuses specifically on the build to ensure that the proposed functionality works close to or as expected. The build is rejected when the test fails, to save

[1] `https://github.com/numenta/nupic/blob/master/tests/regression/run_opf_benchmarks_test.py`

Table 5.1 **The list of 11 Test types/ test methods, and their corresponding test categories, arranged basing on the level of the test (from bottom to top) during the continuous delivery of ML software system.** *Data*: Data Collection, *Clean*: Data Cleaning, *Label*: Data Labelling, *Feat*: Feature Engineering, *Train*: Model training related activities including model fit, prediction, hyper-parameter tuning, *Eval*: Model Evaluation and Post Processing, *Deploy*: Model Deployment activities including Model inspection, model update, pickling and pipeline export, *Moni*: Monitoring including Model Monitoring and Inspection, *Config*: Share configurations and Utility file or frameworks used across ML workflow activities, *Cat*: The test category, *proj:* percentage number of projects with the test method.

	Cat	Test methods	proj	Data	Clean	Label	Feat	Train	Eval	Deploy	Moni	Config	Example of test scenarios and/or related dimension from studied systems separated by commas
11	Black-Box	Manual/ Exploratory test	89%	3.84	9.51	4.22	5.98	20.58	4.99	24.73	3.56	22.6	Explore the application for unexpected behaviors (using the human creativity to hunt possible hidden bugs). The use of print statement instead of assertions to verify the test outcome
10		Performance/ blob Test	67%	4.49	0	3.95	7.41	17.95	0	2.59	54.64	8.98	The system's stability and responsiveness under various workload, computation overhead during Binary Large Object (BLOB) operations (i.e., reshape, read/write, test header such as Canonical Axis Index or offset or Legacy shape, source pointer CPU/ GPU or mutable CPU/ GPU data) in `apollo`, Memory allocation, Time complexity.
9		Compatibility Test	47%	12.97	0	8.64	42.25	12.24	0	7.32	3.35	13.23	System's compatibility with the running environment, identical functionality of algorithm in different language, backward compatibility.
8	White- or Black-Box	End-to-End Test	11%	0	0	0	0	99.19	0	0.81	0	0	Testing for application dependencies to ensure that all integrated components can work together, end to end test for the compute function of HTM spatial pooler algorithm in `Nupic` without any mocking, Testing plumbing: create a random 'dataset', trains the network and then runs inference to ensures the correct classification.
7		Periodic Validation	22%	0	0	0	13.66	20.72	0	65.63	0	0	Frequently checking the program requirements, automated model retraining and continuous learning.
6		API Test	33%	0	0	0	0	62.76	0	0	33.55	3.69	Verifying the results or behaviour produced during the sequence of API calls, observed in the test cases for the studied ML software systems: `Nupic`, `Ray`, `autokeras`
5		Integration Test	33%	0	7.32	0	0	68.25	0	0	9.7	14.72	Testing the combinations of different units or modules to ensures they can work together as expected.
4		Thread testing	22%	0	0	0	0	0	0	74.16	0	25.84	Verifying the key functional capabilities of specific task, concurrent queue Testing, thread pool test, workers threads, performance analysis
3	White-Box	Regression Test	33%	19.25	0	0	51.87	28.88	0	0	0	0	Test that a previous modification or code change has not adversely affected existing feature. For example validating that that the model predictions don't change after modification (i.e., a checkpoint such that changes that affect prediction results will cause the test to fail).
2		Sanity Test	0.09 (11%)	0	0	0	0	100	0	0	0	0	Test the stability of new functionality or code changes in an existing build such as in Finite State Transducer (FST) [99] algorithms implementation in `Deepspeech` project
1		Unit Test	100%	10.43	10.86	8.41	17.95	30.69	5.95	6.22	2.05	7.43	Tests for individual components such as newly developed module, mostly contained within a test folder 'unit tests'.
		AVERAGE		**4.25**	**2.31**	**4.03**	**12.23**	**42.93**	**1.62**	**15.05**	**9.55**	**8.04**	

the cost and time for more rigorous testing. We observed Sanity test cases in only `DeepSpeech` software system. The example of the Regression and Sanity test cases in the `DeepSpeech` include the test for various Finite State Transducer (FST) algorithms[2] that allow for mapping between two sets of symbols and generates a set of relations.

- *Periodic Validation and Verification:* This test periodically or frequently checks the program under test to confirm that the requirement has been fulfilled effectively and that the program achieves its intended purpose given the objective evidence. This type of test happens during automated model retraining and continuous learning. The test ensures that models are retrained with the most recent available data based on a given frequency or other conditions.

- *API Testing:* API Testing aims to check the functionality, performance, reliability, and security of the programming interfaces [100]. In API Testing, the software is used to send calls to the API, the output is collected and the system's response is analyzed. We observed test cases related to API testing in three (3) of the studied ML software system (i.e., `Autokeras`, `Nupic` and `Ray`). The highest percentage of *API test* is observed in model training related test cases (62.76%).

- *Thread Testing:* This is an incremental software testing procedure (also called thread interaction test) performed during software system integration, to verify the key functional capabilities of a specific thread/task. Thread testing is usually conducted at the early stage of the Integration testing phase [101]. Thread testing operations are classified as either (i) single thread testing where only one application transaction is verified at a given time or (ii) multi-thread testing when multiple concurrently active transactions are verified at a time. We observed this type of test in two (2) of the studied ML software systems: `Apollo`, and `Deepspeech` software systems, accounting for an average of 0.53% of the total number of test cases. An example code of thread testing is shown in Listing 5.1. This example is extracted from the `Apollo` software system. In `DeepSpeech` a single threaded test case is used to verify the creation of thread safe producer-consumer queue.

- *Performance/ Blob Testing:* Performance Testing aims to check whether the software under test meets certain performance requirements such as stability, reliability, speed, response time, scalability, and resource consumption.

[2] `https://github.com/mozilla/DeepSpeech/blob/master/native_client/ctcdecode/third_party/openfst-1.6.7/src/test/algo_test.h`

```
1  TEST(TestThread, Test) {
2   MyThread my_thread;
3   EXPECT_EQ(my_thread.get_value(), 0);
4   my_thread.set_thread_name("my_thread");
5   EXPECT_EQ(my_thread.get_thread_name(), "my_thread");
6
7   my_thread.Start();
8   my_thread.Join();
9   EXPECT_EQ(my_thread.get_value(), 100);
10  EXPECT_FALSE(my_thread.IsAlive());
11  MyThread my_thread2;
12  my_thread2.Start();
13  EXPECT_TRUE(my_thread2.IsAlive());
14  my_thread2.Join();
15  EXPECT_EQ(my_thread2.tid(), 0);
16
17  MyThread my_thread3;
18  my_thread3.set_joinable(false);
19  my_thread3.Start();
20  my_thread3.set_joinable(false);
21  EXPECT_TRUE(my_thread3.IsAlive());
22  std::this_thread::sleep_for(std::chrono::milliseconds(100));
23  EXPECT_FALSE(my_thread3.IsAlive());
24 }
```

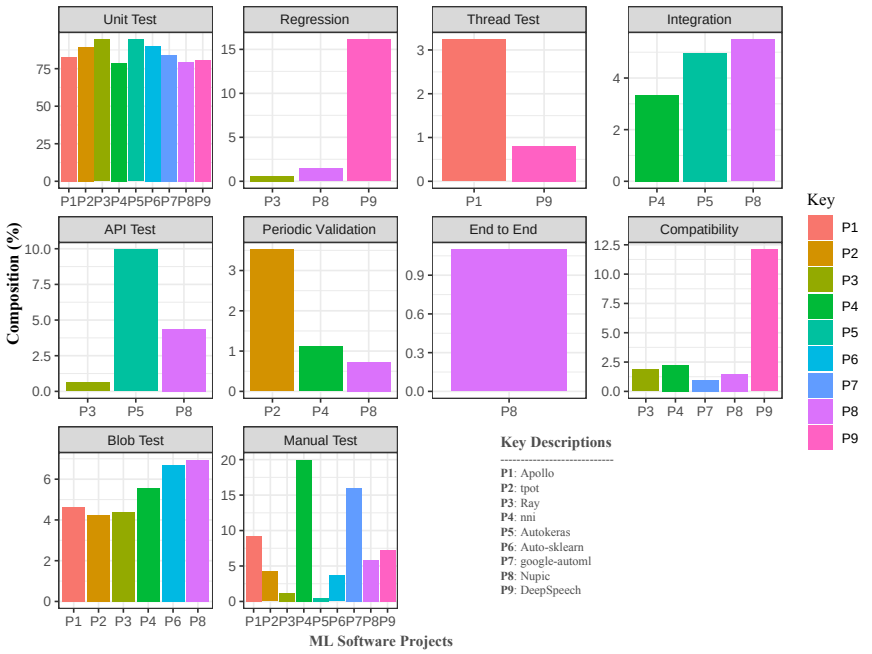
Listing 5.1 Thread Testing example code extracted from Apollo project

The ML engineers should consider including the above newly derived testing methods in the reference Test Pyramid for ML. For the regression test, they may utilise the different regression test selection techniques summarized by Graves, T.L. et al. [102] and make it as one of the required step [103, 104] during the development and testing phases of their ML software system.

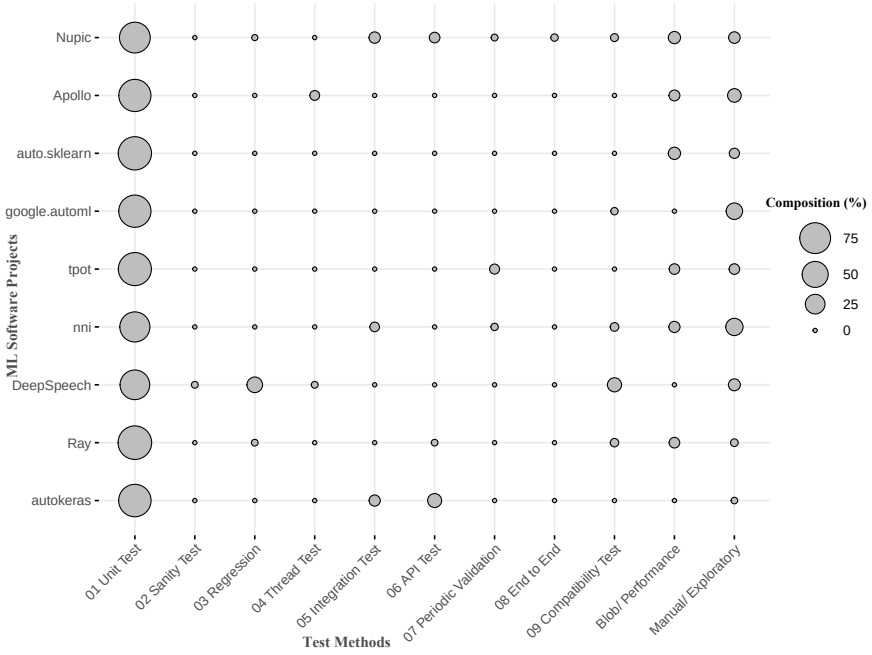
We identified a total of 11 different types of tests, out of which only six (6) are included in the Test Pyramid of the delivery process of ML software system proposed by Sato et al [2]. The newly observed types of tests are: *Regression testing*, *Sanity testing*, *Periodic Validation and Verification*, *Thread testing*, and *Performance/ Blob testing*. Also, the test type *Contract test* described in the proposed Test Pyramid were not implemented by any of our studied ML software systems; suggesting gaps in the current ML testing practices in the field.

5.3.2 RQ2: *Are testing methods used consistently across projects?*

Figure 5.1 (5.1a and 5.1b) compares the usage of identified testing methods across the studied ML software systems. Figure 5.1a summarizes the total number of tests constituting each of the testing methods. Overall, the level of adoption of the identified testing methods varies dramatically across the studied ML software systems (ranging from zero to 16%), except for *Unit Test* which takes the highest percentage of tests ranging between 68% to 91%. Moreover, only *Unit Test* and *Manual/ Exploratory* testing methods are observed in all the studied nine ML software systems. The testing methods such as *End to End* and *Thread*



(a) The bar plot comparison of the testing methods composition across the studied ML software systems.



(b) The tabular comparison of the testing methods composition across the studied ML software systems

Figure 5.1 Comparing the percentage composition of test types/ methods across the studied ML software systems.

testing are implemented in fewer ($\leq$ 22%) of the studied ML software systems. This can indicate that few ML software systems are thoroughly tested during software development. Also, the non-uniform composition of the testing methods along the proposed test levels can potentially indicate that ML engineers do not follow the proposed Test Pyramid for ML software system, but instead only focus on the testing methods that related to their project's goals, or that the proposed Test Pyramid for ML software is inaccurate and needs to be revised. For example, according to our analysis in Figure 5.1, the median test composition of the *Manual/ Exploratory test* and *Blob/ Performance tests* are 5.48% and 4.82%, respectively, representing the second and third highest composition after the unit test, compared to fewer percentage of tests methods such as *Ends to End testing* (1.03%). Similarly, at the software system level, *Nupic* software system which implements most (81%) of the identified testing methods has more *Unit Test* (74.65%) followed by the order *Blob/ Performance Test* (6.69%), *Manual/ Exploratory* (5.63%), *Integration Test* (5.28%), *API Test* (4.23%), *Compatibility Test* (1.41%).

Similarly, for the regression test, Among three studied ML software systems implements *Regression Test* (i.e., `Ray`, `Nupic`, and `DeepSpeech`), the `DeepSpeech` ML software systems has the highest percentage of of test cases corresponding to *Regression Test* (16.1%), according to Figure 5.1a. To highlight the example of the test case corresponding to *Regression test*, in `DeepSpeech` ML software system (a speech recognition system)) an automated test [3] is set to re-run 25 times for every update to check that no regression is introduced for the various algorithms implementing the weighted finite-state transducers (FST) [4], such as lexical/ alphabetical ordering algorithm. The overall testing methods implemented in `DeepSpeech` ML software system are in the order: *Unit Test (80%)*, *Regression Test (16.1)*, *Compatibility Test (12.1%)*, *Manual Test/ Exploratory Test (7.3%) Sanity test (0.8%)*, and *Thread Test (0..8%)*, as shown in Figure 5.1.

[3] `https://github.com/mozilla/DeepSpeech/blob/master/native_client/ctcdecode/third_party/openfst-1.6.7/src/test/algo_test.cc`

[4] `www.openfst.org`

The composition of the identified testing methods varies dramatically (ranging from zero to 16%) across the studied ML software systems, except for *Unit Test* which takes the highest percentage of tests ranging between 68% to 91%. Only the testing methods *Unit Test* and *Manual/ Exploratory* are implemented in each of the nine studied ML software systems. Our findings potentially indicate that, 1. the ML software systems are not thoroughly tested during the development phases, 2. ML engineers do not follow the proposed Test Pyramid for ML software system [2], or 3. the proposed Test Pyramid for ML software system [2] is inaccurate and should be revised.

