ADVANCING SOFTWARE TESTING: MUTATION TESTING IN ACTOR CONCURRENCY AND EMPIRICAL INSIGHTS INTO MACHINE LEARNING TEST PRACTICES

by

MOHSEN MORADI MOGHADAM

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND INFORMATICS

2026

Oakland University
Rochester, Michigan

Doctoral Advisory Committee:

Mehdi Bagherzadeh, Ph.D., Chair
Hua Ming, Ph.D.
Amartya Sen, Ph.D.
Sam Srauy, Ph.D.

To the very few unsung people, whose selfless efforts to make this world a better place have ignited in me the hope that care and compassion still endure.

ACKNOWLEDGMENTS

I wish to express my deepest gratitude to my doctoral advisor, Mehdi Bagherzadeh, Ph.D., for his invaluable guidance, mentorship, and support throughout the course of my doctoral studies. His scholarly insight and encouragement have been instrumental to the successful completion of this work.

I am also indebted to my committee members, Hua Ming, Ph.D., Amartya Sen, Ph.D., and Sam Srauy, Ph.D., for their constructive feedback and contributions, which have significantly strengthened the quality of this dissertation.

I would like to acknowledge my internship mentors at DTE, whose guidance and expertise provided valuable professional experience and contributed meaningfully to my research and skill development.

Finally, I remain profoundly grateful to my family and friends for their steadfast support and encouragement during this journey. Their confidence in me has been a source of enduring motivation.

Mohsen Moradi Moghadam

ABSTRACT

ADVANCING SOFTWARE TESTING: MUTATION TESTING IN ACTOR CONCURRENCY AND EMPIRICAL INSIGHTS INTO MACHINE LEARNING TEST PRACTICES

by

Mohsen Moradi Moghadam

Adviser: Mehdi Bagherzadeh, Ph.D.

Software testing, especially in large-scale projects, is becoming increasingly important in real-world and mission-critical software. This requires the tests to be highly effective at finding bugs that occur in the real world. However, developers find designing such effective tests challenging. My research aims to tackle this challenge by advancing software testing in two key domains: actor concurrency and machine learning that has been widely used across web, mobile, and desktop applications and adopted at scale by companies like Google and Facebook.

This dissertation presents ***μAkka***, a framework for mutation testing of Akka actor concurrency using real actor bugs. The research analyzes 186 real Akka bugs, designs 32 mutation operators, and implements them in an Eclipse plugin. ***μAkka*** generates 11.7k mutants of 10 GitHub applications and runs 7.9k tests. The evaluation compares results to PIT with 26.2k mutants. ***μAkka*** mutants are higher quality, cover more bugs, and tests are less effective in detecting them.

Additionally, this dissertation includes a large-scale study on a set of real-world ML test cases in GitHub to understand their topics, popularities, difficulties, correlations, and historical changes. The study curates a set of 2,525 ML projects and their 136,463 test cases from GitHub; uses topic modeling to group these test cases into ML test topics; groups

similar topics into an ML test topic hierarchy; discusses these ML topics using sample test cases; analyze the popularity and difficulty of these topics; and study the correlation and historical trends over the past 10 years.

Together, these efforts address two critical domains: actor concurrency and machine learning. In actor concurrency, this work introduce a mutation testing framework designed to measure and strengthen test suites. In machine learning, we provide empirical insights into testing practices and gaps through a large-scale analysis. Both contributions advance understanding of how software is tested in areas of growing importance.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iv

ABSTRACT v

LIST OF TABLES xi

LIST OF FIGURES xii

CHAPTER ONE
INTRODUCTION 1

1.1. Motivations 1

1.2. Problem Statement 1

1.3. Proposed Solutions 3

1.3.1. *μAkka*: A Mutation Testing Framework for Actor Concurrency 3

1.3.2. Large-Scale Analysis of ML Test Cases 4

1.4. Contributions 5

CHAPTER TWO
BACKGROUND 8

2.1. Actor Concurrency 8

2.2. Mutation Testing 9

2.3. Types of Mutants 9

2.4. Quality of Mutants 10

2.5. Effectiveness, Coverage, and Bug Simulation 11

CHAPTER THREE
Related Work 12

3.1. Mutation Testing for Actor Concurrency 12

TABLE OF CONTENTS—Continued

3.2. Machine Learning Bug Datasets and Taxonomies 13

CHAPTER FOUR
Methodology 16

4.1. Methodology (Mutation Testing) 16

4.1.1. Akka Actor Bugs 16

4.1.2. Mutation Operators 16

4.1.3. Evaluation 18

4.2. Methodology (ML Study) 19

CHAPTER FIVE
Mutation Operators and a framework for mutation testing of Akka - muAKKA 26

5.1. Mutation Operators and Groups 26

5.1.1. PATH 30

5.1.2. CONFIGURATION 31

5.1.3. COMMUNICATION TYPE 32

5.1.4. LIFE CYCLE 33

5.1.5. EXCEPTION 34

5.1.6. INTERACTION 35

5.1.7. RACE 36

5.1.8. CLUSTER 37

5.2. A framework for mutation testing of Akka - muAKKA 38

5.3. Akka Applications 38

TABLE OF CONTENTS—Continued

CHAPTER SIX
MUTATION TESTING RESULTS FOR ACTOR CONCURRENCY 40

6.1. *μAkka*'s Mutant Generation and Coverage 40

6.2. *μAkka*'s Quality of Mutants 44

6.3. *μAkka*'s Test Effectiveness 46

6.4. PIT's Mutant Quality and Test Effectiveness 46

6.5. Bug Coverage by PIT and Jagannath et al.'s 48

6.6. Implications 50

6.7. Threats to Validity 51

CHAPTER SEVEN
ML TEST TOPICS: RESULTS AND ANALYSIS 53

7.1. RQ1: ML Test Topics 53

7.2. RQ2: ML Test Topic Hierarchy 57

7.2.1. General. 58

7.2.2. Basics. 60

7.2.3. Data Preparation. 62

7.2.4. Architecture. 63

7.2.5. Performance. 65

7.2.6. Reinforcement Learning. 69

7.2.7. Natural Language Processing. 69

7.3. RQ3: ML Test Topic Popularity 69

7.4. RQ4: ML Test Topic Difficulty 72

TABLE OF CONTENTS—Continued

7.5. RQ5: ML Test Topic Popularity and Difficulty Correlation 73

7.6. RQ6: ML Test Topic Historical Popularity and Difficulty 74

7.6.1. Historical Popularity. 74

7.6.2. Historical Difficulty. 76

7.7. Implications 77

7.8. Threats to Validity 79

CHAPTER EIGHT
CONCLUSION 82

REFERENCES 83

LIST OF TABLES

Table 5.1	Mutation operators for actor concurrency, their groups, and descriptions.	26
Table 5.2	Real-world Akka applications from GitHub.	38
Table 6.1	Mutant generation, ***Gen***, and coverage, ***Cov***, for mutation operator groups: Path, Config, Comm, and Life.	40
Table 6.2	Mutant generation, ***Gen***, and coverage, ***Cov***, for mutation operator groups: Exception, Interaction, Race, and Cluster.	41
Table 6.3	Overall mutant generation (Gen) and coverage (Cov) across all mutation operator groups.	41
Table 6.4	Mutant ease-of-killing, duplicity, and subsumption (Part 1).	43
Table 6.5	Mutant ease-of-killing, duplicity, and subsumption (Part 2).	43
Table 6.6	Mutant ease-of-killing, duplicity, and subsumption (Part 3).	44
Table 6.7	Mutant quality and test effectiveness in PIT.	47
Table 7.1	Topic names and top 15 stemmed words of our ML test topics (Part 1).	54
Table 7.2	Topic names and top 15 stemmed words of our ML test topics (Part 2).	55
Table 7.3	Popularity of ML test topics.	70
Table 7.4	Difficulty of ML test topics.	71
Table 7.5	Correlations of ML test topics popularity and difficulty	73
Table 7.6	ML test topics with zero test cases	75
Table 7.7	Validation of topic labeling using GPT-5.	81

LIST OF FIGURES

Figure 4.1	An overview of the methodology of our study.	20
Figure 6.1	Bug coverage by ***μAkka***, Jagannath et al.'s, and PIT.	49
Figure 7.1	A test in *Feature Transformer*topic from *lambda-packs* [106] project.	56
Figure 7.2	A test in *Graph Models*topic from *Pytorch Geometric* project [107].	56
Figure 7.3	A test in *Quantization*topic from *lambda-packs* [106] project.	57
Figure 7.4	ML test topics and their hierarchy.	59
Figure 7.5	Historical popularity of ML test topics.	74
Figure 7.6	Historical difficulty of ML test topics.	76
Figure 7.7	Tradeoff ML test topics by popularity/difficult.	78

CHAPTER ONE

INTRODUCTION

1.1 Motivations

Actor concurrency is becoming increasingly important in the real-world and mission-critical software. For example, PayPal uses actor concurrency to serve more than a billion financial transactions per day [143], Spark to shuffle hundreds of terabytes of big data [34], and Groupon to personalize coupons for 48 million customers in real-time [142]. NASA, Microsoft, Twitter, Verizon, CapitalOne, and Weight Watchers are among many other users of actor concurrency [141, 78]. This requires these applications to be free from actor bugs, that occur in the real world, and have tests that are effective in finding these bugs, which can cost large sums of money or even lives.

Alongside these developments, machine learning (ML) has been increasingly adopted in safety- and mission-critical software, in areas such as self-driving cars [79, 53], medical treatments [146], and aircraft collision avoidance [126], just to name a few. This requires ML code to be free from bugs and have test cases that are effective in finding these bugs. These bugs, if not found by test cases and not fixed by the developers, can lead to disasterous outcomes such as a Tesla crashing into a truck and killing its driver [136], multiple COVID-19 ML diagnosis algorithms failing in identifying the virus [179], and Zillow buying 2000 houses for more than selling price and losing 304 million dollars [161]. However, developers find ML testing challenging [197, 194, 157, 193] because of the statistical, blackbox, and autonomous nature of ML software, among others.

1.2 Problem Statement

Unlike multithreaded concurrency, in which lower-level threads communicate using shared memory and locks, in actor concurrency, higher-level actors communicate using

asynchronous messages [48, 47]. This makes not only actor concurrency but also its bugs [64, 115, 147] fundamentally different from multithreaded concurrency [80, 135]. Previous work [64, 115, 147, 178] discusses these differences.

Mutation testing [94, 16, 125] is a well-established and popular technique that transforms, the source or the binary, code of an application to induce its likely bugs and evaluate the effectiveness of its tests in finding these bugs. Mutation testing that simulates bugs in a software system to assess the effectiveness of its tests in finding the bugs is available for a broad spectrum of applications and their bugs, ranging from web [176] to mobile [145] to machine learning, [118] and is used at scale in companies like Google [172, 173] and Facebook [73]. However, *there still is no mutation testing for actor concurrency that uses real-world actor bugs*. The only relevant work is the work by Jagannath et al. [122] that proposes 12 mutation operators in ActorFoundry [63]. However, these operators are based on the individual experiences of the authors, and not a curated set of real bugs, are not implemented, are not evaluated, and are often specific to ActorFoundry's syntax and semantics and not applicable to today's industrial-strength implementations of actor concurrency. ActorFoundry is a now-defunct research prototype actor language with no industrial use. We discuss this work further in detail.

In parallel, to help developers with ML testing it is necessary to understand the ML topics that they write test cases for, the popularity of these topics, the difficulty of writing these test cases for these topics, and the historical changes of these topics. This understanding can not only help ML developers and their practice but also ML researchers and educators by allowing these communities to better decide when and where to focus their effors. Without such understanding, developers may not prepare themselves for similar ML test topics, researchers may make incorrect assumptions, and educators may teach the wrong topics.

1.3 Proposed Solutions

1.3.1 *μAkka*: A Mutation Testing Framework for Actor Concurrency

We propose ***μAkka***, a framework for mutation testing of Akka actor concurrency using real actor bugs. Akka [144], from LightBend, is an industrial-strength implementation of actor concurrency, among others such as Orleans [74], from Microsoft, and Erlang [62], from Ericsson. We focus on Akka because it is growing faster than others. For example, in the past five years, there are 3,727 Akka questions and answers [60] in Stack Overflow, which is 1.8 and 34.2x more than Erlang and Orleans, respectively. Similarly, there are 10,034 Akka applications [59] in GitHub, which is 1.5 and 7.3x more than Erlang and Orleans.

To design, implement, and evaluate ***μAkka***, we take the following major steps:

1. Manually analyze a recent set of 186 real Akka bugs [64] from Stack Overflow and GitHub to understand their causes.

2. Design a set of 32 mutation operators, with 138 source code changes in Akka API, to emulate these causes and induce bugs.

3. Implement these operators in an Eclipse plugin for Java Akka.

4. Use the plugin to generate 11,736 mutants of 10 real GitHub applications, with 446,444 lines of code and 7,871 tests.

5. Run these tests on these mutants to measure the quality of mutants and the effectiveness of tests in detecting and killing these mutants, using ease-of-killing, duplicity, subsumption, and mutation score metrics that previous work uses often.

6. Use PIT [88] to generate 26,177 mutants to compare ***μAkka*** and PIT mutant quality and test effectiveness.

7. Manually analyze the bug coverage and overlap of ***μAkka***, PIT and Jagannath et al.'s [122] actor operators.

8. Discuss the implications of our findings.

PIT is a popular [137] mutation testing tool with traditional logic operators such as Math that replaces mathematical operands $+, -, *, /, \%$.

While ***μAkka*** addresses mutation testing in the context of actor concurrency, modern software testing challenges extend beyond concurrency frameworks. In particular, understanding testing practices in domains such as machine learning is critical to improving test design and effectiveness at scale.

1.3.2 Large-Scale Analysis of ML Test Cases

Complementing this contribution, we also investigate testing challenges in the rapidly evolving field of machine learning (ML). Specifically, we leverage GitHub used by more than93% of developers, hosting over 420 million projects and repositories, and recording 4.5 billion contributions [90] which has become a valuable source for studying real-world software as well as the interests and behaviors of developers [98, 182, 200, 64]. In this project, we conduct a large-scale study on a set of real-world ML test cases in GitHub to understand their topics, difficulties, and historical changes, by answering the following questions:

- ***ML test topics*** What topics do ML developers write test cases for?
- ***ML test topic hierarchy*** Do ML test topics form a similarity hierarchy?
- ***ML test topic popularity*** How popular are ML test topics among developers?
- ***ML test topic difficulty*** How difficult are ML test topics for developers?

- ***ML test topic historical changes*** How do the popularity and difficulty of ML test topics change over time?

To answer these questions, we first develop a set of 2,525 ML projects and their 136,463 test cases from GitHub and extract the linguistic data [175] of their source codes [174, 86, 188, 189, 183, 55]. Previous work uses the source code linguistic data often for different software engineering tasks such as static prioritization of test cases [55, 188], localization of bugs [189], comprehension of crosscutting concerns that are scattered in the source code[183], and prediction of flaky tests [174], just to name a few. Second, we use topic modeling [77] and open card sort [99] to group the linguistic data of these test cases into ML test topics.

Third, we use the open card sort to group similar topics into categories and construct an ML test topic hierarchy. Fourth, we discuss each topic using three real-world sample test case names and comments. Fifth, we measure the popularity and difficulty of these topics using several well-known metrics from the previous work [18, 42, 47, 59, 74] and study their correlations. Sixth, we study the changes in the popularity and difficulty of these topics over the past 10 years. Finally, we discuss the implications of our findings for the practice, research, and education of ML software testing.

1.4 Contributions

Among others, we find that there is a substantial difference in the generation, coverage, and mutation quality and test effectiveness for different mutation operator groups in ***μAkka*** and between ***μAkka*** and PIT. In particular:

- ***Mutant quality*** ***μAkka*** mutants are high quality: 2x harder to kill, 3x less duplicate, and 1.3x less subsumed than PIT.

- ***Test effectiveness*** Tests are ineffective for ***μAkka*** mutants: 1.3x less effective in covering and 2.3x less effective in killing ***μAkka*** mutants than PIT.

- ***Bug coverage*** ***μAkka*** bug coverage is high: 3.3x more bug coverage than Jagannath et al.'s. And ***μAkka*** bug coverage benefits from the addition of PIT: ***μAkka*** + PIT cover 1.1x more bugs than ***μAkka*** alone and 9.5x more than PIT alone.

A few implications of our findings for ***μAkka*** are in predictive, selective, and sampling mutation testing and actor concurrency testing.

Also, A few findings of our ML study are:

- ***ML test topics*** Developers write ML test cases about a broad spectrum of 25 topics including Optimizers, Feedforward Neural Networks, Diffusion Models, and File Management.

- ***ML test topic hierarchy*** ML test cases that developers write can be grouped into a topic hierarchy with seven categories—Performance, Architecture, General, Basics, Data Preparation, Natural Language Processing, and Reinforcement Learning—with 1/3 of test cases being about Performance and 1/2 about either Performance or Architecture.

- ***ML test topic popularity*** Model Verification and Bounding Box are the most and least popular ML test topics with Model Verification being three times more popular.

- ***ML test topic difficulty*** Reinforcement Learning and Error Handling are the most and least difficult ML test topics, with Reinforcement Learning being four times more difficult.

- ***ML test topic popularity and difficulty correlation*** There is a statistically significant positive correlation between the popularity and difficulty of ML test topics.

- ***ML test topic historical changes*** ML test topics' popularity and difficulty has changed, sometimes dramatically, over the past decade, with some topics, such as Variable Reuse and Feature Transformer, going from respectively the most popular and difficult, to the least popular and difficult in two and one year(s), respectively.

CHAPTER TWO

BACKGROUND

2.1 Actor Concurrency

Basic actor concurrency Unlike multithreaded concurrency, in which a program is a set of threads that communicate using shared memory and locks, in basic actor concurrency [48, 47], the program is a set of *actors* that communicate by *sending*, *receiving*, and *processing* of asynchronous *messages*. An actor has its own thread of execution and behavior and makes its state accessible only through messages, to avoid sharing. To send a message, a *sender* actor sends a fire-and-forget message without *waiting* and *blocking* for its response. To receive the message, a receiver actor enqueues the message in its *mailbox*. To process the message, the receiver dequeues the message from its mailbox and executes it sequentially and to the end before processing the next message in the mailbox. During the processing, an actor can change state and behavior, send a message, or create a new actor.

Akka actor concurrency To allow for the development of real-world applications, Akka extends the basic actor concurrency with several necessary features, most of which are *programmatic* and *dynamic*. These features are *actor path* to locate a local or remote actor, *life cycle* to manage the actor life, *parental hierarchy and supervision* to manage the actor creation and failure, *configuration* to configure an application settings, *actor system* to provide the actor dispatch and scheduling, *dispatch* to assign threads to the actor message processings, *scheduling* to schedule the actor message sendings, *interaction patterns* to support more ways of actor interactions, *stashing* to buffer the actor messages for delayed processing, *deployment* to deploy remote actors, and *clustering* for distribution of actors

over network. These features are often the causes for Akka bugs [64, 115] and are discussed further throughout the paper. ActorFoundry [63] lacks most of these features.

2.2 Mutation Testing

Mutation testing [94, 16, 125] transforms, the source or binary, code of an application to induce its likely bugs and evaluate the effectiveness of its tests in finding these bugs. For each bug, a *mutation operator* slightly changes the application code to emulate the cause of the bug and induce the bug. The operator generates a *mutant* of the application which is the original application plus the induced bug. A test is *effective* if it can *detect* and *kill* a mutant by distinguishing its behavior from the original application behavior; otherwise the mutant stays a*live*. A test cannot kill a mutant that it does not *cover*. A test covers a mutant if its execution executes the mutated part of the mutant. Intuitively, the more mutants the tests cover and kill the more effective the tests are.

2.3 Types of Mutants

There are different types of mutants, including *stillborn*, *easy-to-kill*, *trivial*, *redundant*, *duplicate*, *subsuming*, *subsumed*, and *equivalent*. For a mutant m of an original application p and the set of tests $\mathscr{T}^m_{cover}$ and $\mathscr{T}^m_{kill}$ that, respectively, cover and kill m, m is stillborn if it includes syntactic and semantic errors that a compiler can catch and thus prevent its compilation. A mutant m is easy-to-kill if a large number k of tests in $\mathscr{T}^m_{cover}$ kill it. Following previous work [163], we set k to 97.5%. A mutant m is trivial if all the tests in $\mathscr{T}^m_{cover}$ can kill m. Triviality is a special case of ease-of-killing. A mutant m is redundant if tests that kill another mutant m' can also kill m. A redundant mutant is either duplicate or subsumed. A mutant m is the duplicate of m' if their behaviors are the functional equivalent

of each other, but not equal to p. Duplicate mutants are syntactically different from each other but semantically the same. A subsuming mutant m subsumes a subsumed mutant m' if $\mathscr{T}^{m}_{kill}$ is a subset of $\mathscr{T}^{m'}_{kill}$. m is equivalent if its behavior is the functional equivalent of p. An equivalent mutant is syntactically different from the original application but semantically the same.

