Beyond the Noise: Leveraging Machine Learning vs LLMs to Prioritize ASAT Warnings based on Actionability Probability

Master Thesis

Vivian Yajing Ning

Beyond the Noise: Leveraging Machine Learning vs LLMs to Prioritize ASAT Warnings based on Actionability Probability

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Vivian Yajing Ning
born in Rotterdam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Software Improvement Group
Fred. Roeskestraat 115
Amsterdam, the Netherlands
https://www.softwareimprovementgroup.com

Beyond the Noise: Leveraging Machine Learning vs LLMs to Prioritize ASAT Warnings based on Actionability Probability

Author: Vivian Yajing Ning
Student id: 5273404

Abstract

Automated Static Analysis Tools (ASATs) generate a massive volume of non-actionable warnings. To address this, this thesis investigates the performance and resource trade-offs between classical Machine Learning (ML) models and Large Language Models (LLMs) for generating actionability probability scores. Utilizing the NASCAR dataset of over 1.2 million Java warnings, we evaluate optimized classical models (Random Forest and Logistic Regression) against the Claude 4.x LLM family using classification metrics (F1-score, AUC) and probabilistic calibration (Brier scores), supplemented by a qualitative user study of 15 industry professionals. Empirical results demonstrate that an optimized Random Forest yields superior predictive performance (F1-score: 76.85%, AUC: 0.87) and reliable uncertainty calibration (Brier score: 0.1549), rendering the massive computational overhead of miscalibrated LLMs unnecessary. However, the user study identifies a human-AI feature disconnect: while the Random Forest relies heavily on historical metadata, developers universally demand source code context and severity indicators. Ultimately, an optimized Random Forest provides a significantly more efficient framework for scoring ASAT warnings, provided the scores are tightly coupled with the structural evidence required to sustain human trust.

Thesis Committee:

Chair & University supervisor :	Prof. Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Company supervisor:	Dr. M. Zivkovic, Software Improvement Group
Committee Members:	Dr. A. Costea, Faculty EEMCS, TU Delft
	Dr. Z. Erkin, Faculty EEMCS, TU Delft

Preface

This thesis signifies the culmination of my academic journey at the Delft University of Technology. Here my Master's program in Computer Science provided the invaluable opportunity to deepen my knowledge by exploring how to apply Machine Learning within the context of software engineering. While this journey has seen its share of successes and challenges, navigating through it all has allowed me to learn a great deal about both academia and the professional world. Additionally, this journey has allowed me to discover how to effectively translate theoretical concepts into real-world applications.

None of this would have been possible without the support and guidance of several key individuals. First, I would like to express my sincere gratitude to my university supervisor, Prof. Dr. Andy E. Zaidman, who has provided incalculable value in the form of excellent supervision and feedback. Furthermore, I would like to thank my company supervisor, Dr. Miroslav Zivkovic, for his dedicated weekly supervision at the Software Improvement Group (SIG). Your guidance and practical insights throughout this process were truly indispensable.

I am also deeply grateful to my friends, boyfriend, and family. Their immense love, patience, and encouragement kept me grounded, and I cannot imagine successfully completing this milestone without their unwavering support.

Finally, I dedicate this thesis to the memory of my youngest sister, Jessica.
You were taken too soon, but may you rest in peace.
We will find each other in the next life.

Vivian Yajing Ning
Delft, the Netherlands
June 22, 2026

Contents

List of Figures

Chapter 1

Introduction

In the current digital era, software serves as the foundational infrastructure of daily life, making the assurance of code quality and security a critical engineering priority. The consequences of undetected defects were starkly highlighted by the 2021 Log4Shell vulnerability, where a single Remote Code Execution flaw within a standard Java logging library exposed a vast majority of cloud environments to potential system takeover [54]. While critical vulnerabilities like Log4Shell emphasize the necessity of rigorous code inspection, the massive scale of modern enterprise software makes the manual inspection of every line of code impossible. To systematically identify flaws early in the development lifecycle and mitigate these risks, the software engineering community has established Software Quality Assurance (SQA) as a specialized discipline dedicated to the systematic monitoring and evaluation of code integrity. It relies heavily on automated methodologies to identify flaws early in the development lifecycle [11, 28].

One of these methodologies is Static Code Analysis (SCA), specifically Automated Static Analysis Tools (ASATs). These tools detect not only potential vulnerabilities in written code, but also style violations and structural flaws [28, 56]. Developers use ASATs to scan source code for pre-defined bug-patterns[1], and then report all code units that match these patterns [11]. Many of these tools (e.g., SpotBugs, PMD) are designed as a lightweight and flexible tool, so they can be run from the command line or plug into standard IDEs such as Eclipse or IntelliJ IDEA [28]. This allows developers to extract a set of warnings from their code-base and manually review, understand, and fix them at any point in time [37].

The outputs produced by ASATs are typically presented as warnings, although the literature uses a variety of terms to refer to them: e.g., issues, violations, alerts, bugs, or findings [28, 37]. These warnings represent code elements that match predefined patterns or rules, and are intended to signal potential quality concerns. However, not all reported warnings correspond to real defects or actionable improvements. Many are false positives (FPs), meaning that the tool reports a problem where none exists. Or it concerns non-actionable warnings, where the issue is technically flagged but carries no practical relevance for the developers responsible for the system [28, 56]. This variation in terminology reflects differences in tooling, research areas, and industrial practice. However, all terms refer to the

[1] These patterns are summarized by software-quality experts [28].

same underlying concept: Automated warnings meant to guide developers toward potential problems in code. In the context of this research all FPs are regarded together with non-actionable warnings. Hence forth when talking about non-actionable warnings, we also talk about FPs.

Empirical research [28, 81] has shown that 30-100% of these warnings are non-actionable. Given that developers require approximately five minutes to manually triage a single warning [30, 81], and that static analysis tools can generate upwards of 40 alerts per thousand lines of code (KLOC) [30], the labor cost is substantial. For instance, inspecting 3,000 warnings may require 250 hours of manual effort with at least 75 hours lost to non-actionable warnings [81]. This overhead transforms ASATs from a helpful quality measure into a severe bottleneck within the Software Development Life Cycle (SDLC) [34]. When these tools are integrated directly into Continuous Integration (CI) pipelines, the high volume of low-precision alerts creates operational friction. To maintain necessary delivery velocity, development teams are frequently forced to spot-check results or adopt a blanket policy of suppressing warnings entirely [38]. Consequently, the successful implementation of ASATs within an enterprise environment depends less on their raw detection capabilities and more on providing prioritized, high-context feedback [34, 50, 65, 73]. Minimizing this manual triage burden is essential to lower the barrier to effective ASAT adoption and ensure that critical security threats are systematically addressed rather than bypassed.

This prevalence of non-actionable warnings in ASATs is thus a well-documented bottleneck in the SDLC. Consequently, extensive research has focused on developing systematic mitigation strategies to alleviate this triage burden [28]. While literature explores a wide variety of techniques to refine, test, or group tool outputs, ML has emerged as a particularly prominent approach. By utilizing predictive models, it aims to automatically classify warnings as either actionable or non-actionable [28]. This thesis focuses exclusively on this ML domain, which can be diverged into two distinct methodologies: Classical ML (e.g., Random Forest) and Deep Learning (DL, e.g., Large Language Models (LLMs)) [28].

Within the DL paradigm, this research specifically narrows its focus to LLMs rather than developing or training custom neural network architectures from scratch. The rationale for this is twofold: first, pre-trained LLMs eliminate the need to reinvent complex DL architectures while offering out-of-the-box, state-of-the-art performance on code-related tasks. Second, the software industry is rapidly adopting these foundational models. Enterprise organizations and governance platforms are actively integrating LLM-based generative AI tools and coding agents, such as Claude Code, GitHub Copilot, and Cursor, directly into developer environments [4, 42, 75]. So while classical ML offers high interpretability and lower computational overhead, LLMs provide unprecedented semantic comprehension capabilities. By leveraging deep contextual architectures, LLMs can capture subtle language abstractions, structural dependencies, and developer intent embedded directly within the code and metadata [74]. These are nuances that traditional vectorization techniques often compress or lose [12, 25]. However, the adoption of LLMs introduces a critical trade-off. Despite their advantages, LLMs are computationally expensive and prone to hallucinations [75]. This is a fatal flaw when an incorrect classification results in either a wasted manual effort or a catastrophic oversight.

To bridge the gap between theory and practical application, this research is conducted

in collaboration with the Software Improvement Group (SIG). As an industry leader in software portfolio governance, and recently named a Leader in the 2026 Gartner® Magic Quadrant™ for Technical Debt Management Tools [61], SIG empowers enterprise organizations to continuously measure, manage, and control their software quality. Their flagship platform, Sigrid®, operates the world's only ISO/IEC 17025-accredited software quality lab. It utilizes ASATs to evaluate, among others, architecture quality, software security, and maintainability by benchmarking client code against a massive database of over 30,000 real-world systems and 400 billion lines of code [27, 62].

While Sigrid effectively identifies problematic code segments using expert-curated metrics and presents them as prioritized, actionable insights, the underlying ASATs feeding into these types of enterprise systems still inherently produce a high volume of non-actionable warnings. Because SIG operates at such a massive industrial scale, they provide the perfect real-world environment to study this exact bottleneck. Even with advanced dashboards, the number of non-actionable warnings generated by raw static analysis continues to pose a real productivity and trust challenge for development teams. Therefore, partnering with SIG provides this thesis with the critical industrial context and scale.

1.1 Goals and Research Questions

The primary goal of this thesis is thus to identify the optimal balance between performance and resource efficiency when generating actionability probability scores for ASAT warnings. By comparing ML and LLMs, this research aims to determine which approach most reliably predicts whether a warning is actionable. This in turn would allow software engineers to prioritize their triage efforts while remaining computationally viable for real-world enterprise environments. Crucially, this optimization must be balanced against the absolute necessity of retaining genuinely actionable alerts. A prioritization strategy that successfully suppresses non-actionable noise but inadvertently down-ranks genuinely actionable warnings would be counterproductive and pose a severe risk to the software's integrity. Therefore, the evaluation focuses on maximizing precision without sacrificing the high recall required to ensure that a project's security and maintainability posture remains intact. Ultimately, this research seeks to lower the barrier to effective ASAT adoption by minimizing the friction caused by non-actionable alerts, ensuring that critical issues remain highly visible through high probability scores. Accordingly, the following main research question is addressed:

Main Research Question

What are the performance and resource trade-offs between zero/one-shot LLMs and optimized classical ML models when generating actionability probability scores for ASAT warnings?

To systematically answer this question, the research is divided into two primary subquestion groups, each targeting a specific facet of the main research question:

RQ1: Model Selection and Feature Engineering

Before a comparison can be made, the most effective candidates from each domain must be established. These questions investigate how representative models and feature modalities can be optimally identified.

RQ1.1 Which classical ML model achieves the optimal trade-off between predictive performance and training efficiency?

RQ1.2 Which LLMs provide the most effective representations for actionability prediction under low-resource fine-tuning?

RQ1.3 How does the choice of features (metadata or code) influence model selection?

RQ2: Comparative Performance and Robustness

Once the optimal models are identified, they must be evaluated. These questions thus examine how the selected models differ in terms of predictive quality, reliability, and generalization potential.

RQ2.1 How do the models compare across standard performance metrics (accuracy, precision, recall, and F1) in a controlled environment?

RQ2.2 How well-calibrated are the models' probability estimates (measured via Brier scores), and can they be trusted to signal their own uncertainty?

The remainder of this thesis is structured as follows. Chapter 2 provides the background, alongside a review of related literature, and a detailed explanation of the research gap. Chapter 3 details the methodology, outlining the dataset selection, the distinct preprocessing pipelines for the classical ML and DL models, and the evaluation metrics. Chapter 4 then defines the study setup, specifying the experimental phases and the framework for the qualitative user study. Next, Chapter 5 presents the empirical results of these experiments alongside the findings from the user study. Subsequently, Chapter 6 synthesizes these results to discuss the performance trade-offs, model calibration, and practical software engineering implications, while addressing the potential threats to validity. Finally, Chapter 7 provides the conclusions of the research and outlines potential avenues for future work.

Chapter 2

Background and Related Work

This chapter provides the necessary foundation to contextualize the research conducted in this thesis. It is divided into two primary sections. Section 2.1 begins with the mechanics of Static Code Analysis (SCA) and the challenges associated with Automated Static Analysis Tools (ASATs). Then, Section 2.2 reviews existing literature and contemporary research relevant to this research. It highlights the current state of the art. Finally, Section 2.3 identifies the gaps this thesis aims to address.

2.1 Static Code Analysis

Static Code Analysis (SCA) is a process in which a program is checked for errors without executing it [34, 45]. The domain of SCA can be divided into roughly five categories: Linters, General Bug Finders, Type Checkers, SAST (Static Application Security Testing), and Software Composition Analysis[1]. Linters enforce stylistic consistency and identify basic smells, and Type Checkers ensure data consistency across the whole codebase [13]. In contrast, General Bug Finders[2] employ more sophisticated data-flow and control-flow analysis to detect complex logical errors such as null pointer dereferences [50, 65]. SAST tools focus specifically on security by tracing untrusted data through the application to identify vulnerabilities [24]. Finally, Software Composition Analysis examines third-party dependencies rather than the proprietary source code itself. For the purpose of this thesis, the focus is placed on General Bug Finders (often represented by ASATs such as SpotBugs and PMD), and SAST tools. The emphasis on SAST is driven by the specific industrial context of this research, as the collaborating company (SIG) has provided a specialized evaluation dataset of their SAST tool (Sigrid) for empirical analysis.

ASATs work by comparing the code in the program to a list of known errors [45]. This list is typically curated and maintained by groups of domain experts, security researchers, and community contributors. They program common pitfalls, security vulnerabilities, code style violations, anti patterns, and code smells into formal rules or patterns [70]. However,

[1]This is sometimes referred to as SCA, however for the purpose of this thesis SCA means Static Code Analysis.

[2]Frequently called ASATs in scientific literature. [70]

because these tools rely on predefined rules sets, no single checker is capable of catching every possible defect. In the end, a tool is only as effective as the patterns it has been programmed to recognize. Consequently, different languages have developed specialized sets of ASATs. They can be used through an Integrated Development Environment (IDE), or part of the Continuous Integration (CI) pipeline where they can be set up to stop the build if too many warnings have been found [34, 70]. For the purpose of this thesis, the focus is placed on Java. Specifically the two most prominent ASATs: PMD and SpotBugs [34]. PMD is a widely adopted source code analyzer that identifies common programming flaws, such as unused variables or empty catch blocks, by parsing code into an Abstract Syntax Tree (AST). It comes with 400+ built-in rules and can be used for multiple coding languages [53]. However, for the purpose of this research, only warnings concerning Java are considered. In contrast, SpotBugs (the successor of FindBugs) operates at the bytecode level [63]. So it uses static analysis to look for bug patterns, instances of code that are likely to be errors based on statistical likelihood and known vulnerabilities [9, 63].

These tools provide not just a flat list of warnings, but they also categorize them into distinct groups such as *Performance*, *Multi-threading*, and *Security*. This allows for developers to prioritize critical fixes. Still, a common problem these tools face is that the generated warnings are of high volume (no matter the category) [34, 50, 65]. Making it hard for developers to distinguish between actionable and unactionable warnings, this can lead to a phenomenon known as alert fatigue or the dis- or underuse of the tool [39, 50, 65]. It occurs when a developer is exposed to a high volume of warnings, many of which can be low-priority or unactionable (false positive), leading to sensory overload and a subsequent desensitization to the tool's output [39, 52]. When ASATs flood the development environment, the load required to triage each warning becomes excessive. This can force developers to spot-check results or adopt a blanket policy of suppressing warnings to maintain delivery velocity [34, 35, 65]. Research indicates that this fatigue does not merely impact productivity, but also overall morale as the labor-intensive process of manually validating unactionable warnings creates a frustrating and repetitive workflow [57]. Consequently, the success of ASATs are often less about its raw detection rate and more about its ability to provide actionable, high-context feedback that minimizes this burden.

2.2 Related Work

The challenge of reducing noise in ASAT warnings has driven extensive research into post-processing. This section reviews the evolution of these mitigation strategies, from statistical solutions to classical ML to modern DL models.

2.2.1 The False Positive Problem in ASATs

While ASATs are highly effective at identifying defects early in the SDLC without executing the program [26, 30, 43], developers consistently cite high false positive rates as a primary barrier to their practical adoption [38, 65, 79]. As previously mentioned in Section 2.1 ASATs rely on abstractions and conservative over-approximations to ensure they

can scale to complex codebases without missing real errors. This means that they inherently produce a massive volume of spurious warnings [37, 38, 28].

The scale of this information overload is substantial. Empirical research shows that the false positive or non-actionable rate of these tools typically ranges from 30% to over 90% [28, 30, 32, 79]. In certain enterprise environments, ASATs can generate up to 40 alerts per thousand lines of code (KLOC). Given that developers require an estimated five minutes to manually inspect and validate a single alert, reviewing thousands of warnings can consume hundreds of hours of labor [30, 81]. Faced with this severe overhead, developers often conclude that the time spent painstakingly analyzing thousands of error reports, the vast majority of which are false, outweighs the benefit of catching potential bugs [38, 81].

To combat this bottleneck, the software engineering community has shifted focus toward Actionable Alert Identification Techniques (AAITs) and False Positive Mitigation (FPM) approaches. Rather than refining the highly complex underlying ASAT algorithms, these post-processing techniques aim to automatically filter, classify, or prioritize the generated warnings from the user's perspective [26, 28, 30]. This involves distinguishing between actionable warnings, genuine flaws that developers will actually spend time fixing, and non-actionable warnings, which encompass both factual false positives and trivial issues that developers systematically ignore because they do not impact the software's functionality [26, 41, 79].

Early statistical approaches to this problem demonstrated that the underlying characteristics of the warnings themselves could be exploited to manage this triage burden [28, 30, 32]. For example, the z-ranking technique successfully demonstrated that prioritizing warnings based on the frequency of a rule's successful checks versus failed checks could push true errors to the top of a developer's inspection queue. It does this by establishing that real errors typically exhibit a low number of failed checks compared to successes. So while analysis approximations generally result in an explosion of failures, z-ranking provided an effective mathematical foundation for pushing invalid errors to the bottom of the list [30, 40]. Building upon these statistical foundations, other probabilistic techniques emerged, such as Feedback-Rank, which exploited correlations and code locality among warning reports. As well as models leveraging execution likelihood or historical software change data to prioritize and assign suspiciousness scores [28].

Beyond statistical probability, researchers have also explored static program analysis enhancements like bounded model checking to further prune false positives, dynamic program testing to generate test cases that confirm warnings, and clustering algorithms designed to group similar warnings together so developers only need to review a single dominant issue per cluster [28, 30]. Another strategy for managing the triage burden shifts the focus from statistical suspiciousness to practical refactoring effort. Steidl and Eder [64] proposed prioritizing maintainability defects, specifically code clones and long methods, based on their expected low costs of removal. Rather than strictly filtering false positives, their approach uses heuristic dataflow analysis to recommend defects that present easy refactoring opportunities. This would provide developers with an immediate, actionable starting point. In an evaluation of industrial Java systems, developers were willing to remove 80% of the recommended findings, demonstrating high practical utility. Crucially, their qualitative findings also revealed that the type of context developers require to validate a warning

depends heavily on the scope of the analysis. While local defects like long methods were primarily evaluated based on the nature of the code itself, prioritizing global defects like code clones required significantly more external context information, such as overall design decisions. [64] While these diverse strategies have provided valuable improvements, the most prominent and adaptable solutions have emerged within the fields of classical ML and DL [28], which will be the primary focus of this thesis.

2.2.2 Classical Machine Learning for ASATs

To improve prioritization and mitigate the severe triage burden, researchers framed warning triage as a binary classification problem using classical ML models [28]. These approaches primarily rely on hand-engineered features extracted from warning metadata, software metrics, and code history [2, 38]. Additionally, rather than following explicitly programmed rules, these ML models learn from existing data to build predictive models that can manage and classify new, unseen instances [80]. While defining these diverse features requires significant human expertise [38], they have proven highly effective in distinguishing actionable from non-actionable warnings.

Early evaluations extensively benchmarked various models and feature sets to identify optimal classifiers. For example, Yüksel and Sözer [81] evaluated 34 different ML models over an industrial C/C++ codebase, utilizing ten artifact characteristics. They identified that features such as a warning's lifetime, the developer's idea (feedback), file name, alert code, and severity were the most relevant for classification. They ultimately achieved precision rates up to 90% [81]. Similarly, Alikhashashneh et al. [2] evaluated models like Support Vector Machines (SVM), K-Nearest Neighbors (KNN), Random Forests (RF), and Repeated Incremental Pruning to Produce Error Reduction (RIPPER) using software engineering metrics such as complexity and coupling. They found that tree-based ensembles, particularly RFs, consistently achieved high F-measures ranging from 83% to 98%, and concluded that a function's complexity and coupling heavily influence the likelihood of a tool generating a false positive [2].

Beyond raw metadata, researchers have also incorporated structural and historical characteristics into classical pipelines. Yoon et al. [80] addressed the limitations of purely external features by extracting approximated structural characteristics via Abstract Syntax Trees (ASTs). By training an SVM on these AST-based feature vectors, they successfully reduced false positives by 37.33% while accidentally removing only 3.16% of true positives [80]. From a historical perspective, Heckman [31] proposed the Systematic Actionable Alert Identification (SAAI) process. This demonstrated that models trained on a project's own version history yield the highest precision. However, they also noted a limitation in classical ML scalability. Specifically, that the most predictive artifact characteristics and optimal models are often highly project-specific, and thus require tailored training for different enterprise environments [31].

Despite the necessity of manual feature engineering, a critical finding in this domain was presented by Yang et al. [78], who argued that learning to recognize actionable static code warnings is "intrinsically easy". Building upon a golden set of 23 static warning features, they analyzed the intrinsic dimensionality of massive warning datasets. They discovered

that despite utilizing up to 58 raw features, the data could often be approximated by less than two underlying latent dimensions. Because the underlying data is intrinsically simple, they concluded that employing highly sophisticated DL models on this data is analogous to using "a very big hammer being applied to a very small nail". Consequently, they suggested that classical models, particularly linear SVMs or tree-based ensembles like RFs, are highly sufficient for this task. They concluded that these models would offer superior efficiency without sacrificing performance [78].

2.2.3 Deep Learning and Large Language Models

Despite the success of classical ML, relying on hand-engineered features demands extensive manual effort. Additionally, it relies heavily on human expertise, and often ignores the deep semantic structure of the analyzed source code [37, 38]. To capture this structural context, the literature transitioned towards DL models capable of automated representation learning directly from raw code [28]. For instance, Lee et al. [43] proposed using Convolutional Neural Networks (CNNs) paired with vector embeddings to learn specific lexical patterns in the source code surrounding a warning. By eliminating the need for complex feature extraction, this approach achieved an average precision of 79.72% across multiple C/C++ static analysis checkers [43]. Similarly, Koc et al. [37] applied Long Short-Term Memory (LSTM) networks to program slices. They discovered that Recurrent Neural Networks (RNNs) learning directly over a program's sequential source code could effectively capture the long-term dependencies and context that lead to false positives. This yielded classification accuracies of up to 89.6% [37]. Expanding beyond raw text, Wang et al. [76] leveraged Deep Belief Networks (DBNs) on ASTs to automatically extract latent semantic features. This significantly improved both within-project and cross-project defect prediction compared to traditional metrics.

Building upon these neural architectures, the rapid advancement of Generative AI has recently introduced Pre-Trained Models (PTMs) and LLMs. Unlike traditional DL models that are trained from scratch on limited labeled warnings, PTMs (such as CodeBERT and CodeT5) are pre-trained in a self-supervised fashion on massive corpora of code [26]. This allows them to leverage deep, generic structural knowledge for downstream classification tasks. Recent empirical studies demonstrate the substantial power of this approach. For example, an extensive evaluation by Ge et al. [26] across 12,000 warnings revealed that PTM-based approaches substantially outperform state-of-the-art classical ML models. They do this particularly by drastically improving recall and F1-scores without sacrificing precision. Similarly, Tan and Tian [65] demonstrated that combining CodeBERT embeddings with bidirectional RNNs can achieve exceptional Area Under the Curve (AUC) scores of 98.3% in filtering ASAT warnings.

Furthermore, researchers have begun deploying LLMs not just as classifiers, but as autonomous agents to interact dynamically with ASATs. For example, the CodeCureAgent by Joos et al. [36] utilizes an LLM within an autonomous loop to read warning metadata. It iteratively retrieves surrounding source code context, formulates a plan, and definitively classifies whether a warning is a true or false positive before attempting a repair. Similarly, the SAST-Genius framework by Agrawal and Ahi [1] integrates a fine-tuned LLM as an in-

telligent triage layer. This successfully reduced false positives by roughly 91% by applying semantic reasoning across complex, multi-file data flows that traditional deterministic rules fail to parse.

So even though LLM-based solutions can outperform traditional static analyzers on well-defined benchmarks, existing literature highlights that their performance does not translate uniformly across all vulnerability types [23]. Furthermore, researchers emphasize that deploying these foundational models introduces a significant risk of hallucinations, alongside inconsistent responses and substantial computational overhead [1, 23, 36]. This profound shift towards highly complex, resource-intensive models underscores the core investigation of this thesis: determining whether the high computational cost and complexity of LLMs provide a justifiable performance advantage over optimized classical ML models.

2.3 The Research Gap and Thesis Focus

While the literature demonstrates a clear evolution from statistical methods to classical ML and ultimately to modern DL models, several critical gaps remain in the current research landscape. First, a recurring limitation in existing literature is the scale of the applied datasets. Many prior studies evaluate post-processing techniques on relatively small-scale datasets, often encompassing only around 15 to 20 projects and generating between 1,000 and 10,000 warnings [26, 37, 38, 65, 81]. To draw statistically robust conclusions that reflect the true diversity of coding styles and warning patterns in a real-world environment, evaluations must be conducted on a much larger, high-volume corpus.

Second, there is a pronounced tension in the literature regarding optimal model complexity. The software engineering domain is rapidly shifting toward LLMs because they offer unprecedented semantic comprehension and entirely eliminate the need for manual feature engineering [12, 25, 74]. However, these foundational models introduce substantial computational overhead, high API costs, severe network latency, and a significant risk of inconsistent outputs or hallucinations [1, 23, 36]. Conversely, researchers like Yang et al. [79] have demonstrated that the underlying data for static warnings is often intrinsically simple, arguing that employing highly sophisticated DL models for this task is unnecessary. This contrast highlights a significant empirical gap: it remains unproven whether the massive computational cost and architectural complexity of LLMs provide a justifiable performance advantage over optimized classical ML models.

Finally, existing literature predominantly evaluates predictive models in a vacuum. They rely strictly on standard discrete classification metrics like accuracy, precision, and recall [28]. However, the successful integration of ASATs depends heavily on factors beyond binary correctness. A model's practical utility relies equally on its mathematical robustness. Specifically its probabilistic calibration and ability to accurately signal its own uncertainty. As well as how humans actually perceive, trust, and interact with its output.