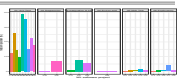
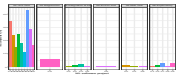
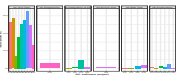
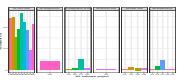
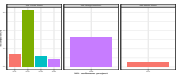
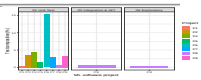
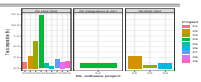
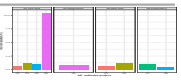
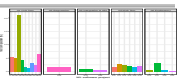
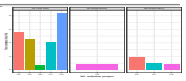
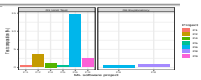
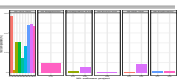
5.3.3 RQ3: *How are ML properties being tested along different testing levels?*

Table 5.2 presents the composition of test methods used to verify each of the identified ML properties. All the identified ML properties are verified by using *Unit Test* methods in at least one of the studied ML software systems, as shown in Table 5.2 (column 'Unit Test'). Also (according to column 'Unit Test' in Table 5.2), 55% of the studied projects has the highest percentage of unit test methods verifying the ML property *Correctness*, and the remaining 44% of the studied ML software systems uses the highest percentage of unit tests method for verifying the ML property *Data Validity*.

Overall, the ML properties *Consistency*, *Completeness*, *Correctness*, *Validity* and *Data Distribution* are continuously tested at different test levels (test methods), across different ML software systems. In contrast, the ML properties such as *Robustness*, *Scalability*, *Efficiency*, *Feature Importance* and *Security* are tested only at some testing level (on the Pyramid) for different ML software systems. Moreover, the test dominance (i.e., the highest composition of the test) targeting specific ML properties varies across ML software systems for different test methods. For example, according to Table 5.2, among the ML software systems that implements *Integration tests* (i.e., `Nupic`, `Autokeras`, and `nni`), the `Nupic` software system has the highest composition (2.2%) of *Integration tests* used to test for the ML property *Data Distribution*. The Integration test cases related to *Data Distributions* in the `Nupic` software system include the tests that check the occurrence of the predicted results returned by the *Sensor Data Record* (SDR) classifier [5] against the expected records of sensors data; each containing multiple class labels. In contrast, the `Autosklearn` software system shares the same highest percentage number (3.31%) of Integration tests used to verify the ML properties *Data Validity* and *Correctness*. Similarly, the `nni` software system uses the same percentage

[5] `https://github.com/numenta/nupic/blob/master/src/nupic/algorithms/sdr_classifier.py`

Table 5.2 **The percentage number of testing strategies aiming to verify the ML properties in the studied ML software project.** *The highlighted values (in yellow) indicate the composition of tests corresponding to test method with the highest value across ML properties, for each projects.* P_1: Apollo, P_2: Tpot, P_3: Ray, P_4: Nni, P_5: Autokeras, P_6: Auto-sklearn, P_7: Automl, P_8: Nupic, P_9: DeepSpeech

	Cat	ML Test Properties	Unit Test	Regression & Sanity	Integration Test	API Test	End-to-end Test	blob/ Performance	Manual/ Exploratory	Visual summary of the analysis
1	Correctness	Consistency	P_1 : 12.5%, P_2 : 26.06%, P_3 : 14.38%, P_4 : 10.0%, P_5 : 38.67%, P_6 : 35.37%, P_7 : 15.0%, P_8 : 22.34%, P_9 : 17.74%	P_8 : 0.37%, P_9 : 7.26%	P_4 : 1.11%, P_5 : 2.21%, P_8 : 1.1%	P_5 : 5.52%, P_8 : 4.4%	P_8 : 0.37%,	P_1 : 0.46%, P_2 : 0.7%, P_3 : 0.62%, P_6 : 1.83%, P_8 : 1.1%	P_1 : 0.46%, P_4 : 1.11%, P_6 : 0.61%, P_7 : 4.0%, P_8 : 0.73%,	
2		Completeness	P_1 : 34.26%, P_2 : 23.94%, P_3 : 15.0%, P_4 : 24.44%, P_5 : 17.68%,P_6 : 10.98%, P_7 : 43.0%, P_8 : 28.57%, P_9 : 16.13%	P_9 : 5.65%	P_3 : 0.62%, P_4 : 1.11%, P_5 : 1.1%, P_8 : 0.73%,	P_5 : 1.1%,	P_8 : 0.37%,	P_2 : 1.41%, P_3 : 0.62%,	P_7 : 3.0%, P_8 : 0.73%, P_8 : 0.37%, P_1 : 0.46%, P_4 : 1.11%, P_9 : 2.42%	
3		Correctness	P_1 : 53.24%, P_2 : 57.04%, P_3 : 14.38%, P_4 : 35.56%, P_5 : 50.28%, P_6 : 55.49%, P_7 : 65.0%, P_8 : 49.08%, P_9 : 26.61%	P_9 : 5.65%	P_4 : 1.11%, P_5 : 3.31%, P_8 : 1.83%	P_3 : 0.62%, P_5 : 6.08%	P_8 : 1.10%	P_1 : 0.46%, P_3 : 0.62%, P_6 : 2.44%, P_8 : 3.3%	P_1 : 0.93%, P_3 : 0.7%, P_4 : 2.22%, P_6 : 1.83%, P_7 : 5.0%, P_8 : 0.37%	
4		Validity	P_1 : 48.15%, P_2 : 49.3%, P_3 : 30.62%, P_4 : 37.78%, P_5 : 53.04%, P_6 : 44.51%, P_7 : 37.0%, P_8 : 18.32%, P_9 : 42.74%	P_9 : 8.06%	P_4 : 1.11%, P_5 : 3.31%, P_8 : 1.47%	P_3 : 0.62%, P_5 : 6.63%	P_8 : 0.37%,	P_1 : 0.93%, P_2 : 2.11%, P_3 : 1.88%, P_8 : 1.1%	P_1 : 0.93%, P_4 : 3.33%, P_7 : 9.0%, P_8 : 0.37%, P_9 : 0.81%	
5		Data Migration Loss & Corruption	P_1 : 1.39%, P_3 : 6.25%, P_8 : 1.1%, P_9 : 0.81%	P_9 : 3.23%				P_1 : 0.46%,		
6		Robustness	P_1 : 0.46%, P_2 : 3.52%, P_4 : 4.44%, P_5 : 1.66%, P_6 : 15.24%, P_7 : 3.0%, P_8 : 0.73%, P_9 : 3.23%		P_8 : 0.73%,				P_8 : 0.37%,	
7		Bias & Fairness	P_1 : 0.93%, P_4 : 5.56%, P_8 : 0.73%							
8		Scalability	P_1 : 1.39%, P_2 : 2.82%, P_3 : 6.25%, P_4 : 12.22%, P_5 : 1.1%, P_6 : 0.61%, P_7 : 2.0%, P_8 : 1.47%, P_9 : 1.61%		P_4 : 1.11%			P_2 : 2.82%, P_3 : 0.62%, P_6 : 1.22%		
9		Compatibility & Portability	P_3 : 0.62%, P_4 : 1.11%, P_8 : 1.1%, P_9 : 10.48%	P_9 : 0.81%,				P_3 : 0.62%, P_4 : 1.11%	P_7 : 1.0%, P_8 : 0.37%	
10		Efficiency	P_1 : 5.56%, P_2 : 4.93%, P_3 : 21.25%, P_4 : 4.44%, P_5 : 1.66%, P_6 : 1.22%, P_7 : 3.0%, P_8 : 2.56%, P_9 : 6.45%	P_9 : 1.61%	P_4 : 1.11%, P_8 : 0.73%			P_1 : 1.85%, P_2 : 2.82%, P_3 : 2.5%, P_4 : 2.22%, P_6 : 1.83%, P_8 : 4.4%	P_3 : 0.62%, P_4 : 3.33%, P_6 : 0.61%, P_8 : 0.37%	
11		Data Uniqueness	P_1 : 4.17%, P_3 : 8.12%, P_4 : 5.56%, P_5 : 0.55%, P_9 : 2.42%						P_1 : 0.46%,	
12		Timeliness	P_1 : 8.8%, P_8 : 0.73%					P_3 : 0.62%,	P_3 : 0.62%,	
13		Data Relation	P_1 : 6.94%, P_2 : 3.52%, P_3 : 0.62%, P_4 : 1.11%, P_5 : 2.21%, P_6 : 1.83%, P_8 : 4.76%, P_9 : 0.81%	P_9 : 4.84%,	P_8 : 0.73%,				P_1 : 0.46%,	
14		Concurrency & Parallelism	P_2 : 1.41%, P_3 : 0.62%, P_4 : 1.11%							
15		Anomaly	P_1 : 5.56%, P_4 : 4.44%, P_6 : 0.61%, P_7 : 4.0%, P_8 : 8.42%	P_9 : 0.81%					P_1 : 1.85%, P_7 : 1.0%, P_9 : 0.81%	
16		Feature Importance	P_1 : 0.46%, P_2 : 3.52%, P_4 : 1.11%, P_5 : 0.55%, P_6 : 14.63%, P_9 : 2.42%						P_6 : 0.61%, P_8 : 0.73%,	
17		Data Distribution	P_1 : 28.7%, P_2 : 1.41%, P_3 : 15.62%, P_4 : 15.56%, P_5 : 7.18%, P_6 : 13.41%, P_7 : 24.0%, P_8 : 24.54%, P_9 : 23.39%	P_9 : 4.84%,	P_8 : 2.2%	P_3 : 0.62%, P_8 : 0.37%	P_8 : 0.37%,	P_1 : 0.46%, P_8 : 2.2%	P_7 : 1.0%, P_9 : 0.81%,	

of Integration Tests (1.11%) to verify the ML properties *Consistency*, *Completeness*, *Correctness*, and *scalability*. The test method *API test* show the highest percentage of usage in testing the ML properties *Consistency* and *Data Validity* for the ML software systems `Autokeras` and `Nupic`.

> ML engineers implement multiple tests at different testing levels (test methods) to verify the ML properties *Consistency*, *Completeness*, *Correctness*, *Validity*, and *Data Distribution*. Moreover, about 75% of ML properties (such as *Robustness*, *Feature Importance*, *Efficiency*, *Anomaly*, and *Scalability*) are tested using less than half ($\leq$ 50%) of the identified test methods.

5.4 Discussion and Implications

In **RQ1**, we examined how ML engineers operationalize the tests described in the Test Pyramid of ML software system proposed by Sato et al. [2]. Our results presented in Table 5.1 show that *Contract test* described in the proposed Test Pyramid were not implemented in any of the studied projects. Moreover, we uncover five (5) new types of test (or testing methods) not included in the Test Pyramid of ML software systems proposed by Sato et al. [2], i.e., *Regression Testing*, *Sanity testing*, *Periodic Validation and Verification*, *Thread testing*, and *Blob test*. Moreover, we highlighted the percentage composition of each of the identified testing methods across the ML workflow in Table 5.1. ML engineering teams should consider including these testing methods (adapted from traditional software testing) into their reference Test Pyramid of ML software system.

For **RQ2**, we compared the tests methods across our studied ML software systems, with the aim to identify potential differences and–or inconsistencies in the application of testing techniques in the field. We showed that the most used testing method in the studied projects is *Unit Testing*, accounting between 68% to 91% of all the test cases across the studied ML software systems. In contrast, the proportion of adoption of the other testing methods vary dramatically from zero to 16% across the studied ML software systems. This finding may indicate that ML software systems may not be currently tested thoroughly during the different phases of their development life cycle.

In Table 5.2 we show that ML engineers implement multiple tests at different testing levels (testing methods) to verify the ML properties *Consistency*, *Completeness*, *Correctness*, *Validity*, and *Data Distribution*.

From the analysis presented in this paper, we make the following suggestions to the ML

engineering teams to consider including the testing techniques such as *Regression Testing*, *Sanity testing*, *Periodic Validation and Verification*, *Thread testing*, and *Performance/ Blob test* (adapted from traditional software testing) into their reference Test Pyramid of Continuous Delivery for ML software system (CD4ML). Also many of the selected ML software systems have no tests, so they need to learn from this and implement more tests.

5.5 Summary

This chapter is an extended part of the study presented earlier in Chapter 4, presenting the first fine-grained empirical study of ML testing practices. Specifically, in this Chapter, we answer three research questions. First, we examined the types of testing (test types/ testing methods) described in the Test Pyramid of ML proposed by Sato et al. [2]. We uncover 5 new types of testing not included in the Test Pyramid, i.e., *Regression Testing, Sanity testing, Periodic Validation and Verification, Thread testing*, and *Performance/Blob test*. In contrast, the *Contract test* described in the proposed Test Pyramid were not implemented in any of the studied projects. Then, we examined the composition of the testing methods and find that *Unit Testing* is the most used testing method in the studied projects, accounting for between 68% to 91% of all the test cases in a given ML software system. In contrast, the proportion of adoption of the other testing methods vary dramatically from zero to 16% across the studied ML software systems. We recommend that ML engineering teams consider including testing techniques such as *Swarming Test*, *Regression Testing*, *Sanity testing*, *Periodic Validation and Verification*, *Thread testing*, and *Performance/ Blob test* (adapted from traditional software testing) into their reference Test Pyramid of ML.

CHAPTER 6 STUDYING RELEASE ENGINEERING CHALLENGES USING STACKOVERFLOW

6.1 Introduction

Due to the market pressure, software industries are continuously required to deliver high-quality software products to the end-users faster. Unlike a few years ago, when software companies could work for months or even years on a release, many software companies now have only a limited time (e.g., a few weeks, days, or even hours) to ship their latest features to end users [6]. For instance, Google Chrome [105], Mozilla Firefox [106] and Facebook Mobile app have reduced their release "cycle time" to between two to six weeks, while Facebook web releases new features (1-2 times) a day [7].

Release engineering deals with all activities in between regular development and delivery of a software product to the end user. Through a series of phases such as code integration from the development branches, build & compilation, package, testing, and signing of the product for release, release engineers transform developers' source code into a product ready for users' consumption [14–16]. Releasing complex ML and traditional software systems with hundreds to thousands of users can be challenging and requires skills that are not always well mastered by ML engineers and release engineers. In fact, release engineers must implement continuous delivery and deployment practices and must be knowledgeable about specialised technologies and tools that support activities like continuous integration & source control management, testing, cloud provisioning, configuration management, application deployment, release orchestration [17]. It is therefore not surprising to see an increase of the prevalence of discussions about release engineering practices and tools on Q&A online developer forums, such as Stack Overflow. Stack Overflow is the most popular Q&A forums for software development. As of May 2020, it has recorded more than 19 million questions and 29 million answers. ML engineers or release engineers turn to Stack Overflow to seek answers from their peers about issues that they face when using different software development technologies. For example, in the Stack Overflow post ID 26440324, a developer asked about "Automated build on Docker Hub" as follows: *(1) I have created an account on Bitbucket which is attached to a repository (no team, no group, just a user on a prepository). (2) In Docker Hub, I tried to link to Bitbucket via button + Add Repository / "Automated Build". (3) I get logged in alright, but it says "No repos available". That is strange as I can see the repository when logged into Bitbucket with this specific user. I have created this Bitbucket*

user for the sole purpose of being able to see that repository.[1]. This question received an answer only after over half a year. Which may be a signal that the issue faced by this engineer is either rare and–or difficult to resolve or not interesting enough to Stack Overflow users. An analysis of release engineering Q&A discussions on Stack Overflow can provide insights about the prevalence, popularity, and difficulty of various release engineering topics and guide the research community towards developing better techniques and tools to support release engineers. Therefore, in this chapter, we present a large-scale empirical study of release engineering related posts on Stack Overflow and apply topic modeling [18] to understand the discussion topics of release engineers and identify the most important challenges that they face. Instead of studying release engineering for ML applications only, we focused on the release engineering in the general software domain, as releasing engineering of ML applications still lack sufficient data and lessons learnt from general software applications can provide insights for releasing ML applications. In particular, we answer the following five research questions.