Automatic determination of all duplicate and equivalent mutants is proven to be undecidable [82, 57]. Our definition approximates duplicity using the execution of tests [164]. Approximation of equivalence requires complex heuristics and analyses, such as weakest precondition [69], and is out of the scope of this work.

Our definitions follow similar definitions from previous work for stillborn [16, 125], easy-to-kill [163], trivial [130, 113], redundant [16, 160, 56, 130], duplicate [16, 165], subsuming [16, 56, 130], subsumed [16, 56, 130], and equivalent [16, 165, 56, 130, 125].

2.4 Quality of Mutants

The quality of mutants is critical to mutation testing since non-quality mutants could bias the testing. For example, previous work [164] shows that using subsumed mutants, which are non-quality, could, on average, bias the conclusions of 62% of arbitrary sets of mutation testings. A mutant m is *quality*, useful [130, 127], or valuable to mutation testing, if it is killable and its killing requires the addition of a new test to set of tests $\mathscr{T}^{m}_{cov}$ that cover it; otherwise it is *non-quality*. Trivial, easy-to-kill, redundant, and equivalent mutants are among the most well-known non-quality mutants [163, 130, 128, 16]. This is because, all and a large subset of tests in $\mathscr{T}^{m}_{cov}$ can kill a trivial and easy-to-kill m, respectively, with no need for a new test; a non-empty subset of $\mathscr{T}^{m}_{cov}$ that kills m' can also kill its redundant m with no need for new tests; and there is no test that can kill an equivalent m. Our definition

follows similar definitions from previous work for quality mutants
[130, 127, 163, 165, 164, 16].

2.5 Effectiveness, Coverage, and Bug Simulation

Effectiveness of tests Mutation testing evaluates the effectiveness of tests in finding bugs that it induces. *Mutation score* [125] is the most well-known metric to measure the effectiveness of tests. For the set of all tests $\mathscr{T}_{cov}$ that cover at least a mutant, the traditional mutation score is the ratio of the number of mutants that tests in $\mathscr{T}_{cov}$ kill to the number of all mutants. Our definitions follow similar definitions from previous work for the mutation score [125, 16, 160].

Mutant coverage Neither all tests cover all the mutants nor all mutants are covered by all the tests. *Mutant coverage* is the metric that measures the coverage of mutants. Our definition follows the standard definition of coverage [114].

Bug coverage A mutation operator covers a bug if it can emulate the cause of the bug and induce the bug. Our definition follows similar definitions from previous work for bug coverage [145].

CHAPTER THREE

Related Work

3.1 Mutation Testing for Actor Concurrency

Concurrency The work by Jagannath et al. [122] is the closest to our work. Jagannath et al. [122] propose 12 mutation operators, in 3 groups, for ActorFoundry [63], that are based on individual experiences of authors and not a curated set of real bugs, are not implemented, are not evaluated, and are specific to the syntax and semantics of ActorFoundry and not applicable to today's industrial-strength implementations of actor concurrency. Six out of our 32 mutation operators (18.8%) and the remaining twenty-six (81.2%) in PATH, CONFIGURATION, LIFE CYCLE, INTERACTION, RACE, and CLUSTER groups are new. There are six operators that are unique to Jagannath et al.'s. Two Remove Constraint (RC) and Modify Constraint (MC) are inapplicable to Akka because Akka does not allow conditional constraints to disable the receiving of messages. For the remaining four, Modify Message Parameter (MMP), Reorder Message Parameter (RMP), Modify Creation Parameter (MCP), and Reorder Creation Parameter (RCP), there were no bugs in $\mathscr{B}$ that required a similar operator.

Previous work proposes ConMan [80] and CCMutator [135] to mutate multithreaded concurrency and its constructs, such as threads, locks, conditional variables, and atomic blocks, in languages like Java and C/C++. However, the fundamental differences between actor and multithreaded concurrency, the syntax and semantics of their constructs, and their bugs make multithreaded mutation testing inapplicable to actor concurrency.

Languages and paradigms Previous work proposes mutation testing for different programming paradigms, such as functional [138], object- [148], aspect- [162], and

declarative-oriented [23] programming, and different programming models, such web [176], mobile [145], and machine learning [118]. Previous work also proposes mutation testing for specification [76], modeling [96] and a broad range of programming languages, such as Java [149], JavaScript [154], C [85], C++ [15], C# [95], and Ruby [139]. However, none of these works design, implement, and evaluate mutation testing for actor concurrency using real bugs.

Mutant quality Previous work propose several techniques, such as subsuming [56, 133], dominator [134], disjoint [132], minimal [56], and surface [112] mutants to reduce non-quality trivial, redundant, and equivalent mutants. Others, quantify the mutant usefulness using the triviality, equivalence and dominance of the mutant and relate the usefulness to the program context of the mutant [127]. However, none evaluate the quality of actor mutants.

Efficiency Previous work proposes Comutation [109] to select a small subset of multithreaded mutation operators with less number of mutants but the same test effectiveness and MutMut [108] for efficient execution of multithreaded mutants. However, selective and efficient mutation testing are outside the scope of this work.

3.2 Machine Learning Bug Datasets and Taxonomies

Machine Learning Bugs Huang et al. [34] manually analyze 3,814 Stack Overflow posts and 1,763 GitHub issues and construct a set of 446 deep learning dependency bugs to understand their symptoms, root causes, and fix pattern. Li et al. [140] manually analyze 1,479 bugs in three deep learning frameworks MXNet, PyTorch, and TensorFlow to understand their impacts on the development of multi-language deep learning frameworks.

Cao et al. [84] manually analyze 210 Stack Overflow posts to characterize the root causes, symptoms, and introduction stages of performance problems in TensorFlow and Keras deep learning systems. Kim et al. [131] analyze and categorize 4,577 bug reports from 193 deep learning projects to construct a bug benchmark. Humbatova et al. [117] manually analyze 1,059 GitHub commits and issues and Stack Overflow posts and use 20 structured interviews to construct a taxonomy of 375 TensorFlow, Keras, and PyTorch deep learning bugs. Islam et al. [121] manually analyze 2,716 Stack Overflow posts and 500 bug fixes from GitHub to understand types, root causes, and impacts and pipeline stages for 970 Caffe, Keras, TensorFlow, Theano, and Torch bugs. Zhang et al. [201] manually analyze 500 Stack Overflow questions and 82 GitHub commits and construct a set of 175 bugs to understand their root causes, symptoms, and challenges in bug localization for TensorFlow bugs.Jia et al. [202, 187] analyze Stack Overflow posts and GitHub messages, pull requests, and issues to construct a set of 202 bugs to understand their symptoms, root causes, and locations for TensorFlow bugs. Thung et al. [190] manually analyze 500 bugs and bug fixes in three machine learning systems Apache Mahout, Lucene, and OpenNLP and categorize their bugs and fixes.

Machine Learning Testing Zhang et al. [197] surveys 144 machine learning testing publications and classifies them into 4 testing categories workflow, properties, components, and applications and their topics. Testing workflow category includes seven topics, such as test input generation, oracle, and prioritization, with a paper sample like DeepXplore [170]. DeepXplore proposes a whitebox differential testing to generate test inputs that trigger inconsistencies among multiple deep neural networks with similar functionalities and identify erroneous behaviors. Testing properties category includes seven topics, such as correctness, robustness, and fairness, with a paper sample like DeepFool [155]. DeepFool computes and adds noise to fool deep neural networks to test and quantify their robustness.

Testing components category includes three topics data, learning program, and framework, with a paper sample like Data Linter [119] similar to code linters, to find data bugs in ML datasets. Testing applications category includes three topics such as autonomous driving and machine translation, and natural language processing, with a paper sample like [191]. DeepTest leverages domain-specific metamorphic relations to find deep neural network bugs in autonomous cars applications.

Topic Modeling of Machine Learning Alshangiti et al. [54] topic model 86,983 Stack Overflow questions into 30 topics to understand the challenges in developing ML software and study their popularity and difficulty. Gao et al. [103] topic model 92,830 Stack Overflow questions and 227,756 READMEs deep learning public repositories into 27 topics to understand the changes in software engineering needs of deep learning developers and study their popularity and difficulty using the number of questions in topics and the number of accepted answers for these questions. Bangash et al. [116] topic model 28,010 Stack Overflow posts to characterize ML topics that developers ask questions about and their popularity and difficulty.

CHAPTER FOUR

Methodology

4.1 Methodology (Mutation Testing)

In this section, we discuss our methodology to analyze Akka actor bugs, design and group ***μAkka*** mutation operators for these bugs, and measure the quality of mutants that ***μAkka*** and PIT operators generate, the effectiveness of tests to identify these mutants, and the coverage of bugs by ***μAkka*** and Jagannath et al.'s operators.

4.1.1 Akka Actor Bugs

For Akka bugs, we use a set $\mathscr{B}$ of 186 real bugs from a recent previous work [64]. $\mathscr{B}$ includes 130 bugs from Stack Overflow questions and answers and 56 bugs from GitHub applications. These bugs cover a broad spectrum of causes, ranging from API confusion to model confusion to missteps in the application logic, and a broad spectrum of symptoms, ranging from incorrect messaging to incorrect termination to unexpected application behavior.

4.1.2 Mutation Operators

Design For each bug in $\mathscr{B}$, we take the following steps to design the mutation operator that induces the bug. First, we manually analyze the bug to understand its cause. For example, there is a bug [19] in $\mathscr{B}$ in which an actor cannot be created with a cause that its name is not unique. Akka requires unique actor names. Second, we manually search Akka API and its documentation to identify methods that can emulate this cause by small changes in their syntax [94, 46, 125]. Several methods may emulate the same cause. For example, *actorOf(String name)* and *actorOf()* are two Akka methods that, their invocations, create an actor with a given and random name, respectively. Changing these invocations to

actorOf(String name') could emulate the cause in which the actor name is not unique, if *name'* is an actor name that exists already. We ensure that the syntactic changes of our operators do not violate the compile-time syntax and semantic requirements of Akka Java API. Third, we select a name for the mutation operator that describes the changes it makes. For example, we give the name CHANGE NAME to the operator that changes *actorOf(String name)* and *actorOf()* to *actorOf(String name')*. For a Stack Overflow bug, we analyze its questions, answers, and comments to understand its cause and design its operator. Similarly, for a GitHub bug, we analyze its commit, messages, original and modified code snippets, pull requests, and issues.

We use the open card sort [99] to identify API methods, their source code changes, and the name of a mutation operator. In the open card sort, there are no predefined API methods, source code changes, and operator names; instead they are developed during the sorting process. To sort, the first and second authors individually analyze the bugs and reiterate and refine until they agree. The second author is a Software Engineer and Programming Languages professor with extensive expertise in actor and multithreaded concurrent systems. The first author is a Ph.D. student with extensive coursework in concurrent and mobile systems. The third and fourth authors are Software Engineer professors with extensive expertise in concurrent and streaming systems, and testing and fault localization, respectively. First three authors have several years of extensive industrial work experience. In total, we design 32 mutation operators using source code changes in 138 Akka API. Table 5.1 shows these operators and their short descriptions. The formalization and API source code changes for these operators can be found in our replication package [58].

Grouping We use the same open card sort to group the mutation operators, based on the semantic relation between the bug causes they emulate. For example, CHANGE PATH groups together CHANGE NAME and CHANGE HIERARCHY operators that change the

related name and the hierarchy parts of the path of an actor. In Akka, the path of an actor denotes its physical location in a system and includes its name and hierarchy, among others. In total, we group our operators into 8 groups. Table 5.1 shows these groups.

Compile-time and logic bugs To avoid the generation of stillborn mutants, we do not design mutation operators for compile-time bugs. Generation and compilation attempts for stillborn mutants decrease the efficiency of mutation testing because they can neither be compiled nor executed. A bug is a compile-time bug if its cause is a syntactic or semantic issue that prevents its successful compilation. For example, there is a bug [18] in $\mathscr{B}$ in which the compilation fails because of using an undeclared variable. There are 21 (11.3%) compile-time bug in $\mathscr{B}$. Similarly, to avoid the redesigning of non-Akka and traditional operators, that already exist [16], we do not emulate logic bugs. A bug is a logic bug if its cause is a misstep in the application logic. For example, there is a bug [186] in $\mathscr{B}$ in which the lookup and message delivery for a remote actor fails because an skipped if conditional creates the wrong path for the actor. The traditional operator Negate Conditional in PIT [88] can emulate this bug by negating the condition of the conditional. There are 65 (35.0%) logic bugs in $\mathscr{B}$. In total, we design mutation operators to emulate the causes of 100 (53.8%) bugs in $\mathscr{B}$ that are neither compile-time nor logic bugs.

4.1.3 Evaluation

Quality of mutants We use three well-known metrics to measure the quality of mutants that ***μAkka*** and PIT operators generate. These metrics are ease-to-killing, duplicity, and subsumption, which section background defines. Intuitively, the lower the number of easy-to-kill, duplicate, and subsumed mutants the higher the quality. Previous work uses the number of ease-of-killing [163], duplicity [16, 165, 93], and subsumption [16, 56, 130] often to measure mutant quality.

Effectiveness of tests We use the well-known metric mutation score, that section background defines, to measure the effectiveness of our tests to identify ***μAkka*** and PIT mutants. Previous work uses mutation score [132, 166] often to measure the effectiveness of tests.

We execute the tests to measure the mutant quality and test effectiveness. However, a test can be flaky and produce different outcomes for different executions. To prevent flaky tests from skewing measurements [167], we follow previous work [72] and use the average of three executions of our tests to calculate our metrics.

Bug coverage We use the same open card sort to identify $\mathscr{B}$ bugs that Jagannath et al.'s [122] operators can induce and cover and their overlap with ***μAkka***.

4.2 Methodology (ML Study)

Figure 4.1 shows an overview of the methodology of our study.

Step 1: Consturct ML project dataset In this step, we construct a set of ML projects from GitHub that are written in Python. A project is a Python ML project if its implementation uses one of Python ML libraries in $\mathscr{M}$ = { *TensorFlow* [45], *Scikit-Learn* [169], *PyTorch* [89], *Keras* [87], *Theano* [52], *Caffe* [124] }. A project uses a library if its implementation invokes a method or references a variable of the library. Python is the most used programming langauge in open-source ML software [81]. According to Gonzalez et al. [110] "*the primary language for AI & ML is Python*". Libraries in $\mathscr{M}$ are the most used ML libraries in open-source Python ML projects [121]. In a recent study [97], 95% of 3,340 Python ML projects in GitHub use the libraries in $\mathscr{M}$.

To develop a dataset of mature and recent real-world projects, we focus on projects with three or more contributors, with 50 or more stars, that are developed between January

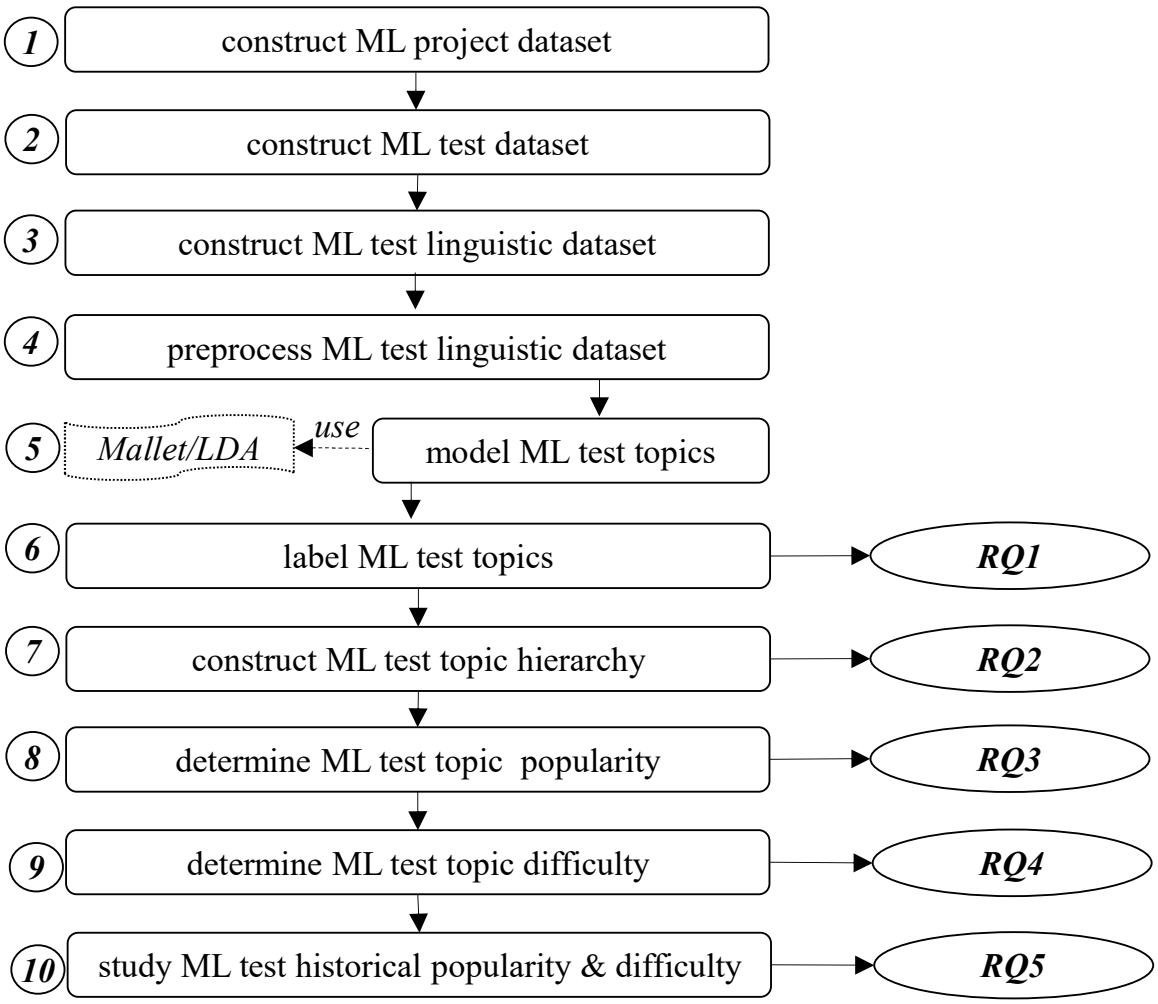

Figure 4.1: An overview of the methodology of our study.

1, 2010 and January 30, 2024 and updated in the past five years. A project is updated in the last five years if its last commit is in the last five years

We write code that uses GitHub API [61] to analyze 40,982 Python projects and extract the latest version of ML projects that satisfy our maturity and recency selection criteria. Our ML projects dataset includes 2,525 Python ML projects developed over 14 years.

Step 2: Consturct ML test dataset In this step, we construct a set $\mathscr{T}$ of test cases of our ML projects. Testing frameworks often assist developers in automated and structured implementation, execution, and management of test cases. To construct a real-world dataset, we focus on test cases that are written in *pytest* [100] and *unittest* [101] which are the two most widely used Python testing frameworks [181, 68]. In *pytest*'s and *unittest*'s test discovery processes [120, 101], a test file is a *.py* file with a name that starts or ends with *test*. A *pytest* test case is a function , in a test file, with a name that starts with *test* and can be either outside of a class or inside a class with a name that starts with *Test* [120]. Similarly, a *unittest* test case is a function with a name that starts with *test* and is inside a class that subclasses *TestCase* class [101]. We write code that recurses through folders and files of each of our ML projects and extracts its test cases according to the above test discovery processes. Our test case dataset $\mathscr{T}$ includes 136,463 test cases with 125.409 (91.9%) *pytest* test cases and 11.053 (8.1%) *unittest*.

Step 3: Extract ML test linguistic dataset In this step, we extract a set of linguistic data $\mathscr{L}$ of our ML test cases in $\mathscr{T}$. The linguistic data of a test case captures the intentions of its developers that are encoded in its comments, strings, and identifier names [175]. Previous work [174, 86, 188, 189, 183, 55] uses comments, strings and identifier names often as the linguistic data of the source code. We write code that uses the Python parser Parso [168] to extract the linguistic data of our ML test cases. Our test case linguistic dataset $\mathscr{L}$ includes 136,463 documents with linguistic data of our test cases. Each document includes the linguistic data of a test case.