To bridge these gaps, this thesis aims to identify the optimal balance between predictive performance and computational efficiency when comparing classical ML models and LLMs for generating actionability probability scores. By leveraging the large-scale NASCAR dataset, which contains over 1.2 million implicitly labeled Java warnings, this research con-

ducts a comparative analysis. It evaluates both model families not only on their raw predictive power (using F1-scores and Precision-Recall Area Under the Curve) but also on their probabilistic reliability (using the Brier score). Furthermore, to ground these results in practical software engineering, this thesis extends the evaluation through a qualitative user study to determine how these probability scores influence triage workflows, automation bias, and trust dynamics in the context of an enterprise environment.

Chapter 3

Methodology

This chapter details the methodological framework designed to evaluate and compare the effectiveness of classical (including ensemble) ML versus LLMs in generating actionability probability scores for ASAT warnings. It outlines the dataset selection and rationale (Section 3.1), the preprocessing pipeline (Section 3.2), and the specific model configurations (Section 3.3) with the chosen evaluation metrics (Section 3.4).

3.1 Dataset Selection and Rationale

This research utilizes supervised classification to predict the probability of the ASAT warning being actionable. A crucial component of this process is the selection of a dataset that balances practical relevance with statistical significance. Although the current implementation is optimized for a single, widely-adopted programming language and a specific dataset, the underlying pipeline is build for extensibility. This design allows for future researchers to integrate additional datasets, diverse languages, or alternative ASAT with minimal reconfiguration. To find a fitting dataset, a choice for a specific programming language needs to be discussed first. Subsequently, the selected dataset can be properly explained.

3.1.1 The Chosen Language

This research focuses exclusively on tools created for Java. While it may no longer hold the absolute top spot in global popularity, its historical and present relevance is undeniable. According to the TIOBE index, Java has consistently maintained a top five position for decades and it shows a level of stability that few other languages can match [69]. This highlights Java's relevance because the index tracks the popularity of programming languages based on search engine volume and developer activity [68]. The importance of Java is further corroborated by comprehensive empirical research by Bissyandé et al. [15] in the open-source ecosystem. They rank Java as one of the most pervasive and impactful languages in real-world development. In a large-scale analysis of 100,000 GitHub projects, Java was identified as a dominant general-purpose language, appearing in over 10% of all repositories surveyed. Unlike scripting languages that are frequently used for supporting tasks, Java is distinguished by its high utilization as a main project language. Furthermore,

the language demonstrates significant professional depth, maintaining one of the largest active developer pools and consistent popularity in terms of total lines of code. Specifically, it accounts for nearly 100 million lines in the studied corpus [15]. This enduring presence ensures that a thesis focused on Java is grounded in a language that is not a passing trend, but a cornerstone of the industry.

Finally, the selection was guided by industrial relevance. Java is the primary development language at the partner company (SIG), and it represents a portion of the analysis performed by their internal tool (Sigrid). Sigrid has been used to analyze more than 400 billion lines of code written in more than 300 different languages. By centering the research on Java, this industrial alignment ensures that the resulting pipeline remains directly applicable within the company's existing infrastructure.

3.1.2 The NASCAR Dataset

As previously mentioned in Section 2.2, many studies have been conducted into reducing warnings for ASATs. However, a recurring limitation in existing literature is the scale of the applied dataset. Many studies rely on relatively small-scale datasets, often encompassing only around 15 to 20 projects with around 1-10K warnings [28, 37, 38, 65, 81]. To ensure the findings of this thesis are as generalizable and statistically robust as possible, it was essential to move beyond this scale and utilize a high-volume, real-world corpus. Hence, the NASCAR (Non-Actionable Static Code Analysis Reports) dataset was selected.

NASCAR is a large-scale, open-source collection specifically designed to address the scarcity of large-scale labeled data for the Java ecosystem. It contains a total of 1,227,763 warnings mined from 102 GitHub projects with at least 200 stars. This minimum star threshold serves as a critical filter to ensure data quality. As focusing on prominent, actively maintained repositories increases the probability that the contributing developers adhere to common software engineering best practices, including the systematic use of ASATs. These are categorized into 196,940 (16%) actionable and 1,030,823 (84%) non-actionable warnings. In this context, actionable warnings are those signaling an actual bug or flaw that developers deem significant enough to actively repair. Conversely, non-actionable warnings are defined as any alerts systematically left unaddressed by developers, irrespective of the underlying reason. This encompasses both outright false positives (FPs)[1], and ignored warnings. While these warnings are technically valid, they are considered irrelevant to the project's safety, functionality, or maintainability (e.g., a warning about a missing Javadoc comment on a private helper method). [41]

To obtain these labels at scale, the NASCAR methodology leverages differential analysis on consecutive version control commits to evaluate implicit developer feedback. By generating and comparing ASAT reports for a specific commit and its parent, the system tracks the lifecycle of each identified issue. A warning from the parent commit is labeled as actionable if its surrounding context is modified by a Java code insertion or deletion, and the warning subsequently disappears from the new commit's report. On the other hand, a warning is classified as non-actionable if it persists across versions. This happens either

[1] FP warnings happen when the tool misinterprets the warning.

because the specific code context was not affected by any Java changes, or because the code was affected but the underlying problem remained unresolved. If a warning conflict occurs where an issue is flagged as both actionable and non-actionable, the actionable classification takes precedence. [41]

To address the occurrence of repeated warnings across hundreds of analyzed commits, a deduplication process was applied by Kószó et al. [41]. This process uses the MinHash and Locality Sensitive Hashing (LSH) schemes with a Jaccard similarity coefficient of 0.95. This process identified 144,690 duplicated warnings, resulting in a refined deduplicated dataset of 1,083,073 warnings. Within this version, the distribution consists of 145,997 (13.48%) actionable and 937,076 (86.52%) non-actionable warnings [41]. A list of all the information the dataset has per warning can be found in Figure 3.1 and the table showcasing the percentages of actionable versus non-actionable is shown in Table 3.1. By utilizing such an extensive dataset, the research avoids the pitfalls of over-fitting to specific project structures and instead captures a wide variety of coding styles and warning patterns.

Dataset	Actionable	Non-actionable	Total
NASCAR	196,940 (16%)	1,030,823 (84%)	1,227,763
Deduplicated NASCAR	145,997 (13.48%)	937,076 (86.52%)	1,083,073

Table 3.1: The distribution of actionable and non-actionable warnings for the original and deduplicated NASCAR dataset.

The choice for NASCAR was further driven by the need for transparency and reproducibility. Because it is an open-source dataset, it allows future researchers to recreate the experiments or extend the pipeline without the barriers of proprietary codebases. Furthermore, the dataset is aligned with the technical requirements of this study. It focuses exclusively on Java and includes warnings generated by industry-standard, widely-adopted ASATs such as PMD and SpotBugs [11]. These tools are among the most widely-adopted and empirically studied instruments in the Java ecosystem [73]. Their inclusion ensures that the classification models developed in this thesis are trained on the same types of warnings that developers encounter in professional, everyday environments.

3.2 Data Setup and Preprocessing Pipeline

This section details the data setup and preprocessing pipeline required to prepare the dataset for analysis. A critical challenge is the fundamental difference in how models consume data. While LLMs can process natural language, classical ML models require structured, numerical representations. To address this, the pipeline is split into two distinct tracks, each tailored to the specific formatting and feature engineering requirements of the ML models and LLMs used in this thesis.

Metadata NASCAR Dataset

tool: The name of the SCA tool, which created the warning.
warning_type: The name of the violated rule of the SCA tool (PMD or Spotbugs).
warning_msg: The detailed explanation of the warning.
commit_sha: The corresponding commit hash that the warning is associated with.
parent_sha: The unique hash of the parent commit in which the warning was reported.
repo: The url of the corresponding project's GitHub repository.
filename: The name of the source file, in which the warning occurred.
positions: The start and the end lines of the narrowest context within the source file, in which the warning occurred; the start and the end columns sometimes are also given.
filepath: The relative path to the file from the files root.
commit_date: The date (converted to UTC and formatted to ISO 8601) when the commit was made.
parent_date: The date (converted to UTC and formatted to ISO 8601) when the previous commit was made.
label: Denoting if the warning is actionable or non-actionable (0 for non-actionable and 1 for actionable).

Figure 3.1: Metadata for the NASCAR Dataset.

3.2.1 Preprocessing for Classical ML

As can be seen in Figure 3.1, multiple different types of textual data exists. To handle categorical variables, such as the tool name, One-Hot Encoding is applied. This technique creates individual binary columns for each tool, preventing the model from assuming an artificial ordinal relationship between different ASAT [59]. Temporal features, such as commit_date and parent_date, are subsequently converted into Unix timestamps. A Unix timestamp is a system for tracking time that represents the total number of seconds elapsed since the Unix Epoch (January 1 1970, at 00:00:00 UTC), excluding leap seconds [20]. By calculating the difference between these two values, an age can be derived for the warning. This provides the model with a quantitative measure of how long a potential vulnerability has persisted.

Crucially, extracting the age metric instead of feeding the raw commit and parent dates directly to the model is necessary to prevent severe temporal data leakage. This leakage stems directly from the NASCAR dataset's data collection and deduplication methodology. To manage the massive volume of non-actionable warnings, the NASCAR methodology applies a strict rule. If the same non-actionable warning appears multiple times across a project's commit history, all earlier duplicates are dropped and only its final occurrence, the warning appearing in the latest commit's report, is retained [41]. While effective at reducing duplicates, this rule introduces a significant artificial timeline bias. Seeing as only

the newest iteration of a persistent warning is kept, there is a massive, artificial concentration of non-actionable warnings towards the end of the data mining window, specifically spiking between August and October 2024 (see Figure 3.2 and Figure 3.3). If the raw commit_date or parent_date were utilized as predictive features, an ML model would quickly exploit this timeline anomaly. It would learn to superficially classify warnings as non-actionable simply because their timestamps fall late in the dataset's chronological window, rather than evaluating the underlying semantic or code-level characteristics. By transforming these absolute dates into an age, this temporal bias introduced by the mining methodology is neutralized, ensuring the model relies on genuine indicators of actionability.

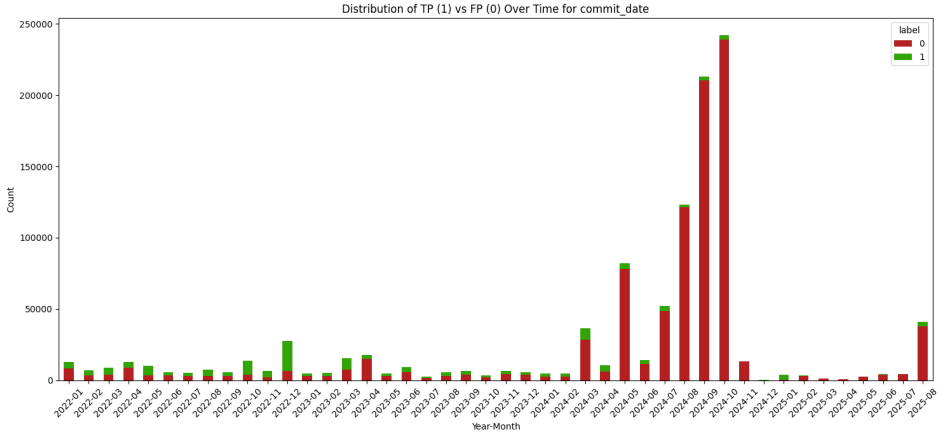


Figure 3.2: Temporal distribution of actionable (TP/1) and non-actionable (FP/0) warnings based on their commit_date.

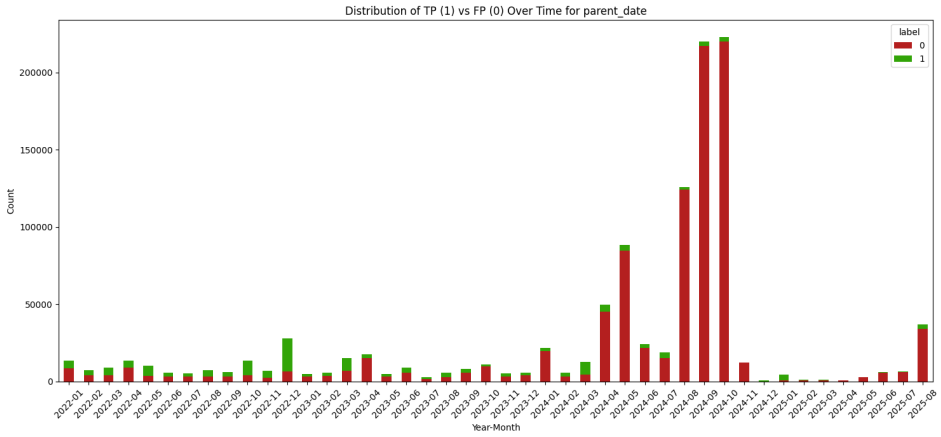


Figure 3.3: Temporal distribution of actionable (TP/1) and non-actionable (FP/0) warnings based on their parent_date.

To capture the context of the warning, a function extracts n-lines of code surrounding the specific position of the warning. So $n = 2$ takes two lines above and below the start and

end line respectively. For this, filename and filepath are needed. It ensures that the correct code from the correct place has been extracted and coupled to the right warning. The then newly created feature (n_source_code) and the warning_msg feature are transformed into TF-IDF (Term Frequency-Inverse Document Frequency) vectors respectively. This means that each feature has their own instantiation of a TF-IDF vectorizer. Seeing as these TF-IDF matrices are often high-dimensional and sparse [46, ch. 6], Latent Semantic Analysis (LSA) has been applied. By utilizing Singular Value Decomposition (SVD), LSA reduces the feature space into a lower-dimensional concept space. This allows the model to capture latent semantic relationships between different technical terms, and condenses sparse word counts into a dense semantic vector. In other words, it mitigates the effects of synonymy, where different words describe the same underlying warning, and it reduces computational noise [21].

Next, the warning_type feature is addressed by mapping the rules to their corresponding Common Weakness Enumeration (CWE) IDs, which is a list of software and hardware weaknesses types [49]. To improve the model's ability to generalize across different tools, the warning_type is also grouped into a category. This category is retrieved from lists that map warning_types to general categories, e.g., Performance, Security. To process these non-numerical features, CWE-ID and categories are transformed using One-Hot Encoding. This feature engineering technique converts categorical variables into a sparse binary matrix by creating an individual binary column for each distinct warning category (assigning a 1 if the category is present, and a 0 otherwise). This transformation is a mathematical necessity for classical ML architectures, as it prevents the models from falsely assuming an artificial ordinal relationship between independent categories (e.g., incorrectly interpreting a categorical identifier of 3 as being mathematically greater or more severe than a 1). By representing these categories as independent binary vectors, One-Hot Encoding allows the model to accurately recognize shared structural patterns across related types of vulnerabilities [14, 29]. Finally, all non-numerical features are dropped and the label (the true answer) is separated from the rest of the data. So in the end, a fully transformed dataset is left (see Figure 3.4).

3.2.2 Preprocessing for LLMs

While classical ML models require an extensive, manually engineered pipeline to convert raw text into dense numerical matrices, the external preprocessing requirements for LLMs are significantly less intensive. However, they still fundamentally process numerical data. Instead of requiring high-dimensional, upfront transformations like TF-IDF or the dimensionality reduction provided by LSA [21], LLMs handle this conversion internally. Consequently, only a specific subset of the previously described preprocessing pipeline is executed. The primary objective in this phase is the derivation of three key structural features: the age of the warning, its broader category, and its mapped CWE-ID. As with the classical approach, the age is calculated as the temporal difference between the parent and current commit Unix timestamps. Similarly, the specific tool rules are mapped to their respective CWE-IDs and general categories (e.g., Security or Performance).

Once these features are generated, the raw textual data are not transformed into vectors. This includes the warning_msg, the category, the originating tool, and the n_source_code

Features Classical ML

tool_i: The SCA tool, this is one-hot encoded so contains a 1 or 0 and is repeated for all tools (e.g., tool_PMD).
category_i: The category of the warning_type, this is one-hot encoded so contains a 1 or 0 and is repeated for all categories (e.g., category_performance).
warning_msg_lsa_i: An LSA component for the original feature called warning_msg.
source_code_lsa_i: An LSA component for the original source code feature (n_soucre_code).
age: The age of the warning in seconds.
CWE-ID_i: The corresponding Common Weakness Enumeration (CWE) code, this is one-hot encoded so contains a 1 or 0 and is repeated for all CWE-IDs (e.g., CWE-ID_CWE:1003)

Figure 3.4: The final structured feature set utilized by the classical ML models. The subscript *_i* serves as a placeholder index indicating that a separate feature column exists for every unique category or component, iterating from 0 through the total number of distinct variables in that specific set (e.g., all one-hot encoded tools or all extracted LSA dimensions).

(the extracted n-lines of code surrounding the warning). Instead, they are maintained in their original string format and organized into a structured prompt. This prompt acts as the direct interface for the model, feeding it the raw code and metadata as a cohesive narrative. LLMs leverage pre-trained architectural knowledge of code semantics and natural language, utilizing distributed representations to maintain the relational integrity of the source code [48]. Unlike classical ML models that require text to be compressed into sparse numerical vectors[2], LLMs perform representation learning. This allows the model to autonomously discover relevant semantic patterns within the data, rather than relying on manually engineered features [12]. Therefore, LLMs can interpret CWE-ID, categories, and source code not as isolated variables, but as contextual signals that interact dynamically with each other through self-attention mechanisms [25, 74].

To ensure a rigorous and fair comparison between the classical ML and LLM approaches, it is imperative to maintain the most consistency possible across the experimental variables. As such, the underlying feature sets exposed to both models are kept as identical as structurally possible. This methodological constraint is the primary reason the raw warning_type is withheld from the prompt. Since the classical ML pipeline transforms the warning_type into generalized categories, providing the raw warning_type to the LLM would grant it a level of granular information not available to its counterpart. The research thus restricts both models to the same set of derived features. This ensures that any variance in performance is attributed to the architectural capabilities of the models themselves, rather

[2]The loss of nuanced context is often a result of compressing text into sparse numerical vectors.

than discrepancies in the underlying data they observed.

3.3 Model Selection

As established in the literature review in Section 2.2, the quest for reducing ASAT warnings has led researchers to explore a wide variety of algorithmic solutions, ranging from traditional statistical methods to modern neural networks [28]. To provide a comprehensive evaluation, this thesis employs a dual-modeling strategy that compares Classical ML against LLMs. By selecting models with fundamentally different underlying assumptions, this research seeks to uncover whether the predictive power lies in complex, non-linear feature interactions. Or if the problem can be addressed by more transparent, linear decision boundaries.

3.3.1 Classical and Ensemble Models

Logistic Regression (LR) and Random Forest (RF) are selected as the representative ML models. This specific pairing is chosen to contrast a simple linear decision boundary against a complex, non-linear one. Here, LR acts as the foundational baseline to determine if actionable and non-actionable warnings are linearly separable [33, ch. 1]. In contrast, RF is chosen as the non-linear alternative to prove whether recursive, multi-dimensional partitioning is necessary because structural metadata and code metrics often interact conditionally [17].

LR serves as the baseline for linear classification. It is highly efficient and provides significant interpretability, as it models the probability of a class by fitting data to a logistic curve [33, ch. 1]. If the relationship between features like warning age, category, and source code semantics is relatively straightforward, a linear model like LR will perform optimally without the risk of over-fitting often associated with more complex architectures. Beyond its simplicity, LR acts as a baseline for calibration because it naturally produces well-aligned probabilities. This characteristic makes it easier to evaluate Brier scores, and thus ensures that the predicted probabilities reflect the true likelihood of a warning being actionable. From a developer's perspective, LR offers high interpretability, allowing one to see exactly how specific metrics (such as a particular CWE-ID or a warning's age) impact the actionability score. Furthermore, it is easy to implement, exceptionally fast to train, and requires minimal memory [29].

In contrast, RF was selected due to its consistent performance as a top performer in existing research [28]. As an ensemble learning method, RF operates by constructing a multitude of decision trees during training and outputting the mode of the classes. This approach is particularly adept at capturing high-order, non-linear interactions between disparate features [17]. Think of the way a specific CWE-ID might interact with a particular code structure to indicate an actionable warning. Furthermore, one of the primary advantages of RF in this context is its high robustness to the noisy labels often found in open-source ASAT datasets. RF naturally handles non-linear relationships between rule metadata and code metrics without requiring extensive hyperparameter tuning. Like its linear counterpart, it remains relatively easy to implement and quick to train, even when processing the over one million datapoints provided by NASCAR [17, 55]. By comparing the performance

of the non-linear RF against the linear LR, this thesis can empirically validate whether the additional complexity of ensemble methods is strictly necessary for the task of warning triaging.

3.3.2 Large Language Models

To investigate the predictive performance on complex static analysis metadata, this thesis incorporates LLMs. Specifically, the Claude model family developed by Anthropic has been selected for this research. It comprises of: Haiku 4.5, Sonnet 4.6, and Opus 4.6. The primary motivation for this selection is institutional access, as the company SIG provides a secure and robust infrastructure for utilizing these specific models. This accessibility ensures that the research remains reproducible within a professional (SIG) environment while adhering to data privacy and security standards necessary for large-scale code analysis.

The Claude 4.x series is characterized by its sophisticated understanding of both structured data and natural language [3]. This makes it uniquely suited for triaging warnings that often contain semantic nuances in source code context. Within this family, the models are differentiated by their scale and reasoning capabilities. This allows for evaluation of the relationship between model size and classification accuracy. Haiku 4.5 serves as the lightweight, high-speed variant designed for efficiency and near-instantaneous processing. This makes it ideal for real-time developer workflows [6]. Sonnet 4.6 represents the mid-tier balance. It offers a significantly increased intelligence and reasoning depth while maintaining the throughput necessary for large datasets like NASCAR [8]. Finally, Opus 4.6 is the most powerful model of the three. It is designed to handle highly complex reasoning tasks and subtle pattern recognition that smaller models might overlook [7].

By employing these three variants, the research can determine whether the additional computational overhead and parameter count of a model like Opus provides a justifiable increase in performance compared to the more efficient Haiku. This tiered approach allows for a granular assessment of how different levels of model complexity handle the intricate metadata associated with ASATs. Ultimately, comparing these state-of-the-art LLMs against RF and LR will reveal if the advanced contextual reasoning of DL can significantly outperform traditional statistical and ensemble methods.

3.4 Evaluation Metrics

Finally, to evaluate these models we need to select metrics that accurately capture the performance aligning with the specific goals of this research. In other words, the metrics need to convey whether the models can reliably distinguish critical signals from noise, and whether they can balance this with capturing all significant actionable warnings. Therefore, even though accuracy is often the default metric for evaluating performance across both classical ML and DL architectures, it serves as a misleading indicator in the context of the current research. Considering that the original NASCAR dataset exhibits a significant class imbalance with 84% non-actionable warnings, a trivial model could achieve 84% accuracy by simply predicting the majority class for every instance. In such a scenario, the reported accuracy provides a false sense of security. That is to say, it fails to reflect the model's actual

utility in identifying the minority class even though this is essential to the analysis. For instance, in a subset of 1000 samples where only 160 are actionable, a model that labels every single data point as non-actionable would yield an 84% accuracy rate despite failing to identify a single actionable warning. While the high percentage suggests a high-performing system, the model's recall for the minority class would be 0%. This renders the model entirely ineffective for any practical application where missing an actionable warning carries a high cost.

To address these limitations, the F1 score is utilized as the primary performance measure. By calculating the harmonic mean of precision and recall, the F1 score ensures that the model is penalized for both excessive non-actionable and missed actionable warnings. Precision and recall are mathematically defined in Equation 3.1 and Equation 3.2, respectively:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3.1}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.2}$$

where TP represents True Positives, FP represents False Positives, and FN represents False Negatives. Using these foundational properties, the F1 score is formulated as shown in Equation 3.3:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.3}$$

This balance is necessary, because the model must remain efficient by reducing non-actionable warnings to minimize unnecessary noise, yet it must remain robust enough to ensure that actionable warnings are not overlooked. Relying on the F1 score thus ensures that the performance of both classical ML models and LLMs is assessed based on their ability to handle these competing priorities effectively. Furthermore, since the F1 score provides a static snapshot at a single decision threshold, Precision-Recall (PR) curves are employed to offer a more comprehensive visualization of model performance. These curves are particularly valuable for imbalanced datasets like the NASCAR set, as they illustrate the trade-off between precision and recall across all possible thresholds. By analyzing the PR curve, it becomes possible to identify the optimal threshold that maximizes detection of the minority class without incurring an unacceptable volume of non-actionable warnings. This allows for a more informed selection of the final operating point, ensuring the model remains tuned to the specific risk tolerances of the research. [60]

In addition to classification performance, the Brier score is incorporated to evaluate the calibration and reliability of the models' probabilistic predictions. Unlike discrete, threshold-dependent metrics, the Brier score calculates the mean squared difference between the assigned probability score and the actual binary outcome [18]. It thus effectively functions as the mean squared error of prediction probabilities, calculated using Equation 3.4:

$$\text{Brier Score} = \frac{1}{N}\sum_{i=1}^{N}(p_i - y_i)^2 \tag{3.4}$$

where N is the total number of samples, p_i is the probability assigned by the model to the i-th instance, and y_i is the actual ground-truth binary outcome to the i-th instance (1 for actionable, 0 for non-actionable). The score strictly ranges from 0.0 to 1.0, where a lower score indicates better performance. A score of 0.0 represents a theoretically perfect model that assigns absolute certainty (1.0 or 0.0) to correct outcomes. Conversely, a score of 0.25 serves as the baseline for non-informative, random guessing (assuming an uninformative 50% probability assignment). Any score exceeding 0.25 thus implies that the model's probability estimates are actively miscalibrated and worse than chance [18]. By using this metric, the framework moves beyond a binary snapshot of precision and recall to quantify exactly how well the model's confidence aligns with reality. The score thus allows for a more nuanced comparison between classical ML and LLM approaches, ensuring that the final deployed model is not merely accurate in its discrete labeling, but also fundamentally dependable in its estimated certainty.

Chapter 4

Study Setup

This chapter outlines the study setup designed to address the research questions established in Chapter 1. Rather than a series of isolated tests, the study is structured as a pipeline that moves from data calibration and model optimization to a final human-centric validation. The eventual purpose of this thesis is to compare classical ML and LLMs, so the pipeline consists of two parts: the ML part and the LLM-prompting part. Additionally, as illustrated in the research workflow (see Figure 4.1), the experiments are categorized into four phases: *Dimensionality Optimization, Baseline Calibration, Model Optimization, and Interpretability Analysis*. After the experiments are defined, a user study is explained to bridge the gap between the experiments and software engineering.

4.1 Experimental Designs

This section provides a detailed breakdown of the experimental pipeline. The proposed pipeline is partitioned into distinct LLM and ML parts for clarity. Although discussed separately, these modules contain the same phases: *Baseline Calibration, Model Optimization* (as shown in Figure 4.1).