RQ1 *What topics are discussed around Release Engineering?*

The goal of this research question is to identify the main issues experienced by release engineers when producing software. By using the Latent Dirichlet allocation (LDA) [18] technique, we group and label StackOverflow posts into 38 topics, from a total of $260,023$ release engineering questions and answers posted in a period of 11 years; i.e., from 2008 to 2019.

RQ2 *What topics are popular among the release engineers?*

The goal of this question is to identify the most prevalent release engineering topics on Stack Overflow. We analyze the popularity of release engineering topics using 3 well-known metrics, and find the most popular topics to be: *Merge Conflict, Branching & Remote Upstream, and Feature Expansion*. This result suggests that despite the many version control systems and source code management tools that exist, developers still struggle with merge conflicts and branching issues. Moreover, we found that the topic *Software Testing* has the highest percentage of questions asked in the CI/CD phase of release engineering process, can potentially indicate the challenge in the software testing practice.

RQ3 *What topics are more and less difficult to find answers?*

The goal of this question is to identify the topics that may be challenging for the release engineers. Using a set of 2 well-known metrics, we find that topics *MobileApp Debug*

[1] `https://stackoverflow.com/questions/26440324/docker-hub-automated-build-linked-to-bitbu`

& Deployment, Continuous Deployment and Docker are among the most difficult, suggesting that novel tools and techniques may be needed to support release engineering teams performing these activities.

RQ4 *How do topic popularity and difficulties correlate?*

Through this question we want to understand if popular release engineering topics are difficult to address.

Results show that the topic *Security* is both popular and difficult, topics *Merge Conflict, Branching & Remote Upstream, Feature Expansion* are among the most popular, while topics *MobileApp Debug & Deployment, Continuous Deployment, and Docker* are among the most difficult. Release engineers must prevent malwares from infecting their products to be released and patch vulnerabilities quickly before they can be exploited by malicious users. This result suggests that release engineers may be struggling with these critical activities. More research, training, and investments are required to better support security management activities.

RQ5 *What types of questions do release engineers ask in StackOverflow?*

This research question aims to deepened our understanding of questions asked by release engineers. We aim to understand for example why they choose StackOverflow over other sources of information, such as official documentations.

By grouping the questions asked by release engineers into $How?$, $Why?$ and $What?$ categories, we find that release engineers frequently ask about *how*? to do things; often seeking clarifications and explanations (i.e., *what*?), and less frequently questioning (i.e., *why*?) certain aspects of release engineering practices, techniques, and tools. The high percentage of questions in the category $How?$ suggests that release engineers need support to create working solutions. Our dataset is available at [19]

The remainder of this Chapter is organised as follows. In Section 6.2, we describe the steps followed in our analysis. In Section 6.3, we discuss the results of our analysis. Section 6.4, we discuss the implications of our findings for the software research community, practitioners and educators. Finally, we summarise the chapter in Section 6.6.

6.2 Methodology

This section summarizes the steps we took to analyze StackOverflow questions and answers, and answer our research questions *RQ1–RQ5*.

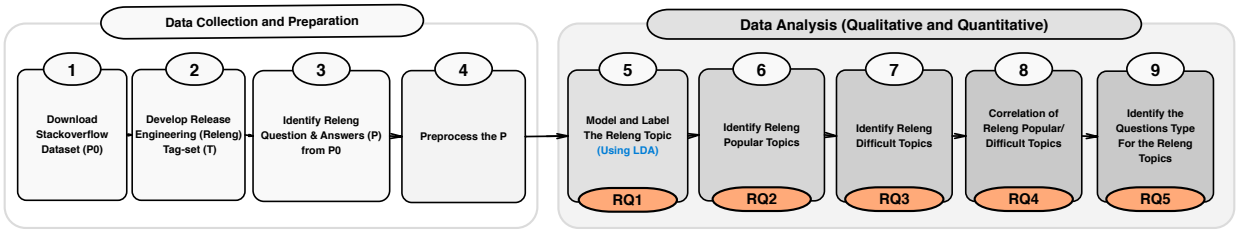


Figure 6.1 Overview of our data analysis process.

(1) *Download StackOverflow Dataset (P_0):* In this first step, we used SOTorrent [107] to download the set of StackOverflow posts P_0, which contains both questions and answers and their metadata. The dataset P_0's metadata includes title, body, tags, creation date, views, identifier, question or answer type, title and favorite counts, score, and the identifier of the accepted answer of a post if the post is a question with accepted answers. A question can have at least one tag and a maximum of five tags. An answer to a question is an accepted answer if the developer who posted the question marked it as accepted.

The dataset P_0 is based on the official StackOverflow data dump released on 2019-09-04[2]. The dataset P_0 has a total 45,919,817 questions and answers for all the post made from September 2008 to September 2019 by $4,479,628$ developer participants of StackOverflow. Among these posts, $18,154,493$ and $27,765,324$ are questions and answers, respectively. In total, $9,531,288$ (34%) of these answers are marked as accepted answers.

(2) *Develop Release Engineering (Releng) tag-set (T):* To extract releng posts from the huge sets of posts using tags, we need a way of identifying the set of tags T used by developers when posting releng questions in StackOverflow. This tag set T is then used to identify and extract only the releng related Questions & Answers from our initial dataset P_0. To develop T, we defined the initial releng tag set T_i[3] based on modern release engineering process described in the previous work [15, 16] as follows:

(1) T_i include tags for the major phases of releng process, i.e., tags “Integration", “Continuous Integration", “Build System", “Infrastructure-as-Code", “Deployment" and “Release" [16]. Note that some tag naming was changed to match the naming in StackOverflow (e.g., Continous Integration changed to continuous-integration) after manually searching the tops posts tagged with the suggested tag name. **(2)** Identify and extract from the initial dataset P_0 all questions Q whose tags match those in T_i. **(3)** We extract the tags of questions in Q to construct a larger candidate tags set T_{i2}. **(4)** Lastly, we select only the tags t relevant

[2] https://archive.org/details/stackexchange
[3] https://drive.google.com/open?id=1upZfJ19d8cWGRJ-1Jbmbdg8ROSCsRMIu

to releng, using two sets of heuristics μ and ν, measuring respectively, the significance and relevance of a tag for a topic. These heuristics were also used in previous works [50,108]. We removed tags that did not meet these criteria as follows.

$$\langle significance \rangle \ \mu = \frac{No.\,of\ questions\ with\ tag\ t\ in\ Q}{No.\,of\ questions\ with\ tag\ t\ in\ P_0} \tag{6.1}$$

$$\langle relevance \rangle \ \nu = \frac{No.\,of\ questions\ with\ tag\ t\ in\ Q}{No.\,of\ questions\ in\ Q} \tag{6.2}$$

We set thresholds limits and for every selected tag t, we considered the tag to be significantly relevant to releng if its μ and ν values are greater than or equal to our thresholds. To identify optimal threshold values for μ and ν, we iteratively experimented with values from intervals $0.05-0.35$ and $0.005-0.035$, respectively. During this experimentation, the authors assessed the quality of the extracted posts and discussed among them until reaching a consensus on the best candidate set. Finally, the optimal threshold values derived for μ and ν was used to select our final tag-sets and extract the studied posts. To illustrate further, we constructed from the initial tag-set T_{i2}: $\{T_{int}\}$, $\{T_{ci}\}$, $\{T_{bs}\}$, $\{T_{IaC}\}$, $\{T_{dep}\}$, $\{T_{rel}\}$ presenting all the six major phases of releng of integration, Continuous Integration, Build System, Infrastructure-as-Code, Deployment and Release, respectively. • T_{int} for example includes the major tags related to branching and merging of code changes between the developer branches and the main project branch, within a distributed version system.

We found that the optimal threshold values of $\mu = 0.3$ and $\nu = 0.010$ allow finding significantly relevant tags for tagset T_{int}, T_{ci} and T_{IaC}, while the threshold values of $\mu = 0.2$ and $\nu = 0.005$ allow finding significantly relevant tagsets for T_{bs}, T_{dep} and T_{rel}. The threshold values pairs of $\mu = 0.2$ and $\nu = 0.005$ are consistent with previous works [50, 108, 109]. Finally, we have a complete tagset T with a total of 64 tags, a union of selected T_{i2}, i.e, $T = T_{int} \cup T_{ci} \cup T_{bs} \cup T_{IaC} \cup T_{dep} \cup T_{rel}$. T is shown in Table 6.1 in light gray background. Note that some generic tags like *continuous-integration, build, deployment* also exist in our final tag sets T. Those are important to identify the possible releng posts that could have been left out, allowing us to cover a wide range. Also, some selected tags t may have the significant and relevant values that are within our threshold limit and yet was manually removed. The manual analysis was particularly done to discover tags that are much broader than just release engineering. Tags which were manually removed include among others *'environment-variables', 'transactions', 'merge', 'release'*[4]. Additionally, the use of tag set T does not prevent releng post with other tags that may not necessary be part of the set. For example

[4] `https://drive.google.com/open?id=1s_ujZJZZ3YKuBAn4CWm_WR0tQsv10bxq`

a tag 'msbuild' may return a post with tag like 'build-tool'. The approach described in step 2 to develop the tag set T has been used by many other previous studies [50, 52, 109, 110].

③ ***Identify and Extract Releng Questions & Answers (P):*** Since StackOverflow is a general questions and answers website for engineering challenges, we now use the tag set T generated in step 2 to extract only the releng posts (questions and answers) P, to be used for the main goal of this study. To do that, we consider a post as related to releng if its tag belongs to T. Applying this extraction criteria, we initially identified a total of 184,830 questions and 196,299 answers. From the extracted answers, 75,193(38.3%) are marked as accepted. To construct P, we included only questions and the accepted answers, following the recommendation from other previous studies [50, 51, 108, 109]. This give us a final set with a total of 260,023 questions and answers, where 184,830(71.0%) are questions and the rest 75,193(29.0%) are accepted answers.

④ ***Preprocess Releng posts P:*** This step, prepare our final posts P for the subsequent analysis of forming clusters of the posts. For every question, we first joined its title and body text to create one final body text. The extracted post text contains the HTML tags (for example, to present a paragraph, code snippet, URL, among others), so our first step was to clean those tags. We therefore remove all possible HTML tags which we could identify, such as $\langle p\rangle\langle/p\rangle$ and $\langle a\rangle\langle/a\rangle$, and code snippets surrounded by $\langle code\rangle\langle/code\rangle$. We then further clean the words identify as stopwords [111] such as numbers, 'a', 'the' and 'is', punctuation marks and non-alphabetical characters, as identified by MALLET's list-of-stopwords [5]. Finally, we used Porter stemmer [112] to reduce words to their stemmed representations. For example 'programmer' was reduced to 'program', 'configuration', 'configure' and 'configured' all were reduced to 'configr'.

⑤ ***Model and Label Releng Topics:*** This step aims to extract the releng topics from the final set of prepossessed questions and answers P. To do that, we build topic models on P using the MALLET [113] toolkit implementing the Gibbs sampling algorithms [114] for latent Dirichlet allocation LDA [18]. LDA is a state of the art, widely adopted topic modeling technique, which models a topic as a set of frequently co-occurring words to approximate real-world situations [50]. Further, LDA is probabilistic. The posts are categorized into K topics after I iterations grouping. A topic is a vector of word probabilities, and a document is a vector of topic probabilities. A topic with the highest proportion value is the most dominant topic.

To improve the quality of the text during classification, we choose uni-gram and bi-grams

[5] `https://github.com/mengjunxie/ae-lda/blob/master/misc/mallet-stopwords-en.txt`

Table 6.1 The Selected significantly relevant tag-sets (in light gray) for 6 releng phases.

$\langle \mu, \nu \rangle$	T_{int}: **Tags set for releng Integration**	# of tags
$\langle 0.3, 0.015 \rangle$	branching-and-merging, branching-strategy, merging-data, manifest-merging, git, branch, version-control, git-branch, feature-branch	9
$\langle 0.3, 0.010 \rangle$	branching-and-merging, branching-strategy, merging-data, manifest-merging, git, branch, version-control, git-branch, feature-branch, svn, merge, mercurial, git-merge, github	13
$\langle \mu, \nu \rangle$	T_{ci}: Tags set for releng Continuous Integration	# of tags
$\langle 0.3, 0.015 \rangle$	continuous-integration, continuous-delivery, azure-devops, continuous-deployment, gitlab, integration-testing, circleci, travis-ci, bitbucket, automated-tests, tfsbuild, jenkins	12
$\langle 0.3, 0.010 \rangle$	continuous-integration, continuous-delivery, azure-devops, continuous-deployment, bamboo, gitlab, integration-testing, circleci, travis-ci, tfvc, bitbucket, automated-tests, tfs2013, tfsbuild, jenkins	15
$\langle \mu, \nu \rangle$	T_{bs}: Tags set for releng Build System	# of tags
$\langle 0.2, 0.010 \rangle$	build,msbuild,build.gradle,phonegap-build,tfsbuild,build-process,build-automation	7
$\langle 0.2, 0.005 \rangle$	build, msbuild, interface-builder, build.gradle, flash-builder, phonegap-build, tfsbuild, build-process, scenebuilder, build-automation	10
(0.2,0.005)	build, msbuild,interface-builder, build.gradle, flash-builder, c++builder, phonegap-build, tfsbuild, build-process, query-builder, stringbuilder, scenebuilder, build-automation, process-builder, powerbuilder	15
$\langle \mu, \nu \rangle$	T_{IaC}: Tags set for releng Infrastructure-as-code	# of tags
$\langle 0.3, 0.015 \rangle$	ansible, chef, puppet, ansible-playbook, chef-recipe, salt-stack, salt,ansible-inventory, cookbook, test-kitchen	9
$\langle 0.3, 0.010 \rangle$	ansible, chef,puppet, ansible-playbook, chef-recipe, salt-stack,salt, chef-solo, knife, ansible-inventory, cookbook, test-kitchen, terraform	13
$\langle \mu, \nu \rangle$	T_{dep}: Tags set for releng Deployment	# of tags
$\langle 0.2, 0.010 \rangle$	deployment,azure-devops,web-deployment,development-environment,environment,devops,production-environment	7
$\langle 0.2, 0.005 \rangle$	deployment, environment-variables, azure-devops, web-deployment, development-environment, environment, devops, production-environment, production, setup-deployment, azure-pipelines	11
$\langle \mu, \nu \rangle$	T_{rel}: tags set for releng Release	# of tags
$\langle 0.2, 0.010 \rangle$	release,rollback,maven-release-plugin,autorelease,release-management	5
$\langle 0.2, 0.005 \rangle$	rollback, maven-release-plugin, autorelease, release-management, nsautoreleasepool, ms-release-management, release-mode,azure-pipelines-release-pipeline,transactions	8

following the previous studies that has shown that transformations such as bi-gram improve the quality [115]. Additionally, the number of topics K and the iterations grouping I are parameters set by user as a way to control the granularity of the discovered topics [51]. To help us discover the right number of topics K, we experimented with varying values of K ranging from 5 to 60 in increments of 5, and I varying from 500 to $3,000$ with increments of 500. We aim to capture a broad range of topics within our dataset while keeping them distinct from each other. Our experiment with $K = 40$ topics and $I = 1,000$ returned meaningful releng topics of medium level of granularity. The topics number K and the iteration I settings is consistent with other previous studies [51, 52, 116] that used StackOverflow posts.