Step 4: Preprocess ML test linguistic dataset In this step, we preprocess the linguistic data of our ML test cases in $\mathscr{L}$ to reduce the noise [49, 65, 70, 195] for the next steps of the analysis. First, we remove URLs, Python keywords [177], and non-english

words [65, 49]. Second, we split camel and snake case words into their composing words [174, 55]. A camel case word uses first capital letters to separate its composing words; similarly, a snake case word uses underlines. For example, we split camel and snake case words "*featureProbability*" and "*feature_probability*" into "*feature*" and "*probability*". Third, we remove numbers, stop words such as "*a*", "*the*", "*is*", punctuation marks, and non-alphabetical characters. We use the list of stop words in Natural Language Toolkit (NLTK) [26]. Finally, we use NLTK's Porter stemmer to reduce words into to their stemmed representations. For example, we stem "*feature*", "*features*", "*featurizer*", and "*featurizers*" words all to "*featur*".

Step 5: Model ML test topics In this step, we use MALLET [153] and its Latent Dirichlet Allocation (LDA) [77] topic modeling [184] to group the linguistic data of our ML test cases in $\mathscr{L}$ into topics. In our topic model, the linguistic data of an ML test case is a probabilistic distribution of several topics with their proportions. A topic is a set of frequently co-occurring words that approximates a real-world concept. MALLET groups tests into K topics after I grouping iterations and returns a set of topics and their proportions for each test case and a set of words for each topic. A test case has a dominant topic that covers the biggest proportion of the test case. To illustrate, the dominant topic for the test case *test_bipartite_batched_negative_sampling* in Figure 7.2 is *Graph Models*. Previous work [65, 49, 86, 188] uses topic modeling often to understand and topic model the linguistic data of the source code and Stack Overflow questions and answers.

We run 40 topic modeling experiments with a broad range of values for K = { 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 } and I = { 500, 1000, 1500, 2000 }. These values are inline with K and I values that previous work [65, 49] uses often.

Our experiments show that $K = 25$ and $I = 2000$ create sufficiently granular topics for the linguistic data of our ML test cases. Smaller K values often lead to an undesirable

combination of multiple unrelated topics into a single topic whereas larger K values lead to an undesirable split of a single topic into multiple similar topics. For example, with $K = 20$ MALLET combines unrelated topics *Feature Transformer*and *Graph Models*into a single topic. Similarly, with $K = 30$ MALLET splits a single topic *Quantization*into related and similar topics *Quantization Schema* and *Quantization Techniques*. MALLET uses hyperparameters α and β to control the distribution of words in topics and the distribution of topics in test cases. Following previous work [65, 49, 67, 70, 180, 195], we use the standard values $\alpha = 50/K$ and $\beta = 0.01$. Biggers et al. [75] show that "α and β have little influence on the accuracy of the LDA [topic modeling]".

Step 6: Label ML test topics In this step, we assign labels to topics that MALLET generates for our linguistic dataset $\mathscr{L}$. MALLET groups the linguistic data of test cases in $\mathscr{L}$ into topics and denotes each topic with a set of words. To illustrate, the word set W = { *net quantiz scale oper blob ref workspac core quant refer sim creat encod per astyp option schema relu param run* } represents a topic. However, MALLET cannot decide about the meaning of a topic and label the topic with a name that best explains its meaning. We use an open card sort [99] to label topics. In an open card sort, there is no predefined topic and topics are developed during the labeling process. To label a topic, we manually inspect the top 20 words in the set of words for the topic as well as 50 random test cases for that topic to decide about a label that best explains these words and tests. For example, we label the word set W with topic name *Quantization*. Previous work [49, 65, 67, 70, 158, 180, 195] uses open card sort, top topic words, and random topic documents, often, to decide the topic labels.

During the card sort process, the first and second author individually assign labels to topics and reiterate and refine until they agree on topic labels. The first author is a Ph.D. student with extensive expertise and a Master's degree in Machine Learning. The second

and third authors are Software Engineering professor with extensive expertise in the refactoring of machine learning software. The authors have several years of extensive industrial work experience.

Step 7: Construct ML test topic hierarchy In this step, we construct a hierarchy of ML test topics. We use a similar card sort process to group similar topics into lower- and higher-level categories of a hierarchy. Figure 7.4 shows the hierarchy of our ML test topics. To illustrate, according to Figure 7.4, topics Optimizers and Bbox Loss are grouped into the lower-level category Loss. Similarly, the lower-level categories Loss and Speed are grouped in to the higher-level category Performance.

Step 8: Determine ML test topic popularity In this step, we determine the popularity of ML test topics. We use the number of test cases that belong to a topic as the metric to measure the popularity of the topic. A test case belongs to a topic if the topic is the dominant topic of the test case. To illustrate, in Figure 7.2, *Graph Models*is the the dominant topic for the test case *test_bipartite_batched_negative_sampling*. Table 7.3 shows the popularity of our ML test topics in which *Model Verification*topic is twice as popular as *Variable Reuse*.

Step 9: Determine ML test topic difficulty In this step, we determine the difficulty of our ML test topics. We use two metrics cognitive complexity [152] and cyclomatic complexity [83] to measure the difficulty in terms of understandability and testability, respectively. A test case is more understandable if a developer takes less time to comprehend its source code. Similarly, the test case is more testable if there are less independent paths in its executions. The cognitive (cyclomatic) complexity of a topic is the average of the cognitive (cyclomatic) complexity of the source code of its test cases.

Table 7.4 shows the difficulty of our ML test topics in which *Reinforcement Learning*is twice as difficult as *Module Management*. Previous work [129, 171, 192] uses cognitive and cyclomatic complexities often to understand the difficulty of the source code. Baron et al. [71] empirically validate the relation between source code understandability and cognitive complexity.

Step 10: Study ML test topic popularity and difficulty correlation

In this step, we use Kendall correlation to identify if there is a correlation between the popularity and difficulty of our ML test topics. We use Kendall because it is less sensitive to outliers and therefore is relatively more stable. We investigate two correlations, between one popularity metric number of test cases in a topic and two difficulty metrics of cognitive and cyclomatic complexity of test cases in the topic.

Step 11: Study ML test topic historical popularity and difficulty

In this step, we analyze the historical trends and changes in the popularity and difficulty of our ML test topics over the past 10 years, from 2014 to 2023. We select 2014 for the starting year for studying these changes because for some topics such as, Graph Models and Convolutional Blocks, there are only a few and sometimes zero test cases before 2014. We do not consider 2024 because the year has not come to and end yet.

CHAPTER FIVE

Mutation Operators and a framework for mutation testing of Akka - muAKKA

5.1 Mutation Operators and Groups

In this section, we discuss and illustrate the mutation operators using real bug examples. In addition, we compare our operators with Jagannath et al.'s [122]. Table 5.1 shows our 32 mutation operators, their eight groups 8 groups PATH, CONFIGURATION, COMMUNICATION TYPE, LIFE CYCLE, EXCEPTION, INTERACTION, RACE, and CLUSTER, and their short descriptions.

Table 5.1: Mutation operators for actor concurrency, their groups, and descriptions.

Group	*Mutation Operator*	*Description*
PATH	name	change the name of an actor to an existing or random actor name
	protocol	change the local protocol of an actor to a remote protocol and vice versa
	actor system	change the actor system name of an actor to an existing or random actor system name
	host	change the host name or IP address of an actor to a random host name or IP address
	port	change the port number of an actor's host to a random port number

Continued on next page

Table 5.1 – continued from previous page

Group	*Mutation Operator*	*Description*
	hierarchy	change the ancestral hierarchy of an actor to a hierarchy with one less random ancestor
CONFIGURATION	application	change the application configuration to an empty configuration or drop it
	fallback	change the fallback configuration to an empty configuration or drop it
	mailbox	change the mailbox configuration of an actor to an empty or random configuration or drop it
	dispatch	change the dispatch configuration of an actor to an empty or random configuration or drop it
	routing	change the routing configuration of an actor to the default or no router configuration or drop it
	deployment	change the deployment configuration of an actor to the default or drop the configuration
COMMUNICATION TYPE	send type	change the set of message types that an actor sends to a set with one more random type

Continued on next page

Table 5.1 – continued from previous page

Group	*Mutation Operator*	*Description*
	receive type	change the set of message types that an actor receives to a set with one less random type
LIFE CYCLE	lookup	change the lookup of an actor to occur in another existing or random actor system
	starting	change the starting pre- and post-behaviors of an actor to empty or supertype behaviors
	stopping	change the stopping pre- and post-behaviors of an actor to empty or supertype behaviors
	restarting	change the restarting pre- and post-behaviors of an actor to empty or supertype behaviors
	termination	change the termination of an actor or actor system by removing or delaying the termination
	monitoring	change monitoring of an actor by removing the monitoring
EXCEPTION	failure	change the failure behavior of an actor to throw an exception
	supervision	change the supervisory behavior of an actor to another random behavior

Continued on next page

Table 5.1 – continued from previous page

Group	*Mutation Operator*	*Description*
INTERACTION	synchrony	change the synchrony of a message that an actor sends from sync to async and vice versa
	blocking	change the blocking behavior of an actor by increasing or decreasing its wait or sleep durations
	timeout	change the timeout behavior of an actor by increasing or decreasing its timeout durations
	forwarding	change the sender of a message that an actor forwards to a random actor reference
	sender	change the sender of a message that an actor sends to an existing actor reference
	receiver	change the receiver of a message that an actor sends to an existing actor reference
	scheduling	change the scheduling of a message that an actor sends by increasing or decreasing its delays
RACE	sharing	change the sharing of data between an actor and its concurrent future or async tasks
	ordering	change the ordering between two messages by delaying the first message

Continued on next page

Table 5.1 – continued from previous page

Group	*Mutation Operator*	*Description*
CLUSTER	subscription	change a cluster event that an actor subscribes for to another Akka cluster event
	unsubscription	change a cluster event that an actor unsubscribes from to another Akka cluster event

5.1.1 PATH

In Akka, an actor has a path that denotes its physical location in a system. The path includes an address and a *name*. The address can be either local or remote. An actor is local if it is in the same Java Virtual Machine (JVM) and is remote otherwise. A local address specifies the *actor system* that the actor resides in and a *hierarchy* of its ancestors, following *akka://actor system/hierarchy/name* format. Actors form a hierarchy in which a parent creates its children. A remote address, also specifies a *host*, *protocol*, and a *port* for remote connections, following *akka.protocol://actor system@host:port/hierarchy/name* format.

An Akka developer is responsible to understand actor paths and their different parts, and their programmatic and dynamic modifications, and manually provide correct values for these parts; otherwise incorrect actor paths can cause bugs, as the previous work [64] shows.

PATH group includes six mutation operators that CHANGE the NAME, PROTOCOL, ACTOR SYSTEM, HOST, PORT, and HIERARCHY parts of an actor path to induce the bugs that are caused by the use of incorrect values for these parts. For example, there are three bugs [14, 19, 1] in $\mathscr{B}$ in which an actor cannot be created because its name is not unique. Akka requires actor names to be unique in a system.

CHANGE NAME incudes these bugs by changing the name of an actor to an actor name that already exists in the system. Similarly, there is a bug [3] in which an actor cannot be looked up and messages sent to it cannot be delivered because its hierarchy is missing an ancestor. CHANGE HIERARCHY induces this bug by changing the hierarchy of an actor to a hierarchy with one less random ancestor. Table 5.1 describes all the mutation operators in PATH and the bug causes they emulate.

Jagannath et al. [122] use their individual experiences to propose 12 unimplemented mutation operators in 3 groups for the research actor language ActorFoundry. These operators, inside parentheses, and their groups, outside parentheses are: Messaging (Remove Send/Receive, Modify Message Parameters, Reorder Message Parameter, Modify Message Name, Modify Message Recipient, and Change Synchronization Type), Constraint (Remove Constraint and Modify Constraint), and Creation/Deletion (Remove Creation/Deletion, Modify Creation Parameter, and Reorder Creation Parameters).

All six mutation operators in PATH are new and cannot be found in the previous work by Jagannath et al.'s.

5.1.2 CONFIGURATION

A configuration defines the properties of an application using a set of parameter and value pair settings. In Akka, a default configuration [50] defines 255 default settings. These settings define the properties of actors and actor systems of an application and their *deployment*, *mailboxes*, *dispatchers*, and *routers*, among others. For example, the mailbox can be configured to be either unbounded or bounded with a specific capacity for the number of its messages. An *application* configuration can override the default settings, define new settings, such as a new dispatcher [13] with a different max and min number of threads in its thread pool, or *fallback* on another configuration for the settings that it does not define. An Akka developer is responsible to understand the default and application

configurations, their parameter and value settings, overriding, and fallback, and their programmatic modifications, and manually configure non-default settings; otherwise incorrect configurations can cause bugs [64].

CONFIGURATION includes six mutation operators that CHANGE APPLICATION, FALLBACK, MAILBOX, DISPATCH, ROUTING, and DEPLOYMENT settings to induce bugs that are caused by using incorrect settings for these configurations. For example, there is a bug [8] in $\mathscr{B}$ in which an actor system cannot be configured and created because its application configuration does not load. CHANGE APPLICATION induces this bug by replacing the application configuration with an empty configuration with no settings. Similarly, there is a bug [29] in which an actor cannot stash its messages because its mailbox is not configured properly. CHANGE MAILBOX induces this bug by removing the mailbox configuration of an actor.

All six mutation operators in CONFIGURATION are new and cannot be found in Jagannath et al.'s.

5.1.3 COMMUNICATION TYPE

In Classic Akka, unlike Akka Typed [51], an actor neither knows about the types of messages that it may receive nor the type of messages it can send. An Akka developer is responsible to manually discover these message types and ensure that the actor can receive all the messages that others send to it and sends only messages that others can receive; otherwise, sending or receiving messages with incorrect types can cause bugs [64].

COMMUNICATION TYPE includes two mutation operators CHANGE RECEIVE TYPE and SEND TYPE that change the type of messages that an actor can receive or send to induce the bugs that are cause by receiving or sending incorrect message types. For example, there are two bugs [21, 22] in $\mathscr{B}$ in which the application behavior is undesired [21] or terminates prematurely [21] because an actor receives a message of a type that it cannot process.

CHANGE RECEIVE TYPE induces these bugs by changing the set of message types that a receiver actor can receive to another set with one less random message type.

Similarly, CHANGE SEND TYPE can induce these bugs by changing the set of message types that a sender actor can send to another set with one more random message type.

Our CHANGE RECEIVE TYPE partially overlaps with Jagannath et al.'s Remove Send/Receive (RSR) operator, in their Messaging group, that "mimics the omission of messages by removing sends/receives". Similarly, our CHANGE SEND TYPE overlaps with their Messaging Modify Message Name (MMN) operator, in Messaging, that "modifies the name [type] of a message being sent".

5.1.4 LIFE CYCLE

In Akka, an actor and its enclosing actor system go through different stages in their life cycle, such as creation, *lookup*, *starting*, *stopping*, *restarting*, *termination*, and *monitoring*. An Akka developer is responsible to understand these life cycles and their programmatic stages and manually manage these stages and their pre- and post-behaviors correctly; otherwise incorrect management of life cycles can cause bugs [64, 115].

LIFE CYCLE includes six mutation operators that CHANGE LOOKUP, STARTING, STOPPING, RESTARTING, TERMINATION, and MONITORING of an actor and its actor system to induce the bugs that are caused by the incorrect managements of these stages. For example, there are two bugs [17, 24] in $\mathscr{B}$ in which the application does not shutdown [17] or consumes all its available memory [24] because its actor system and actors do not terminate. CHANGE TERMINATION induces these bugs by removing the termination of an actor and its enclosing actor system. Similarly, there is a bug [25] in which the termination of an actor goes unnoticed because the actor that is responsible to manage the termination is

not monitoring the terminating actor. CHANGE MONITORING induces this bug by removing the termination monitoring of an actor.

Our CHANGE TERMINATION partially overlaps with Jagannath et al.'s Remove Creation/Deletion (RCD) operator, from their Creation/Deletion group, that "mimics the omission of creation/deletion of an actor by removing an actor creation/deletion". All other five mutation operators in LIFE CYCLE are new.

5.1.5 EXCEPTION

In Akka, a parent actor is not only responsible to create but also supervise its child actors and manage their *failure* when they throw exceptions. The *supervision* strategy of the parent specifies to either stop, resume, or restart the failed child or escalate the exception up the supervision hierarchy to the parent of the parent. An Akka developer is responsible to understand actor failures and exceptions, programmatic supervisory hierarchies, and supervision strategies and handle these exceptions correctly; otherwise, incorrect handling of exceptions can cause bugs [64]. EXCEPTION includes two mutation operators that CHANGE FAILURE and SUPERVISION behavior of an actor to induce the bugs that are caused by incorrect handlings of exceptions. For example, there are two bugs [6, 11] in $\mathscr{B}$ in which the application does not terminate [6] or swallows an exception [11] because an actor throws an exception that its ancestor does not handle properly.

CHANGE FAILURE induces these bugs by changing the behavior of an actor to throw an exception. Similarly, there are two bugs [2, 28], in which an actor does not restart properly because its parent is using the wrong supervision strategy. CHANGE SUPERVISION incudes these bugs by changing the supervision strategy of an actor.

Both mutation operators in EXCEPTION are new.

5.1.6 INTERACTION

In Akka, there are several ways in which an actor can interact with another. In addition to an *asynchronous* fire-and-forget message, a *sender* actor can send a *synchronous* request-response message to a *receiver* and wait and *block* for its response in a future variable for a *timeout* period. A future is a placeholder for an incomplete task with a result that is not ready yet. Similarly, a receiver can *forward* a message it receives to another actor, without changing the sender of the message, or *schedule* to send a message in specific intervals. An Akka developer is responsible to understand these programmatic interactions and their semantics and use them correctly; otherwise incorrect interactions can cause bugs [64, 147].

INTERACTION includes seven mutation operators that CHANGE SYNCHRONY, BLOCKING, TIMEOUT, FORWARDING, SENDER, RECEIVER, and SCHEDULING of actor interactions to induce bugs that incorrect uses can cause. For example, there are two bugs [12, 9] in $\mathscr{B}$ in which a receiver actor cannot respond to its sender more than once because the temporary actor that a request-response message creates to receive the response, terminates after receiving the first response. CHANGE SYNCHRONY induces these bugs by changing an asynchronous fire-and-forget message to a synchronous request-reply message. Similarly, there is a bug [33] in which a receive cannot process its messages because the messages are sent too fast. CHANGE SCHEDULING induces this bug by decreasing the initial and interval delays between messages that a scheduler sends.

Our CHANGE SYNCHRONY overlaps with Jagannath et al.'s Change message Synchronization Type (CST), in Messaging, that "changes a synchronous send to an asynchronous send and vice versa". Similarly, our CHANGE RECEIVER overlaps with their Modify Message Recipient (MMR), in Messaging, that "modifies the recipient of a message". All other five operators in INTERACTION are new.

5.1.7 RACE

A race occurs if two concurrent computations access the same memory and one modifies the memory. In Akka, to avoid races, an actor processes its messages sequentially and one at a time. However, both lower-level data races and higher-level message races [66] are still possible. A data race occurs when a concurrent future or async task that runs outside the actor *shares* memory with the actor and either the actor or the task modifies the memory. Similarly, a message race occurs when two messages arrive at the same actor out of their desired *order* and the processing of either of the messages modifies the actor memory. An Akka developer is responsible to understand data sharings and message orderings among all actors and non-actor concurrent computations of a system and make sure they are free from races; otherwise races could cause bugs [64, 115, 147].

RACE includes two mutation operators that CHANGE SHARING of data and ORDERING of messages

to induce the bugs that incorrect sharing and message ordering can cause. synchronous For example, there are seven bugs [4, 10, 5, 31, 27, 32, 7] in $\mathscr{B}$ in which a response is not delivered to a sender actor [4, 10, 5, 31, 27] or the application does not behave as desired [32, 7] because an actor and its future task share the variable *sender* which the actor modifies. The variable *sender* is the sender of the current message that the actor is processing and changes when the actor starts to process another message. CHANGE SHARING induces this bug by changing a random actor reference in a future or async task with *getSender()*. The method *getSender* accesses, reads and returns the value of the shared variable *sender*. Similarly, there is a bug [20] in which the application produces incorrect results because two messages that initiate the warmup of a simulation and production of the results arrive out of order. CHANGE SHARING induces this bug by delaying the sending of a message to reorder the arrival of messages.

Our CHANGE SHARING overlaps with Jagannath et al.'s Change (message) Reference Type (CRT), in Messaging, that "changes a message sent by reference to a message sent by value and vice versa" to change the sharing between actors. CHANGE ORDERING is new.