The first phase, *Dimensionality Optimization*, focuses on optimizing the feature space for the ML pipeline. This begins with `EXP: Optimal Feature Space` (Section 4.1.1). As the name suggest, it finds the optimal feature space to ensure that the source code data and warning message are semantically dense. Following this, `EXP: Optimize Data Splits` (Section 4.1.1) subjects both models to various class distributions to determine their sensitivity to the high class-imbalance found in the NASCAR dataset. This experiment will establish which dataset split will be used in further experiments. Then we move on to the *Baseline* phase. This phase contains `EXP: Baseline of Standard Models` (Section 4.1.2), it trains and evaluates the default ML models (RF and LR). These models will be used to compare all subsequent experiments happening in the *Model Optimization* phase.

The *Model Optimization* phase contains experiments that conduct hyperparameter tuning for both LR (`EXP: LR Hyperparameter Tuning`, Section 4.1.3) and RF (`EXP: RF Hyperparameter Tuning`, Section 4.1.3). Then the next phase, *Interpretability Analysis*, moves beyond metrics to examine model behavior. In `EXP: Feature Importance` (Sec-

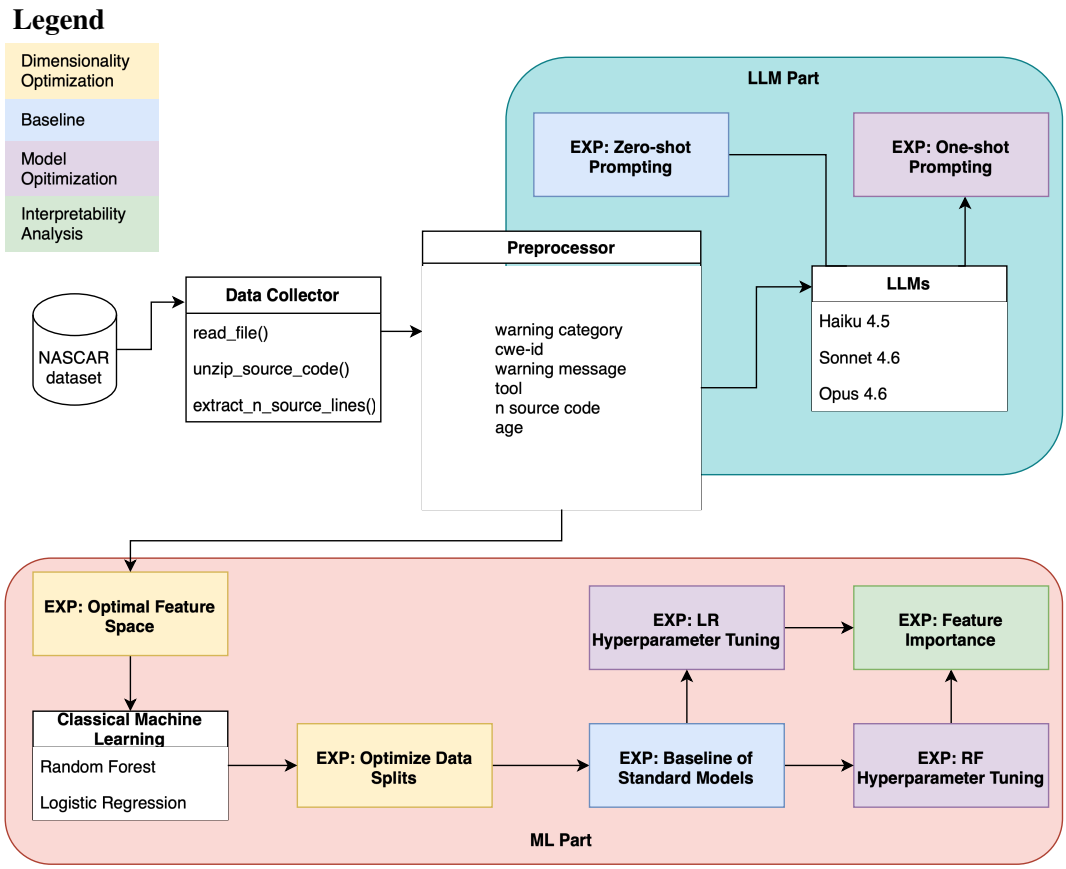


Figure 4.1: The full research workflow of the study. It highlights the flow of the data, and distinguishes between different categories of experiments.

tion 4.1.4), the decision-making logic of the optimized RF and LR are scrutinized via feature importance mapping.

The experimental setup concludes with the LLM component detailed in Section 4.1.5. This part includes experiments for both the Baseline and Model Optimization phases, but are discussed separately from the ML part to maintain clarity. This evaluation utilizes `EXP: Zero-shot Prompting` as the primary baseline for comparison against `EXP: One-shot Prompting`. In this configuration, the zero-shot setup employs a prompt without any prior demonstrations, whereas the one-shot setup is enriched with a single, representative example to facilitate in-context learning. This approach allows for an assessment of how a single demonstration influences the model's ability to classify warnings correctly compared to providing no context at all [19, 71].

4.1.1 Dimensionality Optimization Phase

EXP: Optimal Feature Space

Before proceeding to full model training, it is essential to calibrate the dimensionality reduction process to ensure that the semantic essence of the source code and warning messages is preserved. This calibration is a critical prerequisite, because failing to optimize the feature space risks either overwhelming the models with high-dimensional noise or omitting subtle, critical semantic patterns through over-compression. The first experiment thus focuses on evaluating the Latent Semantic Analysis (LSA) variance. The goal is to determine how various TF-IDF configurations impact the model's ability to capture meaningful information from the NASCAR dataset. These configurations differ in feature counts, n-gram ranges, and frequency filters. The experiment utilizes a series of targeted trials (see Listing 4.1), such as a high-volume wide spectrum approach and a more conservative denoised heavy configuration. This is to observe how text is compressed into a lower-dimensional space. For each trial, the pipeline calculates the explained variance ratio, so it quantifies the amount of information retained by the LSA components. To identify the optimal balance between computational efficiency and data richness, the experiment employs a mathematical elbow point detection. This elbow point signifies the threshold where adding further LSA components yields diminishing returns in captured variance.

The trial is run three times, first with a cap of 50, then 300, and finally with 1000 LSA components. These specific thresholds were established through preliminary empirical exploration of the dataset's variance distribution. To avoid the computational overhead of evaluating every possible dimension, initial scoping runs were utilized to identify the effective lower and upper boundaries of the feature space. Consequently, the selection of 50, 300, and 1000 components is thus designed to capture the full spectrum of the variance curve. It moves from a low-resolution baseline to a high-fidelity semantic representation. The initial 50-component run serves as a baseline for the most dominant latent topics. Its purpose is to isolate the primary structural signals whilst deliberately excluding granular detail. Increasing the cap to 300 components allows for the identification of the expected mathematical elbow point. Finally, the 1000-component iteration acts as a stress test to observe the long tail of the data distribution. It ensures that even the most diffuse patterns in the source code are accounted for and that no significant information density is overlooked in the more complex features. This approach provides a comprehensive view of how information is lost or retained at different levels of compression.

```
trials = [
    {"name": "baseline", "max_features": 5000, "ngram_range": (1, 1), "
    min_df": 2, "max_df": 0.8},
    {"name": "bigrams_10k", "max_features": 10000, "ngram_range": (1, 2),
     "min_df": 5, "max_df": 0.7},
    {"name": "n_gram_specialist", "max_features": 3000, "ngram_range":
    (1, 3), "min_df": 3, "max_df": 0.9},
    {"name": "denoised_heavy", "max_features": 2000, "ngram_range": (1,
    1), "min_df": 10, "max_df": 0.5},
    {"name": "wide_spectrum", "max_features": 20000, "ngram_range": (1,
    2), "min_df": 1, "max_df": 1.0}
]
```

Listing 4.1: Trial configurations for `EXP: Optimal Feature Space`.

Ultimately, the trials are evaluated using an Efficiency Score. This is defined as the ratio of variance captured at the elbow point to the number of components required. The optimal configuration is then locked in for all future experiments. This is methodologically sound because the text's semantic variance is an inherent property, independent of downstream ML models. By decoupling this variance analysis from the classification task, the approach mitigates the noise typical of large-scale repository mining and ensures subsequent models receive the same information-dense baseline. With feature dimensions now optimized to establish a consistent performance floor, the following subsection will discuss the impact of class distribution on default model configurations.

EXP: Optimize Data Splits

Building upon the initial calibration of the feature space, the research moves to optimize the data balance. It investigates the critical impact of class distribution on model reliability and predictive performance. Class imbalance is a challenge in the field of ASATs. As highlighted in the NASCAR study, non-actionable warnings vastly outnumber actionable ones. The non-actionable warnings in this case account for over 86% of the available data [41]. Isolating the threshold where data balancing stops preventing majority-class bias and starts distorting real-world predictive performance is critical. Without this optimization, models risk either collapsing into naive majority-class predictors or failing to generalize to actual deployment environments. To determine how this inherent skewness thus affects the learning behavior of both RF and LR, this experiment subjects the scikit-learn default models[1,2] to a series of controlled dataset splits. These configurations range from perfectly balanced distributions designed to prevent majority-class bias to the original, highly imbalanced ratio found in the NASCAR corpus (see Listing 4.2).

By training the standard models across a spectrum of ratios (see Listing 4.2 and Table 4.1), this experiment aims to quantify the sensitivity of each architecture to different dataset splits. The inclusion of a full-scale, un-sampled training run ensures that the findings are grounded in the actual volume of the 1.1 million labeled datapoints provided by the NASCAR dataset. Among these candidates is a split of 200K warnings. This comprises of 100K actionable and 100K non-actionable warnings. The selection for 200K warnings was driven by the total availability of actionable warnings in the NASCAR corpus, which contains 145,997 actionable warnings. By capping the sample at 200K, the study maintains a perfectly balanced 1:1 class ratio.

Within the dataset sample, the data is further partitioned into training and testing subsets to facilitate model development. This is achieved by allocating 75% of the data for training (utilizing cross-validation for hyperparameter tuning instead of a separate validation set) and 25% for testing. To ensure reproducibility and scientific rigor, this split is executed using the scikit-learn train_test_split method with the default random state of 42. This structured

[1]The RF default model can be found here.

[2]The LR default model can be found here.

Dataset	Actionable	Non-actionable	Total
Split 1	100,000 (50%)	100,000 (50%)	200,000
Split 2	50,000 (50%)	50,000 (50%)	100,000
Split 3	33,612 (̃13%)	226,410 (̃87%)	260,022
Split 4	145,997 (̃13%)	937,076 (̃87%)	1,083,073
Split 5	78,153 (̃30%)	182,759 (̃70%)	260,912
Split 6	104,009 (̃40%)	156,013 (̃60%)	260,022

Table 4.1: The distribution of actionable and non-actionable warnings for the multiple dataset splits.

approach ensures that while the model learns and is refined on the dataset, its true predictive performance is ultimately verified against the distinct test set. Ultimately, this experiment helps identify the optimal data distribution needed to maintain high performance without sacrificing the model's ability to generalize to real-world, imbalanced environments. This optimal distribution is then used in further experiments.

```
# The first split: balanced dataset with even classes
s1_data = helper_split_data(data, 100000, 100000)
# The second split: reduced even dataset with 50k samples per class
s2_data = helper_split_data(data, 50000, 50000)
# The third split: original ratio dataset (~13TP-87FP) but reduced size
s3_data = helper_split_data(data, 33612, 226410)
# The fourth split: full dataset without any downsampling
s4_data = data.copy().reset_index(drop=True)
# The fifth split: 30TP-70FP imbalanced dataset
s5_data = helper_split_data(data, 78153, 182759)
# The sixth split: 40TP-60FP imbalanced dataset
s6_data = helper_split_data(data, 104009, 156013)
```

Listing 4.2: The multiple dataset splits for `EXP: Optimize Data Splits`. To see the code for helper_split_data() see Listing B.1 in Appendix B.

Every specific dataset split utilized in this research is generated through random sampling from the total pool of available data. This is achieved by isolating the positive and negative classes and sampling the required number of instances for each using the default random state of 42 to ensure reproducibility. A significant advantage of this approach is that it ensures that the model is exposed to a diverse variety of warning types and coding patterns from across the entire corpus. However, a notable disadvantage of purely random sampling in the context of ASAT warnings is the potential disruption of project-level context. Because the sampling occurs at the individual warning level rather than the project level, warnings from a single software project may be fragmented. Some can be included in the training set while others from the same project are relegated to the test set or excluded entirely. This fragmentation prevents the model from learning from the cohesive architectural style of a specific codebase and fails to account for the locality of warnings, where similar errors often cluster within specific modules or files.

4.1.2 Baseline Phase

To accurately measure the efficacy of the optimization efforts, such as hyperparameter tuning and custom regularization, it is vital to establish a baseline to compare it to. This is achieved through the Baseline Model experiment, which evaluates the performance of the selected architectures using their standard, out-of-the-box configurations from the scikit-Learn library trained on 200K warnings. This subset of 200K is derived from the results of `EXP: Optimize Data Splits` mentioned in Section 4.1.1. As explained, the subset is 50% actionable and 50% non-actionable warnings, and is capped at 200K because of the limited actionable warnings present in NASCAR.

The results from this baseline are used to quantify the performance gain achieved in subsequent experiments. For the RF, the baseline utilizes a balanced subsample weighting to account for any residual variance in the 200K split. In contrast, the LR is run with its default solver and regularization parameters. This comparison is essential for validating whether the sophisticated tuning of tree depth or L1-regularization actually results in statistically significant improvements in the performance. Without this baseline, it would be impossible to determine if a model's high performance is a result of effective hyperparameter engineering or simply a reflection of the high-quality, large-scale labeling inherent to the NASCAR dataset itself.

4.1.3 Model Optimization Phase

Now we move onto finding the optimal model for this problem space. This is done through hyperparameter tuning of both models. This phase is critical because default model parameters are generalized. So fine-tuning these hyperparameters is necessary to systematically navigate the trade-off between model complexity and generalization, ensuring the models do not overfit to the training corpus or underfit the subtle indicators of actionable warnings. To ensure the model optimization process aligns with the project's performance requirements, the primary objective of the experiments in this phase (`EXP: LR Hyperparameter Tuning` and `EXP: RF Hyperparameter Tuning`), is to maximize the F1-score. As the harmonic mean of precision and recall, the F1-score serves as a critical metric for balancing the model's sensitivity to actionable warnings with the precision necessary to avoid overwhelming developers with incorrect predictions [72]. As mentioned before in Section 3.4, prioritizing the F1-score over standard accuracy is vital, because optimizing for raw accuracy would produce a degenerate model that achieves high scores by simply classifying all warnings as non-actionable in an imbalanced dataset.

Additionally, both experiments utilize a Successive Halving search strategy combined with k-fold cross-validation. Specifically, the implementations employs HalvingGridSearchCV, which operates on a resource-allocation principle. It begins by evaluating all candidate parameter combinations on a small subset of the data and iteratively discards the lowest-performing candidates while increasing the data resources for the remaining survivors. This approach is significantly more efficient than a traditional exhaustive grid search, as it concentrates computational power on the most promising parameter sets early in the process [58]. Employing this resource-constrained search strategy is a mathematical necessity

for this corpus, as an exhaustive grid search would be computationally prohibitive. In contrast, Successive Halving guarantees an optimized hyperparameter configuration within a feasible execution window. Furthermore, to guarantee that the performance metrics used for this selection are not biased by a single fortunate train-test split, a standard 5-fold stratified cross-validation strategy is integrated into the search.

EXP: LR Hyperparameter Tuning

Having obtained the baselines for the default scikit-learn LR model in the previous Section 4.1.2, we now optimize it similar to how we will optimize the RF model. This ensures a fair comparison between ensemble-based and linear-based classification strategies. The goal of this experiment is to identify the optimal linear decision boundary while preventing the model from becoming overly sensitive to noisy features within the NASCAR dataset. This is achieved through a search across a specialized parameter grid (see Listing 4.3), utilizing the same subset of 200K warnings from the previous sections to ensure a statistically stable training environment.

A critical component of this optimization is the tuning of the regularization strength (C) and the penalty type. By evaluating both L1 (Lasso) and L2 (Ridge) penalties, the experiment determines the best method for handling high-dimensional text features. Specifically, the L1 penalty is advantageous for datasets containing large-scale warning messages and source code because it can effectively zero out irrelevant or redundant features. Meaning, it performs a form of automated feature selection that simplifies the final model [67]. To support these diverse penalty types at scale, the SAGA solver was selected due to its computational efficiency with large datasets and its ability to find optimal weights within a maximum of 2,000 iterations [22]. Furthermore, the experiment investigates different class weighting strategies. By testing balanced weights alongside custom ratios, the research calibrates the model to be more sensitive to actionable warnings without losing the well-aligned probability estimates that make LR a superior baseline for Brier score evaluation. This calibration ensures that the linear model provides not just a binary prediction, but a trustworthy probability that accurately reflects the likelihood of a Java warning being actionable in a real-world development workflow.

```
param_grid = {
        'Classifier__C': [0.001, 0.01, 0.1, 1, 10, 100],
        'Classifier__penalty': ['l1', 'l2'],
        'Classifier__solver': ['saga'],
        'Classifier__max_iter': [2000],
        'Classifier__class_weight': [None, 'balanced', {0: 1, 1: 1.5}]
    }
```

Listing 4.3: The parameter grid for Hyperparameter Tuning for LR.

EXP: RF Hyperparameter Tuning

To push the RF beyond its baseline performance as obtained in `EXP: Baseline of Standard Models` (Section 4.1.2), we move from data-level adjustments to model-level optimization.

This experiment is thus about the hyperparameter tuning for the RF model. Beyond mere performance optimization, this experiment is designed to combat the common pitfall of overfitting. It can occur when a complex ensemble model like a RF memorizes noise within the training data [44]. The hyperparameter search space is therefore carefully constrained to prioritize model generalization over raw training accuracy. This is implemented by reducing the permitted max_depth, increasing the min_samples_leaf and min_samples_split thresholds, and introducing an explicit limit on the number of leaf nodes. Furthermore, the experiment incorporates Cost Complexity Pruning (CCP) via the ccp_alpha parameter. This technique prunes low-importance splits within the decision trees, ensuring that each branch contributes significantly to the model's predictive power [58]. By searching through this restricted parameter grid (see Listing 4.4), the experiment identifies a configuration that is both robust and highly sensitive, providing a definitive baseline for the model's performance in a real-world, Java development environment.

```
param_grid = {
        'Classifier__n_estimators': [100, 150, 200],
        'Classifier__max_depth': [5, 8, 12],
        'Classifier__min_samples_split': [10, 15, 20],
        'Classifier__min_samples_leaf': [20, 50, 100],
        'Classifier__max_features': ['sqrt', 'log2'],
        'Classifier__max_leaf_nodes': [50, 100, 150],
        'Classifier__bootstrap': [True],
        'Classifier__ccp_alpha': [0.0, 0.001, 0.01]
    }
```

Listing 4.4: The parameter grid for Hyperparameter Tuning for RF.

4.1.4 Interpretability Analysis Phase

The next component of the classical machine learning evaluation is `EXP: Feature Importance`, which as the name suggests, focuses on feature importance analysis. While previous experiments established the predictive power and optimal configuration of the models, this experiment seeks to open the black box of the RF and LR architectures. We do this to understand which specific metadata and code features drive the classification of actionable warnings. To ensure the most accurate insights, this experiment utilizes the best model identified during the hyperparameter tuning phases. This optimized configuration is then trained on the same balanced subset of 200K records from the previous experiments.

To evaluate the underlying drivers of each architecture, the feature analysis is tailored separately to accommodate the distinct RF and LR models. For the RF model, a game-theoretic approach is utilized by passing a representative 200-instance evaluation sample to a SHAP tree explainer. This SHAP explainer calculates Shapley values to capture non-linear feature interactions and generates a global summary plot alongside a localized force plot for individual instance attribution. Conversely, for the linear LR model, importance is extracted directly from the model's learned coefficients. Here a bar plot displays the magnitude and directional influence of the top 20 parameters, and a localized breakdown is constructed by computing the direct linear contribution of each feature value multiplied by its corresponding weight for an individual sample. Despite these different analysis methods,

both configurations utilize a downstream post-processing pipeline that transforms the test sample through the final pipeline preprocessor to recover explicit feature names, perform a dual-layered correlation analysis via a comprehensive feature clustermap, and a targeted 80-feature heatmap to expose multicollinearity among the most dominant predictors.

By explicitly enabling feature importance mapping, the experiment calculates the relative contribution of every input variable, including the derived age of the warning, the mapped CWE-ID, and the latent semantic (LSA) components of the source code and warning messages. This analysis is crucial for validating the research's underlying assumptions. As it reveals whether the model relies more heavily on historical metadata, such as how long a warning has persisted, or on the deep semantic structure of the Java code itself. It also enables the analysis of feature correlation through cluster maps. This is needed because high-dimensional datasets like NASCAR often contain redundant information, particularly among the LSA components and derived metadata. By visualizing these relationships it becomes possible to identify if certain features are highly collinear, meaning they provide nearly identical information to the model. Furthermore, the cluster map provides a hierarchical view of how features group together. Understanding these clusters or feature groups allow for a more nuanced interpretation of the model's decision-making process. This ensures that the final conclusions are not just based on isolated numbers, but on a clear understanding of the interconnected factors that determine whether a Java static analysis warning is actually worth a developer's time.

Ultimately, this experiment provides the necessary bridge between predictive performance and practical software engineering. By identifying the primary drivers of warning actionability, the findings offer actionable insights for developers and tool maintainers. They could suggest which types of metadata are most effective for predicting actionability, and which are redundant. This interpretability ensures that the resulting pipeline is not only effective but also transparent, and it grounds the predictions in the real-world characteristics of the NASCAR corpus.

4.1.5 LLM Experiments

To evaluate whether advanced semantic processing can surpass the limitations of classical ML, this phase introduces an evaluation framework utilizing LLMs. The setup begins with a structured data extraction process where the same subset of 200K warnings is partitioned into train and test sets. These sets are created by executing the scikit-learn train_test_split method with a fixed random state of 42. From the resulting train dataset, a smaller, representative sample of 1000 instances is randomly selected (random_state=42) to undergo inference through a LLM. This initial small-scale evaluation is repeated for each model type: Haiku 4.5, Sonnet 4.6, or Opus 4.6, to identify the most effective candidate before scaling up. The primary reasoning for this tiered approach is cost efficiency. By benchmarking on a 1000 instance sample first, the study minimizes the financial expenditure associated with unnecessary or redundant API calls to expensive high-tier models. Once the optimal model is identified through this cost-controlled phase, it is then utilized to perform inference on the full test set, which constitutes 25% of the initial 200K warnings. Because the initial 1000 sample was taken from the train dataset, the model will not be exposed to the test set

and the evaluation is thus unbiased.

For inference, only parts of the defined ML pre-processor are used to prepare the necessary architectural and semantic columns. This includes the creation of *Warning Category* and *Age*. Other features like *n Source Code*[3], *CWE-ID*, *Warning Message*, and *Tool* are also injected into a specialized prompter function designed to elicit a single probability score between 0 and 1. This represents the likelihood of a warning being actionable. Following the generation of these probability scores, an evaluation is done by converting the LLM's string outputs into numeric values. Any failures in model response format are handled by a penalty mechanism that fills missing values with a neutral score of 0.5. This effectively indicates that the model was unable to make a determined judgment. To ensure the fairest possible comparison against classical models, the system does not rely on a fixed 0.5 decision boundary. Instead, it dynamically calculates a Precision-Recall curve to identify the optimal threshold that maximizes the F1-score for the specific sample. This best threshold is then applied to generate binary predictions, allowing for the calculation of final accuracy, precision, recall, F1, and Brier scores.

The zero-shot configuration serves as the primary baseline, where the model is presented with a persona-driven prompt defining it as an expert in ASAT triaging (see Listing 4.5). This base prompt was iteratively tuned using examples from the training dataset until the final, optimal configuration was established. The distinction between the zero- and the one-shot experiment is that the one-shot prompt is enriched with a single, specific example of a warning and its correct classification to guide the model's pattern recognition (see Listing 4.5). This example is selected from the training set to serve as a representative demonstration of the task requirements. Foundational research into in-context learning indicates that the transition from zero-shot to one-shot often yields a significant improvement in performance, as it provides the model with a concrete template for the expected output format and reasoning style [19]. By focusing the evaluation on a single example, the experiment assesses the model's ability to adapt rapidly using minimal task-specific data [71]. This approach also serves a practical purpose regarding cost management, as it avoids the increased latency and token consumption associated with larger example sets that may yield diminishing returns in classification performance compared to the initial gain provided by the first demonstration.

```
# Base instructions (System Role & Output Format)
    base_instructions = f"""You are an expert in triaging warnings
    generated from Automated Static Analysis Tools (ASATs). Your task is
    to analyze a provided warning and determine whether it is a true
    positive (actionable issue) or a false positive (safely ignored).

    To make this determination, you will consider:
    1. Warning Message: A description of the warning.
    2. Source Code Context: A snippet of the source code around the
    relevant line.
    3. Category: The specific category of warning (e.g., "Performance", "
    Security", "CodeStyle").
    4. CWE-ID: The Common Weakness Enumeration identifier (or None).
```

[3] n is the amount of lines above and below the warning, so $n = 2$ gives 5 lines of code.

```
    5. Tool: The name of the static analysis tool (e.g., "PMD", "SpotBugs
    ").
    6. Age: The age of the warning in seconds.

    Output Format:
    Produce a single probability score between 0 and 1, where closer to 1
     means true positive, and closer to 0 means false positive.
    CRITICAL: The ONLY output you should provide is a single floating-
    point number (e.g., 0.85). Do not include any additional text,
    thinking, markdown blockquotes, or explanations."""

    # Assemble the prompt based on shot count
    if shot == 0:
        prompt = f"""{base_instructions}

        This is the warning you need to analyze:
        Warning Message: {warning['warning_msg']}
        Source Code Context: {warning['source_code_2']}
        Category: {warning['category']}
        CWE-ID: {warning['CWE-ID']}
        Tool: {warning['tool']}
        Age: {warning['age']}
        Score:"""

    else:
        prompt = f"""{base_instructions}

        Here are {shot} examples of warnings along with their correct
    label:

        {examples}

        Now, analyze this specific warning and generate its probability
    score based on the logic shown above:
        Warning Message: {warning['warning_msg']}
        Source Code Context: {warning['source_code_2']}
        Category: {warning['category']}
        CWE-ID: {warning['CWE-ID']}
        Tool: {warning['tool']}
        Age: {warning['age']}
        Score:"""
```

Listing 4.5: The prompt used for the zero- and one-shot experiments.

4.2 User Study

The objective of this user study is to transition from model performance to practical utility. While the previous section focuses on the model's ability to predict a label, the user study evaluates how these predictions influence a developer's or consultant's workflow. Beyond performance, it is also essential to understand the ways in which developers and consultants perceive the score as adding value or if the model's presence introduces risks of

over-reliance. To address these questions, the study recruits a diverse group of participants consisting of both professional developers and consultants (these are called technical and security consultants in SIG). Additionally, it focuses on individuals with varying levels of experience in code auditing and tool-assisted analysis.