After generating these LDA topics grouping, the next steps were to manually assign a label to all the 40 returned topics groupings (i.e., the output of running MALLET). First, the documents were shared among the labeling team (the first and the second author of this study). Each document in a group contains the corresponding probabilistic value scores and the top 30 keywords extracted from common, occurring words in the group of documents. Intuitively, the higher the probability value of a document, the higher its contribution to that topic's words, and hence its impact on assigning the topic label. To provide the right labels, we sort the documents by the dominant probability values from highest to lowest and randomly read through the top 15 to 20 documents representing each group. We used that information together with the top 30 keywords to assign a topic name to each group. During the topic labeling, the authors discussed and assigned labels after having a common agreement. The first author is a graduate student with many years of experience in software development, the second and third authors are both professors with extensive research experience in releng. Our final list of releng topics has a total of 38 topics and provides the answers to our **RQ1**. Table 6.2 shows the results of our topic label. The topic 9 and 21 were merged into one topic. Topics 38 and 39 were also merged because of their similarity.

⑥ ***Identify Releng Popular Topics:*** This step aims to show the topics obtained in the previous steps that may be more popular among release engineers. To determine the popularity of a topic, we used three metrics adapted from previous works, as follows: average number of views [50, 52, 110, 117, 118], average total number of questions marked as favourite by users [50, 110, 117, 119], and the average question scores [50, 110, 117–119]. Intuitively, a more popular topic is the one that has the higher number of views, favorites, and a higher score. The results of this step is shown in Table 6.3 to answer **RQ2**.

⑦ ***Identify Difficulty of Releng Topics:*** At this step we measure the difficulty of each topic to showcase how challenging they may be to release engineers and call for more attention. To measure the difficulty of a releng topic, we used two (2) known metrics adapted

from previous works. The percentage of questions of a topic that have no accepted answers [50,52,108,110], and the average of median time needed by the questions of a topic to receive an accepted answer [50, 52, 110]. Intuitively, a more difficult topic is the one having fewer accepted answers received after a long amount of waiting time. The analysis of this Step corresponds to our research question **RQ3**. Table 6.4 shows the difficulty of releng topics.

(8) - Determine the correlation of Popular and Difficult Releng Topics: This step identify the correlation between the difficult and the popular releng topics (if any). We used Kendall correlation tests to measure these correlations, since it is considered more stable (since it is less sensitive to outliers). We investigated 6 correlations between the 3 popularity metrics and the 2 difficulty metrics for the releng topics discussed in step 6 and step 7. The results of this step is shown in Table 6.5 and answers **RQ4**.

(9) Determine the Types of Questions in the Topics: To identify the types of questions asked by release engineers on StackOverflow, we used a qualitative coding on a statistically significant random sample of releng questions. We chose for every high-level topic's category shown in Figure 6.2, a random sample. We chose a sample size corresponding to a 95% confidence level and a 5% confidence interval. In total, our random sample has 2,646 questions distributed as: Integration, CI/CD, IaC, Build System, Deployment, release, general topics: = 381, 382, 378, 381, 378, 370, 376, respectively. During the coding process, the authors assigned a category to each question using the following labels : '*How*?', '*Why*?', and '*What*?'. The labels are defined as follows. **(1)** A *How*? type of question seeks better ways to achieve a result. These type of questions ask for guides on how something can be done or the ways to set up an environment. For example *"how to build and run a console application within the build and have it write output files to the output directory of another project?"*. **(2)** A *Why*? type of question examines the reason, or the cause, or the purpose for an occurrence. Developers may ask for reasons why their proposed solution failed to work or why they have an error. An example includes *"Mercury Quick Test Pro and Virtual machines: Works from one client machine but not another,... If I access this virtual machine using another machine (using Remote Desktop), the script starts fine, but stops halfway through...Has anyone had this problem before, or have any idea why the behaviour is different between the two machines?"*. **(3)** Finally, the *What*? type of question aims to get the information related to something (e.g., a clarification).For example, *"Exactly what is integration testing - compared with unit?"*

During our coding, to get the full sense of what a selected post is about, we directly read through the post from StackOverflow using the ID of the post. We then repeatedly go through the posts line per line before assigning the label as either *How*, *Why* or *What*

defined above. For the posts where we identify more than one type of question such as *How*? and *Why*?, we discuss further to identify the most dominant theme and use it. We follow the coding approach used by Treude et al. [120], whereby for their case, they defined the types into 10, i.e., *"how-to, discrepancy, environment, error, decision help, conceptual, review, non-functional, novice, and noise"*. In this case, they were interested in the nature of the question; unlike our case, where we are interested in the general concept. For example, a question which could be *decision help, conceptual, non-functional* fall under the *What*? category in our study. Finally, we label the questions that are not identified under any of the above categories, as *others*. The results of this step is shown in Figure 6.5 and answers **RQ5**.

6.3 Results

6.3.1 RQ1: *What topics are discussed around Release Engineering?*

Table 6.2 shows the topic name, the category, and the top 10 words for releng topics that release engineers discuss in StackOverflow. As explained in step 5 of our methodology, topics *No.* 9 is merged with *No.* 21 into a single topic *Build Failure*, while topic *No.* 38 is merged with *No.* 39 into the topic *Merge Conflict*.

As one can see in Table 6.2, there is a broad range of 38 releng topics that release engineers ask. These topics span across all phrases of releng. We present the topics both at a higher and lower level of granularity. For example, topics *Merge Conflict*, *Web Deployment*, *Web UI Testing*, *Ansible* are fine-grained whereas topics *Software Testing*, *Security* are coarse-grained. In Figure 6.2, we give a visual presentation of the releng topics. For each category and topic, we indicate the percentage of questions asked, arranging them from the highest in the left, to the lowest in the right. The percentage score for a topic indicates how much the topic dominates among others. Summing these scores gives a total percentage for the category. Figure 6.2 was constructed by repeatedly arranging similar topics into groups. We combined the two phases of 'release' and 'production' due to the fewer topics discovered relating to them.

We can see that, the most dominating topics by category are *"Continuous Integration/ Continuous Deployment (CI/CD) (*21.9%*)"*, followed by *"Build System (*19.9%*)"*, then *"Integration (*18.3%*)"*, *"General Topics/ Different Environment* (12.6%*)"*, *"Deployment (*11.1%*)"*, *"Infrastructure-as-Code (IaC)* (11.0%*)"* and finally *"Release* (5.2%*)"*. This may suggest that many IT companies are adapting to CI/CD practice, yet they still find it challenging. Hence maybe a reason for its questions dominating among the rest. Figure 6.2 also shows that

Table 6.2 Release Engineering Topic Label, Category, and their top 10 stemmed words separated by a commas

No	Topic_Name	Category	Topic Words
0	*Build Performance*	Build	issue, time, problem, fix, start, happen, solve, stop, wait, long, day, close, bug, minute, hour, finish
1	*Environment Variables*	Build	set, environment, variable, configuration, default, property, setting, define, global, override, configure
2	*Team Foundation*	CI/CD	version, source, check, control, late, tfs, system, support, update, upgrade, tool, team_foundation
3	*Revision Specifier*	Integration	multiple, number, specific, single, time, revision, separate, date, give, mercurial, current, part
4	*Dependenc Management*	Build	project, dependency, library, include, main, jar, ant, share, maven, external
5	*Security*	Diff. Environment	user, access, key, private, account, public, password, permission, credential, login, username, security
6	*Web UI Testing*	Continuous Integration	page, open, click, show, select, link, view, element, display, browser, text, button, puppeteer, form
7	*Ansible*	IaC	task, ansible, list, host, module, playbook, template, group, variable, role, define, condition
8	*File Transforms*	Deployment	file, content, ignore, include, modify, exclude, xml, replace, rename, edit, filename
9*	*Build Error Debug*	Build	find, follow, type, search, give, resolve, information, miss, problem, documentation
10	*Web Deployment*	CD & Deployment	server, application, deploy, web, deployment, publish, site, azure, website, setup
11	*Virtualization*	Deployment	run, machine, window, start, setup, locally, slave, virtual, running, successfully, computer, runner
12	*Experience with VCS*	Integration	tag, svn, system, tool, large, time, subversion, lot, big, small, good, free, easy, people
13	*Locate Move around Files*	Diff. Environment	folder, copy, directory, path, delete, file, root, location, structure, move, workspace
14	*Team & Project Management*	Distributed Development	team, production, developer, development, code, dev, good, manage, live, product, work
15	*Feature Expansion*	Dev. Environment	change, make, apply, back, modify, switch, original, edit, state, modification, detect
16	*Decision Support around Releg.*	Asking for Guides	base, good, approach, common, easy, handle, component, provide, simple, require
17	*Software Testing*	Continuous Integration	test, integration, unit, testing, automate, write, case, selenium, continuous, result
18	*Docker*	CD and Deployment	service, image, docker, instance, server, container, connect, start, host, connection
19	*Code Review*	Distributed Development	request, pull, send, post, email, status, url, comment, receive, action, event
20	*Continuous Integration*	Continuous Integration	build, agent, vst, definition, building, teamcity, tfs, successful, bamboo, drop
21*	*Build Error*	Build	error, fail, follow, message, exception, throw, unable, failure, occur, log
22	*Documentation*	Diff. Environment	question, answer, understand, bit, read, point, thing, explain, difference, similar
23	*Continuous Deployment*	CI/CD	job, jenkin, pipeline, plugin, trigger, parameter, configure, pass, groovy, slave, schedule
24	*Msbuild*	Buld System	solution, studio, project, visual, msbuild, reference, target, assembly, framework, core, nuget, dll
25	*Rollback*	Release	transaction, rollback, follow, error, call, state, row, insert, work, execute, set, fail, run, record, minion
26	*Text Formatting*	Diff. Environment	case, order, level, note, end, match, rail, block, rule, top, part, simply, fact, space, side, pattern
27	*Code Compilation*	Build	code, generate, compile, report, include, tool, source, link, coverage, binary, static, result, compiler, cmake
28	*Tool Customization*	Build System	add, option, custom, remove, import, edit, enable, follow, extension, section
29	*Artifact Management*	Release	release, step, download, stage, artifact, debug, azure_devop, follow, mode
30	*PowerShell*	IaC	script, execute, process, run, write, program, command, powershell, execution, bash
31	*Work Items*	Distributed Team	work, fine, problem, item, thing, idea, make, correctly, perfectly, properly
32	*Creational Design Pattern*	Diff. Environment	call, method, return, function, object, load, code, pass, memory, array
33	*Update Rules*	Prod Environment	create, update, exist, automatically, check, manually, auto, empty, automatic, follow
34	*MobileApp Debug & Deployment*	Development and Release	android, app, problem, application, phonegap, device, find, support, platform, show
35	*Branching & Remote Upstream*	Branching & Merging	repository, push, local, remote, repo, clone, pull, fork, git, bitbucket, origin, fetch, submodule, locally,
36	*Command Line Interface*	CD, Build	command, line, output, follow, log, show, result, console, print, give
37	*Configuration Management*	IaC	package, instal, install, chef, client, resource, puppet, cookbook, attribute, installation
38*↓	*Merge Conflict*	Integration	branch, merge, master, feature, conflict, develop, trunk, rebase, back, delete
39*↓	*Git Revert Code Changes*	Integration	commit, history, point, git, tree, make, parent, back, current, index, patch, changeset, head,
			CI=Continuous Integration, CD=Continuous Delivery, IaC=Infrastructure-as-Code

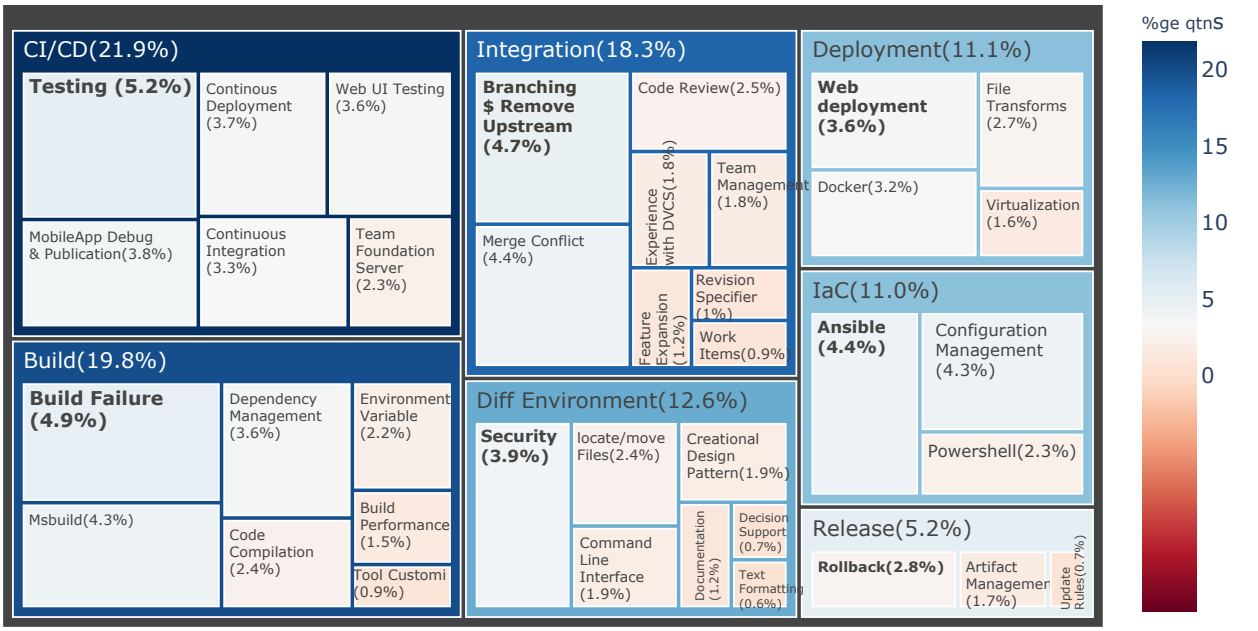


Figure 6.2 Percentage of questions asked in Release Engineering Topics

the topic *Software Testing* is dominating in CI/CD category, while *Build Failure* dominates in the Build System category. *Branching & Remote Upstream* dominates in the Integration category, *Security* dominates across the different environment, *Web Deployment* dominates in Deployment, and *Rollback* dominates in the Release category.