5.1.8 CLUSTER

In Akka, a cluster is a group of actor systems that provide distribution, load balancing, and failover for their actors. An actor system is a logical node of the cluster. An actor in the cluster can *subscribe* for the membership, domain, and reachability events of the cluster, receive messages when these events occur and process these messages accordingly.

Similarly, the actor can *unsubscribe* from these events. An Akka developer is responsible to understand these events, their semantics, and their subscriptions and unsubscriptions and manage them correctly; otherwise incorrect subscriptions or unsubscriptons can cause bugs [64].

CLUSTER includes two mutation operators that CHANGE SUBSCRIPTION and UNSUBSCRIPTION

of actors in a cluster to induce bugs that are caused by incorrect subscriptions. For example, there is a bug [30] in $\mathscr{B}$ in which an actor misses a cluster leader change event because it subscribes for an incorrect member event instead of the correct domain event. The leader change event is a domain event. CHANGE SUBSCRIPTION induces this bug by changing the cluster event that an actor subscribes to another random cluster event. Similarly, CHANGE UNSUBSCRIPTION can induce this bug by changing the cluster event that an actor unsubscribes from.

Both mutation operators in CLUSTER are new.

5.2 A framework for mutation testing of Akka - muAKKA

For real-world applicability, we implement our mutation operators as an Eclipse plugin, in a framework that we call ***μAkka***. ***μAkka*** uses 138 source code changes of Java Akka API [123] to implement our 32 operators. For efficiency, testing in in addition to not generating stillborn mutants, we integrate the following popular techniques, from previous work, into ***μAkka***. First, we use conditional mutation [128] that uses conditional statements to integrate all the mutants and the original application into a single application.

This allows a single compilation to compile all mutants all at once and efficiently instead of one by one and inefficiently. Second, we generate and use coverage information to execute mutants only if they are covered by tests [92]. This allows to not execute the tests that do not cover mutants and mutants that are not covered by tests.

5.3 Akka Applications

Id	***Project***	***LOC***	***Domain***	***Stars***
ditto	ditto [35]	261,951	internet of things (IoT)	390
lms	sunbird-lms [36]	78,633	learning management	29
wot	wot-servient [37]	52,019	web of things	23
flower	flower [38]	21,648	reactive microservices	518
comb	servicecomb [39]	15,527	transactional data	481
rhino	rhino [40]	13,882	web performance testing	16
parc	parallec [41]	13,209	asyncronous web	800
flink	flink-rpc [42]	8,326	web performance testing	19,600
fuse	fuse [43]	3,483	REST server	15
mony	money-transfer [44]	3,136	money transfer API	5
Total		446,444		

Table 5.2: Real-world Akka applications from GitHub.

We use GitHub to randomly select a set of 10 mature and real-world Java Akka applications with a total of 446,444 lines of code. Table 5.2 shows these applications, which cover a broad spectrum of sizes, ranging from 261,951 to 15,527 to 3,136 lines of code, of domains, from internet of things (IoT) to web performance testing to money transfer, and of starts, from 19,600 to 390 to 5. An application is Akka application if it uses Java Akka APIs in its source code. Due to compilation issues in Eclipse, we include *connectivity*, *internal*, *things*, *gateway*, and *base* subsystems of *ditto* and *rpc* of *flink*.

CHAPTER SIX

MUTATION TESTING RESULTS FOR ACTOR CONCURRENCY

6.1 *μAkka*'s Mutant Generation and Coverage

Id	PATH		CONFIG		COMM		LIFE	
	Gen	***Cov***	***Gen***	***Cov***	***Gen***	***Cov***	***Gen***	***Cov***
ditto	117	64	27	15	993	410	1,626	580
lms	0	0	3	1	125	82	706	425
wot	23	19	39	4	84	32	235	90
flower	13	7	7	7	30	9	72	30
comb	12	12	1	1	29	24	43	43
rhino	7	5	1	1	24	11	29	15
parc	17	10	3	1	296	82	410	236
flink	4	4	2	2	34	19	110	61
fuse	68	18	3	0	22	7	53	14
mony	15	12	1	1	98	38	272	24
Total	276	151	87	33	1,735	714	1,518	622
$\boldsymbol{Gen}_{\%}$	2.4		0.8		14.8		30.3	
$\boldsymbol{Cov}_{\%}$		54.8		38.0		41.2		42.7

Table 6.1: Mutant generation, ***Gen***, and coverage, ***Cov***, for mutation operator groups: Path, Config, Comm, and Life.

Table 6.1 and Table 6.2 shows the number of mutants that different mutation operator groups generate, ***Gen***, and the number of these mutants that the tests can cover, ***Cov***. The tables show these numbers in aggregate, for all applications, and separately, for individual applications. $\boldsymbol{Gen}_{\%}$ is the ratio of the number of mutants that an operator generates to the number of mutants that all operators generate. $\boldsymbol{Cov}_{\%}$ is the ratio of the covered mutants of an operator to its generated mutants. Table 6.3 shows the total number

Id	EXCEP		INTER		RACE		CLUSTER	
	Gen	***Cov***	***Gen***	***Cov***	***Gen***	***Cov***	***Gen***	***Cov***
ditto	71	48	2,161	802	724	240	1	1
lms	0	0	759	499	468	320	0	0
wot	5	3	207	91	124	52	2	0
flower	5	4	73	26	28	4	0	0
comb	2	2	49	49	4	4	0	0
rhino	3	2	38	23	8	4	0	0
parc	8	8	450	228	220	112	0	0
flink	6	6	115	56	56	24	0	0
fuse	3	2	32	10	12	0	0	0
mony	4	4	304	23	140	4	0	0
Total	107	79	4,188	1,807	1,784	764	3	1
$\textbf{\textit{Gen}}_{\%}$	1.0		35.7		15.3		0.1	
$\textbf{\textit{Cov}}_{\%}$		73.9		43.2		42.9		33.4

Table 6.2: Mutant generation, ***Gen***, and coverage, ***Cov***, for mutation operator groups: Exception, Interaction, Race, and Cluster.

Id	*Total*	
	Gen	***Cov***
ditto	5,720	2,160
lms	2,061	1,327
wot	719	291
flower	228	87
comb	140	135
rhino	110	61
parc	1,404	677
flink	372	172
fuse	193	51
mony	834	106
Total	11,732	5,067
$\textbf{\textit{Cov}}_{\%}$		43.2

Table 6.3: Overall mutant generation (Gen) and coverage (Cov) across all mutation operator groups.

of mutants that different mutation operator groups generate, ***Gen***, and the number of these mutants that the tests can cover, ***Cov***.

Generation According to Table 6.1, Table 6.2 and Table 6.3 , row $\boldsymbol{Gen}_{\%}$, for all applications in aggregate, the number of mutants that different mutation operator groups generate are substantially different, with INTERACTION alone generating more than a third (35.7%) of the mutants, which is the most, and CONFIGURATION generating the least (0.8%). This excludes the outlier CLUSTER that generates 0.1% of mutants with only 3 mutants. Four operator groups, INTERACTION, LIFE CYCLE, RACE, and COMMUNICATION TYPE, that include 17 out of 32 (53.1%) ***μAkka*** operators, together generate the majority (96.1%) of mutants whereas the other four, CLUSTER, CONFIGURATION, EXCEPTION, and PATH, generate only a small minority (3.9%).

> ***Finding 1:*** $\sim\frac{1}{2}$ of ***μAkka*** operators generate $>\frac{9}{10}$ of mutants.

Coverage Similarly, according to row $\boldsymbol{Cov}_{\%}$, tests can cover less than half (43.2%) of ***μAkka*** mutants, and leave more than half (57.8%) uncovered, with substantially different coverage for mutants of different operators. EXCEPTION mutants are the most (73.9%) covered and CONFIGURATION the least (38.0%), excluding CLUSTER.

> ***Finding 2:*** Tests are ineffective in covering $>\frac{1}{2}$ of ***μAkka*** mutants.

Id	PATH			CONFIG			COMM		
	Eas	***Dup***	***Sub***	***Eas***	***Dup***	***Sub***	***Eas***	***Dup***	***Sub***
ditto	3	10	14	5	2	5	83	46	125
lms	0	0	0	1	0	0	61	0	0
wot	4	5	4	3	2	2	10	8	3
flower	0	2	1	2	3	2	2	3	5
comb	0	0	0	0	0	0	7	4	18
rhino	1	4	4	0	0	0	0	5	5
parc	0	5	9	0	0	0	19	20	32
flink	0	0	1	0	0	0	11	2	8
fuse	4	11	11	0	0	0	1	1	1
mony	1	0	1	0	0	0	2	5	9
Total	13	37	45	11	7	9	196	94	206
$\textbf{\textit{Eas}}_{\%}$	8.7			33.4			27.5		
$\textbf{\textit{Dup}}_{\%}$		24.6			21.3			13.2	
$\textbf{\textit{Sub}}_{\%}$			29.9			27.3			28.9

Table 6.4: Mutant ease-of-killing, duplicity, and subsumption (Part 1).

Id	LIFE			EXCEP			INTER		
	Eas	***Dup***	***Sub***	***Eas***	***Dup***	***Sub***	***Eas***	***Dup***	***Sub***
ditto	130	131	142	11	2	5	142	116	120
lms	124	74	70	0	0	0	81	29	32
wot	9	2	2	2	0	0	8	0	0
flower	1	12	11	1	1	1	2	7	7
comb	9	11	22	0	0	0	7	9	11
rhino	1	3	3	2	1	1	0	1	1
parc	8	51	70	1	0	3	5	53	50
flink	21	13	8	1	1	1	11	3	7
fuse	4	5	5	1	0	1	0	4	4
mony	1	4	4	1	0	2	4	2	5
Total	308	306	337	20	5	14	260	224	237
$\textbf{\textit{Eas}}_{\%}$	20.3			25.4			14.4		
$\textbf{\textit{Dup}}_{\%}$		20.2			6.4			12.4	
$\textbf{\textit{Sub}}_{\%}$			22.3			17.8			13.2

Table 6.5: Mutant ease-of-killing, duplicity, and subsumption (Part 2).

Id	RACE			CLUSTER			*Mutation Score*
	Eas	***Dup***	***Sub***	***Eas***	***Dup***	***Sub***	
ditto	31	19	23	1	0	0	16.1
lms	40	25	25	0	0	0	17.8
wot	7	0	0	0	0	0	9.4
flower	0	0	1	0	0	0	24.6
comb	0	0	0	0	0	0	56.5
rhino	0	0	0	0	0	0	19.1
parc	0	22	12	0	0	0	20.4
flink	5	0	0	0	0	0	21.5
fuse	0	0	0	0	0	0	16.1
mony	0	0	0	0	0	0	5.2
Total	83	66	61	1	0	0	16.6
$\boldsymbol{Eas}_{\%}$	10.9			100.0			17.7
$\boldsymbol{Dup}_{\%}$		8.7		0.0			14.6
$\boldsymbol{Sub}_{\%}$			8.0	0.0			24.2

Table 6.6: Mutant ease-of-killing, duplicity, and subsumption (Part 3).

6.2 *µAkka*'s Quality of Mutants

Table 6.4, Table 6.5 and Table 6.6 show the numbers of easy-to-kill, ***Eas***, duplicate, ***Dup***, and subsumed mutants, ***Sub***, for *µAkka* mutation operator groups. $\boldsymbol{Eas}_{\%}$, $\boldsymbol{Dup}_{\%}$, and $\boldsymbol{Sub}_{\%}$, respectively, are ratios of easy-to-kill, duplicate, and subsumed mutants of an operator to its covered mutants.

Ease-of-killing According to Table 6.4, Table 6.5 and Table 6.6, row $\boldsymbol{Eas}_{\%}$, less than a fifth (17.4%) of mutants are easy-to-kill and non-quality, whereas more than four fifth (82.6%) are hard-to-kill and quality, with substantially different ease-of-killing for mutants of different operators. CONFIGURATION mutants are the most (33.4%) easy-to-kill and PATH the least (8.7%), excluding CLUSTER.

> ***Finding 3:*** $>\frac{4}{5}$ of *µAkka* mutants are hard-to-kill and quality.

Duplicity According to row $\boldsymbol{Dup}_{\%}$, more than a seventh (14.6%) of mutants are duplicate and non-quality, whereas about six seventh (85.4%) are unique and quality, with substantially different duplicity for mutants of different operators. PATH mutants are the most (24.6%) duplicate and EXCEPTION the least (6.4%), excluding CLUSTER.

> ***Finding 4:***
> $\sim\frac{6}{7}$ of ***μAkka*** mutants are unique and quality.

Subsumption According to row $\boldsymbol{Sub}_{\%}$, less than a quarter (24.2%) of mutants are subsumed and non-quality, whereas more than three quarters (75.8%) are subsuming and quality, with substantially different subsumption for mutants of different operators. PATH mutants are the most (29.9%) subsumed and RACE the least (8.0%), excluding CLUSTER.

> ***Finding 5:*** $>\frac{3}{4}$ of ***μAkka*** mutants are subsuming and quality.

Altogether The quality of ***μAkka*** mutants differ substantially for its different operators. RACE mutants are the highest quality as the second least easy-to-kill, third least duplicate, and the second least subsumed, whereas CONFIGURATION mutants are the lowest quality as the seventh easy-to-kill, most duplicate, and subsumed.

> ***Finding 6:*** Race, Interaction, Cluster, Exception, Life, Communication, Path, and Config are highest to lowest ***μAkka*** mutants.

6.3 *μAkka*'s Test Effectiveness

Table 6.4, Table 6.5 and Table 6.6 show the traditional mutation scores for tests of different applications. According to Table 6.4, Table 6.5 and Table 6.6, tests are effective in killing only a sixth (16.6%) of ***μAkka*** mutants, and leave about five sixth (83.3%) alive. In addition, Finding 2 says that the tests are ineffective in covering more than half of ***μAkka*** mutants.

> ***Finding 7:*** Tests are ineffective not only in covering $>\frac{1}{2}$ of ***μAkka*** mutants but also in killing $>\frac{5}{6}$.

6.4 PIT's Mutant Quality and Test Effectiveness

Table 6.7 shows the generation, coverage, and the quality of PIT mutants of our applications. Due to compilation and execution issues in PIT, we include six applications *ditto*, *lms*, *wot*, *flower*, *parc*, and *fuse* that we used previously. Generation, coverage, and quality of ***μAkka*** mutants for these six applications can be easily calculated using Table 6.4, Table 6.5 and Table 6.6.

Id	*PIT Mutations*					*Mutation Score*
	Gen	*Cov*	*Eas*	*Dup*	*Sub*	
ditto	17,652	11,108	5,734	5,727	3,906	44.9
lms	2,436	915	250	265	164	16.8
wot	1,969	1,087	604	404	331	38.4
flower	2,565	470	101	182	155	8.6
parc	1,262	1,232	162	388	229	40.0
fuse	293	106	70	65	55	26.3
Total	26,177	14,909	6,921	7,031	4,840	37.8
$\boldsymbol{Cov}_{\%}$		57.0				
$\boldsymbol{Eas}_{\%}$			46.5			
$\boldsymbol{Dup}_{\%}$				47.2		
$\boldsymbol{Sub}_{\%}$					32.5	

Table 6.7: Mutant quality and test effectiveness in PIT.

Coverage According to Table 6.7, row $\boldsymbol{Cov}_{\%}$, tests cover more than half (57.0%) of PIT mutants, and leave the other half (43%) uncovered, which is a third more than (1.3x) the 44.5% coverage of ***μAkka*** mutants, for these six applications.

> ***Finding 8:*** Tests are 1.3x less effective in covering ***μAkka*** than PIT mutants.

Ease-of-killing, duplicity, and subsumption According to row $\boldsymbol{Eas}_{\%}$, less than a half (46.5%) of PIT mutants are easy-to-kill, which is more than twice (2.64x) the 17.6%

ease-of-killing for ***μAkka*** mutants. Similarly, according to row $\boldsymbol{Dup}_{\%}$, less than a half (47.2%) of PIT mutants are duplicates, which is thrice (3.2x) the 14.7% duplicity for ***μAkka*** mutants. Finally, according to row $\boldsymbol{Sub}_{\%}$, less than a half (32.5%) of PIT mutants are subsumed, which is a third more than (1.3x) the 23.8% subsumption for ***μAkka*** mutants.

Finding 9: ***μAkka*** mutants are higher quality than PIT: 2x harder to kill, 3x less duplicate, and 1.3x less subsumed.

Effectiveness of tests Table 6.7 shows the traditional mutation scores for tests of our six applications. According to Table 6.7, tests are effective in killing more than a third (37.8%) of PIT mutants, which is 2.3x the 16.8% test effectiveness for ***μAkka*** mutants. In addition, Finding 8 says tests are 1.3x less effective in covering ***μAkka*** mutants.

Finding 10: Tests are 1.3x less effective in covering and 2.3x less effective in killing ***μAkka*** mutants than PIT.

6.5 Bug Coverage by PIT and Jagannath et al.'s

Figure 6.1 shows the coverages of non-logic and logic bugs, in our bug set $\mathscr{B}$, by mutation operators in ***μAkka***, Jagannath et al.'s [122], and PIT and the overlap of these coverages. The bug coverage of an operator is the ratio of the number of bugs that it can induce and cover to the number of all bugs in $\mathscr{B}$. Similarly, the coverage overlap of an operator with another is the ratio of the number of bugs that the former covers to the latter. For example, ***μAkka*** INTERACTION covers about one seventh (13.5%) of bugs in $\mathscr{B}$ and

Jagannath et al.'s Messaging overlaps with INTERACTION in covering about one third (32.0%) of bugs that INTERACTION covers.

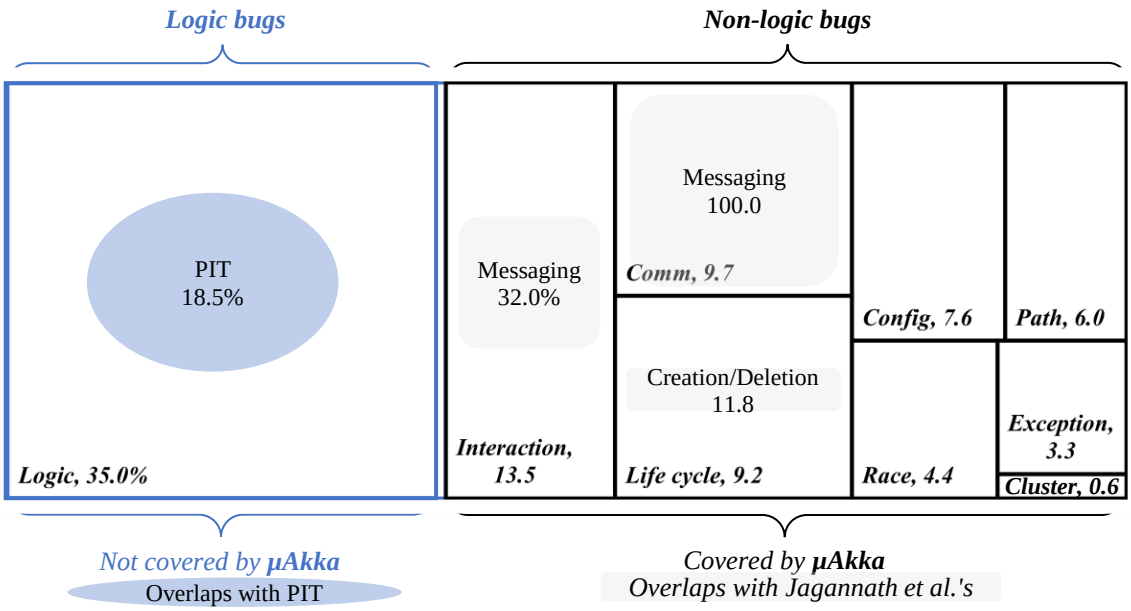

Figure 6.1: Bug coverage by *µAkka*, Jagannath et al.'s, and PIT.

In Figure 6.1, *µAkka* covers $\mathscr{B}$'s non-logic bugs and there is an overlap between bugs that *µAkka*'s COMMUNICATION TYPE, INTERACTION, and LIFE CYCLE cover and Jagannath et al.'s Messaging and Creation/Deletion. In total, Jagannath et al.'s cover about a third (30.7%=31/101) of non-logic bugs that *µAkka* covers.

Finding 11: *µAkka* covers 3.3x more bugs than Jagannath et al.

For non-logic bugs that ***μAkka*** does not cover, PIT covers less than a fifth (18.5%=12/65) of these bugs. Together, ***μAkka*** and PIT cover more than two third (68.1%=113/166) of non-logic and logic bugs which is 1.1x the 60.8% coverage of ***μAkka*** alone and 9.5x the 7.2% coverage of PIT alone.

> ***Finding 12:*** ***μAkka*** + PIT cover 1.1x more bugs than ***μAkka***.