The deliberate choice was made for a qualitative methodology over a quantitative approach for this study. While numerical data can confirm that a model is accurate, it cannot explain the cognitive processes or the trust dynamics that occur when a human interacts with an automated system. One of the primary reasons for this choice is the relatively small population pool available for the study. Since the research is conducted within the specific professional environment of SIG, the number of eligible participants is naturally constrained by the specialized expertise required for ASAT triage. In such a setting, a quantitative survey would lack the statistical power necessary to draw generalized conclusions. However, a qualitative approach allows for a much deeper and more meaningful extraction of data from each individual subject.

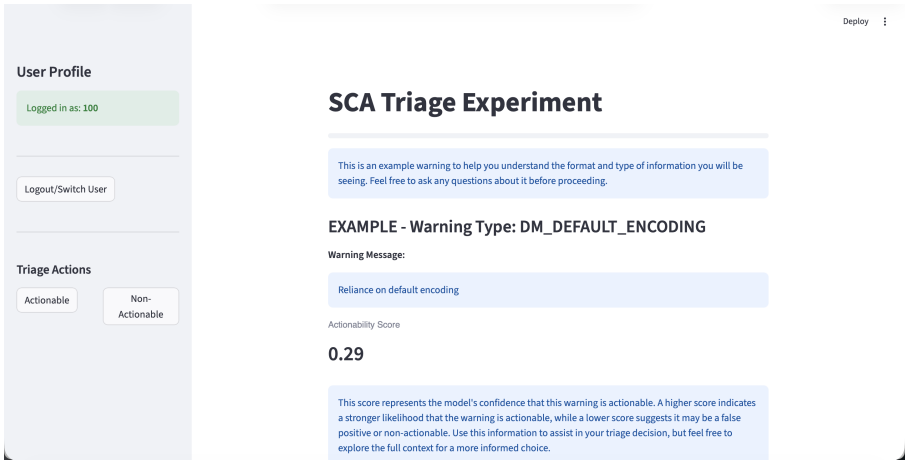
Beyond the constraints of the participant pool, a qualitative framework is better suited for exploring the "how" and "why" behind developer behavior. Quantitative metrics often fail to capture the nuances of professional skepticism or the specific contextual factors that lead a developer to agree or disagree with an algorithmic prediction. By prioritizing depth over breadth, this study can uncover subtle patterns in how consultants or developers interpret confidence scores and whether it aligns with their own technical intuition. Furthermore, the qualitative nature of the study provides the flexibility to follow up on unexpected participant insights in real time. This is essential for identifying the underlying causes of automation bias or trust degradation that a rigid numerical survey would overlook.

4.2.1 The Three Phases

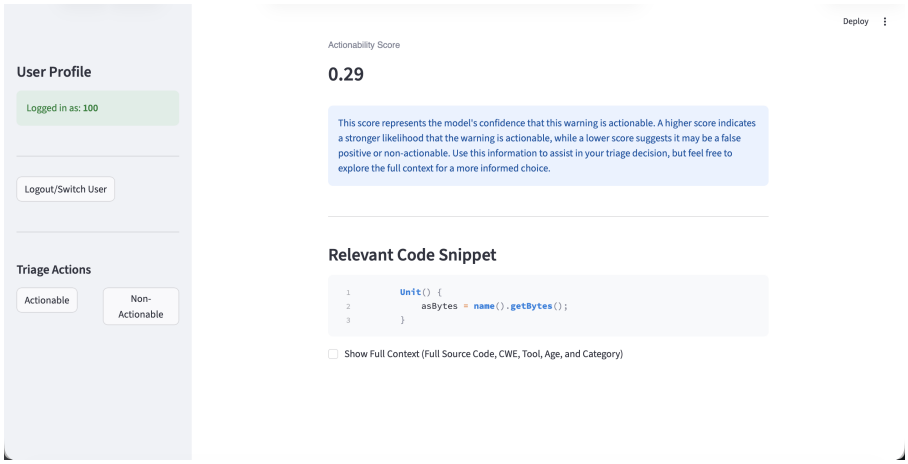
The user study is structured into three distinct phases designed to capture various facets of the user experience. The process begins with the administration of an informed consent form (see Appendix C), which participants must read and sign before proceeding. Following this, the first phase consists of a triage experiment where participants classify warnings as either actionable or non-actionable. For the purposes of this study, an actionable warning is defined as one that the user deems significant enough to warrant a code change. Conversely, a non-actionable warning is one that the user identifies as either a FP or a technically valid issue that does not justify a modification to the source code.

To capture the participants' cognitive processes, a Think Aloud protocol is utilized throughout the experiment. Initially, subjects are presented with two example warnings: one accompanied by a probability score ranging from 0 to 1, and one without. Each warning display includes the warning type, the warning message, the probability score if applicable, and a relevant code snippet (see Figure 4.2). Should the participant require further context to complete the classification, they may activate an information toggle to reveal the CWE-ID, the age of the warning in days, the originating tool, the warning category, and the full source code with start- and end-line of the warning.

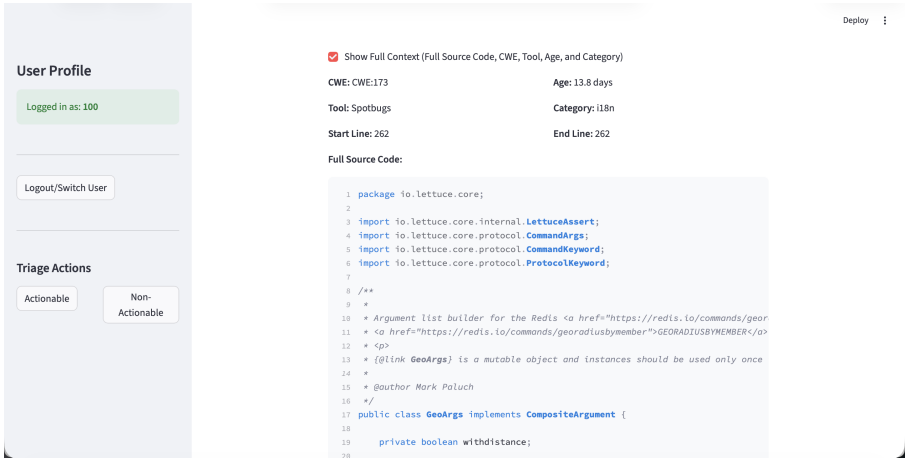
This study also employs a counterbalanced design to mitigate potential order effects and account for the learning curve. Participants are assigned to one of two groups based on



(a) The initial view of the warning triage dashboard interface.



(b) The second expanded view of the warning triage dashboard interface.



(c) The third expanded view of the warning triage dashboard interface.

Figure 4.2: Comprehensive breakdown of the warning triage dashboard interface across its primary views.

their user ID. Even-numbered participants are first exposed to seven warnings with scores followed by seven without, while odd-numbered participants receive the warnings in the reverse order. In total, each participant triages 14 warnings across seven different categories, and is exposed ti two example warnings of the same category. Each category contains one warning with a probability score and one without to ensure the results are not skewed by a single category, and to maintain consistency between the two experimental conditions.

Data collection for this phase is comprehensive. In addition to audio recordings of the Think Aloud process, the system logs final classifications, time spent per warning, and whether the additional information toggle was utilized. Furthermore, all screen activity is recorded to monitor external research behaviors, such as searching via Google or utilizing generative AI tools like ChatGPT. Collecting these metrics allows for a nuanced analysis of the relationship between information seeking, time pressure, and decision-making accuracy. Before commencing, participants are informed that there is no time limit and that they are encouraged to use external resources as they would in a professional setting (see Figure 4.3). Finally, the study incorporates trap warnings to evaluate model bias and over-reliance. These include two non-actionable warnings with various high probability scores. This allows us to determine if participants are critically evaluating the warnings or simply deferring to the model's output.

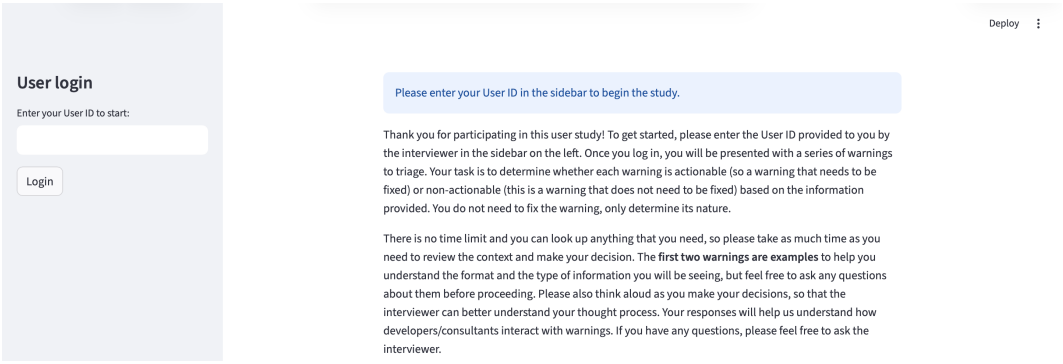


Figure 4.3: The starting page for the triage experiment part of the user study. It details all the information the participants need to start classifying warnings.

After the triage experiment, participants complete a questionnaire hosted on Microsoft Forms via a secure TU Delft institutional account. This survey begins by collecting demographic and professional data, including the participant's specific development role and years of experience to contextualize their triage performance against their industry expertise. It further evaluates their familiarity with how ML models generate probability scores and their experience with specific ASATs. By gathering this information, the study can determine if a participant's background in identifying vulnerabilities or their technical understanding of model outputs correlates with their ability to identify the intentionally misleading scores introduced during the triage phase.

The second half of the questionnaire focuses on the perceived utility and trust of the probability scores provided. Participants rate the importance of various metadata components, such as the CWE-ID, source code, and the specific warning age. This selection of

data points is designed to identify which technical indicators most effectively help a developer validate a model's prediction. Additionally, the survey investigates the threshold for automation bias by asking participants to specify the minimum probability score at which they would feel comfortable skipping manual verification under a tight deadline and under normal circumstances. This specific inquiry helps determine the level of confidence required for developers to trust model scores in high-pressure, real-world scenarios. It then finishes with a question that asks the participant to rate the helpfulness of the score during the triage experiment.

The final stage of the study consists of a semi-structured interview, which is conducted to explore the nuances of the participant's decision-making process. By utilizing a semi-structured format, we can maintain a consistent core set of inquiries (see Appendix D) while allowing the flexibility to pursue spontaneous follow-up questions based on specific observations made during the Think Aloud triage phase. This qualitative approach is essential for understanding the underlying "why" behind user behaviors, such as why a participant chose to ignore a high-probability score or why they prioritized certain metadata like the CWE-ID over the code snippet. By concluding with this open-ended interaction, the study gathers rich, descriptive data on the perceived helpfulness of the scores and identifies potential improvements for the integration of ranked warnings into professional developer workflows.

4.2.2 Theoretical Thematic Analysis

To systematically evaluate the responses collected during the user study, this research employs thematic analysis. As established by Braun and Clarke [16], thematic analysis is a foundational and highly flexible qualitative method utilized for identifying, analyzing, and reporting patterns (also referred to as themes) within a dataset (in our case the recorded responses). Unlike other qualitative methodologies that are strictly bound to specific epistemological or theoretical frameworks, thematic analysis operates independently of theory. Within this method, themes can be identified in either an inductive (bottom-up) or a theoretical (top-down) manner. For the purposes of this study, a theoretical approach is utilized. An inductive approach codes data without trying to fit it into a pre-existing coding frame, allowing themes to emerge organically from the data itself. In contrast, a theoretical approach is explicitly driven by the researcher's predefined analytic interests and specific research questions. [16] This is exactly why this approach was taken. Our user study is not designed to broadly explore the general experiences of developers, but rather to investigate highly specific operational phenomena.

The triage experiment and questionnaire are deliberately structured to test specific cases, such as utilizing trap warnings to evaluate automation bias and querying specific trust thresholds. Therefore, employing a theoretical approach ensures that the coding process remains strictly focused on how developers interact with and validate model scores, rather than attempting to provide a generalized, descriptive account of the entire qualitative dataset. Consequently, the qualitative data is coded and analyzed to extract insights specifically related to two primary, pre-defined theoretical themes:

1. *The Influence of Probability Scores*

   This theme explores the perceived utility and operational impact of the generated scores on the participant's workflow. It focuses on identifying patterns related to how scores alter the chronological triage order, workflow efficiency, and the manifestation of automation bias (e.g., whether a participant blindly relies on a high probability score without adequate critical evaluation).

2. *The Trust and Validation of Probability Scores*

   This theme evaluates trust calibration. It specifically investigates the conditional nature of participant trust, aiming to identify which distinct features (such as Source Code Context, Warning Messages, or Age) developers actively require to manually validate probability scores and establish sustained confidence.

Chapter 5

Results

This chapter presents the results of the study setup discussed in Chapter 4. It begins with the results of the classical ML and LLM experiments. This details the performance trade-offs, optimal configurations, and predictive capacities of both architectures. Following this, the chapter moves on to the results of the qualitative user study. This final phase showcases the practical impact of the model's probability scores on developer workflows, and traces how the participants interact with the probability scores, and what their trust level is in the probability scores.

5.1 Experiments

This section presents the results obtained from the experimental design detailed in Section 4.1. To provide the findings, the presentation of results follows the same logical progression as the experimental design (see Figure 4.1 for the overview again).

5.1.1 Results for the Dimensionality Optimization Phase

Results EXP: Optimal Feature Space

The results of this experiment identify the best configuration for text-based features yielding the highest information density. It evaluates the performance of five distinct TF-IDF trials across varying LSA component caps. As presented in Table 5.1, the evaluation tracks several key metrics: *Total Variance* (the overall semantic information captured by the maximum component cap), the mathematical *Elbow point* (the optimal number of components before diminishing returns set in), *Variance at Elbow* (the amount of information successfully retained at this optimal cutoff), and the primary performance metric, the *Efficiency Score*, which measures the ratio of the variance captured at the elbow point to the number of components required to reach that threshold.

The trial results indicate that configurations with higher frequency filtering correspond to improved performance when processing large-scale software repository data. While the n_gram_specialist and wide_spectrum trials yield high raw Total Variance, they require a significantly larger number of components at their elbow points and introduce greater

noise. Conversely, the denoised_heavy trial consistently records the highest Efficiency Scores across all multi-component evaluations.

Looking specifically at the values in Table 5.1, at the 1000 component cap, the denoised_heavy configuration for Source Code achieves the highest Efficiency Score of 0.0025 by capturing 58.17% of the variance at an elbow of 231 components. In comparison, the broader wide_spectrum trial drops to an efficiency of 0.0015, needing 244 components to capture only 36.25% variance at its elbow. This trend continues for Warning Messages. At the 1000 cap, denoised_heavy achieves an excellent Efficiency Score of 0.0079, successfully capturing 78.73% of the semantic variance using just 100 components.

This denoised_heavy trial restricts the feature set to 2000 tokens and mandates a minimum document frequency of ten. So, it successfully excludes low-frequency, document-specific data points while retaining broader coding patterns and standardized warning structures. Therefore, because the denoised_heavy trial definitively provides the most optimal balance between high semantic information retention and low computational dimensionality, it is explicitly selected as the best text-processing configuration. Consequently, this specific configuration is locked in and utilized for all subsequent classical ML experiments, including the dataset splits, baseline evaluations, and hyperparameter tuning phases.

Additionally, the experiments across the 50, 300, and 1000 component caps reveal a distinct difference in information density between warning messages and source code. For the selected denoised_heavy trial in the 1000 component iteration, the warning message reached a total variance of 0.9593 with an elbow point identified at 100 components, capturing 78.7% of the total information. This degree of compression suggests that the vocabulary used in warning messages is remarkably standardized across different Java projects.

The source code exhibits a more diffuse information distribution compared to the warning messages. In the same 1000 component run for the denoised_heavy configuration, the source code achieved a total variance of 0.8748, with the elbow point occurring at 231 components to capture 58.17% of the variance. This result confirms that the semantic meaning of Java source code is spread across a broader latent space, which in turn could require more dimensions to preserve the semantic intent compared to the more concise warning messages.

To use these findings, three distinct feature selection strategies are formulated (see Table 5.2). Each represent a different level of granularity within the latent space. The derivation of these specific component counts is rooted in the comparative analysis of the elbow points identified across the 300 and 1000 LSA component experimental runs. The Conservative strategy, directly adopts the mathematical elbow points[1] discovered during the 300-component trial. The transition to the Balanced and Rich strategies is then based on the 1000-component stress test. This reveals that while warning message variance plateaus significantly at approximately 100 components, source code information is far more diffuse. Consequently, the Balanced strategy employs 100 components for warnings to capture approximately 79% of its variance and sets a 150-component threshold for source code as a calculated middle ground that captures semantic detail before the Efficiency Score drops sharply. The Rich strategy further extends this for source code to 250 components,

[1] Elbow points represent the initial plateau where the most dominant structural signals are fully stabilized.

Cap (N)	Trial Name	Type	Total Var.	Elbow	Var @ Elbow	Efficiency
50	denoised_heavy	Source	0.3484	16	0.2347	0.0147
50	n_gram_specialist	Source	0.3059	16	0.2048	0.0128
50	baseline	Source	0.2942	16	0.2023	0.0126
50	bigrams_10k	Source	0.2422	17	0.1726	0.0102
50	wide_spectrum	Source	0.2172	17	0.1577	0.0093
50	n_gram_specialist	Warning	0.7468	12	0.5822	0.0485
50	denoised_heavy	Warning	0.7270	12	0.5778	0.0481
50	bigrams_10k	Warning	0.6456	11	0.4890	0.0445
50	baseline	Warning	0.6726	12	0.5310	0.0442
50	wide_spectrum	Warning	0.6025	11	0.4535	0.0412
300	denoised_heavy	Source	0.6291	80	0.4117	0.0051
300	n_gram_specialist	Source	0.5761	81	0.3660	0.0045
300	baseline	Source	0.5254	80	0.3460	0.0043
300	bigrams_10k	Source	0.4391	82	0.2865	0.0035
300	wide_spectrum	Source	0.3858	82	0.2553	0.0031
300	denoised_heavy	Warning	0.8712	35	0.6969	0.0199
300	baseline	Warning	0.8118	35	0.6436	0.0184
300	n_gram_specialist	Warning	0.9009	42	0.7293	0.0174
300	bigrams_10k	Warning	0.7804	38	0.6221	0.0164
300	wide_spectrum	Warning	0.7333	38	0.5804	0.0153
1000	denoised_heavy	Source	0.8748	231	0.5817	0.0025
1000	n_gram_specialist	Source	0.8291	248	0.5420	0.0022
1000	baseline	Source	0.7313	231	0.4864	0.0021
1000	bigrams_10k	Source	0.6321	246	0.4131	0.0017
1000	wide_spectrum	Source	0.5536	244	0.3625	0.0015
1000	n_gram_specialist	Warning	0.9950	102	0.8142	0.0080
1000	denoised_heavy	Warning	0.9593	100	0.7873	0.0079
1000	baseline	Warning	0.8971	101	0.7313	0.0072
1000	bigrams_10k	Warning	0.8659	102	0.7042	0.0069
1000	wide_spectrum	Warning	0.8158	104	0.6607	0.0064

Table 5.1: Comparison of LSA variance and Efficiency Scores across different component caps and trial configurations.

specifically to align with the 231-to-248 component elbow point identified in the 1000 LSA component run. It thereby ensures that even the subtle long tail patterns of the code are represented.

Strategy	Warning Message LSA (n)	Source Code LSA (n)	Total Features
Conservative	35	80	115
Balanced	100	150	250
Rich	100	250	350

Table 5.2: LSA component selection strategies based on explained variance Elbow Points.

Results EXP: Optimal Data Splits

The results of this experiment, as summarized in Table 5.3, demonstrate the significant sensitivity of both default RF and LR architectures[2] to variations in class distribution. The data reveals a clear trade-off between absolute accuracy and the ability to retrieve the minority actionable class as the dataset shifts from balanced to imbalanced ratios.

Both models exhibit their highest recall and F1-scores in the balanced distributions (200K Balanced and 100K Balanced). The RF model maintains a stable F1-score between 83.03% and 84.62%, while the LR model hovers near 72%. Notably, the performance remains similar, even with a reduced sample size of 100K records. This suggests that for balanced data, the models reach a plateau of learning early. The precision-recall balance in both splits is also nearly symmetrical. This means that neither model is predisposed to majority-class bias when the training environment is artificially leveled.

The transition to the original NASCAR ratio in a reduced dataset (200K Original, this is 12.9% actionable and 87.1% non-actionable) and the full dataset highlights a sharp divergence in model behavior. While the accuracy for the RF model reaches its peak at 93.40% (in the full dataset), this metric proves deceptive as a measure of utility. The recall for the RF drops from 82.47% in the balanced split to 57.91% in the full dataset, and even more severely to 44.53% in the reduced original-ratio split. In contrast, the LR model is impacted more heavily with its recall plummeting to 25.97% on the full dataset. This drop in recall confirms that when faced with the raw 87% non-actionable rate of the NASCAR corpus, both models prioritize minimizing global error by favoring the majority class. However, the RF model's precision remains high (89.50%). This suggests that while it identifies fewer actionable warnings in imbalanced settings, the warnings it does flag are highly likely to be correct.

The controlled imbalanced splits (30/70 and 40/60 actionable/non-actionable splits) provide a granular view of the tipping point for model reliability. In the 40/60 split, the RF maintains a robust F1-score of 81.92% and a recall of 77.33%. In contrast, the LR begins to show signs of sensitivity to the imbalance. While it maintains a respectable precision of 74.15%, its recall drops to 58%. This suggests that even a 10% shift away from a balanced distribution starts to impede the LR's ability to find the decision boundary for the minority class. Furthermore, in the 30/70 split, the RF model's recall dips further to 66.95%, but it compensates this with a high precision of 87.62%. However, the LR model suffers a loss in predictive power. Its recall falls to 45.53%, meaning it fails to identify more than half of the actionable warnings. So across both architectures as the proportion of actionable warnings decreases, the models progressively shift toward prioritizing the avoidance of non-actionable warnings at the direct expense of missing actionable warnings.

Finally, across all six configurations, the Brier score consistently favors the RF. It maintains lower (so better) scores, ranging from 0.066 to 0.165. Meanwhile, the LR scores higher (so worse) with 0.114 to 0.266. This suggests that the RF's probabilistic estimates remain more reliable and better calibrated than those of the LR, regardless of how skewed the underlying data distribution becomes. These findings indicate that while training on the full, imbalanced dataset maximizes precision, it creates a failure in recall.

[2]By default, we mean the default values associated for both models as defined in the scikit-learn library.

Data Split	Model	Acc.	Prec.	Rec.	F1	Brier
200k Balanced	RF	85.09%	87.11%	82.47%	84.72%	0.1490
	LR	73.54%	75.57%	69.55%	72.44%	0.2646
100k Balanced	RF	83.52%	85.35%	80.85%	83.03%	0.1647
	LR	73.35%	75.67%	68.65%	71.99%	0.2665
200k Original	RF	91.70%	83.37%	44.53%	58.06%	0.0830
	LR	88.57%	66.89%	24.41%	35.77%	0.1143
Full Dataset	RF	93.40%	89.50%	57.91%	70.32%	0.0660
	LR	88.29%	67.16%	25.97%	37.45%	0.1171
40TP/60FP	RF	86.37%	87.09%	77.33%	81.92%	0.1363
	LR	75.11%	74.15%	58.00%	65.09%	0.2489
30TP/70FP	RF	87.20%	87.62%	66.95%	75.90%	0.1280
	LR	78.52%	72.57%	45.53%	55.95%	0.2148

Table 5.3: Evaluation of RF vs. LR across various data distributions.

5.1.2 Results for Baseline Phase

Following the methodology outlined in Section 4.1.2, the baseline results for both models are presented in Table 5.4. The data reveals distinct performance profiles for the two selected models across training and testing phases. Building upon the findings established in the first experiment `Optimal Feature Space` (Section 5.1.1), this baseline evaluation includes three distinct LSA component strategies. To evaluate performance sensitivity across varying levels of semantic granularity, the baseline experiment is executed over three iterations. These iterations evaluate combinations of warning message and source code LSA components at ratios of 35/80, 100/150, and 100/250, respectively (see Table 5.2).

Strategy	Model	Split	Accuracy	Precision	Recall	F1-Score	Brier
Conservative (35/80)	RF	Train	99.54%	99.35%	99.73%	99.54%	0.0180
		Test	85.53%	86.55%	84.17%	85.34%	0.1099
	LR	Train	71.48%	72.66%	68.83%	70.69%	0.1859
		Test	71.55%	72.85%	68.83%	70.79%	0.1853
Balanced (100/150)	RF	Train	99.50%	99.31%	99.71%	99.51%	0.0185
		Test	85.46%	87.14%	83.05%	85.04%	0.1099
	LR	Train	73.47%	75.51%	69.59%	72.43%	0.1763
		Test	73.38%	75.30%	69.27%	72.16%	0.1765
Rich (100/250)	RF	Train	99.50%	99.30%	99.71%	99.50%	0.0186
		Test	85.55%	87.33%	83.01%	85.12%	0.1105
	LR	Train	74.84%	76.84%	71.21%	73.92%	0.1689
		Test	74.81%	76.65%	71.03%	73.73%	0.1696

Table 5.4: Baseline Performance Comparison across LSA Strategies (train vs. test).

The baseline performance results, as summarized in Table 5.4, demonstrate the varying

sensitivities of the two architectures to the underlying LSA components. The RF establishes a baseline with test F1-scores that remain stable near 85% across all three strategies. However, it exhibits a significant divergence between training and testing metrics. While training F1-scores exceed 99.50%, the approximately 14% performance decrease on the test set indicates a strong tendency toward overfitting. This overfitting trend is consistently reflected across all evaluated metrics. In the Conservative strategy (35/80), the training F1-score of 99.54% drops to a test F1-score of 85.34%, with precision shifting from 99.35% to 86.55% and recall moving from 99.73% to 84.17%. When moving to the Balanced strategy (100/150), the metrics remain tightly bounded, yielding a test F1-score of 85.04%, test precision of 87.14%, and test recall of 83.05% against near-perfect training scores. In the Rich strategy (100/250), the RF achieves a test F1-score and test precision of 85.12% and 87.33% respectively, alongside a test recall of 83.01%. Meanwhile, the training scores are all near-perfect. Furthermore, the model's error calibration mirrors this training-testing divide. Across all three LSA component configurations, the training Brier score remains extremely low. This train Brier score ranges between 0.0180 and 0.0186, whereas the test Brier score increases to a range of 0.1099 to 0.1105.