> Release Engineers ask about a broad range of 38 topics, where most of the questions are related to *Continuous Integration/Continuous Deployment (CI/CD) (*21.9%*)*. Only few questions ask about *Release (*5.2%*)* in StackOverflow.

To illustrate and give meaning to the topics above, we discuss them further by referring to their questions, for the topics identified as most dominating.

Software Testing: Figures 6.2 and 6.3 show that the topic *Software Testing* is the most dominating in the *CI/CD* category; representing 5.2% of questions that release engineers ask in StackOverflow. According to our dataset, the most viewed *Software Testing* question in StackOverflow is the question with identifier (Q_{id}: 5357601) *"What's the difference between unit tests and integration tests?[duplicate]... Are there different names for these tests? Like some people calling unit tests functional tests, etc?"*. This question is the most viewed with 222K, fourth most marked favourites and most scores (in our CI/CD category), and is a duplicate of question Q_{id}: 10752. *Software Testing* questions about "Spring MockMVC" like *"How to test a spring controller method by using MockMvc?"*, questions on "automation test

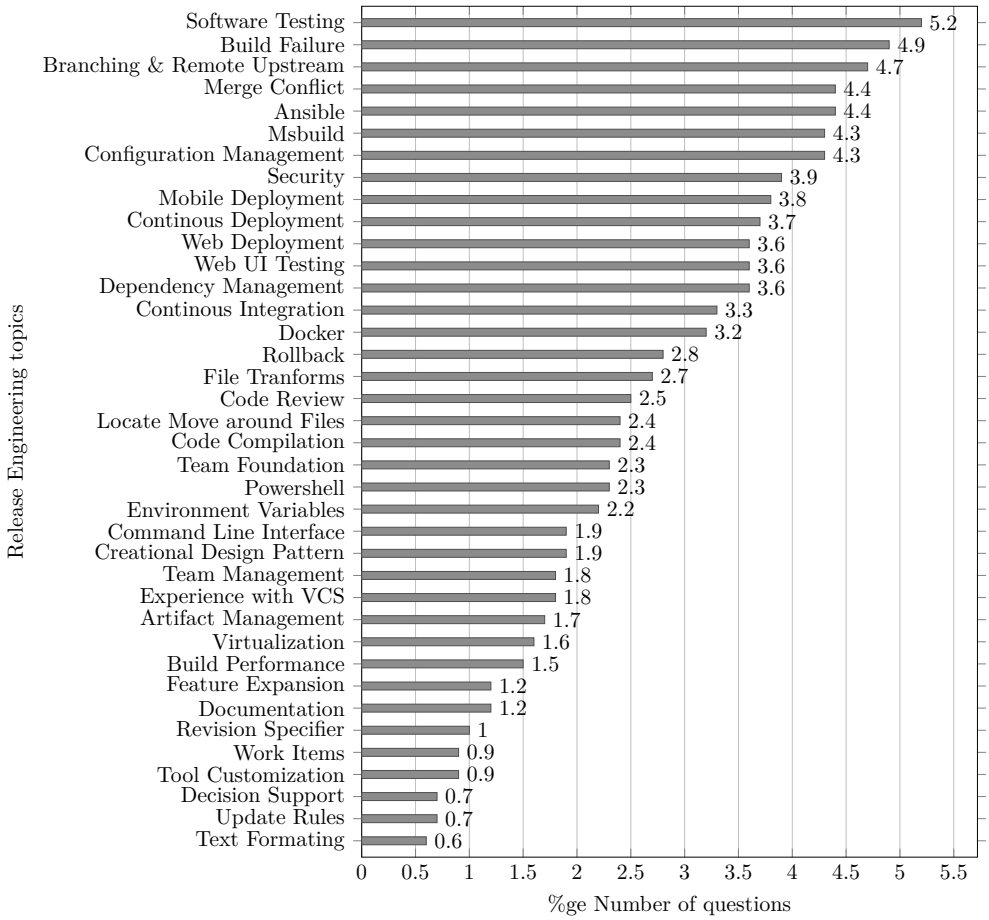


Figure 6.3 Topics and percentage number of their questions.

for IOS" like Q_{id}: 20730917, 402389 "Setting up appium for iOS app test automation" may be due to poor documentation. While *Software Testing* is the most important contributor to quality assurance of the software product and must be performed at different stages of software development. Our study only gives a general view of the area that is challenging to release engineers, understanding the exact challenges in software testing call for future study. Other releng topics that may be interesting to investigate more in the future (in CI/CD category), according to our analysis are *"MobileApp Deployment", "Web UI Testing".*

Build Failure:- Figures 6.2 and 6.3 shows that the topic *Build Failure* is the most dominating in the Build System category and the second most dominating releng question (4.9%) in StackOverflow. Examples of questions related to *Build Failure* include Q_{id}: 6383953 *"We have a situation where our builds have stopped executing in a stable manner. At a rate of about one every three we receive either TF215096 or TF215097 errors & the Build fails. If we then restart the Build controller, it works again - until next time...Server logs provide with little info, at least we 've found nothing that helps us resolve the situation.*

Various searches in the Net were also not productive...", Q_{id}: 41570435 *"Could not find com.android.tools.build:gradle:3.0.0-alpha1 in circle ci ... update the gradle plugin to the latest :... and this error occured.."*.Despite the abundant research that examined the causes of build failures, their automatic resolution, and proposed best practices [121–123], our results show that release engineers frequently seek help about *Build Failure*; which may suggest that they still struggle to obtain working solutions. Hence, more effort is needed to improve build systems, in particular, according to our findings shown on Figure 6.2 and Figure 6.3, more attention should be given to *"MsBuild", "Dependency Management", "Code compilation", "Environment Variables" and "Build Performance"*

Branching & Remote Upstream: Figures 6.2 and 6.3 show that *Branching & Remote Upstream* is the most dominating topic in the *Integration phase* category. It is the third most dominating topic (4.7%) that release engineers discuss on StackOverflow. This topic is about challenges in setting up repository branches and keeping them in sync. The most viewed question on this topic is (Q_{id}: 2765421) : *"...1 want to be able to do the following: 1) Create a local branch based on some other (remote or local) branch...2) Push the local branch to the remote repository (publish), but make it trackable so git pull and git push will work immediately... 1 know about –set-upstream in Git 1.7, but that is a post-creation action. I want to find a way to make a similar change when pushing the branch to the remote repository."*. The question is the most viewed (3.4m) in the topic *Branching & Remote Upstream*. It took an average of 9,480 hours to receive the right answers, and the second highest scores after question Q_{id}: 6591213 which is asking about changing branch name. Other examples of questions on this topic includes Q_{id}: 8345141 *"Keep histories in sync between local and remote mercurial repositories... The problem is then that the personal server-side clone is totally out of sync with the local repository because it was not rebased. We aren't using proper branches, so merge+rebase and transplant/graft seem to not be what we would use to get the repositories back in sync..."*.

Ansible:- is a lightweight general-purpose tool for automating the configuration of application deployment, cloud provisioning, and intra-service orchestration, among others [124]. Figures 6.2 and 6.3 show that the topic *Ansible* has the highest number of questions (4.3%) in IaC category and the fourth most number of releng questions in StackOverflow.

The most viewed questions about *Ansible* include Q_{id}: 32627624 *"Ansible: how to run task on other host inside one playbook? ... Is it possible to include one playbook in another? ..."*, and Q_{id}: 31152102 *"Is it a good idea to make Ansible and Rundeck work together, or using either one is enough? ... Seems both Ansible and Rundeck can be used to do configuration/management/deployment work, maybe in a different way So my questions are: ... If they can be used*

together, what's the best practice?... ". The large number of questions about *Ansible* suggests that it may not be as easy to understand and use as reported in the literature [124]. According to our findings, other important topics related to the IaC category include *Configuration Management and Powershell*.

Security:- According to Figures 6.2 and 6.3, the topic *Security* is the most dominating topic in *Topic across different Environment* and eighth most dominating (3.9%) releng topic on StackOverflow. An example of question posted in StackOverflow about *Security* is Q_{id}: 56081795 *"how to restrict pipeline deployment by user?..right now, my pipeline works but anyone can trigger a deploy to any environment... the only thing I found is to block the entire project via user management, but it's not quite what I want to achieve... what I want to achieve is to have: User1 and User2 be able to deploy to production, but User3 can't trigger a Prod deployment, or at least group them and allow Group A to deploy and Group B to not deploy to Prod"*. There are other security questions related to 'automating the code signing and server certificate', i.e., Q_{id}: 22431526, 35821245, 23534429. Other general topics that cross between different environment in our findings, in order of dominance, include *Locate/ Move around Files, Command Line Interface, Creational Design Pattern, Documentation, Decision Support, and Text Formatting*.

Web Deployment:- Figures 6.2 and 6.3 show that *Web Deployment* is the most dominating topic in the Deployment category and the 11^{th} most discussed releng topic (3.6%) on StackOverflow. An example of *Web Deployment* question is Q_{id}: 49726235 which complain about a 'lack of tools': *"Web-based complex data-center automation tool..After evaluating existent tools like Ansible Tower, rundeck and others, it seems that no tool can fulfill the needed requirements. We have complex data-center servers, cluster of DB and web servers, the data-center has a lot of client-systems, +100, and other tools like solr, redis, kafka... deployed there across the physical servers, not to mention that the same data-center servers have different accounts, linux users, (QA,stag,production..etc), for now the meta-data about these environments alongside their web-apps, source code to be used, servers of the cluster are all defined on xml and there is a bash scriptsreads from that XML that operated manually to run any operation/task (like checkout the source, build, deploy, start, stop... and other customized operations)....And if this is not the right place, can you please point out where cat I ask such question?..."*. This question did not receive a right answer after two years and have not attracted many views. Other posts include Q_{id}: 51619 *"How to set up Git bare HTTP-available repository on IIS"*, Q_{id}: 47539115 *"How to GitLab Shared Runners deploy to server"*, Q_{id}: 33446277 *"Deploying the same site N times"*. "Troubleshooting Web Management Services" Q_{id}: 56559488, 56233303, 44929985, 54100188, 44573433, 37665540, 34457183, "Installing application in the

remote server" Q_{id}: 33012702, 37221920, 54975830, "Web application and windows services" Q_{id}: 1789316, 9590422, 29808512 among others. Finally, other important topics in the *Deployment* category according to our findings include *"Docker", "File Transformation" and "Virtualization"*.

> *Software Testing, Build Failure, Branching & Remote Upstream, Merge Conflict, Web Deployment, Ansible & Configuration Management, Security and Rollback* are the top 9 releng topics discussed. They cover all releng phases. The *Software Testing* has the most asked questions and *Text Formatting* is least discussed topic on StackOverflow.

6.3.2 RQ2: *What topics are popular among the release engineers?*

Table 6.3 shows the results of our releng topic's popularity. The popularity of topics is measured as explained in step 6 of our analysis, and sorted by average total of views from the highest to the lowest. Intuitively, a more popular topic is one having more views, with more questions marked as favorites and receiving higher scores [50,108–110,118]. According to Table 6.3, the topic *Merge Conflict* has the highest average number of views, average number of favorite questions, and average number of scores. In contrast, topic *Web Deployment* has the lowest average number of views, sixth least favorites, and least scores. Also, topic *Creational Design Pattern* is the second least viewed, the least marked as favorite and the second least score.

> *"Merge Conflict", "Repositories Structure & Remote Upstream", "Feature Expansion"* of Integration category are among the most popular releng topics, while *Web Deployment* and *Creational Design Pattern* are amongst the least popular topics.

6.3.3 RQ3: *How do topic popularity and difficulties correlate?*

Table 6.4 shows the difficulties of the releng topics, which is measured as described in step 7 of our data analysis. Table 6.4 is sorted in the order from the highest percentage number of questions without accepted answers.

"..., a topic with higher percentage of its questions not receiving accepted answers or taking longer time to receive accepted answers is more difficult" [109]. According to Table 6.4, topic *MobileApp Debug & Deployment* has the highest percentage of questions without accepted answers, and is in eighth position in terms of time required to receive an accepted answer. *Continuous Deployment* has the third-highest percentage number of questions with no ac-

Table 6.3 Releng Topics Popularity

Topic_Name	Avg. Views	Avg. Favourites	Avg. Scores
Merge Conflict	9166.72	5.5	17.74
Branching & Remote Upstream	7618.43	3.6	10.83
Feature Expansion	6913.07	3.53	12.18
Security	5077.71	1.85	5.95
File Transforms	4988.98	3.00	10.03
Experience with VCS	4705.83	3.67	10.04
Locate Move around Files	4452.63	1.48	5.31
Team Foundation	4293.11	2.27	7.32
Tool Customization	4226.45	1.46	5.12
Update Rules	3957.25	1.87	6.32
Documentation	3892.68	2.41	6.23
Command Line Interface	3542.88	1.13	4.34
Web UI Testing	3365.99	1.28	4.78
Continuous Deployment	3066.3	0.71	2.9
Build Failure	3064.18	0.82	3.54
Environment Variables	3048.08	0.7	2.92
Text Formatting	3014.6	2.42	5.91
Msbuild	2937.23	1.05	4.36
Build Performance	2769.02	1.02	4.52
Configuration Management	2720.93	1.05	4.27
Code Review	2676.75	1.23	4.94
Dependenc Management	2649.33	0.95	3.42
PowerShell	2512.86	0.59	2.34
Revision Specifier	2452.26	1.25	4.58
Work Items	2444.42	1.00	3.61
Decision Support around Releg.	2377.9	1.67	5.22
Virtualization	2334.51	0.66	2.56
Rollback	2319.14	1.03	3.21
Artifact Management	2309.59	0.81	3.13
Docker	2254.28	0.87	2.94
Code Compilation	2222.25	2.84	3.39
Team & Project Management	2031.85	1.66	4.19
Ansible	2031.2	0.88	3.25
MobileApp Debug & Deployment	1923.66	1.01	3.32
Continuous Integration	1829.94	0.7	2.79
Software Testing	1749.85	0.93	2.99
Creational Design Pattern	1489.96	0.52	2.04
Web Deployment	1363.5	0.7	2.03
releng Topics	**3894.43**	**1.87**	**6.10**

Table 6.4 Releng Topics Difficulty

Topic_Name	% No acc. Ansers	Hrs to acc. Answers
MobileApp Debug & Deployment	59.66	6.00
Continuous Deployment	59.09	9.00
Docker	59.09	6.00
Virtualization	58.58	9.00
Web Deployment	56.97	9.00
Build Failure	56.5	5.00
Code Compilation	55.78	9.00
PowerShell	54.76	3.00
Software Testing	54.2	7.00
Build Performance	53.74	4.00
Artifact Management	53.48	4.00
Code Review	53.35	4.00
Web UI Testing	52.42	4.00
Environment Variables	51.99	3.00
Command Line Interface	51.21	3.00
Dependenc Management	50.96	2.00
Update Rules	50.51	2.00
Security	50.44	4.00
Rollback	50.28	1.00
Configuration Management	49.61	4.50
Tool Customization	49.46	1.00
Decision Support around Releg.	48.92	2.00
Work Items	48.5	3.00
Locate Move around Files	47.84	1.00
Continuous Integration	47.74	11.00
Creational Design Pattern	47.32	0.00
Text Formatting	46.77	1.00
Ansible	46.33	6.00
File Transforms	45.52	1.00
Team & Project Management	43.9	1.00
Msbuild	43.39	8.00
Revision Specifier	43.01	3.00
Branching & Remote Upstream	41.61	0.00
Documentation	41.08	1.00
Feature Expansion	40.24	0.00
Experience with VCS	40.21	1.00
Team Foundation	39.73	1.00
Merge Conflict	39.12	0.00
releng Topics	**49.82**	**3.00**

cepted answers, and the third-highest average median time needed to receive an accepted answer. In contrast, topic *Merge Conflict* has the least percentage of questions with no accepted answers and the least amount of hours waited to receive accepted answers.