6.6 Implications

In this section, we discuss a few implications of our findings and ***μAkka*** usages in predictive, selective, and sampling mutation testing and actor concurrency testing.

Predictive mutation testing for actor concurrency[156] To improve efficiency, previous work [196, 151] uses characteristics of mutants as features to build and train models that can predict mutant killing without executing the mutants. For example, Mao et al. [151] uses characteristics such as the coverage of a mutant (numTestCover) and the operator group that generates the mutant (MutatorClass) for their 0.85 accurate predictions of mutant killing in 654 real-world Java applications with more than 4 million lines of code. While still non-existent, future prediction mutation testing tools for actor concurrency can use ***μAkka*** mutants, and their characteristics, such as mutation operator group, coverage, ease-of-killing, duplicity, and subsumption, to build and train models that predict the killing of actor concurrency mutants.

N-selective and N%-sampling mutation testing for actor concurrency Similarly, to improve efficiency, previous work [159, 150] approximates mutation testing using a select set of mutation operators, that excludes N operators that generate the most/more mutants, while maintaining the mutation score. For example, Offutt and Rothermel [159] use 2- and

4-selective mutation testing, with 22 operators, in 10 Fortran applications and achieve mutant reductions of 24.0% and 41.4%. While still non-existent, future actor concurrency mutation testing can use ***μAkka*** operators, mutants, and statistics about their generation for N-selective testing. According to Table 6.1, Table 6.2 and Table 6.3, a 4-selective testing that excludes the four operators in COMMUNICATION TYPE and RACE groups, could potentially reduce our mutant numbers by 30.1%, if their exclusion maintains the mutation score. Similarly, previous work [111, 198] approximates mutation testing by sampling N% of mutants per criteria such mutation operator, method, or class. Future actor concurrency mutation testing can use ***μAkka*** operators, operator groups, and mutant quality as new guiding criteria for sampling.

Actor concurrency testing To improve tests, previous work [91, 102] uses mutation testing to guide *where* to test and *what* to improve. For example, Fraser and Zeller [102] μTEST uses alive mutants to generate oracles and tests that kill 75% of all mutants in 10 Java libraries with 1,416 classes. Future actor concurrency testing can use ***μAkka*** to guide similar oracle and test generations.

6.7 Threats to Validity

Some of our decisions during this work could be a threat to its validity. The bug set $\mathscr{B}$ that we use to understand Akka bugs and their causes may not be representative of all Akka bugs and could be a threat. However, the large number of bugs in $\mathscr{B}$ from both popular Stack Overflow and GitHub and the large number of Stack Overflow posts and GitHub commits that the previous work [64] uses to construct $\mathscr{B}$ could help mitigate this threat. The manual analysis that we use to understand the bugs, design and group mutation operators, and understand bug coverages can be a threat. To minimize this threat, we use well-known sorting techniques [99], with multiple sorters, that previous work proposes and often uses [64, 65, 49]. The metrics that we use to measure the mutant quality and test

effectiveness can be another threat. To minimize this threat, we use well-known metrics that previous work proposes and often uses for mutant quality [56, 163, 16, 165, 93, 130, 125] and test effectiveness [132, 166, 164, 56, 125]. The applications that we use in our evaluations may not be a representative of all Akka applications and could be a threat. To minimize this threat, we select a random set of applications that are different in sizes, domains, number of stars, and developers.

CHAPTER SEVEN

ML TEST TOPICS: RESULTS AND ANALYSIS

In this chapter, we present a comprehensive discussion of the results obtained from our study, structured around our research questions RQ1 through RQ6. Our analysis aims to provide a detailed understanding of the characteristics, prevalence, and focus areas of machine learning (ML) test topics as observed in real-world projects.

7.1 RQ1: ML Test Topics

Table 7.1 and Table 7.2 show the ML test topics that real-world developers write tests about. According to Table 7.1 and Table 7.2 , developers write tests about a broad spectrum of 25 ML test topics including Optimizers, Feedforward Neural Networks, Diffusion Models, and File Management.

The meaning of an ML test topic can be best understood by looking at ML tests that belong to the topic. To illustrate, Figure 7.1 shows part of a test case in Feature Transformer topic. The test case is from lambda-packs project [106] that provides a set of precompiled packages for Amazon web services (AWS) lambda which is Amazon's event driven serverless computing platform. This test case constructs the NumPy array X and converts it to the Panda dataframe X_df with column names col0 and col1. Then it constructs the passthrough column transformer ct and applies it to X_df. Finally it asserts that column names col0 and col1 are not changed by the transformation of the column transformer. A column transformer applies transformations to columns of a dataset. A passthrough transformer does not change the columns (features) of a dataset. NumPy and Panda are Python libraries to work with multidimensional arrays and for data analysis and manipulation, respectively.

No.	Topic name	Topic words
0	Feature Transformer	featur transform fit array column frame tran seri forecast train valu select pipelin step encod
1	Error Handling	error rais valu pytest warn msg match regex runtim err messag invalid one must fail
2	Gradient	allclos gradient atol diff grad qml rtol numpi order backend param symbol circuit math shape
3	Module Management	modul arg forward graph jit node inp script function randn add trace mod init foo
4	Natural Language Processing	embed encod token length decod emb text vocab sequenc pad seq batch word logit tag
5	Bbox Transformation	box point allclos depth field rotat mode video cam rang bbox lidar flip mat rot
6	Variable Reuse	variabl var constant shape run sess batch session get graph initi width num updat valu
7	Bbox Loss	loss target bbox label empti pred weight assign result fake item ignor box zero reduct
8	Optimizers	optim param model train step state paramet epoch mock trainer get parallel batch rank loader
9	Convolutional Blocks	conv channel shape block feat stride pad kernel num randn net convd modul pool norm
10	Model Verification	fit clf predict estim train classifi sampl state array pred regress est iri coef proba
11	Tensor Operations	devic grad cuda cpu requir ref npu backward randn clone rtol atol item shape view
12	Array Operations	array spars indic shape axi numpi dens csr matrix rank row result mesh face one

Table 7.1: Topic names and top 15 stemmed words of our ML test topics (Part 1).

Similarly, Figure 7.2 shows a test case in *Graph Models*topic. The test case is from *Pytorch Geometric (PyG)* project [107] which is a library to work with Graph Neural Networks (GNNs). This test case constructs the Torch tensor edge_index to represent a bipartite graph and divides the graph into smaller src_batch and dest_batch batches for batch processing. Then it generates negative edges for the graph. Finally, it asserts that the number of negative edges does not exceed the number of the original edges. A graph is bipartite if its nodes can be divided into two sets with edges between

No.	Topic name	Topic words
13	Bounding Box	result img cfg head cuda shape meta bbox instanc num label pred mask score xyz
14	Reinforcement Learning	val env action spec shape mode state polici space theano batch arg reward actor observ
15	File Management	path file load save dataset join tmp checkpoint exampl key temp config obj pipelin exist
16	Evaluation Metrics	score label metric pred grid array distanc search cluster neighbor sampl almost averag result precis
17	Dimensionality Reduction	state rng fit compon alpha array almost sampl mean featur transform pca scaler nan init
18	Data Distribution	sampl dist mean log prob shape kernel normal actual prior std num exp dim scale nois
19	Feedforward Neural Networks	weight layer bia norm hidden linear state kera activ shape cell mlp batch init rnn
20	Quantization	net quantiz scale oper blob ref workspac core quant refer sim creat encod per astyp
21	Attention Mechanism	dim mask batch shape key num attent head anchor seq attn gener queri valu pad
22	Model Configuration	model config onnx train get convert target predict shape pretrain dummi kera opset infer backend
23	Graph Models	dataset num index batch edg node split idx train group graph sampler sampl loader src
24	Diffusion Models	imag pipe gener shape img seed diffus crop devic patch clip num schedul manual infer

Table 7.2: Topic names and top 15 stemmed words of our ML test topics (Part 2).

nodes from different sets. Batch processing processes larger datasets by breaking them down into smaller batches. A negative edge of a graph is an edge that does not exists in the graph. Negative edges allows for effective training of graph models. Limiting the number of negative edges ensures the cost effectiveness of negative sampling in a graph model.

Finally, Figure 7.3 shows a test case in Quantization topic. The test case is from AIMET (AI Model Efficiency Toolkit) project [105] which “provides advanced model quantization and compression techniques for trained neural network models”. This test case

```
# from test file test_column_transformer.py
def test_column_transformer_get_feature_names_dataframe():
 # passthough transformer with a dataframe
 pd = pytest.importorskip('pandas')
 X = np.array([[{'a': 1, 'b': 2}, {'a': 3, 'b': 4}],[{'c': 5}, {'c': 6}]], dtype=object).T
 X_df = pd.DataFrame(X, columns=['col0', 'col1'])
 ct = ColumnTransformer([('trans', 'passthrough', ['col0', 'col1'])])
 ct.fit(X_df)
 assert ct.get_feature_names() == ['col0', 'col1'] ...
```

Figure 7.1: A test in *Feature Transformer*topic from *lambda-packs* [106] project.

```
# from test file test_negative_sampling.py
def test_bipartite_batched_negative_sampling():
 edge_index1 = torch.as_tensor([[0, 0, 1, 1], [0, 1, 2, 3]])
 edge_index2 = edge_index1 + torch.tensor([[2], [4]])
 edge_index3 = edge_index2 + torch.tensor([[2], [4]])
 edge_index = torch.cat([edge_index1, edge_index2, edge_index3], dim=1)
 src_batch = torch.tensor([0, 0, 1, 1, 2, 2])
 dst_batch = torch.tensor([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2])
 neg_edge_index = batched_negative_sampling(edge_index,(src_batch, dst_batch))
 assert neg_edge_index.size(1) <= edge_index.size(1) ...
```

Figure 7.2: A test in *Graph Models*topic from *Pytorch Geometric* project [107].

constructs the quantizer quantizer and configures its properties, including its encoding encoding. Then, it creates the Torch tensor inp_tensor and quantizes it. Finally, it asserts that the quantized output is the same as a predefined expected output. Quantization lowers the precision of parameters of a model to decrease its size and improve itsefficiency. To illustrate, quantization could convert weights and activations parameters of a model from 32 bit high-precision floating-points to 8 bit lower-precision integers. A tensor is a multiway array for representation and processing of multidimensional data, with foundations in multilinear algebra.

```
# from test file test_qc_quantize_op.py
def test_quantize_only_asymmetric_cpu(self):
 # Test tensor quantizer quantize only asymmetric functionality
 # Note: Quantize−only mode uses symmetric range for output tensor by default
 quantizer = StaticGridPerTensorQuantizer(bitwidth=8, round_mode='nearest',
  quant_scheme=QuantScheme.post_training_tf, use_symmetric_encodings=False,
  enabled_by_default=True, data_type=QuantizationDataType.int)
 encodings = libpymo.TfEncoding()
 encodings.bw, max, min, offset = 8, 2.23, −5.19, −178
 quantizer.encoding = encodings
 inp_tensor = torch.tensor([−7, −5, −3, 0, .1, 2.5])
 quant_out = quantizer.quantize(inp_tensor, MAP_ROUND_MODE_TO_PYMO['nearest'])
 expected_out = torch.tensor([−128., −122., −53., 50., 53., 127.], dtype=torch.float32)
 assert torch.equal(quant_out, expected_out)
```

Figure 7.3: A test in *Quantization*topic from *lambda-packs* [106] project.

Finding 1: Developers write ML test cases about a broad spectrum of 25 topics including Optimizers, Feedforward Neural Networks, Diffusion Models, and File Management.

7.2 RQ2: ML Test Topic Hierarchy

Figure 7.4 shows the hierarchy of our ML test topics. The figure shows the topics, in gray, and their categories, in white, and the percentages for the number of their test cases. The hierarchy extends outwards from higher to lower level categories and then topics. According to Figure 5, ML test cases that developers write can be grouped into a hierarchy with seven high level categories Performance, Architecture, General, Basics, Data Preparation, Natural Language Processing (NLP), and Reinforcement Learning (RL). More than a third of test cases are about Performance, near a fifth are about Architecture, and the

least are about Reinforcement Learning. More than half of test cases are about either Performance or Architecture.

Finding 2: ML test cases that developers write can be grouped into a topic hierarchy with seven categories Performance, Architecture, General, Basics, Data Preparation, Natural Language Processing, and Reinforcement Learning.

In the following, we discuss each ML test higher level category and its lower categories and topics using three real-world sample test cases for each topic. A sample test case is shown in quotations “” and consists of the name of the test case inside brackets [] followed by a comment written by the programmer of the test case to describe its functionality.

7.2.1 General.

General category is about general software development practices, with more than a seventh of ML test cases in it. General includes three topics Module Management, Error Handling, and File Management.

Module Management is about the composition, exporting, packing, unpacking, and freezing of Python code modules and submodules and their forward methods with test cases like `[test_module_forward_preforward_hook_removable]`: “This test is to test when multiple pre-forward hook functions can be registered successfully and used correctly, and if the handle can be removable during the pre-forward hook function call,” `[test_module_direct_forward_invocation]`: “Test that hooks are only invoked when the module is called directly and not when `forward` is called,” and

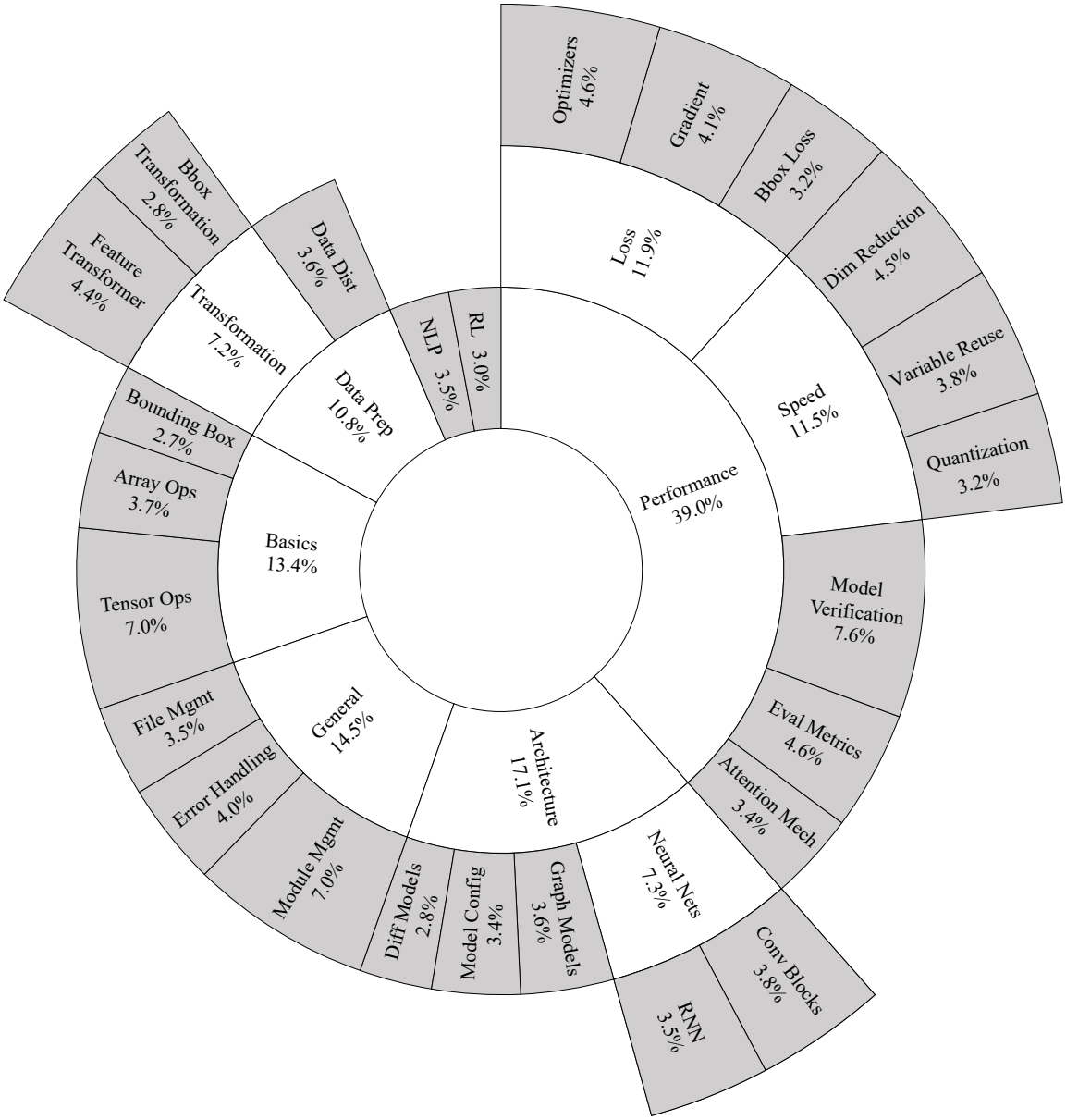

Figure 7.4: ML test topics and their hierarchy.

`[test_trace_return_dataclass]`: "Test case for (traced) Module that returns dataclass."

Nearly half of general test cases are about Module Management. In Python, a module is a file with definitions and statements. A hook is a designated point in the execution of the module where registered functions can execute. For example, a pre-forward hook is the point before the execution of the `forward` method of a module or class. In PyTorch, `forward` defines the forward pass of a neural network that computes the output of the network using its input. A module is traced if its computation graph is recorded. A module should produce the same output before and after tracing.

Error Handling is about raising, catching, and handling of errors and exceptions with test cases like `[test_safe_divide]`: "Test that _safe_divide handles division by zero," `[test_scale_input_finiteness_validation]`: "Check if non-finite inputs raise ValueError," and `[test_invalid_types]`: "Make sure invalid type strings raise an error."

File Management is about storage, validity, and retrieval of data files and directories with test cases like `[test_load_files_allowed_extensions]`: "Check the behaviour of `allowed_extension` in `load_files`," `[test_data_home]`: "Check `data_home` will point to a pre-existing folder," and `[test_validate_file_sha256]`: "Validate SHA256 hash of a file."

Finding 3: 1/7 of ML test cases are about General with 1/2 in Module Management.

7.2.2 Basics.

Basics category is about basic ML constructs and their operations, with more than an eighth of ML test cases in it. Basics includes three topics Tensor Operations, Array Operations, and Bounding Boxes.

Tensor Operations is about the creation, allocation, type, and value of a tensor with test cases like `[test_storage_method]`: "Test TypedStorage, FloatStorage and so on," `[test_iRshift_tensor]`: "Test in-place right shift operation," and `[test_qr_common_shape_format]`: "Test QR decomposition operation across various tensor shapes and formats." QR decomposition divides a tensor into a tensor with orthogonal columns and an upper triangular tensor. More than half of Basics test cases are about Tensor Operations. A tensor is a multiway array for representation and processing of multidimensional data, with foundations in multilinear algebra. In PyTorch, `TypedStorage` represents the storage of a tensor with elements of type `dtype`. In `FloatStorage`, `dtype` is the 32-bit floating point `float32`.

Array Operations is about creation, ranking, shape, compression, and density of a matrix with test cases like `[test_copy]`: "Test copying a Fortran array," `[test_set_order_sparse]`: "Check that _set_order returns sparse matrices in promised format," and `[test_sparse_to_sparse_compressed]`: "This test tests conversion from COO to CSR." Coordinate List (COO) and Compressed Sparse Row (CSR) are popular formats to represent and store sparse matrices.

Bounding Boxes is about prediction, labeling, conversion, and implementing bounding boxes with test cases like `[aug_test_bboxes]`: "Test bboxes with test time augmentation," `[test_bbox_post_process]`: "Test the length of detection instance results is 0," and `[test_mask_scoring_roi_head_forward]`: "Tests trident roi head forward." A bounding box is a rectangle for specification and detection of an object in an image. Test time augmentation creates multiple samples of the original test input and averages them for better predictions. A bounding box is the core of an object detection instance, with its length being zero when no object is detected. A Region of Interest (ROI) is an area of an image where an object is likely to exist. ROIs can be specified using

bounding boxes. TridentNet utilizes a multi-scale feature pyramid and a specialized Trident head to improve detection accuracy across different object sizes and scales.

Finding 4: 1/8 of ML test cases are about Basics with 1/2 in Tensor Operations.

7.2.3 Data Preparation.

Data Preparation category is about preprocessing, transformation, and preparation of raw data for downstream stages of a machine learning pipeline, with more than a tenth of ML test cases in it. Data Preparation includes three topics Feature Transformer, Bbox Transformation, and Data Distribution with Feature Transformer and Bbox Transformation forming the lower level category Transformation.