In contrast, the LR model starts with a lower test F1-score of 70.79% in the Conservative strategy. However, its predictive power improves steadily as the LSA components increase. It achieves a test F1-score of 72.16% in the Balanced configuration (100/150) and eventually reaches its highest test F1-score of 73.73% in the Rich configuration (100/250). Unlike the RF, the LR baseline exhibits high consistency between the training and testing phases. All metrics typically remain within a narrow margin of each other across all iterations. Specifically, in the Conservative strategy, train and test F1-scores are nearly identical at 70.69% and 70.79% respectively. Notably, this is the only strategy in which the test F1-score is better than the train F1-score. Furthermore, the Conservative strategy has a train precision of 72.66% and test precision of 72.85%, and an identical recall of 68.83% for both train and test. This tight margin persists in the Balanced strategy, where test F1-score hits 72.16% (compared to 72.43% train) and test precision reaches 75.30%. It then also extends into the Rich strategy, where test F1-score peaks at 73.73% and test precision reaches 76.65%. Furthermore, an analysis of the Brier scores reveals that probabilistic calibration remains highly consistent between the splits, even as it shifts across strategies. As the feature space expands from Conservative to Rich, the LR training Brier score decreases from 0.1859 to 0.1689, and the test Brier score similarly decreases from 0.1853 to 0.1696. This indicates that as more components are introduced, the linear model simultaneously improves both its discrete classification metrics and its probabilistic error calibration.

Comparing the two architectures, the RF model consistently outperforms the LR model across all primary classification metrics and strategies on the test set. Even in the Rich configuration where the LR model reaches its peak performance with a test F1-score of 0.7373, the RF model maintains an absolute lead of 0.1139 (scoring 0.8512). Additionally, the RF model produces lower test Brier scores across all evaluations, such as 0.1099 versus 0.1853 in the Conservative configuration. This demonstrates a lower overall probability error floor in its leanest form. However, the data highlights a severe generalization gap for the RF model that is entirely absent in the linear model. While the LR model's training and testing metrics remain tightly bounded within fractions of a percent of one another across all strate-

gies, the RF model exhibits a stark divergence. The RF training accuracy and F1-scores consistently exceed 0.9950 across all three iterations, yet these drop by approximately 14% on the test sets. This demonstrates that while the RF extracts greater predictive utility from the feature space than the LR model, it operates under overfitting. These baseline findings establish a benchmark for the subsequent optimization of hyperparameter depth.

Based on these comparative trends, the Conservative strategy (35/80 LSA components) is selected to carry forward into the hyperparameter tuning phase. This choice is primarily driven by the observation that increasing the dimensionality of the semantic feature space yields no substantial added value to the classification metrics of the top-performing architecture. For the RF model, expanding the feature space from the Conservative to the Rich strategy results in a negligible test F1-score variance of only 0.22% (85.34% to 85.12%), while the test Brier score remains effectively stagnant. While the LR model does demonstrate a marginal upward trajectory in predictive power with more components, its peak test F1-score (73.73%) remains substantially below the baseline established by the RF with the least LSA components. Consequently, proceeding with the 35/80 configuration aligns with the principle of parsimony (Occam's razor), which dictates that a simpler model structure with fewer parameters should be preferred if it yields equivalent predictive performance [29].

5.1.3 Results for Model Optimization Phase

Results EXP: LR Hyperparameter Tuning

Moving on to optimizing the previous discussed baseline models, the results of this experiment demonstrate the efficiency of the successive halving approach. As shown in Figure 5.1, the HalvingGridSearchCV begins at iteration 0 with 36 candidate parameter combinations evaluated on an initial resource of 5,555 samples. Unlike the more complex RF search, the LR search converges rapidly. By iteration 3, with approximately 149,985 samples, the scores for the remaining candidates has tightly clustered. This indicates that the linear model has reached a stability point where further data does not significantly alter the weights of the L1 or L2 penalties. The right plot in Figure 5.1 confirms this stability. It shows a very narrow band of performance where the F1-score hovers consistently around 0.72 throughout the entire halving process.

The final performance metrics for the optimized LR are shown in Table 5.5. The model achieves a test F1-score of 72.67%, and a notably high test recall of 80.73%. This emphasizes that the tuning of the C parameter and the inclusion of custom class weights successfully pushes the model toward high sensitivity. This in turn allows the model to capture a large majority of actionable warnings. However, it comes at the cost of a lower test precision (0.6607). This suggests that the linear decision boundary, even when regularized with an L1 penalty, is more prone to false alarms than the non-linear ensemble approach. The extremely low delta between training and testing accuracy (only 0.03%) further validates the effectiveness of the SAGA solver and the regularization in preventing overfitting, despite the high-dimensional nature of the text-derived features.

Another observation from the results is the model's calibration as reflected by the brier score of 0.1917. While this score is higher than the RF baseline, the consistent performance

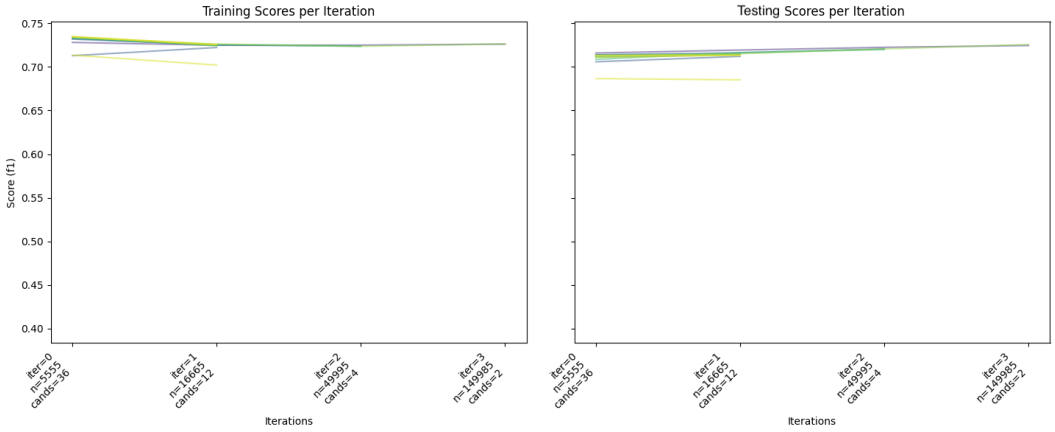


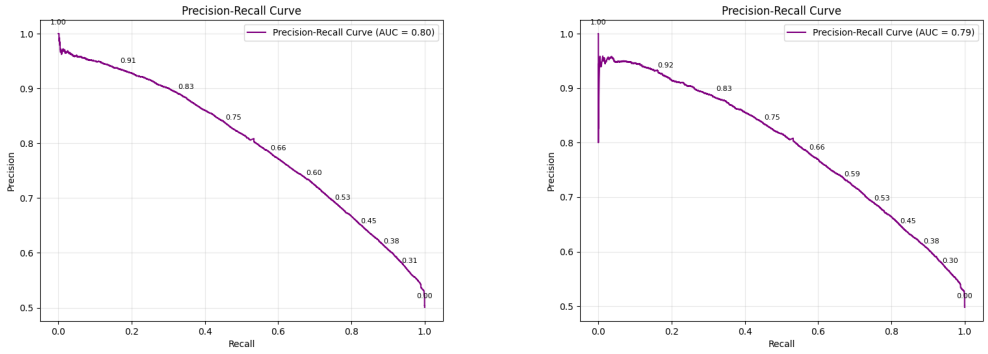
Figure 5.1: Successive halving search progression for Logistic Regression hyperparameter optimization. The training plot (left) and test plot (right) illustrate how the F1-score stabilizes early in the search.

across training (0.1912) and testing (0.1917) phases indicates that the LR model provides highly predictable probabilistic outputs. So while the LR model is stable and recall-heavy, it lacks the precision-focused filtering capability seen in the optimized RF results discussed in the next section.

Metric	**Train Set**	**Test Set**
Accuracy	69.76%	69.73%
Precision	66.17%	66.07%
Recall	80.95%	80.73%
F1-Score	72.82%	72.67%
Brier Score	0.1912	0.1917

Table 5.5: The performance metrics for the optimal Logistic Regression model found during hyperparameter tuning.

Furthermore, the precision-recall characteristics of the optimized LR model are illustrated in Figure 5.2. It presents the metric tradeoffs across the training and testing environments. The training phase yields an Area Under the Curve (AUC) of 0.80, while the testing phase achieves a near-identical AUC of 0.79. Both curves share a matching downward trajectory as recall increases. At the origin where recall is minimal, both plots demonstrate a peak precision starting marker of 1.00. Notably, at the very beginning of the test curve, the precision score drops sharply down to 0.80 at a recall near 0.0, but it recovers immediately back to its stable trajectory above 0.90 within a fraction of a recall point. Aside from this initial transient dip, the precision for both splits stays above 0.90 until recall reaches approximately 0.25. After this, the curve maintains roughly a linear decline for precision across the remaining recall spectrum.



(a) Train set performance.

(b) Test set performance.

Figure 5.2: Precision-Recall Curves for the optimized Logistic Regression model across training and testing splits.

An examination of the discrete threshold markers along the curves further reveals a high degree of numerical consistency between the splits. At a recall value of approximately 0.20, the threshold is marked at 0.91 on the training curve and 0.92 on the testing curve, keeping precision near 0.93 in both instances. Moving to the center of the trade-off curve, the threshold marker for 0.75 aligns identically at a recall of approximately 0.47 and a precision of 0.83 for both sets. Minor variations emerge only at higher recall boundaries. For instance, at a recall around 0.68, the training curve records a threshold marker of 0.60 compared to 0.59 on the test curve. Similarly, near a recall of 0.95, the threshold value reads 0.31 for the train set and 0.30 for the test set, before both curves terminate at a final threshold marker of 0.00 where recall reaches 1.00 and precision drops to 0.52.

Results EXP: RF Hyperparameter Tuning

The results of this experiment demonstrate the progression of the successive halving search and the final performance of the optimized RF model. By constraining the search space to prioritize generalization, the experiment successfully prevents the overfitting problem observed in the previous baseline discussed in Section 5.1.2.

As illustrated in Figure 5.3, the tuning process begins at Iteration 0 with 1,458 unique candidate parameter combinations evaluated on a resource of 205 samples per fold. As the iterations progresses, the HalvingGridSearchCV discards lower-performing candidates and reallocates resources to the top-performing configurations. In the initial phases of the search (from $iter = 0$ to $iter = 2$), the left plot shows that the training F1-scores for the top candidates experience an upward trajectory. It rises from approximately 77% toward a peak near 85% as the sample size increases from $n = 205$ to $n = 1845$. However, as the available training data expands more aggressively in subsequent iterations, scaling up to $n = 149,445$ by $iter = 6$, the training scores consistently decreases and it ultimately stabilizes just below 80%. This behavior occurs because the model is forced to generalize across a much larger and more diverse data footprint rather than fitting to a small sample subset. Conversely, the testing scores displayed in the right plot of Figure 5.3 indicate a steady upward trend

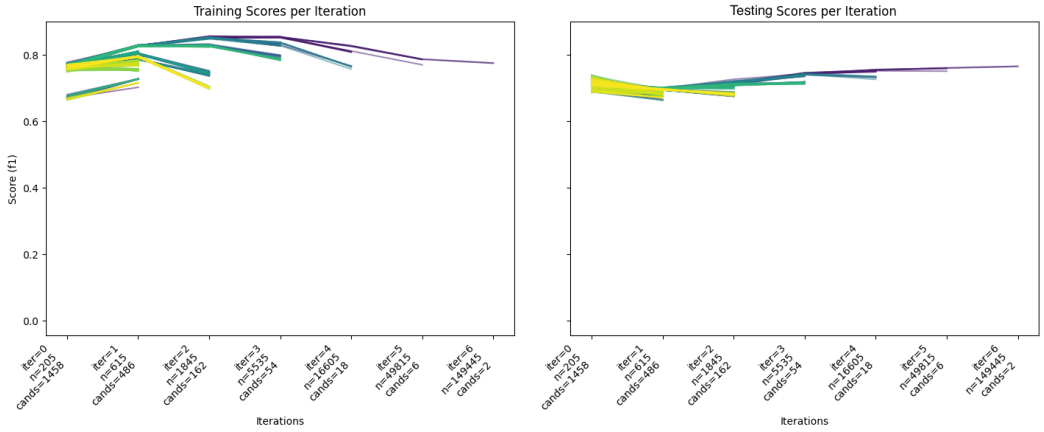


Figure 5.3: Successive Halving search results for Random Forest hyperparameter tuning. The left plot shows the convergence of training F1-scores, while the right plot illustrates the improvement in test F1-scores as the resource n increases across iterations.

as the iterations progress. The initial mean testing scores cluster tightly between roughly 68% and 74% during the earliest iterations before climbing steadily to a peak of 76.85% in the final configuration. As a result of these opposing trajectories, the training scores and testing scores converge closely in the final stages of the halving process. This tight alignment between the two metrics demonstrates that the applied hyperparameter tuning effectively prevents the model from memorizing noise. So the model can maintain stable generalization boundaries even as the training volume increases to its maximum capacity

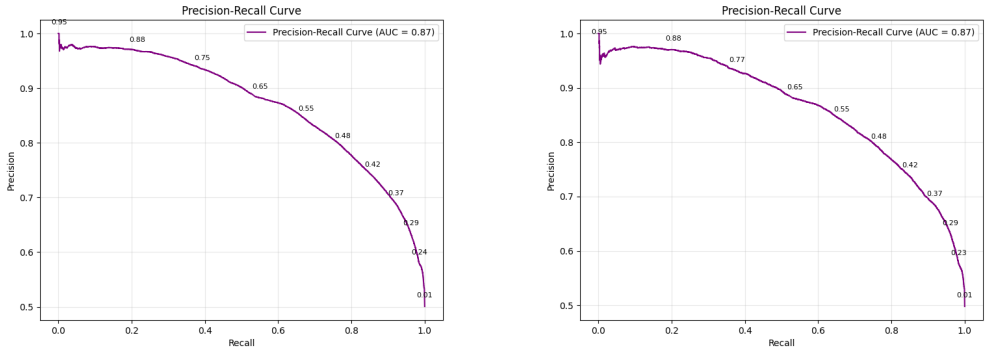
The final performance metrics for the best candidate RF model are detailed in Table 5.6. The optimized RF achieves a test F1-score of 76.85%, supported by a test precision of 80.07% and a test recall of 73.88%. The closeness of the train F1-score (77.51%) to the test F1-score (76.85%) confirms the success of the hyperparameter tuning. Specifically, it represents a significant improvement in generalization compared to the overfitting baseline. Furthermore, the model maintains a Brier Score of 0.1549 on the test set. This indicates that despite the introduction of pruning and depth limits, the model remains well-calibrated in its probability estimates.

Metric	**Train Set**	**Test Set**
Accuracy	78.39%	77.70%
Precision	80.75%	80.07%
Recall	74.52%	73.88%
F1-Score	77.51%	76.85%
Brier Score	0.1532	0.1549

Table 5.6: The performance metrics for the optimal Random Forest model found during hyperparameter tuning.

Finally, the precision-recall curves of the optimized RF across both training and testing

phases are illustrated in Figure 5.4. Both the training and testing evaluations yield an identical AUC of 0.87. The curves follow a matching trajectory across the entire recall spectrum, demonstrating a consistent trade-off between precision and recall across splits. At a low recall threshold close to 0.0, both plots show a high precision starting point labeled at 0.95. As recall scales upward, precision decreases incrementally, maintaining a value above 0.90 until the recall extends past approximately 0.45 to 0.50.



(a) Train set performance.

(b) Test set performance.

Figure 5.4: Precision-Recall Curves for the optimized Random Forest model across training and testing sets.

A detailed inspection of the threshold markers along the curves confirms the shared metric alignment between the training and testing sets. At a recall of approximately 0.20, the threshold is marked at 0.88 for both splits, where precision remains roughly at 0.97. Moving across intermediate thresholds, the marker at 0.65 sits near a recall of 0.53 and a precision of 0.88 in both plots. A minimal divergence occurs around a recall of 0.40, where the threshold marker reads 0.77 on the test curve compared to 0.75 on the train curve. At the highest recall boundaries, the curves drop sharply. So at a recall of approximately 0.95, the threshold reaches 0.29 with precision near 0.65, ultimately terminating at a threshold marker of 0.01 as recall reaches 1.0 and precision hits 0.50.

5.1.4 Results Interpretability Analysis Phase

Feature Importance Analysis for RF

The results of the feature analysis demonstrate the complex interplay between historical metadata and semantic code representations. The SHAP summary plot (Figure 5.5) provides a clear ranking of the primary drivers behind the model's predictions. The warning age emerges as the most influential feature, as shown by its high SHAP values and broad distribution. High feature values for age are associated with negative SHAP values. This indicates that older warnings are significantly less likely to be classified as actionable. Which in turn suggests that the model is successfully identifying warnings that have likely been ignored or dismissed by developers over time. Following age, several latent semantic com-

ponents from the warning messages, such as LSA indices 9, 4, and 14, show substantial impact. These semantic features exhibit mixed SHAP values, implying that specific linguistic patterns within the warning descriptions are strong indicators of whether a warning is actionable or not. The full list of the LSA components mapped to the top 10 words can be found in appendix E.

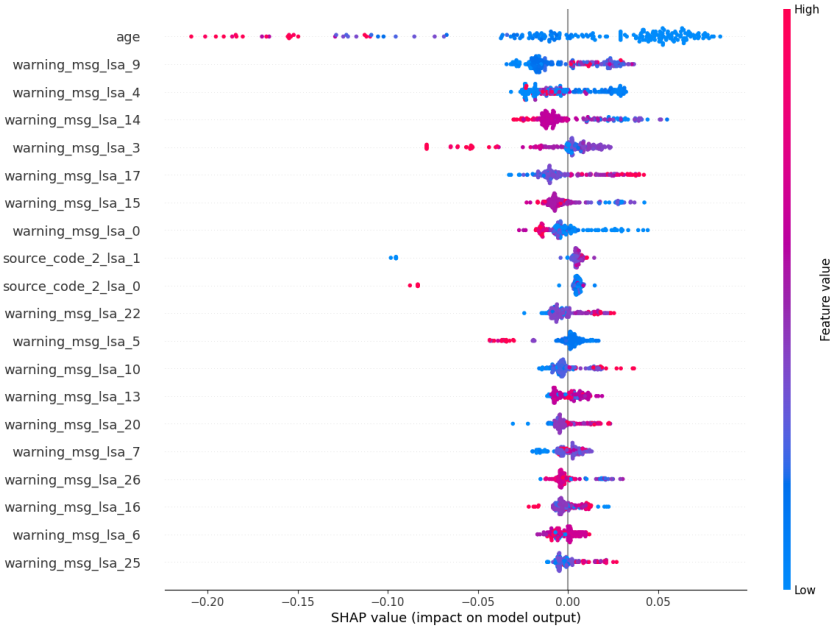


Figure 5.5: SHAP summary plot for the optimized Random Forest model. Features are ranked by their mean absolute SHAP value, indicating their global impact on model output. The color represents the feature value (red for high, blue for low), while the horizontal position indicates whether that value increased (positive SHAP) or decreased (negative SHAP) the probability of a warning being classified as actionable.

The correlation matrix of the top 100 SHAP features (Figure 5.6) further clarifies how these variables interact with one another. The heatmap reveals that the majority of the top-performing features are relatively independent, as evidenced by the dominance of neutral blue tones across the off-diagonal cells. There is very little evidence of strong positive or negative correlation between the age of a warning and the various LSA components. This independence is a positive indicator for the model's robustness, as it suggests that the metadata and the semantic features are providing distinct, non-redundant information to the RF model. Some localized clusters of light correlation appear among specific LSA components derived from the source code snippet, yet these remain weak enough to prevent significant dilution of the feature importance scores. From a software engineering perspective, these correlation patterns establish that the warning age metadata and the semantic LSA components function as independent inputs. For an engineer maintaining or scaling the system, this provides a direct indication that these distinct data streams do not require multi-collinearity mitigation techniques or deduplication protocols. Furthermore, the low correlation between

features confirms that individual variables can be updated, modified, or isolated in future development cycles without a high probability of inducing correlated errors or unpredictable feature-interaction shifts across the broader pipeline.

The hierarchical cluster map or the full correlation matrix (Figure 5.7) provides a broader perspective on the global feature space. It organizes all input variables into a dendrogram based on their statistical similarity. Due to the high dimensionality of the feature set, this visualization serves as a structural overview where the labels are selectively rendered to prioritize the legibility of the overarching clusters. What can be gleaned from this global view is that the LSA components for source code and warning messages tend to group into separate sub-clusters. This validates the decision to treat these as distinct feature sets.

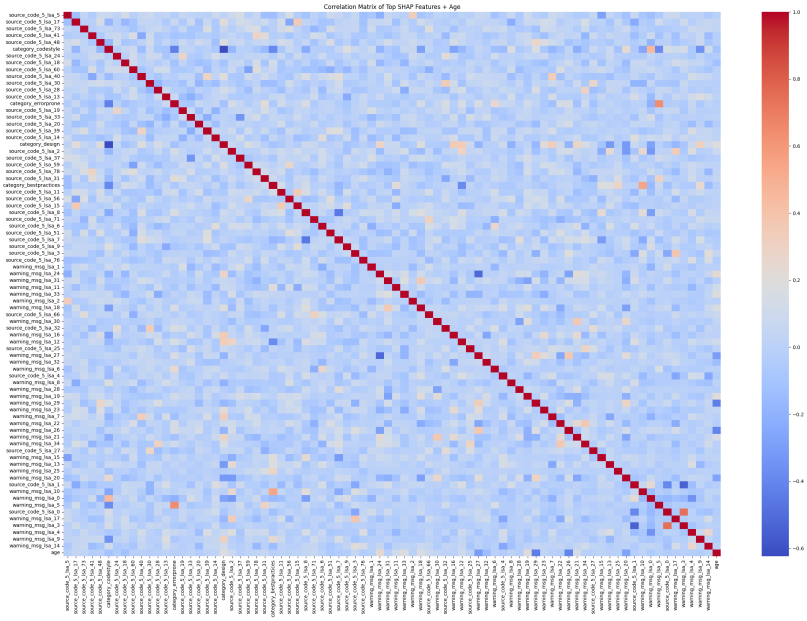


Figure 5.6: Correlation matrix of the top 100 features identified by SHAP analysis including the warning age. The heatmap illustrates the Pearson correlation coefficients between the most influential variables.

A notable observation is the separation of categorical features, such as specific design or best-practice categories, from the denser semantic vector blocks. The presence of a few highly correlated pairs, represented by deep red and blue spots at the edges of the clusters, identifies specific LSA dimensions that may capture overlapping semantic concepts. By viewing the hierarchy as a whole, it is evident that the model is leveraging a diverse set of feature groupings rather than relying on a single monolithic block of information. This hierarchical structure ensures that the final classification is supported by a variety of independent factors ranging from broad technical categories to fine-grained code patterns.

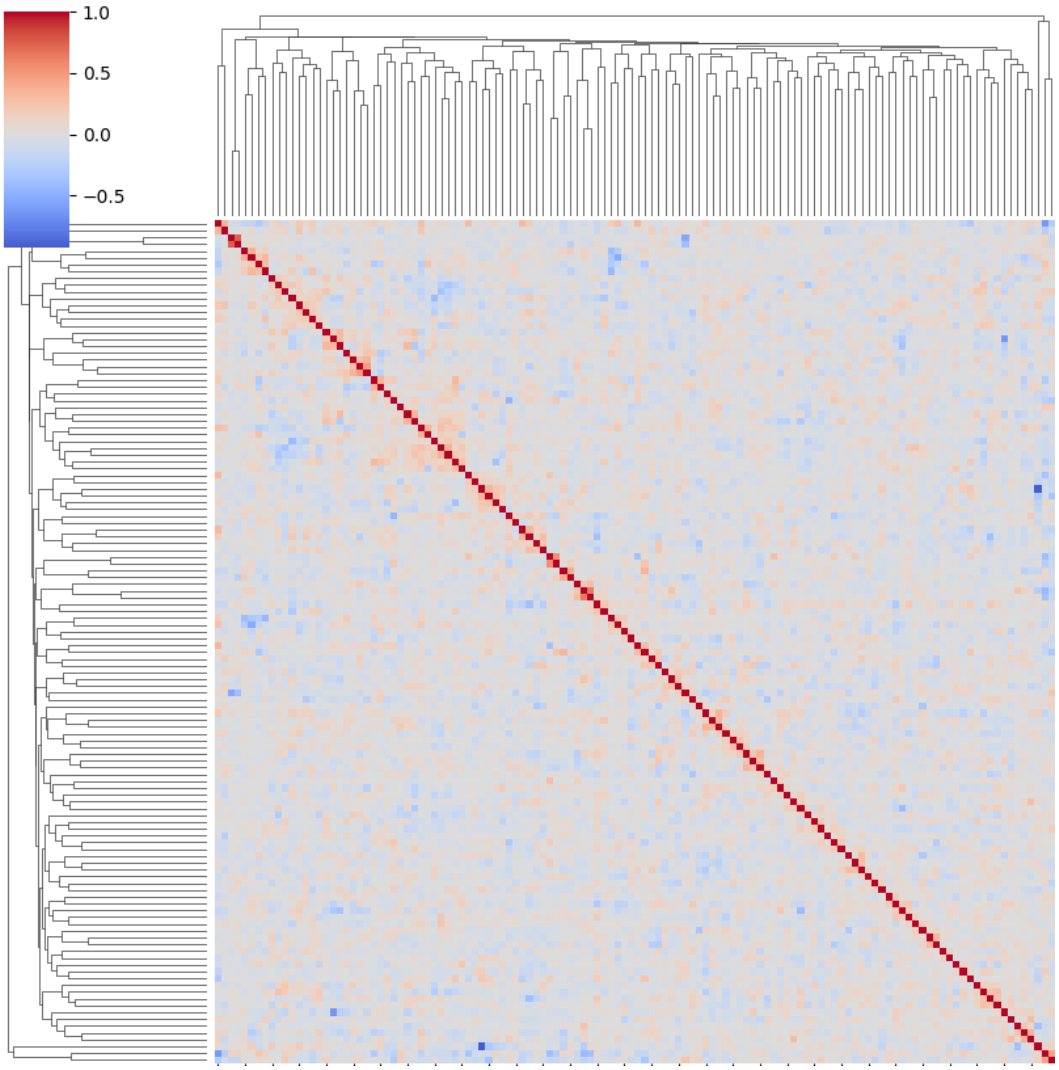


Figure 5.7: Hierarchical cluster map of the feature space. This visualization groups features by their statistical similarity, with the dendrogram on the left and top indicating the hierarchy of clusters.

Feature Importance Analysis for LR

The empirical results of the feature importance analysis for the optimized LR model reveal specific structural and semantic variables driving the likelihood of warning actionability. As illustrated in the top 20 coefficient distribution in Figure 5.8, the model leans heavily on semantic representations extracted via LSA from warning messages and source code context. Negative coefficients indicate features that suppress the predicted probability of a warning being actionable, they are led prominently by `warning_msg_lsa_0` (-0.67), `warning_msg_lsa_5` (-0.51), and `warning_msg_lsa_3` (-0.44). The chronological `age` feature also exhibits a strong negative relationship (-0.34), establishing that older warnings are statistically less likely to be classified as actionable by the architecture. Conversely, positive parameters expand the predicted probability. It is spearheaded by individual LSA dimensions such as `warning_msg_lsa_10` (0.22), `source_code_2_lsa_1` (0.22), and categorical indicators such as the static analysis execution tool `tool_SpotBugs` (0.13).