MobileApp Debug & Deployment:- Table 6.4 shows *MobileApp Debug & Deployment* having the highest percentage of non-accepted answers (59.66% higher than the rest of the releng topics in StackOverflow). This may signal that it is difficult for release engineers to find answers related to *MobileApp Debug & Deployment* and therefore that it is a difficult topic. If we analyze question Q_{id}: 11934125 which is about "Automated testing of Hybrid apps" : *"If I want to perform automated testing of a PhoneGap app (for now, only on iOS), what options do I have (if any)?, I had also looked at using Jasmine with the Jasmine-jQuery plugin for much the same thing but it requires duplicating the app HTML (and the overhead of keeping the two in sync etc)"*. This question despite being popular did not receive a clear answer after more than seven years. The question was asked more than once (Q_{id}: 18739352) in StackOverflow. This situation may tell us (i) Mobile app developers are migrating from native apps to hybrid mobile development apps, due to its numerous benefits such as simplicity, portability, cross-platform support, reusing web development knowledge and cheap development processes [125, 126] among others. However, it seems that (2) they still lack efficient tools (e.g., for automation tests) to help in the migration and development of hybrid app.

Continuous Deployment:- Similarly, Table 6.4 shows that *Continuous Deployment* related questions are difficult to answer, with the percentage of questions with no accepted answers being 59.09%. An example of *Continuous Deployment* question that took a very long time (4 years) to get an acceptable answer is Q_{id}: 31806108: *"Is there any method or plugins available to retrieve deleted Jenkins job? I have mistakenly deleted one job from Jenkins. So please give a suggestion to undo the delete"*. However, this difficult question was viewed over 20k times between August 2015 and January 2019.

> *"MobileApp Debug & Deployment" and "Continuous Deployment"* are among the most difficult releng topics.

6.3.4 RQ4: *How do topic popularity and difficulties correlate?*

Our analysis indicates that topics such as *Merge Conflict*, *Branching & Remote Upstream* and *Feature Expansion* [127] are more popular in Table 6.3, but the least difficult in Table 6.4. These results inspired us to investigate the correlations between popular and challenging

Table 6.5 Correlations of releng topics popularity/difficulty.

coefficient / p-value	Avg. views	Avg. Favourites	Avg. Scores
% w/o acc. answer	**-0.338/0.002**	**-0.460/4e-05**	**-0.492/1e-05**
hrs to acc. answer	0.042/0.708	0.046/0.681	0.053/0.637

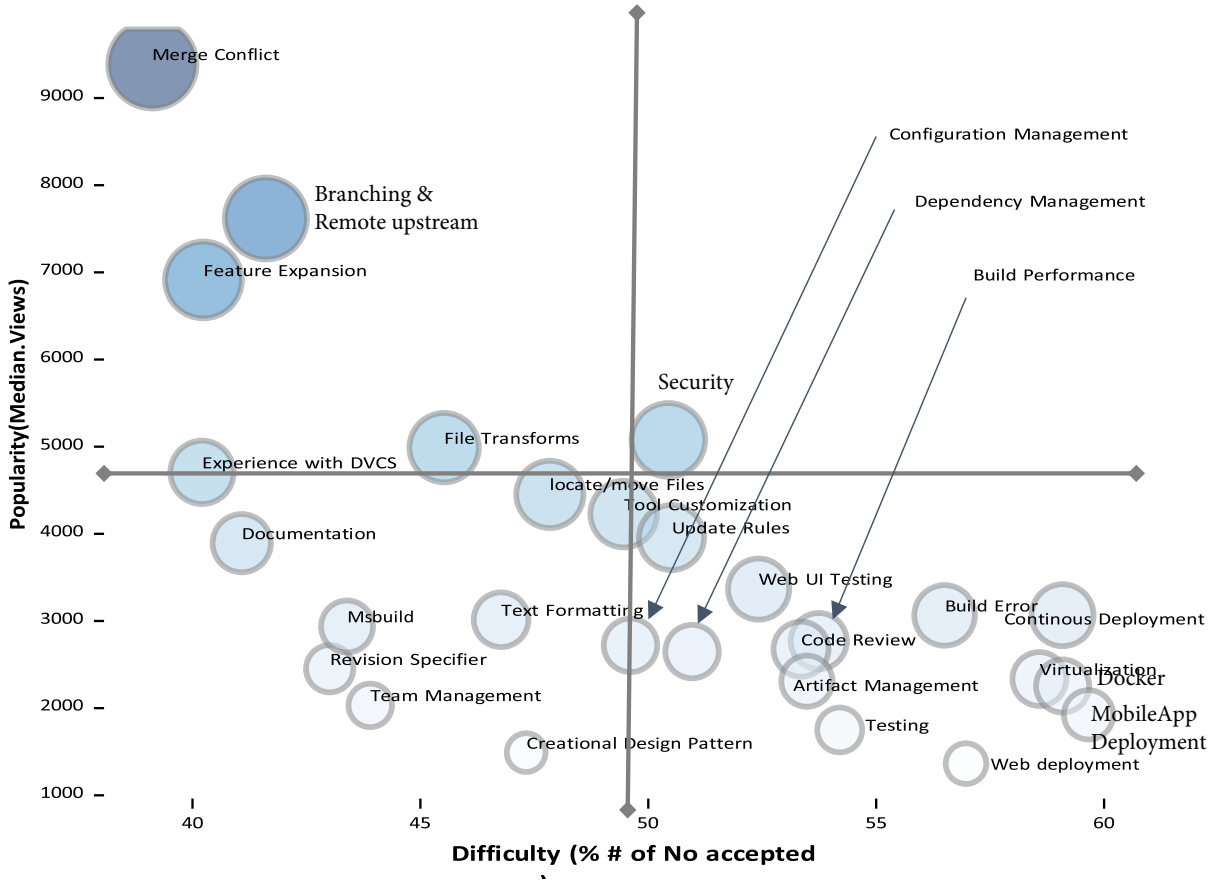


Figure 6.4 Comparison of releng topics by popularity & difficulty

topics. As explained in step 8 of our analysis approach, we investigate the relationship using Kendall correlation by measuring the six correlations against the three popularity and two difficulty metrics of releng. Table 6.5 show the correlation measurement results of our analysis. The results in Table 6.5 shows a statistical negative correlation between the popularity and difficulty of releng topics even at 80% confident level. We further discuss the

tradeoff between popularity and difficulty in Section 6.4.

> There is a statistically significant negative correlation between the popularity and difficulty of releng topics.

Our results, however, do not imply that difficulty and popularity topics in StackOverflow are always negatively correlated. For example, Bagherzadeh and Khatchadourian [50] found that big data topics in StackOverflow are not negatively correlated. Similarly, the topic of general mobile development which is found to be popular by Barua et al. [51] is also seen as difficult by Rosen and Shihab [52].

6.3.5 RQ5: *What types of questions do release engineers ask in StackOverflow?*

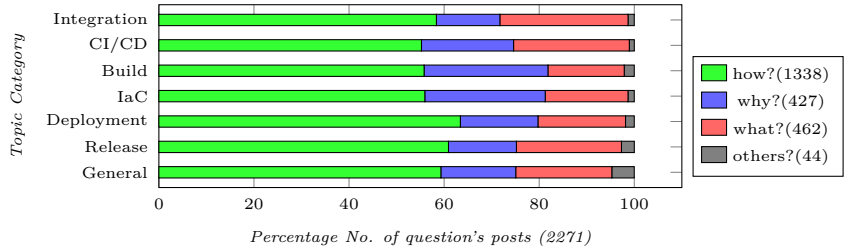


Figure 6.5 Distributions of questions type in Releng topics category

Figure 6.5 shows the distributions of question types for the studied releng topic's category. We performed a Chi-squared test and found that there is no statistically significant difference between the types of questions across the topic's categories with a $p-value < 0.05$. Overall, as shown in Figure 6.5, the most prevalent type (58.9 %) of questions is "How", followed by "What" (20.3 %) and "Why" (18.8 %) for all releng categories. This result shows that release engineers are looking for more specific help to their problems, concepts, and errors. The high percentage of "*How*?" questions in our results is consistent with the findings by Rosen and Shihab [52].

The majority of "*how*?" questions are related to *'Deployment'* (63.42%). The category IaC and Build has more "*why*" questions (26.84% and 25.33%, respectively), implying that better tools to help debugging could be most useful to improve releng pipelines, for example, Q_{id}: 32871956 asks "Why Aren't My Chef Normal Attributes Persisted?". The topic category *Integration* shows the most "*what*" questions (26.58%); meaning that developers need some guidelines, documentation, and other useful information, to guide them in the integration phase of releng, for example, Q_{id}: 17466933 asks "What is the Git branching strategy with

agile process?".

> Overall, the most prevalent type of questions is "How" (58.9 %), followed by "what" (20.3 %) and then "Why" (18.8 %), for all releng categories.

6.4 Discussion and Implications

So far in this Chapter, we have highlighted the different releng issues that are discussed in StackOverflow, pointed out the most popular and difficult topics, and finally, the types of questions that releng engineers ask. In this section, we summarise and discuss the impactful issues and their implications for the researchers, practitioners, and educators.

Impactful Topics:- Although all the topics presented in this study are important in their specific area. We particularly summarized the releng popular and difficult topics, which we believe should be given more attention by researchers and practitioners. Figure 6.4 shows the popular topics in the y-axis as a measure of average views and the difficult topics in the x-axis as a percentage of questions with no accepted answers, and further divided into four quadrants showing more clearly, the relative popularity and difficulty of the issues. As can be seen in the top right quadrant, the topic *Security* is found to be both popular and difficult. This is therefore an important direction of research. Other topics we believe that need more attention as shown in Figure 6.4, include *Merge Conflict, MobileApp Deployment, Virtualization, Continuous Deployment, Debugging, Code Review, Web Deployment, File Transforms, Docker, Web UI Testing*. Identifying the actual problems related to each of these topics is out of the scope of this study and we call for more investigations by the research community. We also found that many developers are looking for the right solutions for the problems at hand (shown by the high number of "How" questions), concepts ("what" questions), and errors ("why" questions). This suggests the need for a general effort from both researchers, tool builders, or practitioner, and the educators.

To Researchers:- The empirical results of our study provide general views about the trends in releng; highlighting both the popular and challenging releng issues that are being discussed. We encourage researchers to tackle the top 10 difficult releng topics that we have identified, i.e., *MobileApp Debug & Deployment, Code Review, Docker, Continuous Deployment, Virtualization, Web deployment, Build Error Debug, Web UI Testing, Build System Performance, Security and Configuration Management*. According to Figure 6.4, the *Security* topic is both difficult and popular. We also found in Chapter 4 that Security & Privacy as one of the ML properties not consistently tested potentially indicate the challenge this area. This calls

for more attention on releng security challenges. By infecting a build, malicious users could distribute their malware to thousand and even millions of users. Researchers should invest in developing efficient techniques and tools to support release engineers.

To Practitioners:- According to our findings, we recommend the team lead to always take the difficult topics in to consideration when they are distributing works between the project team members. Release activities related to *Security, Docker, Virtualization, Code Review and Continuous deployment* are more difficult and they should be assigned to the more experienced team members. Whereas the tasks related to code integration (with the exception of code review), could be assigned to even a less experienced team members. For the high percentage of the topic *"Software Testing"* in the CI/CD release engineering phase, the ML practitioners can also use the results on ML software testing practices in Chapter 4 and Chapter 5 to learn about the various testing strategies and the test methods.

To Educators:- Release engineering topics with higher percentage of questions on StackOverflow such as *"Software Testing"*, *"Branching & Remote Upstream"* and *"Merge Conflict"* should be better covered by training and course materials. Educators should also pay a particular attention to "Security" issues and ensure that challenges related to security in releng pipelines are covered in course materials.

6.5 Threats to validity

To identify and extract the releng related posts in StackOverflow, we entirely relied on the selected releng tags, which is a threat. Our analysis may have missed to identify some releng posts. To mitigate this threat, we adapted methodologies used by many previous studies [50, 52, 109, 110]. We are confident that this resulted in a significantly relevant releng tag set. Another threat is due to manual labeling, where we read the questions posts to appropriately map them to the topics. There is no tool that we could use to perform this task automatically. We further minimized this threat by both relating the topics keywords and randomly selecting at least 15 questions in the order of highest probability value to the category, and relating them to the user-defined tags. This approach has also been used by many previous works [50, 109, 117]. Another possible threat could be when choosing the optimal number of topics K and iterations value I. We, therefore, followed a well-defined technique adapted from [50–52, 117] together with multiple experiments to ensure that we identify reasonable K and I values. Also, LDA a probabilistic method may lead to random posts for the topics (to some extent), which is a threat. To mitigate this threat, we compared the returned 40 topics for the potential difference by running our final model at least four

times; in all, we did not find a significant difference. Also, considering StackOverflow for understanding releng topics may be a threat. However, given the enormous popularity of Stack Overflow and the large number of release engineers using it, we believe that this threat is minimal. Nevertheless, future investigations with other crowed-source platforms are desirable to make our findings more generic.