Feature Transformer is about extraction, encoding, and reduction of data features, with test cases like `[test_column_transformer_get_feature_names_dataframe]`: "Passthrough transformer with a dataframe," `[test_ordinal_encoder_infrequent_three_levels_user_cats]`: "Test that the order of the categories (features) provided by a user is respected," and `[test_ohe_infrequent_multiple_categories_dtypes]`: "Test infrequent categories with a pandas dataframe with multiple dtypes." A passthrough transformer forwards its input data to its output without modifications. One-hot encoding (OHE) transformer transforms categorical data into numerical.

Bbox Transformation is about rotation, scaling, translation, and reorientation of bounding boxes with test cases like `[test_lidar_boxes3d]`: "Test box rotation," `[test_bbox_xyxy_xywh]`: "Test (to) convert boxes to xywh and back. Make sure they are the same," and `[test_boxes_conversion]`: "Test CAM to LIDAR and DEPTH

(conversions)." CAM, LiDAR, and DEPTH coordinate systems capture and represent 3D spatial data.

Data Distribution is about distribution, sampling, and noisiness of data with test cases like `[test_joint_noise_sample_log_prob]`: "Test that sampling and log probability work as expected for pairs of noise," `[test_Bernoulli]`: "Tests Bernoulli distribution," and `[test_TorchDeterministic]`: "Tests the TorchDeterministic distribution." Bernoulli distribution describes the outcome of a binary experiment with exactly two possible outcomes. In PyTorch, `TorchDeterministic` ensures deterministic behavior for random variable operations. A third of Data Preparation test cases are about Data Distribution.

Finding 5: 1/10 of ML test cases are about Data Preparation with 1/3 in Data Distribution.

7.2.4 Architecture.

Architecture category is about building blocks, models, and configurations of ML systems, with a sixth of ML test cases in it. Architecture includes five topics Convolutional Blocks, Feedforward Neural Networks (FNN), Graph Models, Model Configuration, and Diffusion Models, with Convolutional Blocks and Feedforward Neural Networks forming Neural Networks lower level category.

Convolutional Blocks is about building blocks of neural networks that sequence a convolutional, batch normalization, activation, pooling, and dropout layer, with test cases like `[test_up_conv_block]`: "Test BasicConvBlock structure and forward," `[test_mobileoneblock]`: "Test MobileOneBlock with kernel_size 3," and `[test_Clifford1d_conv_shapes]`: "Test shapes of Clifford1d convolution module."

MobileOneBlock is a convolutional block designed to operate on mobile and edge devices. Clifford1d convolutional block allows for processing of one-dimensional data using Clifford algebra.

Feedforward Neural Networks is about cycle-free connection, activation, state, and layering of neurons, with test cases like `[test_global_pruning]`: "Test that global L1 unstructured pruning over 2 parameters removes the 'amount=4' smallest global weights across the 2 parameters," and `[testMultiLayerPerceptron]`: "Tests MultiLayerPerceptron class." L1 unstructured pruning reduces the size of a neural network model by removing its weights with smallest absolute values. Multi-Layer Perceptron (MLP) is a feedforward neural network with multiple layers of fully connected neurons. Nearly half of Architecture test cases are about Neural Networks.

Graph Models is about construction, loading, sampling, and splitting a graph model into positive and negative edges, with test cases like `[test_hard_edge_dst_negative_sample_generate_complex_case]`: "Test GSHardEdgeDstNegativeSampler._generate with slow track when not all the positive edges have enough hard negatives defined," `[test_dataset_split_custom]`: "Transductive split with node task," and `[test_SubgraphSampler_Node_seed_nodes]`: "Test subgraph sampler." Transductive split divides a graph into training and evaluation subsets without changing the structure of the graph.

Model Configuration is about the configuration of layers, operations, and parameters of a model with test cases like `[test_backend_config]`: "Test set backend pattern config," `[test_peft_causal_lm_extended_vocab]`: "Check if kwargs are passed correctly," and `[test_non_default_adapter_name]`: "Test using custom adapter names." Backend configuration of a model configures its computational infrastructure. An adapter customizes and adapts a model to a specific task or training scenario by modifying its configuration.

Diffusion Models is about probabilistic generative models that progressively destruct data by injecting noise and then learn to reverse it for sample generation with applications in image synthesis, with test cases like `[test_stable_diffusion_vae_tiling]`: "Enable VAE tiling, disable VAE tiling," `[test_stable_diffusion_img2img_pipeline_multiple_of_8]`: "Test image-to-image translation pipeline using AltDiffusion," and `[test_mlsd]`: "Test stable diffusion with MLSD." Stable Diffusion is a deep learning text-to-image generative model. Variational AutoEncoder (VAE) improves the images that Stable Diffusion generates. AltDiffusion is an image-to-image translation diffusion model. Machine Learning Sketch-to-Image Diffusion (MLSD) allows for conversion of sketch images into more detailed and realistic images.

Finding 6: 1/6 of ML test cases are about Architecture with 1/2 in Neural Networks.

7.2.5 Performance.

Performance category is about the reduction of input size and the loss and the increasing of speed to improve performance of ML systems, with more than a third of ML test cases in it. Performance includes nine topics Optimizers, Gradient, Bbox Loss, Dimensionality Reduction, Variable Reuse, Quantization, Model Verification, Evaluation Metrics, and Attention Mechanism. Optimizers, Gradient, and Bbox Loss form the lower level category Loss. More than a quarter of Performance test cases are about Loss. Similarly, Dimensionality Reduction, Variable Reuse, and Quantization form Speed. More than a quarter of Performance test cases are about Speed.

Optimizers is about adjusting parameters of a model to reduce its loss function and improve its performance, with test cases like `[test_setup_optimizers]`: "Test that `setup_optimizers()` can handle one or more optimizers," `[test_fabric_optimizer_load_state_dict]`: "Test that the FabricOptimizer (wrapper) can load the state dict on the wrapped optimizer and update its internal __dict__," and `[test_clone_optimizer()]`: "Tests optimizer cloning."

Gradient is about the direction and magnitude of changes that are needed to reduce the loss, with test cases like `[test_tf_grad]`: "Tests that gradients are computed correctly using the TensorFlow interface," `[test_torch]`: "Tests that the output of the Hadamard gradient transform can be differentiated using Torch, yielding second derivatives," and `[test_globalphase_tf_grad]`: "Test the gradient with TensorFlow for a controlled GlobalPhase." Hadamard gradient is the gradient of a function involving a Hadamard product, which is the element-wise multiplication of matrices or vectors. In quantum computing, a global phase is a constant complex multiplier applied to a quantum state without affecting measurement probabilities but can impact phase relationships between quantum states in superpositions.

Bbox Loss is about reducing the loss difference between the predicted and ground truth bounding box of an object, with test cases like `[test_bbox_head_loss]`: "Tests bbox head loss when truth is empty and non-empty," `[test_pisa_reitnanet_head_loss]`: "Tests PISA RetinaNet head loss," and `[test_max_iou_assigner_with_empty_boxes_and_ignore]`: "Test corner case where a network might predict no boxes and ignore_iof_thr is on." A bbox head predicts the coordinates and size of the bounding box. RetinaNet is an object detection model addressing challenges like foreground-background imbalance. PISA RetinaNet incorporates Precise IoU-aware Sample Assignment (PISA) to focus on higher-quality samples. MaxIoUAssigner matches predicted and ground truth boxes.

Dimensionality Reduction is about reducing features (input variables) of high-dimensional data, with test cases like `[test_explained_variances]`: "Test that PCA and IncrementalPCA calculations match," `[test_nmf_float32_float64_consistency]`: "Check that the result of NMF is the same between float32 and float64," and `[test_singular_values]`: "Check that the TruncatedSVD output has the correct singular values." Principal Component Analysis (PCA) maximizes variance in dimensionality reduction. Non-Negative Matrix Factorization (NMF) decomposes a non-negative matrix into two lower-dimensional non-negative matrices. Truncated Singular Value Decomposition (TruncatedSVD) approximates top singular values and vectors to reduce dimensionality.

Variable Reuse is about reuse of variables, such as weights and parameters, across models and training iterations, with test cases like `[testReuseVars]`: "Although we feed different images, the moving_mean and moving_variance shouldn't change," `[testReuseVars]`: "Test variable reuse for batch normalization," and `[testTrainEvalWithReuse]`: "Test reuse of variables between training and evaluation phases." Batch normalization normalizes inputs to have mean zero and variance one to improve training performance.

Quantization is about reducing the precision of data, with test cases like `[test_quantize_only_asymmetric_cpu]`: "Test tensor quantizer quantize-only asymmetric functionality," `[test_tensor_quantizer_bitwidth_32]`: "Test tensor quantizer with bitwidth 32," and `[test_quantize_and_dequantize_op]`: "Test quantization and dequantization for a given bit rate." Quantization can be symmetric or asymmetric. Dequantization is the reverse of quantization, recovering approximated original values.

Model Verification is about verifying that a model's actual and expected behaviors are similar, with test cases like `[test_multinomial_binary]`: "Test multinomial LR on a

binary problem," [test_ovr_partial_fit]: "Test if partial_fit is working as intended," and [test_parallel_regression]: "Check parallel regression." Multinomial logistic regression (LR) is a supervised classification model. partial_fit allows incremental updates without retraining from scratch.

Evaluation Metrics is about metrics such as precision, recall, and F1 score, with test cases like [test_precision_recall_f1_score_multilabel_1]: "Test precision_recall_f1_score on a crafted multilabel example," [test_permute_labels]: "All clustering metrics do not change score due to permutations of labels," and [test_exactly_zero_info_score]: "Check numerical stability when information is exactly zero." Numerical stability measures accuracy in the presence of small data changes.

Attention Mechanism is about focusing a model on specific parts of the input data and masking others, with test cases like [test_att_mask_ignored]: "If an sxs (square self) attention mask is passed in, it should be ignored. Results should be the same as if no mask was passed," and [test_cross_attention_with_dropout]: "Test forward pass (of cross-attention) with dropout." A square self mask is a binary matrix in self-attention that prevents a token from attending to future tokens. Self-attention allows each token to weigh and attend to all others. Cross-attention integrates data from two sources (e.g., secondary context and primary input). Dropout randomly drops input nodes to prevent overfitting.

Finding 7: 1/3 of ML test cases are about Performance with 1/4 in Loss and another 1/4 in Speed.

7.2.6 Reinforcement Learning.

Reinforcement Learning (RL) is for autonomous experience-driven learning where an agent learns a behavior via trial-and-error interactions with a dynamic environment, with test cases like `[test_learns_from_experiences_with_accelerator]`: "Learns from experiences and updates network parameters,"
`[test_clone_returns_identical_agent]`: "Test that clones the agent and returns an identical agent," and `[test_mutation_applies_rl_hp_mutations]`: "Test that the mutation method applies RL hyperparameter mutations to the population and returns the mutated population." Mutation allows for exploration of different hyperparameter values by modifying them through either random or controlled changes.

7.2.7 Natural Language Processing.

Natural Language Processing (NLP) is a tract of machine learning and linguistics for the representation and analysis of human language, with test cases like `[test_strip_accents]`: "Check some classical Latin and Arabic accentuated symbols," `[test_skip_gram_sample_emit_self]`: "Tests skip-gram with `emit_self_as_target = True`," and `[test_max_length_truncation]`: "Test truncation of sentences longer than `max_length`." Skip-gram is an NLP technique for learning word embeddings. A word embedding captures the relations and meaning between words based on their context and usage. Truncation allows handling long texts by reducing their length to a uniform size.

7.3 RQ3: ML Test Topic Popularity

Table 7.3 shows the popularity of our ML test topics, in a descending order of the number of test cases in a topic, with the average in gray. The popularity of a topic is

Topic	*No. of Tests*
Model Verification	10,257
Module Management	9,477
Tensor Operations	9,436
Evaluation Metrics	6,218
Optimizers	6,189
Dimensionality Reduction	6,130
Feature Transformer	5,956
Gradient	5,553
Error Handling	5,458
Variable Reuse	5,138
Convolutional Blocks	5,129
Array Operations	4,913
Data Distribution	4,857
Graph Models	4,811
Natural Language Processing	4,698
Recurrent Neural Networks	4,676
File Management	4,674
Model Configuration	4,591
Attention Mechanism	4,505
Bbox Loss	4,355
Quantization	4,345
Reinforcement Learning	3,986
Diffusion Models	3,812
Bbox Transformation	3,686
Bounding Boxes	3,613
Average	***5,458***

Table 7.3: Popularity of ML test topics.

measured using the number of test cases in the topic, as described in step eight of our methodology. The higher the number of test cases in a topic the more popular the topic.

According to Table 7.3, the popularity of ML test topics are not uniform. Model Verification is the most popular topic with 10,257 (7.6%) of tests whereas Bounding Box is the least popular with 3,613 (2.7%) of tests. Model Verification is near three times more

Topic	*Cognitive*	*Cyclomatic*
Reinforcement Learning	2.56	5.51
Convolutional Blocks	2.32	7.57
Graph Models	2.17	5.35
Quantization	2.01	3.14
Array Operations	1.83	3.42
Optimizers	1.81	4.05
Attention Mechanism	1.67	4.19
Bounding Boxes	1.64	7.77
Tensor Operations	1.63	2.59
Recurrent Neural Networks	1.49	3.35
File Management	1.35	3.59
Natural Language Processing	1.33	3.53
Module Management	1.28	2.08
Gradient	1.22	3.14
Data Distribution	1.13	3.12
Evaluation Metrics	1	2.94
Dimensionality Reduction	0.95	2.7
Model Configuration	0.86	2.81
Bbox Loss	0.82	3.62
Bbox Transformation	0.8	3.98
Model Verification	0.79	2.52
Diffusion Models	0.77	2.86
Variable Reuse	0.71	1.93
Feature Transformer	0.67	2.93
Error Handling	0.65	1.86
Average	***1.34***	***0.4***

Table 7.4: Difficulty of ML test topics.

popular than Bounding Box. Error Handlingis average in popularity. The average number of tests for an ML test topic is about 5,458. About a fifth of ML test cases are about three topics Model Verification, Module Management, and Tensor Operations, with Module Management and Tensor Operations in Basics category. Eight top topics are more popular than average and the remaining 17 are less popular. Practitioners, researchers, and educators can use topic popularity to tradeoff one topic against another, as discussed.

Finding 8: Model Verification is the most popular ML test topic whereas Bounding Box is the least, with Model Verification three times more popular than Bounding Box.

7.4 RQ4: ML Test Topic Difficulty

Table 3 shows the difficulty of our ML test topics, in a descending order of cognitive complexity, with the average in gray. The difficulty of a topic is measured using the cognitive and cyclomatic omplexities of its test cases, as described in step nine of our methodology. The higher the cognitive and cyclomatic complexity of test cases in a topic the more difficult the topic. According to Table 3, the difficulty of ML test topics are not uniform. Reinforcement Learning has the highest cognitive and the third highest cyclomatic complexity whereas Error Handling has the lowest cognitive and cyclomatic complexity. Reinforcement Learning is near four times more difficult than Error Handling. Natural Language Processing is average in difficulty. The average cognitive and cyclomatic complexities are 1.34 and 3.63. Eleven top topics in Table 3 are more difficult than average and the remaining 14 are less difficult. Practitioners, researchers, and educators can use topic difficulty to tradeoff one topic against another, as discussed.

Finding 9: Reinforcement Learning is the most difficult ML test topic whereas Error Handling is the least, with Reinforcement Learning being four times more difficult than Error Handling.

7.5 RQ5: ML Test Topic Popularity and Difficulty Correlation

Table 7.5 shows the coefficients and p-values for the correlations between the popularity and difficulty of our ML test topics in the past 14 years. The popularity of a topic is measured using the number of its test cases; similarly, the difficulty of the topic is measured using the two metrics cognitive and cyclomatic complexity of its test cases. According to Table 7.5, there is a moderate positive correlation between the popularity and difficulty of our ML test topics when the difficulty is measured using cognitive complexity. And this correlation is statistically significant at 95% significant level. Similarly, there is a statically significant weak positive correlation between the popularity and difficulty when it is measured using cyclomatic complexity. This means that more popular ML test topics are usually more difficult.

Coefficient / p-value	**Cognitive**	**Cyclomatic**
No. of test cases	0.287 / 0	0.191 / 0

Table 7.5: Correlations of ML test topics popularity and difficulty

Finding 10: There is a statically significant positive correlation between the popularity and difficulty of our ML test topics.

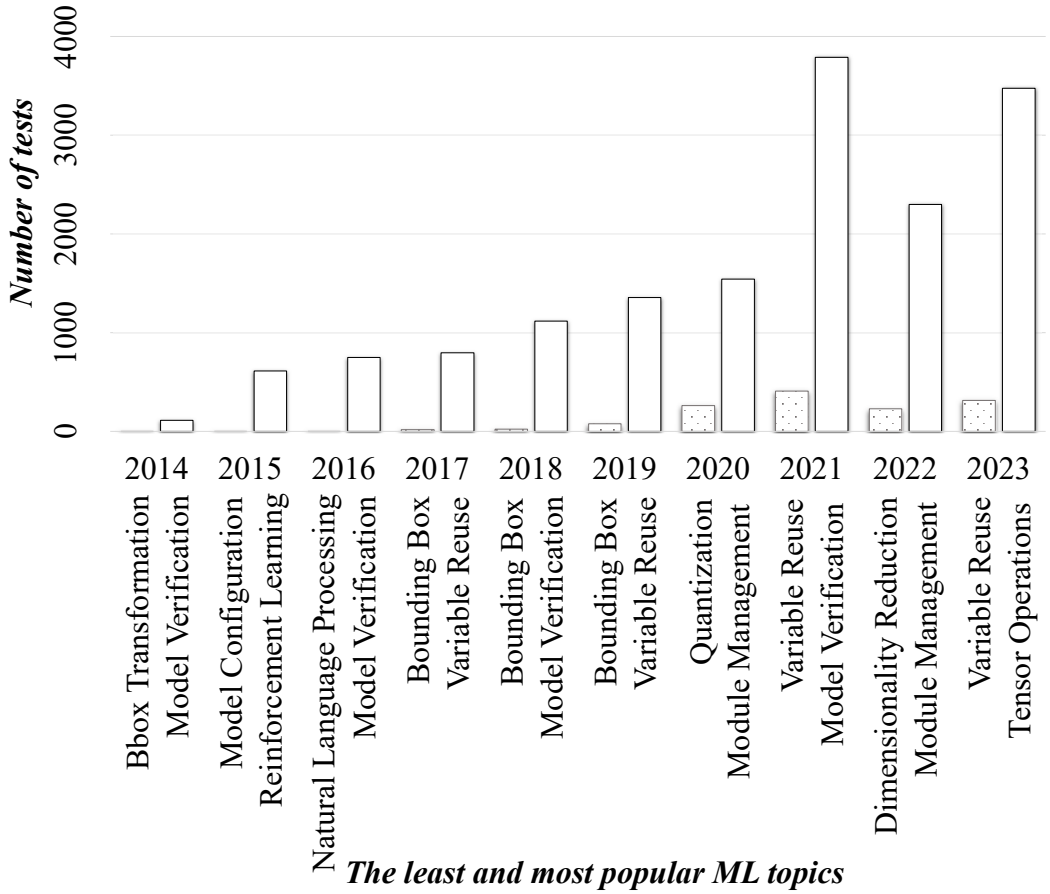

Figure 7.5: Historical popularity of ML test topics.

7.6 RQ6: ML Test Topic Historical Popularity and Difficulty

7.6.1 Historical Popularity.

Figure 7.5 shows the historical popularity of ML tests and their changes over the past ten years, from 2014 to 2023. The figure shows the most popular topics, in solid white bars, and the least popular topics, in dotted gray bars. Table 7.6 shows topics with zero test cases in this decade.

According to Figure 7.5, the popularity of ML test topics over time has not been uniform and has changed, sometimes dramatically. Model Verification, Reinforcement

Learning, Variable Reuse, Module Management, and Tensor Operations have been the most popular ML test topics in past decade whereas Bbox Transformation, Model Configuration, Natural Language Processing, Bounding Box, Quantization, Variable Reuse, and Dimensionality Reduction have been the least. Model Verification has been the most popular for four out of ten years whereas Bounding Box has been the least popular for three. Model Verification that has been popular in the first half of the decade, has been replaced by Module Management in popularity in the second half. Similarly, Bounding Box has been replaced by Variable Reuse in unpopularity. Variable Reuse has gone from the most popular in 2019 to the least in 2021 in less than two years. Conversely, Module Management has gone from non-existent with zero test cases in 2014 to the most popular in 2022. Practitioners, researchers, and educators can use the historical topic popularity to tradeoff one topic against another, as discussed.