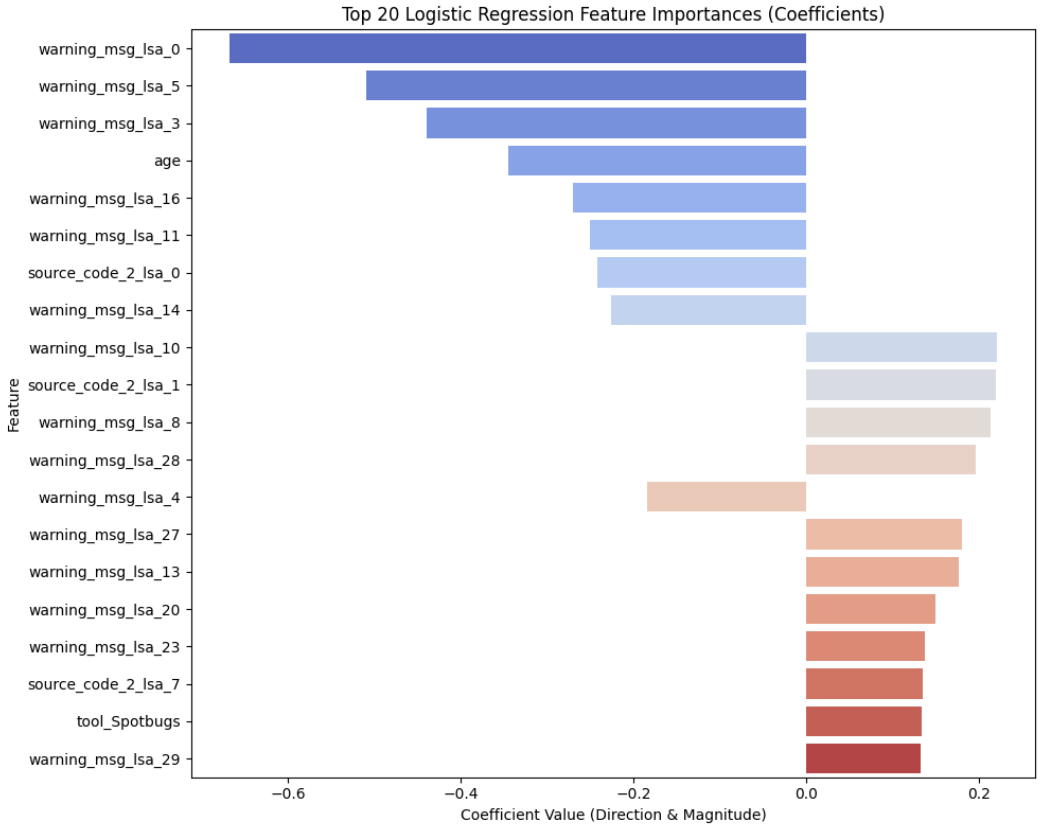


Figure 5.8: The top 20 impactful features of the optimized Logistic Regression model.

The complete feature correlation clustermap in Figure 5.9 provides a global diagnostic of the dataset's underlying dependencies by applying hierarchical clustering to the pairwise correlation matrix of the active feature space. This visualization demonstrates an overall

low degree of systemic multicollinearity across the entire feature space, evidenced by a predominantly neutral baseline of muted, low-magnitude hues hovering near zero ($r \approx 0$). While the hierarchical dendrograms along the top and left axes mathematically bind all variables into a single tree structure, the extreme height at which the branches merge confirms that the features lack strong mutual correlation. Instead, the features remain largely orthogonal and independent of one another. This is punctuated exclusively by the sharp, prominent primary red diagonal representing deterministic self-correlation ($r = 1.0$). Consequently, these visual patterns confirm that the preprocessing pipeline successfully feeds a well-conditioned, non-redundant feature matrix into the LR model. Thereby preventing variance inflation and ensuring the structural stability and interpretability of the LR coefficients. From a software engineering perspective, the absolute independence between the metadata and semantic features proves they capture entirely distinct, non-redundant facets of software quality. Furthermore, this structural independence guarantees mathematical stability in the model's coefficients. This protects the system from erratic priority shifts across code commits and provides developers with stable, predictable, and interpretable justifications they can trust.

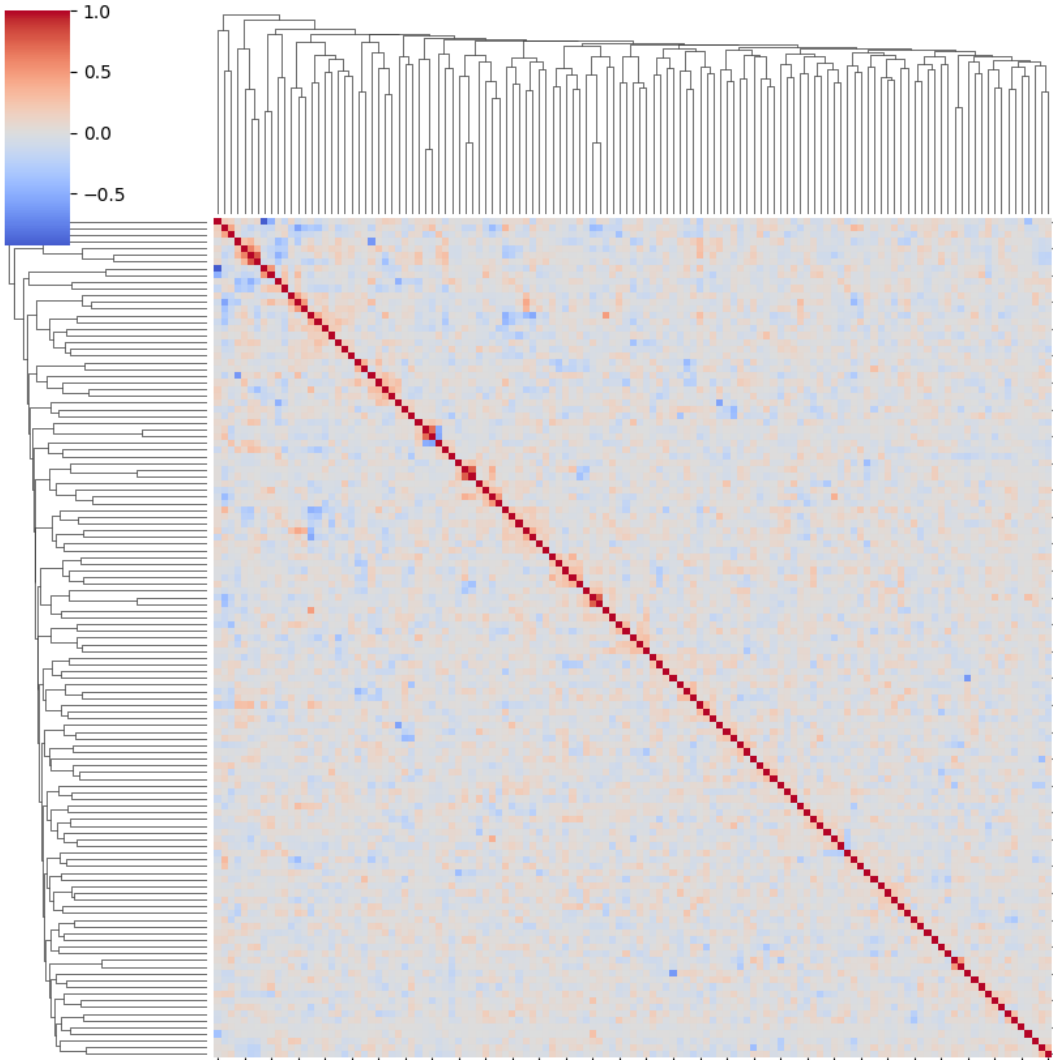


Figure 5.9: The complete feature correlation clustermap of the optimized Logistic Regression model.

However, when evaluating the targeted heatmap of the top 80 most heavily weighted features in Figure 5.10, distinct horizontal and vertical vectors of absolute zero correlation appear. These are shown as solid white crosshair bands in the plot, and include categorical variables such as `CWE-ID:_CWE-609`, `tool_PMD`, and `CWE-ID:_CWE-453`. As mentioned in Section 4.1.4, this visualization is constrained to a small random slice of 200 examples. The presence of these solid white rows occurs because the model assigns high global importance weights to these features during training, yet they do not manifest a single positive instance within this specific 200-sample testing window. Consequently, these features exhibit zero variance within the localized slice, preventing the calculation of any meaningful pairwise correlations. These variables are intentionally preserved in the visualization to illustrate the structural sensitivity of localized interpretability tools when dealing with sparse categorical

feature spaces. Rather than dictating global feature distributions, this phenomenon highlights the contrast between the model's comprehensive structural parameters and the limited data density capture inherent to small validation samples. While increasing the evaluation sample size ($N > 200$) would naturally introduce variance to these sparse categorical features and fill the zero-correlation bands, the sample size was strictly capped to maintain strict parity with the SHAP analysis executed on the RF model. Because TreeExplainer scales with high computational complexity, a uniform, controlled sample size was enforced across both interpretability workers to ensure a mathematically fair and computationally feasible comparative framework.

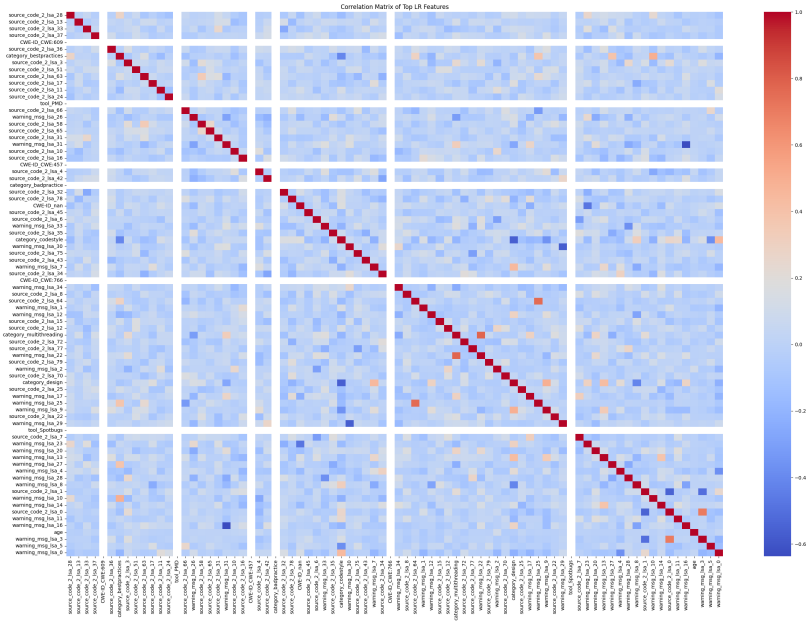


Figure 5.10: The top 80 features correlation clustermap of the optimized Logistic Regression model.

5.1.5 Results LLM Experiments

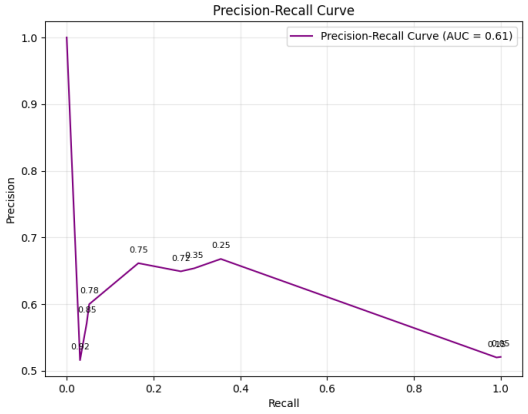
Results for Zero-shot Experiments

The results of the zero-shot baseline evaluation conducted on the preliminary sample of $N = 1000$ instances shown in Table 5.7 demonstrate varying degrees of predictive capability among the three Claude models: Haiku 4.5, Sonnet 4.6, and Opus 4.6. Performance across the models is assessed using precision, recall, the F1-score, the Brier score, and the AUC. Haiku 4.5 and Sonnet 4.6 achieved identical macro-level classification metrics. Both yield an accuracy of 52.1%, a precision of 52.1%, a perfect recall of 100%, and an identical maximum F1-score of 68.5%. Both models also optimized their F1-scores at a dynamic

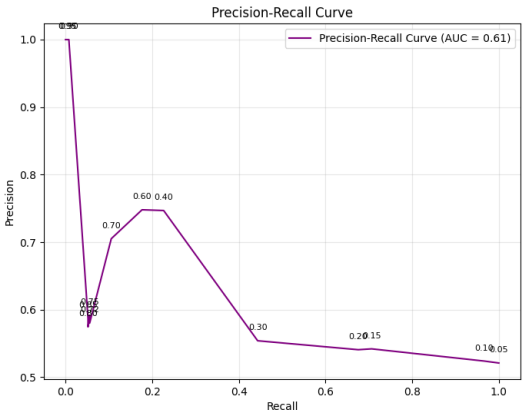
decision boundary threshold of 0.05. However, Sonnet 4.6 demonstrates a slightly lower Brier score of 0.328 compared to the 0.332 observed for Haiku 4.5. This indicates that Sonnet 4.6 has a slight advantage in probability calibration. Analysis of the Precision-Recall curves in Figure 5.11 further reveals that both models achieve an identical AUC of 0.61. This indicates that both models have a comparable overall trade-off between precision and recall across all potential threshold configurations.

Model	Optimal Threshold	Accuracy	Precision	Recall	F1	Brier
Haiku 4.5	0.05	52.1%	52.1%	100%	68.5%	0.332
Sonnet 4.6	0.05	52.1%	52.1%	100%	68.5%	0.328
Opus 4.6	0.08	55.9%	54.2%	99.8%	70.2%	0.348

Table 5.7: Zero-shot performance metrics across LLM candidates ($N = 1000$)



(a) The Precision-Recall Curve for Haiku 4.5 ($N = 1000$).



(b) The Precision-Recall Curve for Sonnet 4.6 ($N = 1000$).

Figure 5.11: Precision-Recall Curves for Haiku 4.5 and Sonnet 4.6.

In contrast, Opus 4.6 outperforms both alternative models across the majority of discriminatory metrics. Operating at an empirically derived optimal decision threshold of 0.08, Opus 4.6 achieves a higher accuracy of 55.9% and a higher precision of 54.2%. While its recall experienced a negligible decrease to 99.8%, its overall classification balance improved. This all culminates into the highest peak F1-score of 70.2%. This superior predictive performance is further validated by its Precision-Recall curve in Figure 5.12, specifically because of the AUC of 0.65. Opus 4.6 thus demonstrates a stronger capacity to maintain higher precision at comparable levels of recall than either Sonnet 4.6 or Haiku 4.5. Although Opus 4.6 exhibits the highest Brier score at 0.348, indicating slightly less precise absolute probability calibration, its superior discriminatory power and higher overall F1-score establish it as the most effective model on this sample. Consequently, based on these empirical findings from the cost-controlled preliminary phase, Opus 4.6 is selected as the optimal model to execute zero-shot inference on the full 25% test set.

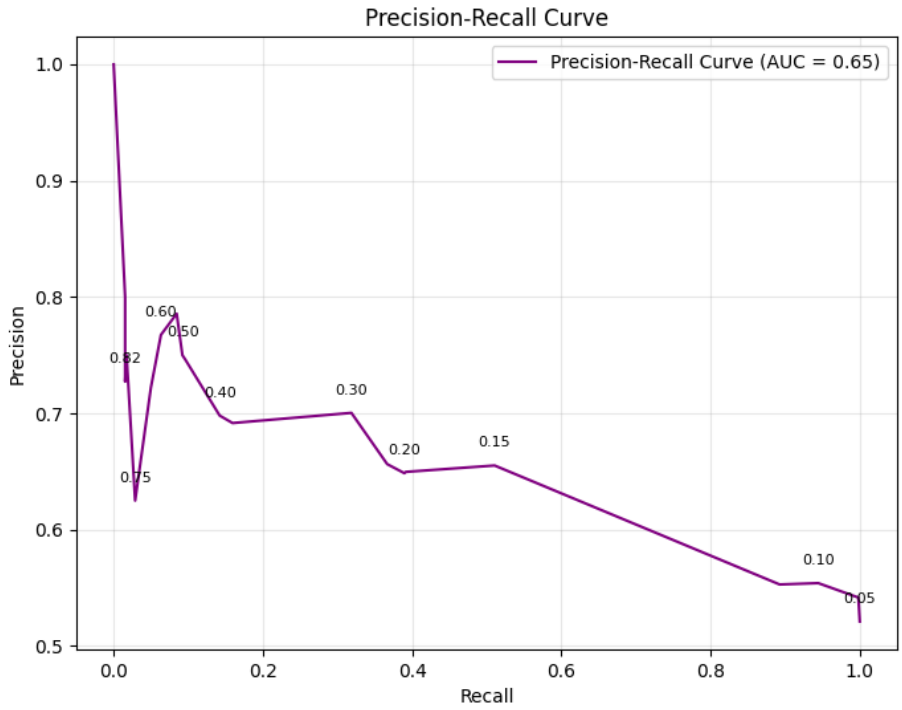


Figure 5.12: The Precision-Recall Curve for Opus 4.6 ($N = 1000$).

To make a fair comparison with the classical ML model, Opus 4.6 is deployed to execute zero-shot inference on the full 25% test dataset. This comprises of $N = 50,000$ ASAT warnings. The performance metrics obtained from this large-scale evaluation are reported in Table 5.8. The model, similarly to the small sample experiment, operates at a decision threshold of 0.08. It achieves a precision of 50.5%, and crucially a high recall of 99.78%. This means it successfully captures nearly all actionable warnings, and culminates in an overall F1-score of 67.0%.

Model	Optimal Threshold	Accuracy	Precision	Recall	F1	Brier
Opus 4.6	0.08	51.1%	50.5%	99.8%	67.0%	0.324

Table 5.8: Zero-shot performance metrics for Opus 4.6 on the full test set ($N = 50,000$).

The comprehensive trade-off between precision and recall across all potential operating boundaries is visualized via the Precision-Recall curve in Figure 5.13. The curve yields an AUC of 0.60. At the chosen threshold anchor point of 0.08, the curve highlights a distinct profile where extreme sensitivity (near-perfect recall) is prioritized and it has a precision of approximately 50.5%. Furthermore, the probabilistic calibration of the model registers an improvement on the larger set. The Brier score decreases from the preliminary 0.348 down to 0.324 on the full dataset. While a score of 0.324 remains above the uninformative 0.25

random-guessing baseline, the downward shift demonstrates that the model's confidence scores stabilize and align more closely with real-world target distributions when evaluated over large-scale, representative open-source data streams. Still the achieved Brier score of 0.324 indicates that the absolute probability estimates remain structurally overconfident or miscalibrated.

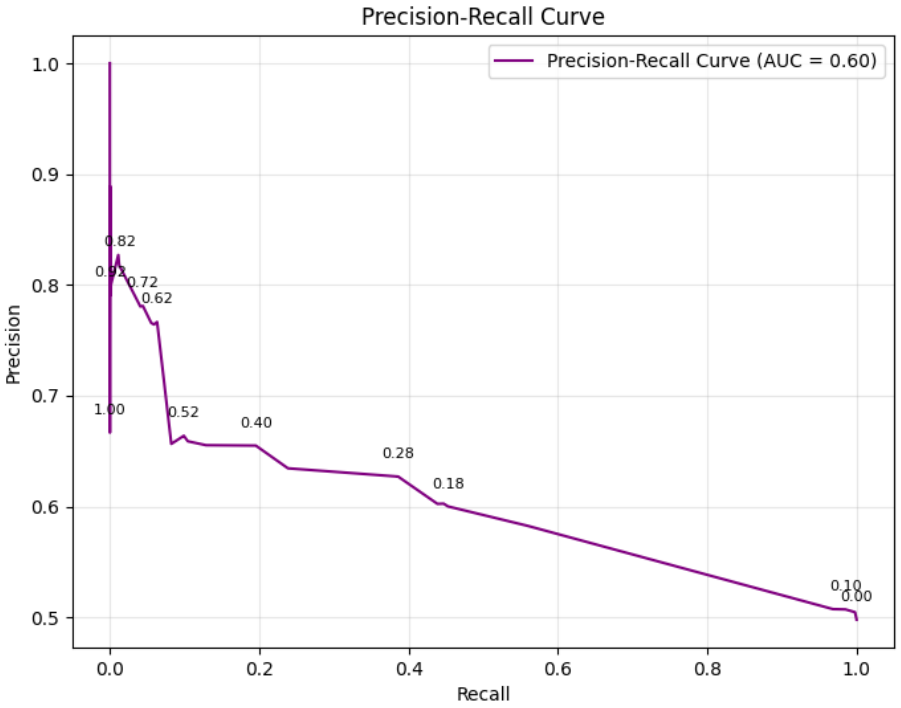


Figure 5.13: The Precision-Recall Curve for Opus 4.6 ($N = 50,000$).

Results for the One-shot Experiment

Following the zero-shot baseline assessment, a 1-shot evaluation is conducted on the preliminary sample size of $N = 1000$ instances to analyze the impact of contextual example provision. The metrics obtained for the three Claude configurations: Haiku 4.5, Sonnet 4.6, and Opus 4.6, are shown in Table 5.9.

Model	Optimal Threshold	Accuracy	Precision	Recall	F1	Brier
Haiku 4.5	0.05	52.1%	52.1%	100.0%	68.5%	0.338
Sonnet 4.6	0.10	53.1%	52.6%	99.2%	68.8%	0.327
Opus 4.6	0.15	54.9%	53.7%	98.3%	69.4%	0.269

Table 5.9: One-shot performance metrics across LLM candidates ($N = 1000$).

Haiku 4.5 registers an optimal decision threshold of 0.05, this gives a precision of 52.1%, a perfect recall of 100.0%, and an F1-score of 68.5%. Its Brier score is 0.338. The corresponding trade-off curve across all threshold boundaries is presented in Figure 5.14, this shows an AUC of 0.59.

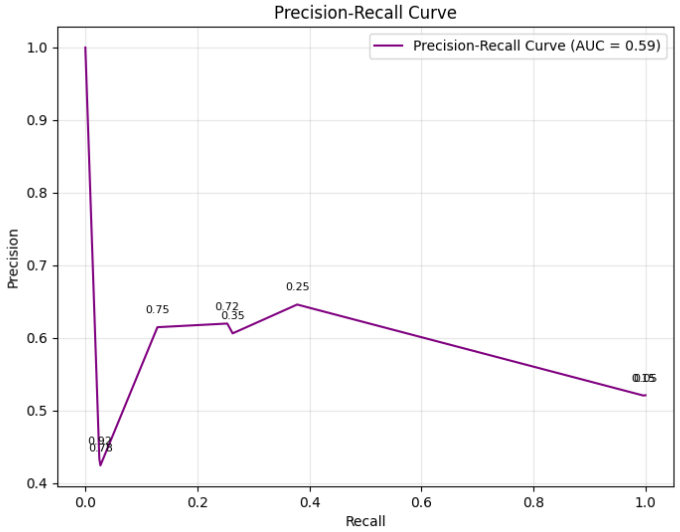
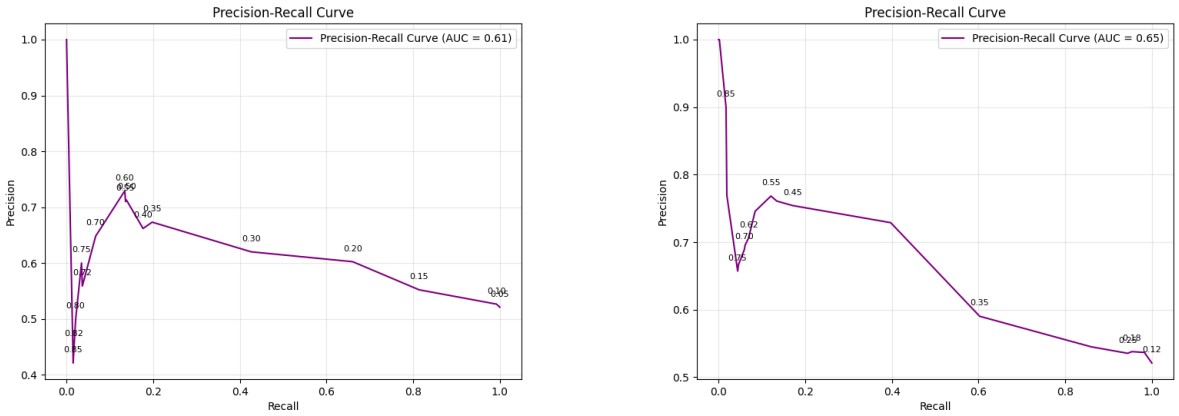


Figure 5.14: The one-shot Precision-Recall Curve for Haiku 4.5 ($N = 1000$).

Sonnet 4.6 optimizes its classification performance at a higher threshold boundary of 0.10 compared to Haiku 4.5. At this configuration, the model achieves a precision of 52.6%, a recall of 99.2%, and an F1-score of 68.8%. Furthermore, it has a Brier score of 0.327. As shown in the Precision-Recall curve in Figure 5.15a, Sonnet 4.6 produces an overall AUC of 0.61. In contrast, Opus 4.6 establishes its optimal F1-score at a decision threshold of 0.15. This model records the highest precision of 53.7% and a recall of 98.3%. These metrics combined for a peak F1-score of 69.4%, which is the highest compared to the other two models. Additionally, Opus 4.6 achieves the lowest Brier score at 0.269. The full threshold distribution is plotted in Figure 5.15b, establishing an overall AUC of 0.65.

Based on these findings from the one-shot preliminary phase, Opus 4.6 is selected as the optimal model to execute large-scale inference on the full 25% test set ($N = 50,000$). While Opus 4.6 demonstrated superior discriminatory capabilities in earlier zero-shot trials, the inclusion of a contextual example successfully addresses its previous probability calibration limitations. Specifically, Opus 4.6 achieves a Brier score of 0.269, marking a substantial reduction in forecast error compared to both its own zero-shot baseline (0.348) and the 1-shot configurations of Haiku 4.5 (0.338) and Sonnet 4.6 (0.327). When evaluating this enhanced calibration alongside its superior classification metrics, like the highest precision (53.7%), thus the highest F1-score (69.4%), and a robust overall AUC of 0.65, Opus 4.6 establishes the most dependable and mathematically sound framework for processing the complete large-scale dataset.



(a) Sonnet 4.6 ($N = 1000$).

(b) Opus 4.6 ($N = 1000$).

Figure 5.15: Comparison of one-shot Precision-Recall Curves between Sonnet 4.6 and Opus 4.6 models.

Model	Optimal Threshold	Accuracy	Precision	Recall	F1	Brier
Opus 4.6	0.12	50.7%	50.2%	99.3%	66.7%	0.313

Table 5.10: One-shot performance metrics for Opus 4.6 on the full test set ($N = 50,000$).