6.6 Summary

This empirical study presents the results of a large-scale study conducted using StackOverflow, to understand what release engineers ask about and identify the questions that are the most challenging during the six major phases of the release engineering process. We have identified popular and difficult topics and examined the relationship between them. We have categorized the questions that release engineers asked into three types (i.e., "How?", "Why?", "$What$?"). These results can be extended to identify what is being asked for any topic.Finally, we have discussed our findings and formulated recommendations for researchers, practitioners, and educators teaching release engineering. Researchers can also apply our approach in a similar way to help them get data and information related to their study.

CHAPTER 7 CONCLUSION

In this chapter, we provide the summarise our findings and conclude the thesis. And discuss our future works.

7.1 Summary

This thesis presents empirical studies on the practices of testing ML software systems and modern release engineering topics. In Chapter 4, we presented the first part of our empirical studies. We provided the first fine-grained empirical study that studies the adoption of ML testing in practices by analyzing the testing strategies and ML properties for the ML software systems. First, we examine different ML testing strategies implemented in ML workflows deployed in the field. We derived a total of nine (9) main categories of ML testing strategies used in the ML workflow, out of which six testing strategies are marked as ML specific (i.e., testing strategies for only ML related code). We find than on average, most testing is associated with Model Training (32.68%), followed by Feature engineering activities of the ML workflow. Second, we studied the specific ML properties that are tested in a ML workflow and identified 20 ML properties that ML engineers commonly test. We find that some of the identified ML properties, i.e., *Uncertainty*, *Security & Privacy*, *Concurrency*, and *Model Bias and Fairness* are tested in less than half ($\leq 50\%$) of ML workflow activities. In contrast, the ML properties *Consistency*, *Completeness*, *Correctness*, *Data Validity*, *Robustness* and *Data Distribution* are tested in a large majority ($\geq 80\%$) of ML workflow activities. Third, we compared the testing strategies, and ML properties, across our studied ML software systems, to identify any potential differences and–or inconsistencies in the application of testing techniques in the field. We showed that, there is a non-uniform use of different testing strategies and ML properties within and across the studied ML software systems. We find that, at least 80% of the studied ML projects consistently use the testing strategies: *Absolute Relative Tolerance (Oracle Approximation), Error bounding (Oracle Approximation)*, *Instance and Type Checks*, *Negative Test*, *State Transition*, *Value Range Analysis*, *Decision & Logical Condition*, *Membership Testing*, and *Value Error (Fault Injection)*. For the ML properties, we found only about 20% to 30% of the ML properties such as *Correctness*, *Consistency*, *Completeness*, *Data Distribution*, *Data Validity*, and *Efficiency* are consistently tested across at least in 90% of the studied ML software systems. The ML properties *Bias and Fairness*, *Compatibility and Portability*, *Security and Privacy*, *Data Timeliness* and *Uncertainty* are not consistently tested in about 80% of the studied ML software systems. Fourth, by exam-

ining the testing strategies used across the ML properties, we observed that at least two (2) testing strategies are used to verify a single ML property, as observed in at least 80% of the identified ML properties.

In Chapter 5, we study the adoption of testing methods throughout the development phases of ML software systems. Specifically we focused on three main aspects. First, we examined the types of testing (test types/ methods) described in the Test Pyramid of ML proposed by Sato et al. [2]. We uncover 5 new types of testing not included in this Test Pyramid, i.e., *Regression Testing, Sanity testing, Periodic Validation and Verification, Thread testing*, and *Performance/Blob test*. In contrast, the *Contract test* and *Service test* described in the proposed Test Pyramid were not implemented in any of the studied projects. Secondly, we examined the composition of the testing methods across projects and found that *Unit Testing* is the most used testing method in the studied projects, accounting for between 68% to 91% of all the test cases in a given ML software system. In contrast, the proportion of adoption of the other testing methods varies dramatically from zero to 16% across the studied ML software systems, potentially indicating that ML software systems are not currently tested thoroughly during the different phases of their development life cycle. Finally, we compared the testing methods verifying a given ML properties and showed that ML engineers implement multiple tests at different testing levels (testing methods) to verify the ML properties *Consistency*, *Completeness*, *Correctness*, *Validity*, and *Data Distribution*.

In Chapter 6, we studied the release engineering topics to understand what questions are asked during release and deployment of the traditional and ML software systems and identify the questions that are the most challenging during the six major phases of the release engineering process. We identified popular and difficult topics and examined the relationship between them. Also, we have categorized the questions that ML and release engineers asked into three types (i.e., "How?", "Why?", "$What$?"). These results can be extended to identify what is being asked for any topic.

We encourage researchers, to build on our finding of 20 common ML properties to develop novel testing techniques and better tool support to help ML engineers tests for these identified properties. We also invite more studies on the evaluation of the effectiveness of the identified ML testing strategies in future. We recommend that ML engineers use our presented taxonomy, to learn about the existing ML testing strategies, and implement them in their ML workflow, especially the most used testing strategies such as *Absolute Relative Tolerance (Oracle Approximation), Error bounding (Oracle Approximation)*, *Instance and Type Checks*, *Negative Test*, *State Transition*, *Value Range Analysis*, *Decision & Logical Condition*, *Membership Testing*, and *Value Error (Fault Injection)*. We also encourage ML

maintenance teams to test for our identified ML properties in their projects in order to ensure their systems' trustworthiness. Finally, we recommend that ML engineering teams consider including testing techniques such as *Swarming Test*, *Regression Testing*, *Sanity testing*, *Periodic Validation and Verification*, *Thread testing*, and *Performance/ Blob test* (adapted from traditional software testing) into their reference Test Pyramid of ML. Finally, we urge the searchers, practitioners, and educators to explore further and use the release engineering topics discussed in Chapter 6 in their respective field. Also, we encourage the researchers our apply our approach to extract topics in StackOverflow in a similar way to help them get data and information related to their study.

7.2 Future Works

As this thesis report the testing practice used by the ML engineers in securing the quality of their ML software systems and the challenges they may face during their release engineers process. In the following, we highlight some of the potential future research directions resulting from the contributions of these empirical studies.

- We have explored and highlighted eight new testing strategies, six new test types/ testing methods, 20 ML properties, and 35 release engineering topics. It is, therefore, a new research direction for researchers in the future to explore these areas in detail.
- Evolution of the test practices: In future we hope to study how the studied tests are maintained and evolved through out the ML software development life-cycle. We still don't know when ML engineers first introduced the studied tests and how their testing strategies may have changed overtime.
- The efficiency of these testing practices: To evaluate the efficiency of the identified ML testing practice in future will help the ML engineers to choose the most adequate testing strategies for their ML software systems.
- We also invite more future works to explore the ML properties (such as *Security and Privacy*, *Data Uniqueness*, *Timeliness* or *Scalability*) which are not consistently used across the projects.

7.3 Authors remarks

It is worth noted that part of this study was peer-reviewed and published at the IEEE International Conference on Software Maintenance and Evolution (ICSME). Also, the other part

of this work is currently under the review process at the ACM Transactions on Software Engineering and Methodology (TOSEM) journal. Once again, the authors wish to exceptionally thank prof. Foutse, for his overall supervision of these studies, and inviting other external professionals consisting of both ML practitioners and researchers with extensive research experience in empirical software engineering, ML software testing and release engineering; to review and provide insightful comments throughout these studies, that greatly improved the results reported in this thesis.

REFERENCES

[1] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 291–300.

[2] "Continuous delivery for machine learning," https://martinfowler.com/articles/cd4ml.html, 2019.

[3] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 2722–2730.

[4] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[5] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Medical image analysis*, vol. 42, pp. 60–88, 2017.

[6] HP Application Handbook. (2012) Shorten release cycles by bringing developers to application lifecycle management. [Online]. Available: http://bit.ly/x5PdXl

[7] C. 2014. (2015) Moving to mobile: The challenges of moving from web to mobile releases," keynote at releng 2014. [Online]. Available: https://www.youtube.com/watch?v=Nffzkkdq7GM

[8] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 100–111.

[9] D. Cheng, C. Cao, C. Xu, and X. Ma, "Manifesting bugs in machine learning code: An explorative study with mutation testing," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 313–324.

[10] H. Shin, D. Kim, Y. Kwon, and Y. Kim, "Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 445–467.

[11] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, "Adversarial sensor attack on lidar-based perception in autonomous driving," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 2267–2281.

[12] M. Nejadgholi and J. Yang, "A study of oracle approximations in testing deep learning libraries," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 785–796.

[13] A. D'Amour, K. Heller, D. Moldovan, B. Adlam, B. Alipanahi, A. Beutel, C. Chen, J. Deaton, J. Eisenstein, M. D. Hoffman *et al.*, "Underspecification presents challenges for credibility in modern machine learning," *arXiv preprint arXiv:2011.03395*, 2020.

[14] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 141–150. [Online]. Available: https://doi.org/10.1145/1985793.1985813

[15] B. Adams, S. Bellomo, C. Bird, T. Marshall-Keim, F. Khomh, and K. Moir, "The practice and future of release engineering: A roundtable with three release engineers," *IEEE Software*, vol. 32, no. 2, pp. 42–49, 2015.

[16] B. Adams and S. McIntosh, "Modern release engineering in a nutshell – why researchers should care," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5. IEEE, 2016, pp. 78–90.

[17] XebiaLabs. (2015) Periodic table of devops tools. [Online]. Available: https://xebialabs.com/periodic-table-of-devops-tools/

[18] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, no. null, p. 993–1022, Mar. 2003.

[19] M. Openja, "Release engineering posts," Aug. 2020, – A Large-Scale Study using StackOverflow –. [Online]. Available: https://doi.org/10.5281/zenodo.3980266

[20] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, 2020.

[21] M. Fink, "The eu artificial intelligence act and access to justice," *EU Law Live*, 2021.

[22] V. Christina, "What is the best programming language for machine learning?" https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7, 2017.

[23] Pytest, "Pytest, about fixtures," https://docs.pytest.org/en/latest/explanation/fixtures.html, 2021.

[24] a.-s. automl, "test metrics," https://github.com/automl/auto-sklearn/blob/master/test/test_metric/test_metrics.py, 2021.

[25] ——, "test automl," https://github.com/automl/auto-sklearn/blob/master/test/test_automl/test_automl.py, 2021.

[26] Googletest, "Googletest primer," http://google.github.io/googletest/primer.html, 2021.

[27] R. Gennadiy and E. Raffi, "Boost c++ libraries," https://www.boost.org/, 2020.

[28] cpputest, "Cpputest: Cpputest unit testing and mocking framework for c/c++," https://cpputest.github.io/, 2021.

[29] V. Mark, K. Mike, and W. Greg, "Unity unit testing for c (especially embedded software)," http://www.throwtheswitch.org/unity, 2015.

[30] C. Google, "Mlops: Continuous delivery and automation pipelines in machine learning," https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning, 2021.

[31] IBM, "The machine learning development and operations," https://ibm-cloud-architecture.github.io/refarch-data-ai-analytics/methodology/MLops/, 2020.

[32] C. Mike, "The forgotten layer of the test automation pyramid," https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid, 2009.

[33] B. C. Society, "Glossary of terms used in software testing (version 6.3)," http://www.testingstandards.co.uk/bs_7925-1_online.htm, 1998.

[34] S. P. Berczuk, S. Berczuk, and B. Appleton, *Software configuration management patterns: effective teamwork, practical integration*. Addison-Wesley Professional, 2003.

[35] E. Shihab, C. Bird, and T. Zimmermann, "The effect of branching strategies on software quality," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, 2012, pp. 301–310.

[36] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[37] D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *Journal of Systems and Software*, vol. 87, pp. 48–59, 2014.

[38] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski, "Evolution of the linux kernel variability model," in *International Conference on Software Product Lines*. Springer, 2010, pp. 136–150.

[39] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[40] L. Bass, I. Weber, and L. Zhu, *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[41] H. B. Braiek and F. Khomh, "On testing machine learning programs," *Journal of Systems and Software*, vol. 164, p. 110542, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220300248

[42] S. Masuda, K. Ono, T. Yasue, and N. Hosokawa, "A survey of software quality for machine learning applications," in *2018 IEEE International conference on software testing, verification and validation workshops (ICSTW)*. IEEE, 2018, pp. 279–284.

[43] A. Nikanjam, H. B. Braiek, M. M. Morovati, and F. Khomh, "Automatic fault detection for deep learning programs using graph transformations," *arXiv preprint arXiv:2105.08095*, 2021.

[44] S. Galhotra, Y. Brun, and A. Meliou, "Fairness testing: testing software for discrimination," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 498–510.

[45] H. K. Wright and D. E. Perry, "Release engineering practices and pitfalls," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 1281–1284.

[46] M. Castelluccio, L. An, and F. Khomh, "An empirical study of patch uplift in rapid release development pipelines," *Empirical Software Engineering*, vol. 24, no. 5, pp. 3008–3044, 2019. [Online]. Available: https://doi.org/10.1007/s10664-018-9665-y

[47] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou, "Understanding the impact of rapid releases on software quality - the case of firefox," *Empirical Software Engineering*, vol. 20, no. 2, pp. 336–373, 2015. [Online]. Available: https://doi.org/10.1007/s10664-014-9308-x

[48] T. Karvonen, W. Behutiye, M. Oivo, and P. Kuvaja, "Systematic literature review on the impacts of agile release engineering practices," *Information and software technology*, vol. 86, pp. 87–100, 2017.

[49] N. Kerzazi and B. Adams, "Who needs release and devops engineers, and why?" in *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, ser. CSED '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 77–83. [Online]. Available: https://doi.org/10.1145/2896941.2896957

[50] M. Bagherzadeh and R. Khatchadourian, "Going big: A large-scale study on what big data developers ask," in *In Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '19. City University of New York (CUNY): ACM, 2019. [Online]. Available: https://academicworks.cuny.edu/hc_pubs/524/

[51] A. Barua, S. Thomas, and A. Hassan, "What are developers talking about? an analysis of topics and trends in stack overflow," in *Empir Software Eng 19*. "-": Springer Link, 2014, p. 619–654. [Online]. Available: https://doi.org/10.1007/s10664-012-9231-y

[52] C. Rosen and E. Shihab, "What are mobile developers asking about? a large scale study using stack overflow," *Empirical Software Engineering*, pp. 1–32, 01 2015.

[53] N. V. Ivankova, J. W. Creswell, and S. L. Stick, "Using mixed-methods sequential explanatory design: From theory to practice," *Field methods*, vol. 18, no. 1, pp. 3–20, 2006.

[54] I. GitHub, "The github search api lets you to search for the specific item efficiently." https://docs.github.com/en/rest/reference/search, 2021.

[55] (2021) Github rest api. [Online]. Available: https://developer.github.com/v3/

[56] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[57] J. Businge, M. Openja, S. Nadi, E. Bainomugisha, and T. Berger, "Clone-based variability management in the android ecosystem," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 625–634.

[58] J. Businge, M. Openja, D. Kavaler, E. Bainomugisha, F. Khomh, and V. Filkov, "Studying android app popularity by cross-linking github and google play store," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 287–297.