Year	Topic
2014	Module Management, Variable Reuse, Convolutional Blocks, Bounding Box, Quantization
2015	Bbox Transformation

Table 7.6: ML test topics with zero test cases

Finding 11: ML test topic popularity has changed, sometime dramatically, over the past decade, with some topics such as Variable Reuse going from the most popular to the least in less than two years.

7.6.2 Historical Difficulty.

Figure 7.6 shows the historical difficulty of ML tests and their changes over the past ten years, from 2014 to 2023, using their cognitive complexity. The figure shows the most difficult topics, in solid white bars, and the least difficult topics, in dotted gray bars.

According to Figure 7.6, the difficulty of ML test topics over time has not been uniform

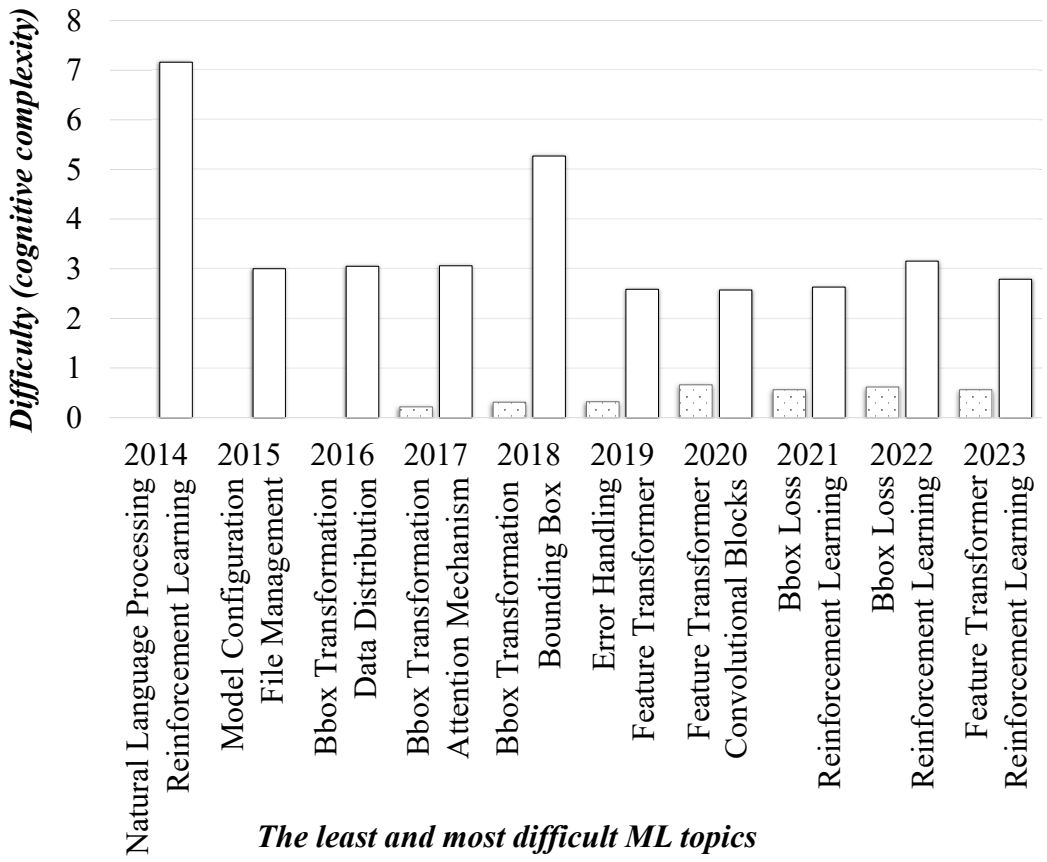

Figure 7.6: Historical difficulty of ML test topics.

and has changed, sometimes dramatically. Reinforcement Learning, File Management, Data

Distribution, Attention Mechanism, Bounding Box, Feature Transformer, and Convolutional Blocks, have been the most difficult ML test topics in past decade whereas Natural Language Processing, Model Configuration, Bbox Transformation, Error Handling, Feature Transformer, and Bbox Loss have been the least. Reinforcement Learning has been the most difficult for three out of ten years whereas Bbox Transformation has been the least difficult for three. Bbox Transformation that has been the least difficult for the first half of the decade has been replaced by Bbox Loss. Feature Transformer has gone from the most difficult in 2019 to the least difficult in 2020, in less than a year. Practitioners, researchers, and educators can use the historical topic popularity to tradeoff one topic against another, as discussed.

Finding 12: ML test topic difficulty has changed, sometime dramatically, over the past decade, with some topics such as Feature Transformer going from the most difficult to the least in less than a year.

7.7 Implications

The results of our study can not only help ML test developers and their practice but also the research and education communities in ML software testing to better decide where to focus their efforts. Members of these communities can use tradeoffs between ML test topics as one of several factors in their decision making process. One way to tradeoff a ML test topic against another is using popularity and difficulty of these topics and their changes over time.

Figure 7.7 shows the four most popular ML test topics for the more recent year 2023, in larger solid white circles, and less recent year 2018, in smaller dotted circles.

Practice According to Figure 7.7, a developer with interest in ML software testing may decide to start their learning on the more popular, less difficult, and more recent

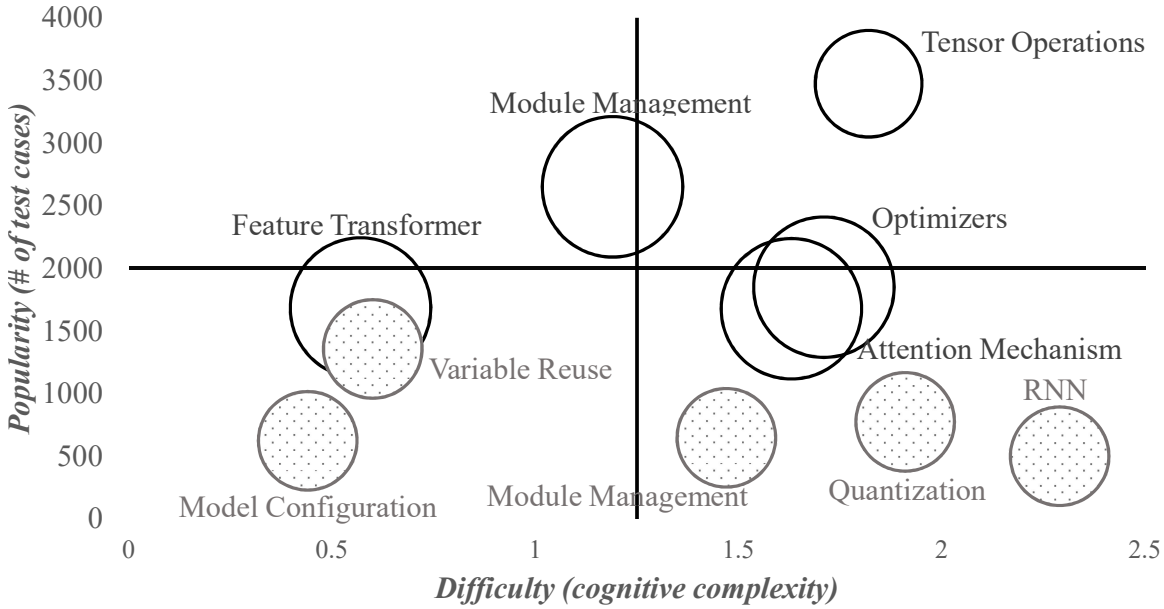

Figure 7.7: Tradeoff ML test topics by popularity/difficult.

Feature Transformer topic, instead of the less popular, more difficult, and less recent Feedforward Neural Networks (FNN). For topics like Feature Transformer and Variable Reuse with similar popularity and difficulty, the developer may decide to focus on the more recent Feature Transformer instead of Variable Reuse. With the recent and substantial increase in the popularity of Module Management, the developer may decide to learn more about this topic, if they have not been yet. A manager of an ML testing team may decide to assign a less difficult Model Configuration testing task to a less experience developer and a more difficult Feedforward Neural Networks task to a more experienced developer.

Research According to Figure 7.7, an experienced ML testing researcher may decide to focus their research on the more difficult, less popular, and less recent and probably more crowded Feedforward Neural Networks, instead of the less difficult, more popular, and more recent and probably less crowded Attention Mechanism in the hope of making more impactful contributions. Conversely, a researcher looking to transition anew to

ML testing research may decide to start on the more popular, less difficult, and more recent Feature Transformer, instead of Quantization.

Education According to Figure 7.7, an ML testing educator may decide to teach the material for the less difficult and more popular Module Management before the more difficult and less popular Attention Mechanism, if there are no dependencies between them. The educator may also decide to prepare more material and spend more time both in the class and lab on the more difficult Attention Mechanism than Feature Transformer topic. The educator may continue teaching Module Management, because of its increased popularity over time and staying recent, and replace the less recent Variable Reuse with more recent Feature Transformer.

There are many more factors that go into tradeoffs that practitioners, educators and researchers make to decide where to focus their efforts. However, we believe our findings can contribute to inform and improve these decisions.

7.8 Threats to Validity

Some of our decisions during this study could be a threat to its validity. Using Python, ML libraries in $\mathscr{M}$, maturity and recency requirements, and GitHub to identify our ML projects and test cases could be a threat, by not selecting a complete set of ML projects. However, our use of similar criteria that previous work [81, 110, 97, 90]. uses often to identify similar projects and the large sizes 2,525 and 136,463 of our selected ML projects and test cases, respectively, could help mitigate this threat. Python and ML libraries in M are the most used programming language and libraries in opensource ML software [81, 110, 97]. GitHub, that is used by 93% of developers is the largest open-source software repository in existence [90]. Using the linguistic data of test cases to understand their topics could be a threat, by not representing the intention of their developers. However, our use of well-known approaches that previous work [55, 86, 174, 175, 183, 188, 189] uses often to

understand source code topics using its linguistic data could help mitigate this threat. Manual labeling of topic word sets and grouping of topics into categories could be a threat, by choosing incorrect labels and categories. However, our use of the well-known open card sort [99] that previous work uses often [64, 65, 49, 67] with the help of multiple authors, and the use of 20 topic words and 50 random test cases for labeling each topic could help mitigate this threat. Deciding the values for K topic number, I iterations number, and α and β hyperparameters of MALLET topic mining could another threat, for not selecting the most optimal values. However, our extensive 40 topic modeling experiments with different values of these parameters could help mitigate this threat. Previous work [64, 65, 49] uses similar techniques and values often.

To evaluate the consistency and reliability of topic naming process, I conducted an additional validation using a large language model. This approach follows the methodology used in several recent studies that employ foundation models to validate or label software-related artifacts, including Configuration Bugs Classification using LLMs and Encoders [199], Foundation Models for Automatic Issue Labeling [185], and Using Large Language Models to Support the Workflow of Differential Testing [104]. The prompt was specifically designed to function as an AI expert and only providing the model with: A test function and the chosen category assignment. The model's task was to grade (0–100%) how strongly it agreed with the chosen category without exposing any definitions that were already provided during the card sort process.

The GPT-5 model was used to evaluate 25 representative topics across five independent test cases drawn from diverse projects. The results demonstrated a high overall level of agreement: 20 out of 25 topics achieved strong consistency scores (above 85%), with an average agreement of approximately 90%. Several topics, including *Error Handling*, *File Management*, *Evaluation Metrics*, *Dimensionality Reduction*, *Graph Models*, and *Array*

Operations, received perfect or near-perfect validation scores, confirming that their definitions and assigned labels are well aligned with standard machine learning concepts.

Only two topics *Variable Reuse* and *Feedforward Neural Networks* showed lower agreement, while three others (*Module Management*, *Optimizers*, and *Attention Mechanism*) had moderate agreement levels. Table 7.7 shows the detailed results of the model's evaluation, illustrating the reasoning behind partial matches and supporting potential refinement of topic descriptions.

Topic	**Average Agreement (%)**	**Evaluation Outcome**
Error Handling	100	Fully Consistent
Array Operations	100	Fully Consistent
Graph Models	100	Fully Consistent
Variable Reuse	41	Low Agreement
Feedforward Neural Networks	54	Low Agreement
(remaining topics)	*85–100*	High Consistency

Table 7.7: Validation of topic labeling using GPT-5.

To further test robustness, intentionally incorrect label assignments were introduced. For example, assigning a *Graph Models* test case to *Diffusion Models* resulted in a zero agreement score with a detailed justification for rejection. This behavior confirmed that the model distinguishes clearly between semantically distinct categories. Overall, the high validation scores across most topics provide strong evidence that the topic labeling process is consistent, interpretable, and not substantially influenced by researcher bias.

CHAPTER EIGHT

CONCLUSION

In this work, we propose ***μAkka*** for mutation testing of Akka actor concurrency. To design, implement, and evaluate ***μAkka***, we manually analyze a set of 186 real Akka bugs, design 32 mutation operators to induce these bugs, implement these operators in an Eclipse plugin, generate 11.7k mutants of 10 real applications, measure the quality of mutants and effectiveness of tests, generate 26.2k PIT mutants and compare with ***μAkka***, analyze the bug coverage and overlap between ***μAkka***, PIT, and actor operators in Jagannath et al.'s, and discuss a few implications of our findings.

In addition, we conduct a large-scale study on GitHub to understand the ML test topics that developers write tests for, the popularity of these topics, the difficulty of writing tests for these topics, the correlation between the popularity and difficulty of these topics, the historical changes of popularity and difficulty of these topics, and the implications of such understanding for the practice, research, and education of ML software testing.

Future work could explore leveraging program context to predict and avoid the generation of lower-quality mutants, developing predictive mutation testing techniques for actor concurrency using machine learning, and triangulating our findings with prior research on understanding ML-related bugs.

REFERENCES

[1] Stack Overflow. Actor name is not unique invalidactornameexception. https://www.stackoverflow.com/questions/11693562.

[2] Stack Overflow. Actor supervised by BackoffSupervisor loses stashed messages after restart. https://www.stackoverflow.com/questions/53933363.

[3] Stack Overflow. actorselection with relative path messages going to dead letters. https://www.stackoverflow.com/questions/23835157.

[4] Stack Overflow. Akka actor - sender points to dead letters. https://www.stackoverflow.com/questions/43396596.

[5] Stack Overflow. Akka actor cannot send back the message. https://www.stackoverflow.com/questions/28211255.

[6] Stack Overflow. Akka actor not terminating if an exception is thrown. https://stackoverflow.com/questions/6170227.

[7] Stack Overflow. Akka actors always times out waiting for future. https://www.stackoverflow.com/questions/36219778.

[8] GitHub. Akka .conf file configuration to .properties file. https://stackoverflow.com/questions/43517113.

[9] Stack Overflow. Akka context.parent unexpected value. https://www.stackoverflow.com/questions/19972524.

[10] Stack Overflow. Akka Dead Letters with Ask Pattern. https://www.stackoverflow.com/questions/25402349.

[11] Stack Overflow. Akka exception handling. https://stackoverflow.com/questions/14820489.

[12] Stack Overflow. Akka: waiting for multiple messages. https://www.stackoverflow.com/questions/20151944.

[13] Stack Overflow. akka.io dispatcher configuration exception. https://stackoverflow.com/questions/21686327/akka-io-dispatcher-configuration-exception.

[14] GitHub. Appmaster failed to recover. https://github.com/gearpump/gearpump/commit/02b234363e1fcb3f799e864205ccb05e8a53ee1e.

[15] Assessment of class mutation operators for C++ with the MuCPP mutation system. *Information and Software Technology '17*, 81.

[16] Chapter six - mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers '19*.

[17] Stack Overflow. correctly terminate akka actors in scala. https://www.stackoverflow.com/questions/12324055.

[18] Stack Overflow. could not find implicit value for parameter system: akka.actor.ActorSystem. https://www.stackoverflow.com/questions/35202570.

[19] Stack Overflow. Dead-letter in akka scala actors. https://www.stackoverflow.com/questions/23327675.

[20] GitHub. Delay datawriters initialization after warmup. https://github.com/gatling/gatling/commit/ba139ec991fb05a282b07ddbf20287d83ca32ce0.

[21] GitHub. error log when launching local. https://github.com/gearpump/gearpump/commit/8065694e5b7006953eeaa2cf21f4cbe8d02fd652.

[22] GitHub. fix occasional deathpactexception in httphostconnector. https://github.com/spray/spray/commit/e34da115fa43d4d46db0e7ae06eea7fbcbc4fdfd.

[23] Mutating database queries. *Information and Software Technology '7*, 49(4).

[24] Stack Overflow. Outofmemoryerror using akka actors. https://www.stackoverflow.com/questions/31995994.

[25] Stack Overflow. Parent actor doesn't receive termination message when child is stopped with poisonpill. https://stackoverflow.com/questions/49736277.

[26] Porter stemming algorithm in NLTK 3.4.5. https://www.nltk.org/api/nltk.stem.html.

[27] Stack Overflow. Scala Akka Actor - Dead Letters encountered. https://www.stackoverflow.com/questions/53200356.

[28] Stack Overflow. Send message to actor after restart from Supervisor. https://www.stackoverflow.com/questions/48446194.

[29] Stack Overflow. Setting bounded mailbox for an actor that uses stashing. https://stackoverflow.com/questions/20419990.

[30] GitHub. small fix for Leader role handover. https://github.com/rkrzewski/akka-cluster-etcd/commit/9ca03223eea1a989927fcb57fd024e4741515d33.

[31] Stack Overflow. Spray Dead Letter msg. https://www.stackoverflow.com/questions/30740132.

[32] GitHub. The restart application REST api returns false. https://github.com/gearpump/gearpump/commit/c6ab9f9f07b9a81f976e6e5d7e5389d18f68b0b7.

[33] GitHub. Too fast messaging. https://github.com/codingteam/horta-hell/commit/f7b104e57deb399044304ebf667b5a708d579d27.

[34] Akka actors in Spark. https://issues.apache.org/jira/browse/SPARK-5293, 2015.

[35] GitHub. https://github.com/eclipse/ditto, July 2022.

[36] GitHub. https://github.com/project-sunbird/sunbird-lms-service, July 2022.

[37] GitHub. https://github.com/sane-city/wot-servient, July 2022.

[38] GitHub. https://github.com/zhihuili/flower, July 2022.

[39] GitHub. https://github.com/apache/servicecomb-saga-actuator, July 2022.

[40] GitHub. https://github.com/ryos-io/Rhino, July 2022.

[41] GitHub. https://github.com/eBay/parallec, July 2022.

[42] GitHub. https://github.com/apache/flink/tree/master/flink-rpc, July 2022.

[43] GitHub. https://github.com/gibffe/fuse, July 2022.

[44] GitHub. https://github.com/kiamesdavies/money-transfer, July 2022.

[45] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[46] Allen T. Acree, Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, Georgia Institute of Technology, 1979.

[47] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. 1986.

[48] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In *Foundations of Software Technology and Theoretical Computer Science '85*.

[49] Syed Ahmed and Mehdi Bagherzadeh. What do concurrency developers ask about? a large-scale study using stack overflow. In *ESEM '18*.

[50] Akka Actor Reference Config File. `https://github.com/akka/akka/blob/master/akka-actor/src/main/resources/reference.conf`, April 2020.

[51] Akka Typed. `https://doc.akka.io/docs/akka/current/typed/index.html`, May 2020.

[52] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv e-prints*, pages arXiv–1605, 2016.

[53] Bruno Correia Almeida and Paulo André Lima de Castro. Autonomous driving on a direct perception system with deep recurrent layers. In *Proceedings of the 2nd International Conference on Applications of Intelligent Systems*, APPIS '19, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3309772.3309790`.

[54] Moayad Alshangiti, Hitesh Sapkota, Pradeep K. Murukannaiah, Xumin Liu, and Qi Yu. Why is developing machine learning applications challenging? a study on stack overflow posts. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2019. `doi:10.1109/ESEM.2019.8870187`.

[55] Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. A comparative study of vectorization-based static test case prioritization methods. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 80–88, 2020. `doi:10.1109/SEAA51224.2020.00023`.

[56] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *ICST '14*.

[57] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 2008.

[58] Anonymous. Replication package. `https://www.dropbox.com/sh/m8hovmstq42r6n4/AACoFmpYJ6QKlB9llIgdRUEga?dl=0`.

[59] Anonymous. Number of Akka, Erlang, and Orleans projects in GitHub, 2022. Accessed on 2022-08-04, Akka: `https://tinyurl.com/yshr4ypn`, Erlang: `https://tinyurl.com/yz9zz25b`, and Orleans: `https://tinyurl.com/4jj9czry`.

[60] Anonymous. Number of akka, erlang, and orleans questions and answers in Stack Overflow, 2022. Accessed on 2022-08-04. URL: `https://tinyurl.com/ycxjjbb6`.

[61] GitHub API. https://docs.github.com/en/rest?apiVersion=2022-11-28, February 2024.