Consequently, a large-scale one-shot inference is executed on the full 25% test dataset. To reiterate, this test dataset comprises of 50,000 warnings. The performance metrics obtained from this extensive evaluation are reported in Table 5.10. Operating at an empirically derived optimal decision threshold of 0.12, the model achieves a precision of 50.2%. and it maintains a near-perfect recall of 99.3%. This means that the model successfully captures almost all actionable warnings, culminating in an overall F1-score of 66.7%. The comprehensive trade-off between precision and recall across all potential operating boundaries is visualized via the Precision-Recall curve in Figure 5.16, which yields an AUC of 0.60. Furthermore, the probabilistic calibration of the model registers a Brier score of 0.313. While this represents a slight improvement in forecast error compared to its zero-shot large-scale baseline of 0.324, the score remains consistently above the 0.25 random-guessing baseline. Additionally, it is higher than the Brier score obtained during the small sample run of the one-shot experiment. A likely explanation for this is that a single contextual example is simply insufficient to represent the vast structural and semantic diversity present across 50,000 warnings. While one demonstration may adequately calibrate the model's probability estimates for a limited, localized sample. Scaling up introduces a massive variety of edge cases, distinct code patterns, and unfamiliar warning types that the single example cannot generalize to. This indicates that even with contextual demonstration, the model's absolute probability estimates remain structurally miscalibrated when processing large-scale data streams.

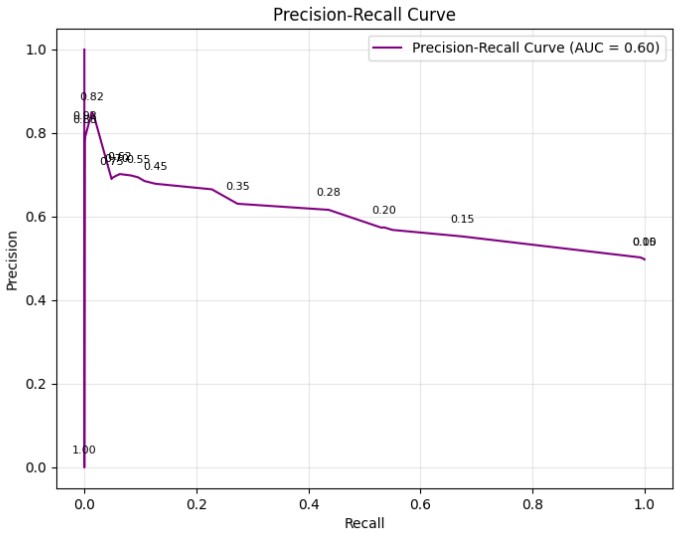


Figure 5.16: The one-shot Precision-Recall Curve for Opus 4.6 ($N = 50,000$).

5.2 User Study

While the preceding experiments evaluated the predictive capabilities of the classical ML models and LLMs, true software engineering utility depends heavily on how they are integrated and perceived by human practitioners. To bridge the gap between model performance and practical application, this qualitative user study investigates how developers interact with, interpret, and trust generated probability scores during the warning triage process.

Following the theoretical thematic analysis methodology established in Section 4.2, the transcribed recording sessions and survey data are systematically coded. To execute this analysis from theory into practice, a deductive, top-down coding strategy was applied. This study aims to investigate highly specific operational phenomena, like trust calibration and automation bias. Consequently, the qualitative data was not allowed to emerge inductively into new themes. Instead, the two overarching themes were predefined based on the core research questions: *The Influence of Probability Scores*, and *The Trust and Validation of Probability Scores*.

This practical execution of the thematic analysis involved systematically reviewing the transcripts, and extracting key participant quotes and behaviors. These data points were then assigned to a set of specific, descriptive labels (or codes) that map directly into the two predefined themes [16]. For *Theme 1: The Influence of Probability Scores* (Section 5.2.2), the data was coded using four primary labels:

- *Workflow Prioritization*: Instances where participants used the score to decide the chronological order of their tasks.
- *Affirmation and Second Opinion*: Behaviors where the score was used to validate a

user's own thoughts or the output of an external tool.

- *Role/Experience Resistance*: Remarks where highly experienced developers deliberately ignored the probability score in favor of their own technical intuition.
- *Automation Bias*: Instances observed during the trap warnings where high probability scores successfully overrode the participant's manual validation process.

For *Theme 2: The Trust and Validation of Probability Scores* (Section 5.2.3), the data was coded using four distinct labels to capture trust dynamics:

- *Code Context Dependency*: The universal requirement for source code snippets to validate the model's performance.
- *Metadata Dismissal*: Explicit statements dismissing certain historical data points, like the age of a warning, as irrelevant to their trust.
- *Conditional Security Skepticism*: The phenomenon where trust thresholds completely collapsed when dealing with high-risk vulnerability categories like Security.
- *External AI Validation*: The emerging workflow where developers copy-paste warnings into external GenAI tools (like ChatGPT) to establish enough context to trust the probability score.

The following subsections detail the participant pool, and the patterns and findings within these two overarching themes.

5.2.1 Participant Pool

The study gathers rich qualitative data from a final pool of 15 professional software and security experts, all working at the collaborating company SIG. The analysis of their responses outlines explicit patterns regarding the perceived utility, risk, and cognitive impact of incorporating these probability scores into a real-world workflow. The demographic composition (see Figure 5.17) represents a balanced distribution across distinct professional roles within the organization. It is split between seven Technical or Security Consultants, seven Software Engineers (Back-end, Front-end, and Dev-ops), and one Researcher.

The participant pool exhibits a diverse range of technical maturity. While the largest single cohort consists of early-career professionals (40% in the 1–3 years bracket), more than half of the total participant pool (53.3%) possesses 4 or more years of professional experience. Furthermore, specific technical fluency in the target programming language (Java) is highly pronounced in 12 out of 15 participants. Additionally, all developers (with the exception of one Front-end Developer) and the Researcher have indicated during the interview phase to possess a good understanding of Java. They have at least 4+ years of industry experience. Conversely, experience among the Consultants is more variable. During the interview phase, three of the Technical Consultants clarified that their background is predominantly in Python with almost no exposure to Java. The rest indicated to have a good understanding of Java and at least 1 year of professional developing experience.

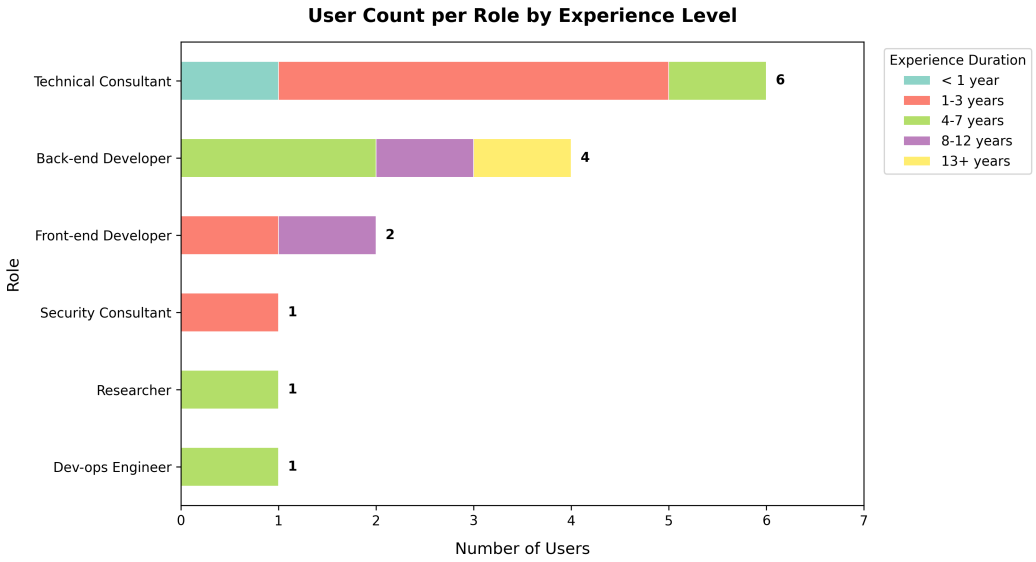


Figure 5.17: Breakdown of the roles and their corresponding experience in years for the 15 participants of the user study.

5.2.2 Theme 1: The Influence of Probability Scores

The qualitative data reveals that the integration of probability scores does not typically replace the manual triage process, but rather acts as a prioritization mechanism. When asked how the scores would impact their workflow if implemented in their dashboard, participants consistently emphasized its value as a sorting tool for navigating high-volume warnings. As Participant 14 noted, "knowing where to start is important, right? Where can I gain the most in terms of security or in terms of maintainability". Similarly, Participant 5 stated, "I would really mostly use the scores for triaging large [sets of warnings]... for ranking".

Interestingly, the scores also served as a second opinion or affirmation tool, particularly for developers lacking specific domain expertise or when utilizing other AI agents. Participant 11, who frequently relied on ChatGPT to understand Java warnings, explained that the probability score "basically helped me to affirm if what ChatGPT was saying [was] correct or not in a way". Other participants viewed the score as an automated safety net, with Participant 4 explicitly saying that they viewed it as "a teammate that catches things [I] missed." However, the influence of the score was not uniform. Several experienced participants (with 4+ years of experience in Java) admitted to unintentionally or deliberately ignoring the probability scores during the experiment. This was often driven by a strong sense of professional ownership over the code. When asked why they became no more or less critical with the introduction of the scores, Participant 5 admitted, "I was too much focused on the task and felt responsible to do it myself," later adding, "I'm too stubborn to adjust my own judgment". This suggests that while probability scores successfully influence the triage order, they face resistance in influencing the final verdict for more experienced developers who rely heavily on their own technical intuition.

To further research this perceived utility, participants are also explicitly asked in the survey to rate the overall helpfulness of the model during their triage process (see Figure 5.18). The responses reflect a cautious optimism. The majority of the group (eight participants) rated the model as either "Very helpful" (three participants) or "Somewhat helpful" (five participants). Four participants remained neutral ("Neither helpful nor unhelpful"), while the remaining three rated the tool as "Somewhat unhelpful." However, the subsequent semi-structured interviews provided critical context for this skepticism. When asked to elaborate on their negative ratings, all three participants who answered "Somewhat unhelpful" clarified that their rating stemmed from the model's inability to definitively dictate the final verdict. They felt the score was unhelpful for explicitly telling them whether a warning was actionable or not, as that still required manual validation. However, they conceded that the probability scores would actually be helpful for organizing and prioritizing how they approach their triage tasks. This distinction reinforces the overarching theme. While participants reject the model as an absolute oracle for actionability, they still recognize its structural value as a workflow prioritization mechanism.

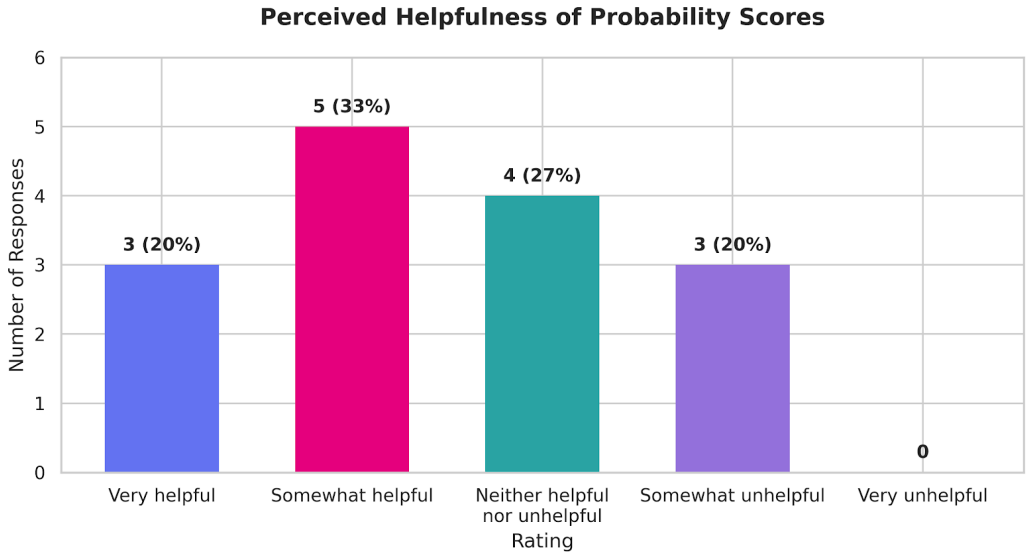


Figure 5.18: The perceived helpfulness of the probability scores for the participants in the user study.

Finally, the study's implementation of two trap warnings[3] reveals a complex relationship between model confidence and participant reliance. For the first trap warning, which was assigned an exceptionally high 98% score, only six participants (Participants 6, 7, 11, 13, 14, and 15) correctly identified it as non-actionable. The remaining nine participants incorrectly trusted the model. For instance, Participant 1 demonstrated clear automation bias, reacting to the 98% score by deferring immediately: "Probability Score 98. Oh, yeah, of

[3]These trap warnings were specifically designed to test for automation bias by presenting non-actionable warnings with artificially high probability scores.

course. Yeah, yeah, yeah. (Laughs) Yeah, yeah, this is very obvious." In contrast, the second trap warning, assigned a 75% probability score, saw a much lower rate of blind reliance. Only five participants (Participants 1, 2, 4, 13, and 14) answered incorrectly, while the remaining ten correctly dismissed the warning. Those who caught the traps often emphasized readability or context over the algorithmic score, as Participant 7 noted when dismissing the 98% trap, "it's probably because it's very confident that it actually found the thing, but that doesn't mean that we should care about it". This indicates that an excessively high score is significantly more likely to override an analytical validation process than a moderately high score. Though a subset of participants will still critically evaluate the code regardless of the model's confidence.

5.2.3 Theme 2: The Trust and Validation of Probability Scores

Next, we research participant trust in the probability scores. This proved to be highly conditional, and was driven almost entirely by the availability of explicit code context and the severity of the warning category. When explicitly asked in the survey to rank which pieces of information were most important to validate a model's accuracy, participants' responses (see Figure 5.19) overwhelmingly favored structural context over historical metadata. The Code Snippet (Source Code) emerged as the absolute most critical feature, with 14 out of 15 participants classifying it as "Very Important" and the remaining rating it as "Important". As Participant 24 explained, "most of my inspection was more based on the code and less on the sort of the metadata... the [probability] score served as a summary of the metadata." Without the ability to read the actual code snippet, participants felt they lacked the necessary context to determine the validity of the warning, with Participant 24 further noting that "without that code text I wouldn't know where the finding comes from."

Beyond the raw code, features denoting the impact of the warning were heavily prioritized. CVSS severity was rated as "Very Important" by eight participants and "Important" by four, while the Warning Category and Warning Type/Rule ID similarly received high importance ratings across the majority of the group. Participant 24 summarized this priority, stating, "CVSS severity is important, uh probably very important... if it's a nine... it doesn't matter what it is, you want to confirm it." Conversely, Warning Age was universally dismissed by the developers. Six participants rated it as completely "Not Important" and five as merely "Slightly Important". Participants argued that the age of an alert does not inherently correlate with its validity. As Participant 24 pointed out, "there's things people miss all the time... I don't think the warning age is important." Participant 22 echoed this sentiment, stating, "Actually, for Warning Age I'm putting that way but way farther down." This reveals a critical operational disconnect, seeing as the results from the feature analysis of both Ml models (see Section5.1.4) show Warning Age to be an important feature when deciding the actionability of a warning. So, while historical metadata like warning age serves as a powerful statistical predictor for these models, the participants actively disregard it during manual validation. Instead, they require transparent source code and severity indicators to establish genuine trust in an automated tool.

To understand reliance on the probability scores, participants were first asked about their likelihood to manually inspect warnings when shown a certain probability score. As

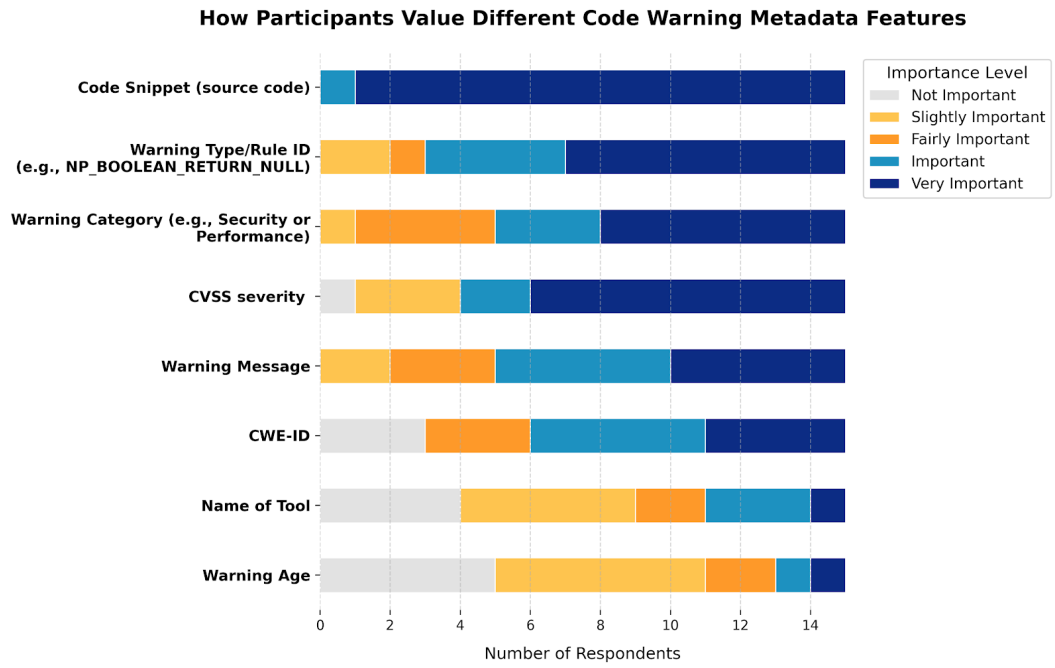


Figure 5.19: Participant perception of the importance of various code warning metadata features. Features are ordered from highest to lowest average perceived importance.

illustrated in Figure 5.20, the general consensus for inspection remains high across all probability score ranges. For warnings with high probability scores (e.g., 80-100%), the vast majority of respondents (14 participants) indicated they would "Always" or "Usually" inspect the warning. They treat the high score as a strong signal for necessary action. Conversely, as the probability score decreases, the commitment to inspect drops. The lowest score ranges (0-40%) are the only brackets where responses severely fracture. This has the highest concentration of participants who would "Never," "Usually Not," or "Rarely" inspect the alert. This suggests that the participants use low probability scores to deprioritize or bypass probable non-actionable warnings, while high scores solidify their decision to manually review the code.

However, when time constraints are introduced, this reliance shifts significantly. When explicitly asked to define the minimum probability score required to skip manual verification under a tight deadline, participant responses (summarized in Figure 5.21) revealed a mix of responses. Rather than a uniform consensus, the answers painted a clear spectrum of trust. The majority of the group set an exceptionally high barrier, with nine participants requiring a score between 75% and 95% before they would bypass a manual check. For these individuals, the model's confidence had to be nearly absolute. Participant 1 demanded a peak threshold of 95%, while Participants 3 and 7 both required scores of "about 90." Similarly, Participants 8 and 11 hovered in the same cautious range, with Participant 8 concluding, "eighty-five... or ninety, I guess. Ninety probably," and Participant 11 stating "eighty-five percent." Even those who were slightly more lenient still required certainty.

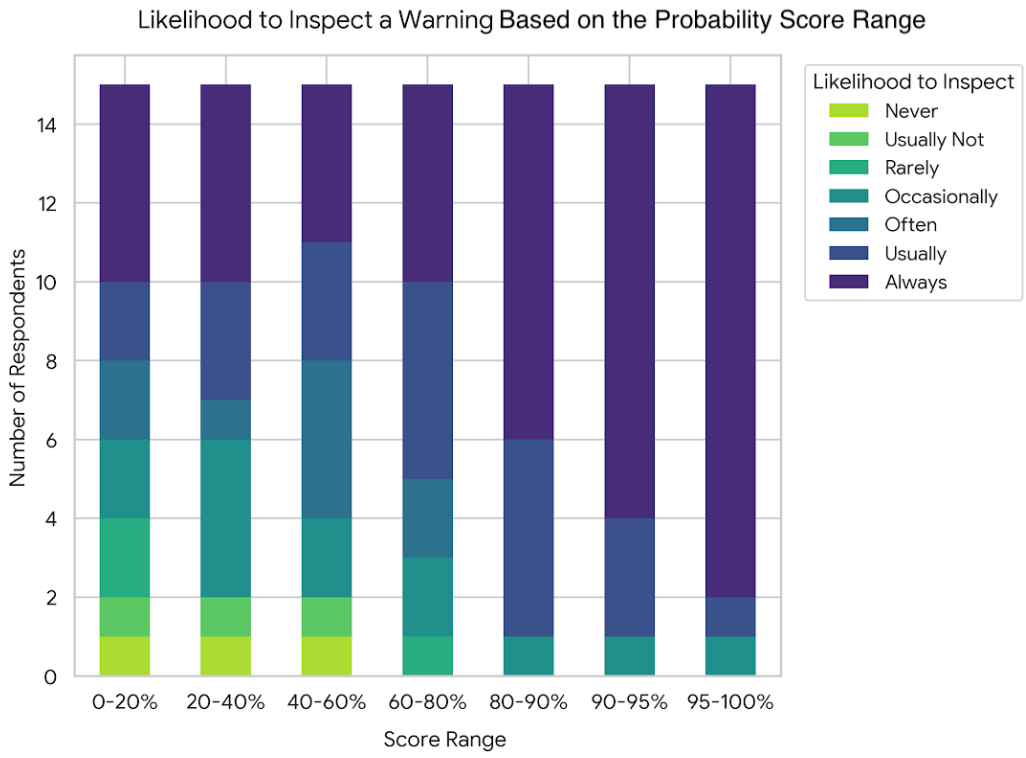


Figure 5.20: The likelihood a warning will be inspected by a participant based on the probability scores, shown in ranges.

Participant 4 placed their threshold at 80%, explaining that "eighty percent is a pretty good thing if you trust your model," a sentiment directly mirrored by Participant 12. Furthermore, Participants 24 and 25 indicated a threshold of 75%.

In contrast, a smaller subset of three participants exhibited significantly lower thresholds, though their reasoning often stemmed from uncertainty rather than blind faith. Participants 6 and 10 selected 50%, with Participant 10 explicitly attributing this choice to the model's black-box nature: "this is because I don't know its internals and how it works and how trustworthy it is." Participant 2 set the lowest threshold at 40%, noting that for scores "40% and below, I would not even check." However, the most telling indicator of skepticism came from a third faction who refused to surrender their autonomy to the score at all. Participants 15, 19, and 23 firmly stated they "would never skip manual verification regardless of the score". Furthermore, the study revealed that even when a participant does establish a numerical trust threshold, that baseline instantly becomes obsolete when dealing with high-stakes categories. Participant 14 perfectly summarized this conditional reliance by noting they would skip manual verification at "75% for other [categories]" but "Never for security". Majority of the participants (10 out of 15) also indicated something similar to this during the interview section. They expressed a willingness to trust the model and skip manual verification for lower-stakes code style or maintainability issues, but universally drew a hard line at vulnerable categories like Security. Participant 2 perfectly captured

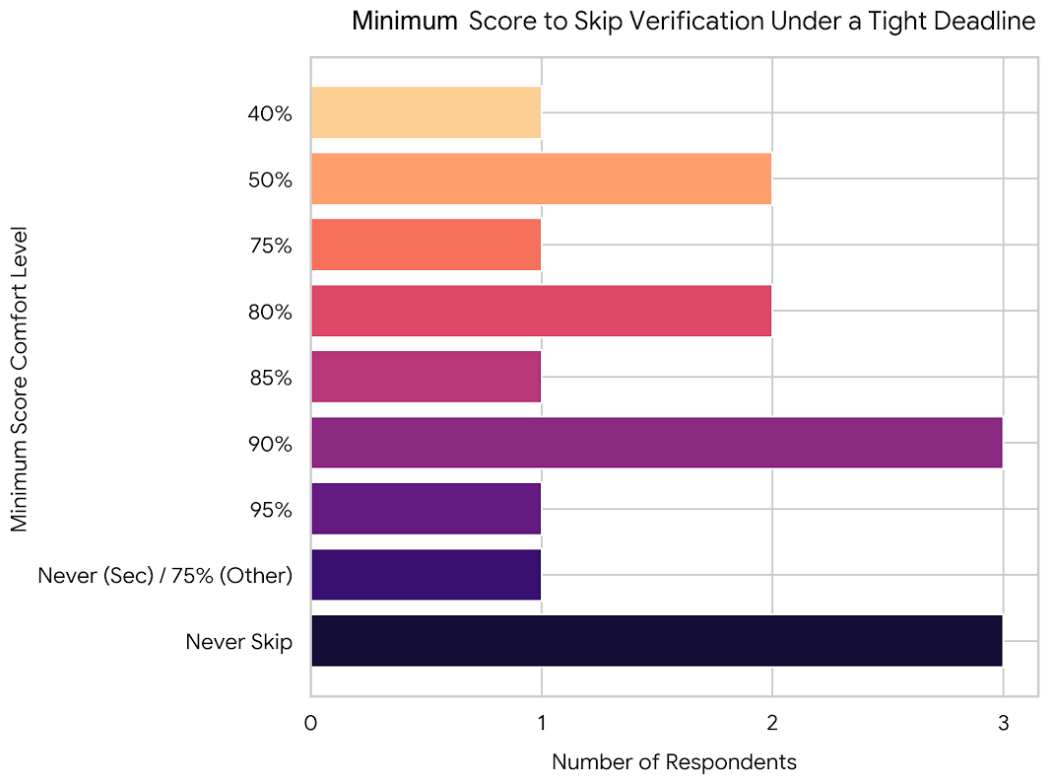


Figure 5.21: The minimum probability score participants are willing to skip when under time constraint.

this: “I have higher faith for maintainability... Security requires more of a manual check”. Ultimately, this spectrum of responses reinforces the finding that a model’s confidence is treated merely as a supplementary guiding metric, rather than an absolute source of truth.

An unexpected theme emerged through the participants’ active reliance on external Generative AI tools. When the provided Source Code and the Warning Message were insufficient to make a conclusive decision, several participants actively copied the code snippets into external LLMs like ChatGPT or Google Gemini to seek explanations. Participant 7 explicitly vocalized this workflow stating, “I just don’t know the Java, so then I ask ChatGPT, right?” and “Am I the first person to just completely copy-paste it in ChatGPT? [No]”. Similarly, Participant 11 repeatedly verified the warnings against AI outputs, stating, “I’m not so familiar with [Java], so I need to rely on [ChatGPT] a bit”. Whilst Participant 24 turned to Gemini, “Let me ask Gemini what it thinks about this”. In these instances, the provided probability score acted as a benchmark to compare against the external LLM’s assessment. Participant 11 highlighted this comparative dynamic, explaining that “the scores basically helped me to affirm if what ChatGPT was saying was correct or not in a way”. This behavior highlights a shift in modern developer workflows. Specifically, to validate the output of an internal predictive model, developers increasingly leverage external generative models to bridge their own knowledge gaps before making a final triage decision.

Finally, the study highlights that trust in a model cannot be demanded upfront. It must be empirically proven over time. When asked if the probability scores would reduce workload, Participant 15 emphasized that "these kind of tools need to build trust... You do the next one, it's right again, and it's right again and it's right again. [Then there] comes a moment where you start thinking this actually works". This indicates that developers require a sustained, iterative track record of the model aligning with their own manual conclusions before they will fully integrate its probability scores into their accepted workflow.