[59] StackExchange, ""github stars" is a very useful metric. but for *what*?" https://opensource.stackexchange.com/questions/5110/github-stars-is-a-very-useful-metric-but-for-what, 2017.

[60] T. T. Le, W. Fu, and J. H. Moore, "Scaling tree-based automated machine learning to biomedical big data with a feature set selector," *Bioinformatics*, vol. 36, no. 1, pp. 250–256, 2020.

[61] R. S. Olson, R. J. Urbanowicz, P. C. Andrews, N. A. Lavender, L. C. Kidd, and J. H. Moore, *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part I.* Springer International Publishing, 2016, ch. Automating Biomedical Data Science Through Tree-Based Pipeline Optimization, pp. 123–137. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-31204-0_9

[62] R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore, "Evaluation of a tree-based pipeline optimization tool for automating data science," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16. New York, NY, USA: ACM, 2016, pp. 485–492. [Online]. Available: http://doi.acm.org/10.1145/2908812.2908918

[63] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2019, pp. 1946–1956.

[64] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: An exploratory

study," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '19. USA: IBM Corp., 2019, p. 193–202.

[65] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of production amp; test code," in *2008 1st International Conference on Software Testing, Verification, and Validation*, 2008, pp. 220–229.

[66] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on software engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[67] J. Lawrence, S. Clarke, M. Burnett, and G. Rothermel, "How well do professional developers test with code coverage visualizations? an empirical study," in *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE, 2005, pp. 53–60.

[68] NumPy, "Numpy: The fundamental package for scientific computing with python," https://numpy.org/, 2021.

[69] "Tosem-2021-replication," https://github.com/openjamoses/TOSEM-2021-Replication, 2021.

[70] N. Gv, "Memory errors in c++," https://www.cprogramming.com/tutorial/memory_debugging_parallel_inspector.html, 2019.

[71] A. Causevic, R. Shukla, S. Punnekkat, and D. Sundmark, "Effects of negative testing on tdd: An industrial experiment," in *Agile Processes in Software Engineering and Extreme Programming*, H. Baumeister and B. Weber, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 91–105.

[72] K. Ilgun, R. Kemmerer, and P. Porras, "State transition analysis: a rule-based intrusion detection approach," *IEEE Transactions on Software Engineering*, vol. 21, no. 3, pp. 181–199, 1995.

[73] W. H. Harrison, "Compiler analysis of the value ranges for variables," *IEEE Trans. Softw. Eng.*, vol. 3, no. 3, p. 243–250, May 1977. [Online]. Available: https://doi.org/10.1109/TSE.1977.231133

[74] R. E. Moore, *Interval analysis*. Prentice-Hall Englewood Cliffs, 1966, vol. 4.

[75] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA

2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 78–88. [Online]. Available: https://doi.org/10.1145/2338965.2336763

[76] W. Sebastian, "Membership testing," https://switowski.com/blog/membership-testing, 2021.

[77] ICS-33, "Complexity of python operations," https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt, 2021.

[78] B. W. Boehm, "Verifying and validating software requirements and design specifications," *IEEE software*, vol. 1, no. 1, p. 75, 1984.

[79] M. Glinz, "Improving the quality of requirements with scenarios," in *Proceedings of the second world congress on software quality*, vol. 9, 2000, pp. 55–60.

[80] Y. Belinkov and Y. Bisk, "Synthetic and natural noise both break neural machine translation," *arXiv preprint arXiv:1711.02173*, 2017.

[81] V. Prabhakaran, B. Hutchinson, and M. Mitchell, "Perturbation sensitivity analysis to detect unintended model biases," *arXiv preprint arXiv:1910.04210*, 2019.

[82] M. Srivastava, B. Nushi, E. Kamar, S. Shah, and E. Horvitz, "An empirical analysis of backward compatibility in machine learning systems," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3272–3280. [Online]. Available: https://doi.org/10.1145/3394486.3403379

[83] G. Bansal, B. Nushi, E. Kamar, D. S. Weld, W. S. Lasecki, and E. Horvitz, "Updates in human-ai teams: Understanding and addressing the performance/compatibility tradeoff," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 2429–2437.

[84] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance estimation for neural network pruning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 264–11 272.

[85] V. Tjeng, K. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming," *arXiv preprint arXiv:1711.07356*, 2017.

[86] G. P. Hammer, J.-B. d. Prel, and M. Blettner, "Avoiding Bias in Observational Studies," *Dtsch Arztebl International*, vol. 106, no. 41, pp. 664–668, 2009. [Online]. Available: https://www.aerzteblatt.de/int/article.asp?id=66288

[87] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 2736–2744.

[88] C. Batini, C. Cappiello, C. Francalanci, and A. Maurino, "Methodologies for data quality assessment and improvement," *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–52, 2009.

[89] L. Liu and L. Chi, "Evolutional data quality: A theory-specific view." in *ICIQ*, 2002, pp. 292–304.

[90] M. W. Bovee, "Information quality: A conceptual framework and empirical validation," Ph.D. dissertation, University of Kansas, 2004.

[91] M. Scannapieco and T. Catarci, "Data quality under a computer science perspective," *Archivi & Computer*, vol. 2, pp. 1–15, 2002.

[92] F. Pesarin and L. Salmaso, "The permutation testing approach: a review," *Statistica*, vol. 70, no. 4, pp. 481–509, 2010.

[93] W. J. Vetter, "Matrix calculus operations and taylor expansions," *SIAM review*, vol. 15, no. 2, pp. 352–369, 1973.

[94] J. Petit, B. Stottelaar, M. Feiri, and F. Kargl, "Remote attacks on automated vehicles sensors: Experiments on camera and lidar," *Black Hat Europe*, vol. 11, no. 2015, p. 995, 2015.

[95] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, "The computational limits of deep learning," *arXiv preprint arXiv:2007.05558*, 2020.

[96] R. Nishant, M. Kennedy, and J. Corbett, "Artificial intelligence for sustainability: Challenges, opportunities, and a research agenda," *International Journal of Information Management*, vol. 53, p. 102104, 2020.

[97] A. Galakatos, A. Crotty, and T. Kraska, *Distributed Machine Learning.* New York, NY: Springer New York, 2018, pp. 1196–1201. [Online]. Available: https://doi.org/10.1007/978-1-4614-8265-9_80647

[98] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *arXiv preprint arXiv:2102.00554*, 2021.

[99] E. Roche and Y. Schabes, *Finite-state language processing*. MIT press, 1997.

[100] Isha, A. Sharma, and M. Revathi, "Automated api testing," in *2018 3rd International Conference on Inventive Computation Technologies (ICICT)*, 2018, pp. 788–791.

[101] P. C. Jorgensen, *Software testing: a craftsman's approach*. Auerbach Publications, 2013.

[102] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 2, pp. 184–208, 2001.

[103] L. Baresi and M. Pezze, "An introduction to software testing," *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 1, pp. 89–111, 2006.

[104] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ml test score: A rubric for ml production readiness and technical debt reduction," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 1123–1132.

[105] S. Shankland. (2010) Google ethos speeds up chrome release cycle. [Online]. Available: https://www.cnet.com/news/google-ethos-speeds-up-chrome-release-cycle/

[106] ——. (2011) Rapid-release firefox meets corporate backlash. [Online]. Available: http://cnet.co/ktBsUU

[107] S. Baltes, L. Dumani, C. Treude, and S. Diehl, "Sotorrent: Reconstructing and analyzing the evolution of stack overflow posts," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 319–330. [Online]. Available: https://doi.org/10.1145/3196398.3196430

[108] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web? (nier track)," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 804–807. [Online]. Available: https://doi.org/10.1145/1985793.1985907

[109] S. Ahmed and M. Bagherzadeh, "What do concurrency developers ask about? a large-scale study using stack overflow," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser.

ESEM '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3239235.3239524

[110] X.-L. Yang, D. Lo, X. Xia, Z. Wan, and J.-L. Sun, "What security questions do developers ask? a large-scale study of stack overflow posts," *Journal of Computer Science and Technology*, vol. 31, pp. 910–924, 09 2016.

[111] H. Joshi, J. Pareek, R. Patel, and K. Chauhan, "To stop or not to stop — experiments on stopword elimination for information retrieval of gujarati text documents," in *2012 Nirma University International Conference on Engineering (NUiCONE)*, 2012, pp. 1–4.

[112] K. Sparck Jones and P. Willett, Eds., *Readings in Information Retrieval*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[113] A. K. McCallum, *A Machine Learning for Language Toolkit*, 2002. [Online]. Available: http://mallet.cs.umass.edu

[114] S. Geman and D. Geman, "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, no. 6, pp. 721–741, 1984.

[115] C.-M. Tan, Y.-F. Wang, and C.-D. Lee, "The use of bigrams to enhance text categorization," *Inf Process Manag*, vol. 38, no. 4, pp. 529–546, 2002.

[116] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 522–531.

[117] K. Bajaj, K. Pattabiraman, and A. Mesbah, "Mining questions asked by web developers," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 112–121. [Online]. Available: https://doi.org/10.1145/2597073.2597083

[118] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: Why do java developers struggle with cryptography apis?" in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 935–946. [Online]. Available: https://doi.org/10.1145/2884781.2884790

[119] G. Pinto, W. Torres, and F. Castor, "A study on the most popular questions about concurrent programming," in *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 39–46. [Online]. Available: https://doi.org/10.1145/2846680.2846687

[120] C. Treude, O. Barzilay, and M. Storey, "How do programmers ask and answer questions on the web?: Nier track," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 804–807.

[121] S. Mcintosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan, "A large-scale empirical study of the relationship between build technology and build maintenance," *Empirical Softw. Engg.*, vol. 20, no. 6, p. 1587–1633, Dec. 2015. [Online]. Available: https://doi.org/10.1007/s10664-014-9324-x

[122] F. Hassan, "Tackling build failures in continuous integration," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2019, pp. 1242–1245.

[123] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 41–50.

[124] RedHat Ansible. (2020) Ansiblefest 2020 virtual experience. [Online]. Available: https://www.ansible.com/

[125] Native, "Web or hybrid mobile-app development. white paper," in *IBM Corporation. Document Number: WSW14182USEN*, no. WSW14182USEN, April 2012.

[126] I. Malavolta, "Beyond native apps: Web technologies to the rescue! (keynote)," in *Proceedings of the 1st International Workshop on Mobile Development*, ser. Mobile! 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–2. [Online]. Available: https://doi.org/10.1145/3001854.3001863

[127] J. Businge, M. Openja, S. Nadi, E. Bainomugisha, and T. Berger, "Clone-based variability management in the android ecosystem," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 625–634.

APPENDIX A ASSERTION API AND EXAMPLE CODE REPRESENTING THE TEST STRATEGIES

Table A.1 API code example for expressing *Value Range analysis* and *Decision & Logical Condition Test*

Strategy	**Test scenarios and Assertion**	**Framework**
Value Range Analysis	Interval Analysis (i.e., $[a, b] = \{x \in \mathbb{R} : a \leqslant x \leqslant b\}$)	***
	assertTrue(0 ≤ a and b ≤ n)	unnittest
	assertTrue(0 ≤ b4 ≤ n)	unittest
	assertion inside loop statement	***
	value-range datatype assertion eg assert*list*, assert*set*, assert*dict*, assert*turple*	unittest
Decision & Logical Condition	Control flows (statement, branch or path)	***
	Conditional assertions eg EXPECT_TRUE(a exp b && x exp y)	***
	Check IsValidRowCol e.g row ≥ 0 && row < rows && col ≥ 0 && col < cols	deepspeech

Table A.2 Assertion API example code expressing the selected testing strategies

(a) The API code example for expressing Negative Test, Instance Checks and Sub component test strategies

Strategy	Assertion API	Framework
Negative Test	assertNot**	python
	assertFalse(statement), assert not (statement)	unittest, pytest
	assert_not_equal(statement)	numpy
	assertTrue(a != b), assert a != b	unittest, pytest
	ASSERT(a != b)	C/C++
	CHECK(!statement), CHECK(a != b)	gLog
	RAY_CHECK(a != b)	Ray
	ACHECK(!statement)	C/C++
	**_NE(a, b)	gTest
	**_FALSE(statement)	gTest
Instance and Type Checks	assertIsInstance(a,b)	python
	**equal(a.astype(datatype), typeid(a))	numpy
	**equal(a.dtype, typeid(a))	unittest
	**equal(type(a), typeid(a))	unittest
	assert a.dstype == typeid(a)	pytest
	assert **_type(a) == typeid(a)	nupic, apollo
	_EQ(a::Instance(), instance(a))	apollo
	_EQ(a::Type(), typeid(a))	gTest
	EXPECT_**(→type, typeid(a))	apollo
	CHECK(a**::Type() == typeid(a))	gLog
Membership Testing	assertIn(a, A)	unittest
	assert a in A	pytest
	CHECK(Subset(A, B));	gLog
	EXPECT_TRUE(A.has**(a));	apollo
	**TRUE(a, indexof(a));	gtest, unittest
	assert a == indexof(a));	pytest
	Checking Sub Component using loop statement (e.g., in a List, Set, dictionary)	*

(b) The API code example for expressing Oracle approximation

Strategy		Assertion API	Framework
Oracle Approximation	Absolute Relative Tolerance/ Absolute Tolerance	assert_allclose(result, expect, rVal,aVal)	numpy
		assert allclose(result, expect, rVal,aVal)	numpy
		assertTrue(isclose(result, expect, rVal,aVal))	numpy
		assert isclose(result, expect, rVal,aVal)	numpy
		assert torch.all(torch.isclose(result, expect, rVal))	torch
		assert torch.allclose(result, expect, rVal)	torch
		assert math.isclose(result, expect, aVal)	python
		assertAllClose(result, expect, rVal,aVal)	numpy
		assertAllCloseAccordingToType(re expect, aVal)	numpy
		SLOPPY_CHECK_CLOSE(resul expect, rVal,aVal)	deepspeech
		BOOST_CHECK_CLOSE((resul expect, rVal,aVal)	Boost
		EXPECT_NEAR(result, expect, aVal)	gTest
		ASSERT_NEAR(result, expect, aVal)	gTest
	Rounding Tolerance	assert_almost_equal(result, expect, dp)	numpy
		assertAlmostEquals(result, expect, dp)	unittest
		assertAlmostEqual(result, expect, dp)	unittest
		assertListAlmostEqual(result, expect)	unittest
		assert_array_almost_equal(result expect, dp)	numpy
		ApproxEqual(result, expect, dp)	deepspeech
		CHECK(ApproxEqual(result, expect, dp))	deepspeech
		EXPECT_TRUE(almost_equal(r expect, dp))	gTest
	Error Bounding	assertTrue(x < y)	numpy
		assertLess(x,y)	numpy
		assertLessEqual(x,y)	numpy
		assert_array_less(x,y)	numpy
		EXPECT_LT(x,y), ASSERT_LT(x,y)	gTest
		EXPECT_TRUE(EXPECT_LT(ASSERT_TRUE(ASSERT_LT(x,y))	gTest