[62] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. 2007.

[63] M. Astley. The actor foundry: A Java-based actor programming environment. Open Systems Laboratory. University of Illinois at Urbana-Champaign, 1999.

[64] Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. Actor concurrency bugs: A comprehensive study on symptoms, root causes, api usages, and differences. *Proc. ACM Program. Lang.*, 4(OOPSLA).

[65] Mehdi Bagherzadeh and Raffi Khatchadourian. Going big: A large-scale study on what big data developers ask. In *ESEC/FSE 2019*.

[66] Mehdi Bagherzadeh and Hridesh Rajan. Order types: Static reasoning about message races in asynchronous message passing concurrency. In *AGERE '17*.

[67] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. Mining questions asked by web developers. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 112–121, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2597073.2597083, doi:10.1145/2597073.2597083.

[68] Lívia Barbosa and Andre Hora. How and why developers migrate python tests. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 538–548, 2022. doi:10.1109/SANER53432.2022.00071.

[69] Sebastien Bardin, Mickael Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis, Yves Le Traon, and Jean-Yves Marion. Sound and quasi-complete detection of infeasible test requirements. In *ICST '15*.

[70] Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. What are developers talking about? an analysis of topics and trends in Stack Overflow. *Empirical Softw. Engg.*, 19(3):619–654, June 2014. URL: http://dx.doi.org/10.1007/s10664-012-9231-y, doi:10.1007/s10664-012-9231-y.

[71] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. An empirical validation of cognitive complexity as a measure of source code understandability. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '20)*, pages Article 5, 12 pages, Bari, Italy, 2020. Association for Computing Machinery. doi:10.1145/3382494.3410636.

[72] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. Deflaker: Automatically detecting flaky tests. In *ICSE '18*.

[73] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry - a study at facebook. In *ICSE-SEIP '21*.

[74] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, 2014.

[75] Lauren R. Biggers, Cecylia Bocovich, Riley Capshaw, Brian P. Eddy, Letha H. Etzkorn, and Nicholas A. Kraft. Configuring latent dirichlet allocation based feature location. *Empirical Softw. Engg.*, 19(3):465–500, June 2014. URL: `http://dx.doi.org/10.1007/s10664-012-9224-x`, `doi:10.1007/s10664-012-9224-x`.

[76] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In *ASE '00*.

[77] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003. URL: `http://dl.acm.org/citation.cfm?id=944919.944937`.

[78] Robert L. Bocchino, Edward Gamble, Kim P. Gostelow, and Raphael R. Some. Spot: A programming language for verified flight software. *Ada Lett. 14*, 34(3).

[79] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars, 2016. `arXiv:1604.07316`.

[80] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. Mutation operators for concurrent Java (J2SE 5.0). In *Mutation '06*.

[81] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. The open-closed principle of modern machine learning frameworks. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 353–363, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3196398.3196445`.

[82] Timothy A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Inf. '82*, 18(1).

[83] G. Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 International Conference on Technical Debt (TechDebt '18)*,

pages 57–58, Gothenburg, Sweden, 2018. Association for Computing Machinery. doi:10.1145/3194164.3194186.

[84] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. Understanding performance problems in deep learning systems. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, 2022.

[85] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *ICSE '17*.

[86] Tse-Hsun Chen, Stephen W. Thomas, and Ahmed E. Hassan. A survey on the use of topic models when mining software repositories. *Empirical Software Engineering*, 21(5):1843–1919, 2016. doi:10.1007/s10664-015-9407-6.

[87] François Chollet et al. Keras: The python deep learning library. *Astrophysics source code library*, pages ascl–1806, 2018.

[88] Henry Coles. http://pitest.org/, 2023.

[89] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[90] Kyle Daigle. https://github.blog/2023-11-08-the-state-of-open-source-and-ai/, November 2023.

[91] Meenu Dave and Rashmi Agrawal. Mutation testing and test data generation approaches: A review. In *Smart Trends in Information Technology and Computer Communications '16*.

[92] M. E. Delamaro. *Proteum–A Test Environment Based on the Mutation Analysis*. PhD thesis, Masters thesis, University of Sao Paulo, Brazil, 1993.

[93] Pedro Delgado-Pérez, Sergio Segura, and Inmaculada Medina-Bulo. Assessment of C++ object-oriented mutation operators: A selective mutation approach. *Software Testing, Verification and Reliability '17*, 27.

[94] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer '78*, 11(4).

[95] Anna Derezinska and Karol Kowalski. Object-oriented mutation applied in common intermediate language programs originated from C. In *ICST '11*.

[96] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Featured model-based mutation analysis. In *ICSE '16*.

[97] Malinda Dilhara, Ameya Ketkar, and Danny Dig. Understanding software-2.0: A study of machine learning library usage and evolution. *ACM Trans. Softw. Eng. Methodol.*, 30(4), jul 2021. doi:10.1145/3453478.

[98] Mohamed Raed El aoun, Heng Li, Foutse Khomh, and Moses Openja. Understanding quantum software engineering challenges an empirical study on stack exchange forums and github issues. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 343–354, 2021. doi:10.1109/ICSME52107.2021.00037.

[99] Sally Fincher and Josh Tenenberg. Making sense of card sorting data. *Expert Systems '05*, 22(3).

[100] Pytest Testing Framework. https://docs.pytest.org/en/8.0.x/, February 2024.

[101] Unittest Testing Framework. https://docs.python.org/3/library/unittest.html, February 2024.

[102] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *ISSTA '10*.

[103] Kai Gao, Zhixing Wang, Audris Mockus, and Minghui Zhou. On the variability of software engineering needs for deep learning: Stages, trends, and application types. *IEEE Transactions on Software Engineering*, 49(2):760–776, 2023. doi:10.1109/TSE.2022.3163576.

[104] Marco Garcia, Dhruv Patel, and Linh Nguyen. Using large language models to support the workflow of differential testing. https://fse2025.org/papers/llm-differential-testing, September 2025. To appear; Accessed: 2025-10-25.

[105] GitHub. Ai model efficiency toolkit (aimet). https://github.com/quic/aimet, March 2024. Accessed: 2025-09-16.

[106] GitHub. Lambda packs. https://github.com/ryfeus/lambda-packs/, March 2024. Accessed: 2025-09-16.

[107] GitHub. Pytorch geometric. https://github.com/pyg-team/pytorch_geometric, March 2024. Accessed: 2025-07-24.

[108] Milos Gligoric, Vilas Jagannath, and Darko Marinov. Mutmut: Efficient exploration for mutation testing of multithreaded code. In *ICST '10*.

[109] Milos Gligoric, Lingming Zhang, Cristiano Pereira, and Gilles Pokam. Selective mutation testing for concurrent code. In *ISSTA '13*.

[110] Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. The state of the ml-universe: 10 years of artificial intelligence & machine learning software development on github. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*, pages 431–442, Seoul, Republic of Korea, 2020. Association for Computing Machinery. doi:10.1145/3379597.3387473.

[111] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. How hard does mutation analysis have to be, anyway? In *ISSRE '15*.

[112] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. Measuring effectiveness of mutant sets. In *ICSTW '16*.

[113] Rahul Gopinath, Philipp Görz, and Alex Groce. Mutation analysis: Answering the fuzzing challenge. *arXiv preprint arXiv:2201.11303*, 2022.

[114] Bernhard J. M. Grün, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *ICSTW '09*.

[115] Brandon Hedden and Xinghui Zhao. A comprehensive study on bugs in actor systems. In *ICPP '18*.

[116] Abram Hindle and Karim Ali. What do developers know about machine learning: A study of ml discussions on stackoverflow. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 260–264, 2019. doi:10.1109/MSR.2019.00052.

[117] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*, pages 1110–1121, Seoul, South Korea, 2020. Association for Computing Machinery. doi:10.1145/3377811.3380395.

[118] Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. Deepcrime: Mutation testing of deep learning systems based on real faults. In *ISSTA '21*.

[119] Nick Hynes, D. Sculley, and Michael Terry. The data linter: Lightweight, automated sanity checking for ml data sets. In *Proceedings of the NIPS Workshop on Machine Learning Systems (MLSys)*, volume 1, 2017.

[120] Test Discovery in Pytest. https://docs.pytest.org/en/7.1.x/explanation/goodpractices.html, February 2024.

[121] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 510–520, New York, NY, USA, 2019. ACM. URL: http://doi.acm.org/10.1145/3338906.3338955, doi:10.1145/3338906.3338955.

[122] Vilas Jagannath, Milos Gligoric, Steven Lauterburg, Darko Marinov, and Gul Agha. Mutation operators for actor systems. In *ICSTW '10*.

[123] Java Akka API. https://doc.akka.io/japi/akka/current/, June 2022.

[124] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[125] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *TSE '11*, 37(5).

[126] Kyle D. Julian, Jessica Lopez, Jeffrey S. Brush, Michael P. Owen, and Mykel J. Kochenderfer. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–10, 2016. doi:10.1109/DASC.2016.7778091.

[127] René Just, Bob Kurtz, and Paul Ammann. Inferring mutant utility from program context. In *ISSTA '17*.

[128] Rene Just, Franz Schweiggert, and Gregory M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *ASE '11*.

[129] Nadia Kasto and Jacqueline Whalley. Measuring the difficulty of code comprehension tasks using software metrics. In *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136 (ACE '13)*, pages 59–65, Adelaide, Australia, 2013. Australian Computer Society, Inc.

[130] Samuel J. Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. Prioritizing mutants to guide mutation testing. In *ICSE '22*.

[131] Misoo Kim, Youngkyoung Kim, and Eunseok Lee. Denchmark: A bug benchmark of deep learning-related software. In *2021 IEEE/ACM 18th International Conference*

on Mining Software Repositories (MSR), pages 540–544, 2021. doi:10.1109/MSR52588.2021.00070.

[132] Marinos Kintis, Mike Papadakis, and Nicos Malevris. Evaluating mutation testing alternatives: A collateral experiment. In *APSEC '10*.

[133] Bob Kurtz, Paul Ammann, Marcio E. Delamaro, Jeff Offutt, and Lin Deng. Mutant subsumption graphs. In *ICSTW '14*.

[134] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. Are we there yet? how redundant and equivalent mutants affect determination of test completeness. In *ICSTW '16*.

[135] Markus Kusano and Chao Wang. Ccmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In *ASE '13*.

[136] Fred Lambert. Understanding the fatal tesla accident on autopilot and the nhtsa probe. https://electrek.co/2016/07/01/understanding-fatal-tesla-accident-autopilot-nhtsa-probe/, July 2016.

[137] Thomas Laurent, Mike Papadakis, Marinos Kintis, Christopher Henard, Yves Le Traon, and Anthony Ventresque. Assessing and improving the mutation testing practice of pit. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 430–435, 2017. doi:10.1109/ICST.2017.47.

[138] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. Mucheck: An extensible tool for mutation testing of haskell programs. In *ISSTA '14*.

[139] Nan Li, Michael West, Anthony Escalona, and Vinicius H. S. Durelli. Mutation testing in practice using ruby. In *ICSTW '15*.

[140] Zengyang Li, Sicheng Wang, Wenshuo Wang, Peng Liang, Ran Mo, and Bing Li. Understanding bugs in multi-language deep learning frameworks. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, pages 328–338. IEEE, 2023.

[141] Lightbend. Customer case studies. https://www.lightbend.com/case-studies#filter:akka, 2019.

[142] Lightbend. How Groupon scales personalized offers to 48 million customers on time. https://www.lightbend.com/case-studies/groupon-scalability-personalized-offers-to-48-million-customers, 2020.

[143] Lightbend. PayPal blows past 1 billion transactions per day using just 8 VMs with Akka, Scala, Kafka and Akka Streams. https://www.lightbend.com/case-studies/paypal-blows-past-1-billion-transactions-per-day-using-just-8-vms-and-akka-scala-kafka-and-akka-streams, 2020.

[144] Lightbend. Akka. https://www.lightbend.com/akka-platform, 2022.

[145] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Enabling mutation testing for android apps. In *ESEC/FSE '17*.

[146] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen A.W.M. van der Laak, Bram van Ginneken, and Clara I. Sánchez. A survey on deep learning in medical image analysis. *Medical Image Analysis*, 42:60–88, 2017. URL: `https://www.sciencedirect.com/science/article/pii/S1361841517301135`, `doi:10.1016/j.media.2017.07.005`.

[147] Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. A study of concurrency bugs and advanced development support for actor-based programs. In *Programming with Actors - State-of-the-Art and Research Perspectives '18*.

[148] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for java. In *ISSRE '02*.

[149] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: An automated class mutation system: Research articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, jun 2005.

[150] José Carlos Maldonado, Ellen Francine Barbosa, Auri Marcelo Rizzo Vincenzi, and Márcio Eduardo Delamaro. *Evaluating N-Selective Mutation for C Programs: Unit and Integration Testing*.

[151] Dongyu Mao, Lingchao Chen, and Lingming Zhang. An extensive study on cross-project predictive mutation testing. In *ICST '19*.

[152] Thomas J. McCabe. A complexity measure. In *Software Quality Analysis and Guidelines*, pages 22–51. McGraw-Hill, Inc., USA, 1993.

[153] Andrew Kachites McCallum. Mallet: A machine learning for language toolkit. `http://mallet.cs.umass.edu`, 2002.

[154] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Efficient javascript mutation testing. In *ICST '13*.

[155] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2574–2582, 2016.

[156] Mohsen Moradi Moghadam, Mehdi Bagherzadeh, Raffi Khatchadourian, and Hamid Bagheri. muakka: Mutation testing for actor concurrency in akka using real-world bugs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 262–274, New York, NY, USA, 2023. Association for Computing Machinery. URL: `https://doi.org/10.1145/3611643.3616362`.

[157] Mohammad Mehdi Morovati, Amin Nikanjam, Foutse Khomh, and Zhen Ming (Jack) Jiang. Bugs in machine learning-based systems: a faultload benchmark. *Empirical Softw. Engg.*, 28(3), apr 2023. `doi:10.1007/s10664-023-10291-1`.

[158] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do Java developers struggle with cryptography APIs? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 935–946, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2884781.2884790`.

[159] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *ICSE '93*.

[160] A. Jefferson Offutt and Ronald H. Untch. *Mutation 2000: Uniting the Orthogonal*.

[161] Thor Olavsrud. 9 famous analytics and ai disasters. `https://www.cio.com/article/190888/5-famous-analytics-and-ai-disasters.html`, September 2023.

[162] Elmahdi Omar and Sudipto Ghosh. An exploratory study of higher order mutation testing in aspect-oriented programming. In *ISSRE '12*.

[163] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. Mutant quality indicators. In *ICSTW '18*.

[164] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *ISSTA '16*.

[165] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE '15*.

[166] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *ICSE '18*.

[167] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. A survey of flaky tests. *ACM Trans. Softw. Eng. Methodol. '21*, 31(1).

[168] Parso Python Parser. `https://pypi.org/project/parso/`, February 2024.

[169] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[170] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. *Communications of the ACM*, 62(11):137–145, October 2019. `doi:10.1145/3361566`.

[171] Norman Peitek, Janet Siegmund, Sven Apel, Christian Kästner, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. A look into programmers' heads. *IEEE Transactions on Software Engineering*, 46:442–462, 2020. `doi:10.1109/TSE.2018.2877766`.

[172] Goran Petrović and Marko Ivanković. State of mutation testing at google. In *ICSE-SEIP '18*.

[173] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and Rene Just. Practical mutation testing at scale: A view from google. *TSE '21*.

[174] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. What is the vocabulary of flaky tests? In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*, pages 492–502, 2020. `doi:10.1145/3379597.3387482`.

[175] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007. `doi:10.1109/TSE.2007.1016`.

[176] Upsorn Praphamontripong, Jeff Offutt, Lin Deng, and JingJing Gu. An experimental evaluation of web mutation operators. In *ICSTW '16*.

[177] Python Software Foundation. Python language reference: Keywords. `https://docs.python.org/3/reference/lexical_analysis.html#keywords`, March 2024. Accessed: 2025-07-24.

[178] Henrique Rebêlo, Gary T. Leavens, Mehdi Bagherzadeh, Hridesh Rajan, Ricardo Lima, Daniel M. Zimmerman, Márcio Cornélio, and Thomas Thüm. Modularizing crosscutting contracts with aspectjml. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, page

21–24, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2584469.2584476.

[179] Michael Roberts, Derek Driggs, Matthew Thorpe, Julian Gilbey, Michael Yeung, Stephan Ursprung, Angelica I Aviles-Rivero, Christian Etmann, Cathal McCague, Lucian Beer, et al. Common pitfalls and recommendations for using machine learning to detect and prognosticate for covid-19 using chest radiographs and ct scans. *Nature Machine Intelligence*, 3(3):199–217, 2021.

[180] Christoffer Rosen and Emad Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Softw. Engg.*, 21(3):1192–1223, June 2016. doi:10.1007/s10664-015-9379-3.

[181] Ebert Schoofs, Mehrdad Abdi, and Serge Demeyer. Ampyfier: Test amplification in python. *Journal of Software: Evolution and Process*, 34(11):e2490, 2022. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2490, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2490, doi:10.1002/smr.2490.

[182] Gian Luca Scoccia, Patrizio Migliarini, and Marco Autili. Challenges in developing desktop web apps: a study of stack overflow and github. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 271–282, 2021. doi:10.1109/MSR52588.2021.00039.

[183] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD '07)*, pages 212–224, 2007. doi:10.1145/1218563.1218587.

[184] Camila Costa Silva, Matthias Galster, and Fabian Gilson. Topic modeling in software engineering research. *Empirical Software Engineering*, 26(6):62, November 2021. doi:10.1007/s10664-021-10026-0.

[185] Alan Smith, Yue Chen, and Naila Rahman. Foundation models for automatic issue labeling. https://icse2025.org/papers/foundation-models-labeling, May 2025. To appear; Accessed: 2025-10-25.

[186] Stack Overflow. Dead-letter in akka scala actors. https://stackoverflow.com/questions/23327675. Accessed: 2025-07-24.

[187] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. An empirical study of refactorings and technical debt in machine learning systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 238–250, 2021. doi:10.1109/ICSE43902.2021.00033.

[188] Stephen W. Thomas, Hadi Hemmati, Ahmed E. Hassan, and Dorothea Blostein. Static test case prioritization using topic models. *Empirical Software Engineering*, 19(1):182–212, 2014. doi:10.1007/s10664-012-9219-7.

[189] Stephen W. Thomas, Meiyappan Nagappan, Dorothea Blostein, and Ahmed E. Hassan. The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering*, 39(10):1427–1443, 2013. doi:10.1109/TSE.2013.27.

[190] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 271–280, 2012. doi:10.1109/ISSRE.2012.22.

[191] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*, pages 303–314, Gothenburg, Sweden, 2018. Association for Computing Machinery. doi:10.1145/3180155.3180220.

[192] Asher Trockman, Keenen Cates, Mark Mozina, Tuan Nguyen, Christian Kästner, and Bogdan Vasilescu. "automatically assessing code understandability" reanalyzed: Combined metrics matter. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*, pages 314–318, Gothenburg, Sweden, 2018. Association for Computing Machinery. doi:10.1145/3196398.3196441.

[193] Tatiana Castro Vélez, Raffi Khatchadourian, Mehdi Bagherzadeh, and Anita Raja. Challenges in migrating imperative deep learning programs to graph execution: an empirical study. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 469–481, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3524842.3528455.

[194] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, pages 1–27, 2024. doi:10.1109/TSE.2024.3368208.

[195] Xin-Li Yang, David Lo, Xin Xia, Zhi-Yuan Wan, and Jian-Ling Sun. What security questions do developers ask? a large-scale study of Stack Overflow posts. *Journal of Computer Science and Technology*, 31(5):910–924, Sep 2016. doi:10.1007/s11390-016-1672-0.

[196] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. Predictive mutation testing. In *ISSTA '16*.

[197] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Trans. Softw. Eng.*, 48(1):1–36, jan 2022. doi:10.1109/TSE.2019.2962027.

[198] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. Operator-based and random mutant selection: Better together. In *ASE '13*.

[199] Wei Zhang, Priya Kumar, and Thomas Lee. Configuration bugs classification using llms and encoders. https://msr2025.org/papers/config-bugs-llm, June 2025. To appear; Accessed: 2025-10-25.

[200] Yang Zhang, Yiwen Wu, Tingting Chen, Tao Wang, Hui Liu, and Huaimin Wang. How do developers talk about github actions? evidence from online software development community. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3597503.3623327.

[201] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 129–140, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3213846.3213866, doi:10.1145/3213846.3213866.

[202] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, pages 129–140, Amsterdam, Netherlands, 2018. Association for Computing Machinery. doi:10.1145/3213846.3213866.