In summary, the thematic analysis of the user study reveals a dual narrative. While generated probability scores successfully optimize the triage workflow by acting as a powerful prioritization and affirmation tool (Theme 1), they do not independently command absolute developer trust (Theme 2). Trust remains highly conditional, dependent on source code, warning categories, and the gradual, iterative proof of the model's reliability over time. Ultimately, these qualitative findings provide the crucial human-centric context required to evaluate the true utility of the predictive models tested in the preceding experiments.

Chapter 6

Discussion

The primary objective of this research is to identify the optimal balance between performance and resource efficiency when generating actionability probability scores for Automated Static Analysis Tool (ASAT) warnings. By comparing classical Machine Learning (ML) architectures against Large Language Models (LLMs), this thesis seeks to provide a reliable triage method without inadvertently filtering out actionable warnings.

To achieve this, the research is guided by the following main research question:

Main Research Question

What are the performance and resource trade-offs between zero/one-shot LLMs and optimized classical ML models when generating actionability probability scores for ASAT warnings?

This is then further divided into two sub-question groups:

RQ1 Model Selection and Feature Engineering: this investigates the optimal classical models (RQ1.1), effective LLM representations (RQ1.2), and the influence of feature choice (RQ1.3).

RQ2 Comparative Performance and Robustness: this examines the models across standard metrics (RQ2.1) and probabilistic calibration (RQ2.2).

This chapter thus combines the empirical experiments and qualitative user study results to answer these questions and discuss their broader implications. The chapter is structured to systematically address the RQs as follows: First, Section 6.1 addresses the core of the main research question alongside RQ1.1, RQ1.2, and RQ2.1. It evaluates the fundamental predictive capabilities and computational cost trade-offs between the optimized classical ML models and the LLMs. By doing so, it establishes which candidates achieve the optimal trade-off between training efficiency and predictive performance across standard metrics like the F1-score and AUC.

Next, Section 6.2 expands beyond classification metrics to directly answer RQ2.2. It analyzes the probabilistic calibration and robustness of these architectures, specifically utilizing Brier scores. This is to establish whether the models are well-calibrated and can

be genuinely trusted to accurately signal their own uncertainty. Subsequently, Section 6.3 grounds these statistical findings in practical software engineering while addressing RQ1.3. By combining the empirical feature importance results with the qualitative user study, this section explores how the choice of features (such as metadata versus source code) influences the model's logic. Furthermore, it addresses how those resulting actionability scores impact developer workflows, cognitive bias, and trust dynamics in a real-world environment. Finally, Section 6.4 addresses the internal, construct, and external threats to validity that contextualize these findings, and Section 6.5.1 discusses how generatie AI was used in this research.

6.1 Performance and Efficiency Trade-offs

The experiments establish the Random Forest (RF) model as the optimal performance-efficient model when compared to LLMs. This section addresses the core predictive capabilities of these models to answer ***RQ1.1*** *(optimal classical ML model)*, ***RQ1.2*** *(most effective LLM)*, and ***RQ2.1*** *(comparison across standard metrics)*. In summary, the optimized RF model achieves the best trade-off between training efficiency and predictive performance (**RQ1.1**). Among the LLMs, Opus 4.6 utilizing one-shot prompting provides the most effective representations (**RQ1.2**). However, when comparing both paradigms across standard metrics, the classical RF significantly outperforms the LLMs in both classification power and computational efficiency (**RQ2.1**). To understand why this classical ensemble model outperforms models like Anthropic's Claude Opus 4.6, it is necessary to systematically evaluate their performance across three distinct operational metrics: the F1-score, the Area Under the Curve (AUC), and computational efficiency.

6.1.1 Evaluating the F1-Score

In the context of ASATs, raw accuracy serves as a highly misleading metric. Because datasets like NASCAR consist of roughly 84% to 86% non-actionable warnings, a model could achieve artificially high accuracy simply by classifying everything as non-actionable. To address this limitation, the F1-score is utilized because it calculates the harmonic mean of precision and recall. We care about this metric because it directly measures a model's ability to minimize non-actionable warnings (precision) without accidentally filtering out actionable warnings (recall).

The experimental results demonstrate that the optimized RF model achieves a superior test F1-score of 76.85%, boasting an effective balance with 80.07% precision and 73.88% recall (Table 5.6). In contrast, the zero-shot Opus 4.6 model reaches a peak F1-score of only 70.2% on a small 1000-instance sample (see Table 5.7). When this evaluation is scaled up to the full 50,000-warning test set, the LLM's performance degrades further to an F1-score of 67.0% in the zero-shot configuration (see Table 5.8) and 66.7% in the one-shot configuration Table 5.10). These findings reveal that the classical RF model fundamentally outperforms the LLMs in raw classification power.

Furthermore, comparing the RF to the classical Logistic Regression (LR) model reveals why non-linear decision boundaries are required. While the LR model achieves a stable F1-

score of 72.67% and a high recall of 80.73%, its precision is severely restricted at 66.07% (see Table 5.5). This performance gap indicates that the geometric boundaries separating actionable and non-actionable warnings are complex and non-linear. Because LR cannot effectively isolate these complex feature interactions, it fails to filter false positives accurately and thus produces an unacceptable volume of incorrect actionable warnings. This ultimately establishes the non-linear RF as the more robust choice with respect to performance metrics.

6.1.2 Precision-Recall Curves and Area Under the Curve (AUC)

While the F1-score provides a snapshot at a single decision threshold, analyzing the Precision-Recall (PR) curves and their respective Area Under the Curve (AUC) visualizes how reliable each model remains across all possible operating boundaries. This metric is needed, because enterprise environments frequently need to adjust their risk tolerance. Like demanding higher recall thresholds for security-critical applications. A high AUC and a stable PR curve would then ensure that the model's trade-off between precision and recall remains robust, regardless of where the specific threshold is set.

The experimental results indicate that the optimized RF model maintains a smooth, hyperbolic trajectory on its PR curve, culminating in an AUC of 0.87 (see Figure 5.4). This smooth contour demonstrates a well-conditioned, non-linear decision boundary capable of consistently prioritizing actionable warnings. Additionally, the model sustains a precision above 0.90 well into the recall spectrum and only begins to incrementally drop after recall extends past approximately 0.45 to 0.50. The LR model similarly achieves a generally stable curve with a matching downward trajectory, resulting in an AUC of 0.79 (see Figure 5.2). While the LR model features an initial transient dip in precision down to 0.80 near a recall of 0.0 (see Figure 5.2b), it immediately recovers to stay above 0.90 until a recall of approximately 0.25, after which it enters a roughly linear decline.

On the other hand, the LLMs demonstrate significantly weaker threshold stability. Opus 4.6 yields a substantially lower AUC of 0.65 on the preliminary sample (see Figure 5.12) and a further reduced AUC of 0.60 on the full test dataset during zero-shot experiments (see Figure 5.13). Further examining these PR curves reveal highly volatile and jagged shapes. This stands in stark contrast to the stable curves of the classical ML models. Across the various LLM configurations, the PR curves exhibit severe, immediate drops in precision at very low recall boundaries (Figure 5.11-5.16). They sometimes even plummet down to 0.66 or below 0.50 before erratically climbing back up. This pronounced instability at the extremes of the probability distribution shows that LLMs struggle to consistently rank their most confident predictions, establishing the classical RF model as a fundamentally more predictable and reliable prioritization mechanism for dynamic triage environments.

To further understand why a lightweight model like RF can so effectively outperform Opus 4.6, one must also look at the nature of the data itself. The structural characteristics of ASAT warnings explain why this happens. As established by the Latent Semantic Analysis (LSA) variance trials (Table 5.1), the vocabulary in the Warning Message feature is standardized across different Java projects. Specifically, the denoised_heavy configuration reaches a total variance of 95.93% for Warning Message, with an elbow point capturing 78.7% of the total information using just 100 components (Table 5.1). Because

of this extreme information density, classical feature engineering effectively solves the semantic representation problem without requiring massive transformer-based self-attention layers. Consequently, utilizing computationally expensive LLMs to parse highly structured, expert-curated warning patterns amounts to an unnecessary over-allocation of resources. Traditional ensemble architectures (like RF) successfully map these dense semantic matrices to the actionability of a warning.

6.1.3 Computational Efficiency and Economics

For a prioritization pipeline to be viable in enterprise platforms like Sigrid, which processes hundreds of billions of lines of code, it must scale efficiently. Minimal operational latency and low financial overhead are absolute prerequisites for this real-world integration. So, deploying high-tier LLMs like Claude Opus 4.6 incurs massive compounding API costs. The models are officially priced at $5 per million input tokens and $25 per million output tokens [5]. Since evaluating a single warning requires feeding the model source code, metadata, warning messages, and potential examples, analyzing millions of records results in severe network latency and thousands of dollars in recurring cloud inference fees.

Conversely, the classical RF architecture is exceptionally fast to train and requires minimal computational overhead. During the experimental phase, training the RF model on the full dataset (see Section 4.1.1) required a maximum of only 30 minutes, here 75% was utilized for training and 25% was held out for testing. Furthermore, executing the comprehensive hyperparameter tuning process took at most two hours.

Crucially, this time spent only applies to the initial training phase. Using the fully trained RF model to perform inference on new warnings is nearly instantaneous [44]. Additionally, the RF model requires minimal memory. The entire pretrained model can be serialized and saved as a single .joblib file. Since the hyperparameter tuning phase explicitly constrains the model's complexity, this resulting .joblib file is highly compact. It typically ranges in size from just a few megabytes to a few dozen megabytes [66]. This negligible memory footprint allows the model to be loaded and executed seamlessly on standard infrastructure without the API latency delays associated with LLMs. Utilizing computationally expensive LLMs to generate probability scores is ultimately thus an unnecessary over-allocation of resources.

6.2 Model Calibration, Uncertainty, and Robustness

While classification metrics such as the F1-score and AUC successfully evaluate a model's discrete predictive boundaries, the ultimate utility of the models depends heavily on its mathematical robustness and its ability to accurately signal its own uncertainty. This section directly answers ***RQ2.2*** *(How well-calibrated are the models' probability estimates, and can they be trusted to signal their own uncertainty?).* In summary, optimized classical ensemble models like RF provide highly reliable uncertainty signals that can be genuinely trusted. Whereas all evaluated LLMs consistently exhibit structural miscalibration and mathematical overconfidence. To fully understand this reliability across the classical ML models and the

LLMs, it is necessary to systematically evaluate them across two distinct robustness metrics: the Generalization Gap and the Brier score.

6.2.1 Evaluating the Generalization Gap

In predictive modeling, the generalization gap measures the variance in performance between the data a model was trained on and the unseen test data it is evaluated against. We care about this metric because a model that simply memorizes the noise within its training corpus will fail unpredictably when deployed in a live enterprise environment. This phenomenon is also referred to as overfitting. Thus, a reliable model must demonstrate a tight alignment between its training and testing metrics to prove it has learned genuine, generalizable patterns rather than dataset-specific artifacts.

To achieve this structural stability, both classical ML models underwent a hyperparameter tuning phase to balance complexity and generalization. The LR model demonstrated an absolute resistance to overfitting from the very beginning. Unlike the RF model, the LR model's training and testing metrics consistently remained tightly bounded within fractions of a percent of one another during the baseline phase (see Table 5.4). Nevertheless, tuning the regularization strength (C) alongside L1 (Lasso) and L2 (Ridge) penalties ensured the linear model did not become overly sensitive to noisy, high-dimensional features. This optimization successfully achieved a higher F1-score by roughly 2% (see Table 5.5) compared to the baseline without introducing any generalization gap.

Conversely, the necessity of hyperparameter tuning is most apparent when evaluating the RF model. During the baseline phase, the default RF model exhibited a severe generalization gap. While it initially achieved a high test F1-score near 85% (see Table 5.4), this performance was the direct result of massive overfitting. The baseline model's training F1-scores consistently exceeded 99.50%, creating an approximate 14% generalization gap when applied to the test set. However, by intentionally constraining the model's complexity during the tuning phase, specifically by introducing depth limits and Cost Complexity Pruning, the hyperparameter search successfully forced the ensemble to generalize rather than memorize training noise. Consequently, while the optimized RF model yields a lower absolute test F1-score of 76.85% compared to the baseline, its training F1-score of 77.51% now aligns almost perfectly with its test performance (see Table 5.6). This closed generalization gap confirms that the optimized model possesses the structural robustness necessary for real-world deployment without the risk of an overfitted decision boundary.

6.2.2 Evaluating Probabilistic Calibration and Uncertainty

To determine how reliably these probability scores can guide the triaging of ASAT warnings, the evaluation next transitions to the Brier score. This measures the mean squared difference between a model's assigned probability and the actual outcome [18]. Assessing calibration is essential because the ultimate utility of these models relies on their ability to provide accurate probability scores. If a model assigns high certainty to incorrect predictions, it introduces systematic miscalibration that can misdirect software engineers. Because a Brier score of 0.25 represents the baseline for uninformative, random guessing, any score

exceeding this threshold indicates that a model is structurally overconfident. This renders its assigned probabilities mathematically worse than chance [18]. Furthermore, the lower this score is, the better the calibration.

The empirical results reveal a stark contrast in probabilistic reliability between classical ML models and LLMs. The optimized classical RF model demonstrates the highest degree of calibration, achieving a test Brier score of 0.1549 (see Table 5.6). Furthermore, it maintains this probabilistic dependability even when subjected to highly skewed, imbalanced data distributions, consistently yielding Brier scores between 0.066 and 0.165 (see Table 5.3). The LR model similarly remains highly stable, though it operates with a slightly higher probability error floor of 0.1917 (see Table 5.5).

In sharp contrast, the LLMs struggle significantly with uncertainty and calibration across all model scales. During the zero-shot experiments, the lightweight Haiku 4.5 and mid-tier Sonnet 4.6 yielded high Brier scores of 0.332 and 0.328, respectively (see Table 5.7). The most powerful model, Opus 4.6, generated an overconfident score of 0.348 on the preliminary sample and 0.324 during large-scale inference on 50,000 warnings (see Table 5.8). While providing a single contextual demonstration (one-shot prompting) improved Opus 4.6's calibration to 0.269 on a small sample (see Table 5.9), scaling this up to the full test set caused the model to regress to a Brier score of 0.313 (see Table 5.10). This regression probably occurs because a single contextual example is insufficient to represent the vast structural diversity present across 50,000 warnings. Ultimately, because all evaluated LLMs consistently produced Brier scores above the 0.25 threshold, they exhibit severe structural miscalibration. This makes their probability estimates mathematically unreliable.

6.3 The Link to Software Engineering

The empirical findings from both the feature engineering experiments and the qualitative user study highlight that successfully integrating predictive models into ASAT workflows requires balancing efficiency with human cognitive needs. This section grounds the statistical results in practical software engineering to directly answer ***RQ1.3*** *(How does the choice of features influence model selection?)*. In summary, there is a critical operational disconnect regarding feature choice. While classical ML models rely heavily on historical metadata (like warning age), human developers actively disregard this. Instead, they universally demand source code context and severity indicators to validate a model's output. To fully understand this practical utility, the findings are systematically evaluated across four operational dimensions: Workflow Prioritization, Trust Calibration, Conditional Reliance, and System Maintenance.

6.3.1 Workflow Prioritization and Automation Bias

This section evaluates how probability scores alter the methodology software engineers and consultants use to address ASAT warnings. Evaluating this dimension is critical, as the primary software engineering value of this automated pipeline lies in its ability to reduce the time spent on non-actionable items. The qualitative user study (see Section 5.2) shows that presenting participants with probability scores provides a highly effective, structured

starting point for navigating ASAT warnings. Rather than serving as an absolute oracle that dictates the final verdict, participants utilize these scores primarily as a prioritization and sorting mechanism. Survey results from the study support this cautiously optimistic view as well. Majority of participants rated the scores as helpful for organizing their tasks, noting that the scores help them determine "where to start." Even those who rated the scores as unhelpful for making the final classification decision conceded their structural value for ranking high-probability warnings to be tackled first.

Furthermore, the study highlights that probability scores often serve a secondary function as an affirmation tool or a "second opinion". This proved particularly valuable for participants navigating unfamiliar codebases or relying on external AI agents, with some users treating the model as a teammate that catches overlooked issues. However, this influence is not uniform across all experience levels. The results demonstrate that experienced developers, specifically those with four or more years of Java experience, frequently ignored the probability scores. Driven by a strong sense of professional ownership and technical intuition, these participants preferred to rely entirely on their own manual judgment. This indicates that while probability scores successfully optimize the initial triage order, they face significant resistance in influencing the final verdict.

While the scores offer clear organizational benefits, their introduction also carries severe risks of automation bias. Blind reliance on the probability scores could lead to oversights, making it imperative to understand how participants react to perceived model certainty. The study's trap warnings vividly illustrated this danger. When presented with an exceptionally high probability score, such as a 98% probability assigned to a non-actionable warning, the score successfully overrode the analytical validation process for a majority of the participants. This caused nine out of fifteen developers to incorrectly trust the score. The verbal remarks during the experiment revealed that users frequently deferred to the 98% score immediately, assuming the tool's confidence guaranteed a real defect. Conversely, when a moderate score of 75% was presented on a similar trap warning, it triggered a much lower rate of blind reliance. In this case, only five participants were fooled, while the remaining ten correctly dismissed the issue. This sharp contrast shows that high probability scores could directly impair scrutiny and encourage blind deference, whereas moderate scores successfully preserve the human's critical evaluation process.

6.3.2 Trust Calibration and the Feature Disconnect

Trust calibration evaluates the specific structural criteria and contextual information a software engineer requires to genuinely trust a probability score. If they do not trust the model's output, the probability scores become obsolete, leaving manual triage completely unchanged. A critical finding of this research is thus the pronounced operational disconnect between the statistical drivers of the ML models and the specific criteria developers require to build trust.

The feature importance analysis (see Section 4.1.4) reveals that the age of a warning is the single most powerful mathematical predictor of actionability for the RF model. Yet, the empirical results from the user study demonstrate that developers rate a warning's age as the absolute least useful piece of information for manual validation (see Figure 5.19).

Specifically, six participants universally dismissed it by rating it as completely "Not Important", while five others rated it as merely "Slightly Important". Participants argued that the age of an alert does not inherently correlate with its validity. As Participant 14 pointed out, "there's things people miss all the time... I don't think the warning age is important," a sentiment explicitly echoed by Participant 12 who placed warning age "way farther down" their list of priorities.

Instead, the qualitative data reveals that developers overwhelmingly rely on the Source Code and severity indicators (like CWE-ID and CVSS severity) to understand the structural reasoning behind an issue. The code snippet emerged as the absolute most critical feature, with 14 out of 15 participants classifying it as "Very Important" and the remaining participant rating it as "Important". Participant 14 perfectly summarized this reliance, explaining that "most of my inspection was more based on the code and less on the sort of the metadata... without that code text I wouldn't know where the finding comes from". Furthermore, features denoting the direct impact of the warning, such as CVSS severity, were also heavily prioritized to validate the tool's output, with eight participants rating it as "Very Important" and four as "Important".

This highlights a fundamental software engineering challenge: while historical metadata, such as how long a warning has persisted, is highly effective for predictive modeling, it provides zero explanatory power to human reviewers. Therefore, any practical implementation must tightly couple the generated probability scores with code snippets and severity indicators. Without explicitly providing the structural context that developers rely on, the model risks severe trust degradation.

6.3.3 Conditional Reliance and External AI Integration

We then move on to evaluate the conditional reliance on these probability scores, defined as the behavioral threshold at which software engineers are willing to surrender their autonomy to the probability scores. Evaluating this aspect is critical, because understanding when a participant will actually skip a manual review determines the true time-saving potential of the model.

The empirical user study demonstrates that participant trust is highly conditional and sensitive to the stakes of the specific warning. When asked to indicate a minimum probability score they would feel comfortable skipping under tight deadlines (see Figure 5.21), the responses revealed a clear spectrum rather than a uniform consensus. The majority of developers (nine participants) demanded near-absolute certainty, setting strict probability thresholds between 75% and 95% before they would comfortably skip a manual code review. As Participant 4 rationalized, "eighty percent is a pretty good thing if you trust your model". However, a distinctly more skeptical faction emerged alongside this majority. Three participants firmly stated they "would never skip manual verification regardless of the score," refusing to surrender their autonomy entirely.

Crucially, the study reveals that even when a numerical trust threshold is established, it instantly collapses when dealing with high-risk categories such as Security. In these critical instances, the majority of the participants (10 out of 15) universally indicated that they refuse to bypass manual verification regardless of how confident the probability score

is. This conditional reliance is perfectly captured by Participant 14, who noted they would skip a manual check at "75% for other [categories]" but "Never for security". Similarly, Participant 2 emphasized that while they have "higher faith for maintainability... Security requires more of a manual check". This indicates that a probability score is treated merely as a supplementary guiding metric rather than an absolute source of truth when system integrity is on the line.

This inherent skepticism has also driven a shift in modern developer workflows. The user study revealed that when the provided code context or their domain knowledge is insufficient, participants increasingly copy-paste warnings into external Generative AI tools like ChatGPT or Google Gemini to bridge this gap. Participant 7 explicitly vocalized this workflow when navigating unfamiliar code, asking, "Am I the first person to just completely copy-paste it in ChatGPT? [No]". In these scenarios, the internal probability score is actively repurposed as a comparative benchmark to validate or affirm the external LLM's assessment. As Participant 11 highlighted, "the scores basically helped me to affirm if what ChatGPT was saying was correct or not in a way".

Ultimately, participants with more professional experience (4+ years) strongly emphasize that baseline trust in a model cannot simply be demanded upfront. Instead, they argue that this trust must be earned iteratively. It requires a sustained, transparent, and empirically accurate track record that consistently aligns with their own manual conclusions over time. Participant 15 perfectly summarized this necessary progression, stating that "these kind of tools need to build trust... You do the next one, it's right again, and it's right again and it's right again. [Then there] comes a moment where you start thinking this actually works."

6.3.4 System Maintenance and Feature Independence

While the software engineer's willingness to trust and act on these predictions hinges heavily on transparency and tracking over time, the long-term viability of the tool relies equally on its structural stability behind the scenes. From a systems engineering perspective, this section evaluates the degree to which individual data inputs overlap or correlate with one another. Analyzing this structural property is critical for deploying a model in Sigrid, as it ensures the overall pipeline remains robust, maintainable, and stable against future codebase shifts or architectural updates.

The empirical results from the interpretability analysis yield a significant advantage. Namely, the extracted feature space is fundamentally orthogonal and non-redundant. For the optimized RF model, the correlation matrix of the top 100 SHAP features (see Figure 5.6) reveals a strict independence between the most influential variables. Specifically, the age of a warning, which serves as the model's single most powerful statistical predictor, exhibits near-zero correlation with the dense LSA components derived from the text. Furthermore, the global hierarchical cluster map (see Figure 5.5) confirms that the LSA components for Source Code and the LSA components for Warning Message actively separate into distinct sub-clusters, while categorical metadata (such as design categories or best practices) branches off entirely on its own. This structural separation validates the preprocessing design. It shows that raw code, warning descriptions, and historical metadata each

capture entirely distinct, non-overlapping facets of software quality rather than relying on a single monolithic block of information.

This systemic independence is equally pronounced and verified within the LR. The complete feature correlation clustermap for this model (see Figure 5.9) demonstrates an overwhelmingly neutral baseline, with pairwise correlations consistently hovering near zero ($r \approx 0$). Because the hierarchical dendrograms mathematically merge at extreme heights, it confirms that the features lack strong mutual correlation and are largely orthogonal.

For future development cycles, this structural independence provides crucial operational guarantees for system maintenance. It ensures that individual features[1] can be modified or isolated without the risk of inducing severe multicollinearity errors or causing unpredictable shifts in the model's underlying decision-making logic. Consequently, the model does not need aggressive variance inflation mitigations or complex deduplication protocols. Ultimately, this structural stability ensures that the deployed model will consistently provide the stable, predictable, and interpretable probability scores required to build and maintain trust.

6.4 Threats to Validity

To properly contextualize the empirical and qualitative findings of this research, it is essential to acknowledge the methodological limitations and boundaries of this research. This section thus provides transparency regarding the experimental design and ensures that the conclusions drawn regarding model performance, probability calibration, and software engineer behavior are interpreted within their appropriate scope. Therefore, we systematically address the internal, construct, and external threats that may impact the reliability and generalizability of the results.

6.4.1 Internal Validity

Internal validity concerns the rigor of the experimental design and the degree to which confounding variables or data processing choices may have influenced the outcomes [47]. The first threat lies in how the labels for actionable and non-actionable warnings were originally generated by Kószó et al. [41]. Because the NASCAR dataset was labeled at scale, it relies on a heuristic differential analysis of consecutive version control commits. It assumes that if a warning's surrounding code is modified and the warning disappears, it was actionable. Conversely, if it persists, it is non-actionable [41]. This reliance on implicit user feedback introduces inherent noise into the ground truth. For instance, a warning might disappear simply because a file or method was deleted for unrelated architectural reasons. This would result in falsely labeling it as an actionable fix. Similarly, a severe security vulnerability might persist simply because the developers lacked the time or budget to address it. So it falsely labels a genuinely actionable threat as non-actionable noise.

[1] Features such as updating a specific LSA dimension, introducing a new ASAT tool category, or removing a deprecated metric.

Another internal threat in this study arises from the data partitioning strategy. The dataset splits were generated using purely random sampling (42). While this ensures a mathematically balanced and diverse training environment, random sampling inherently fragments project-level context. Because warnings from a single repository may be split arbitrarily across the training and testing sets, the models are prevented from learning cohesive architectural styles or accounting for the locality of warnings that frequently cluster within specific modules.

A third internal threat stems from the deduplication methodology employed by the underlying NASCAR dataset. To manage the massive volume of non-actionable warnings, Kószó et al. [41] drops earlier duplicates of unaddressed warnings. This resulted in retaining only their final occurrence, which introduces a severe artificial timeline bias. It creates a massive concentration of non-actionable warnings late in the mining window. So while this research successfully neutralized direct temporal data leakage by engineering a relative age metric rather than feeding the model absolute timestamps, the underlying skewed distribution could still subtly influence the data landscape [41].

Finally, the interpretability analysis was bounded by computational limitations. To ensure a fair methodological comparison against the computationally expensive SHAP TreeExplainer used for the RF, the localized feature correlation sample for the LR model was strictly capped at 200 instances. This small sample size caused sparse categorical features to exhibit zero variance, limiting the ability to calculate meaningful pairwise correlations for certain variables within that specific evaluation window.

6.4.2 Construct Validity

Construct validity evaluates whether the theoretical concepts being studied are accurately measured by the experimental operationalizations [10]. One prominent threat is the manner in which abstract cognitive concepts, like trust and automation bias, are evaluated in the user study. While the study utilized trap warnings and a Think Aloud protocol to observe real-time decision-making, the reported trust thresholds (e.g., participants demanding 90% certainty to skip a manual review) are inherently self-reported. Participants in an observed, simulated experimental environment may act significantly more cautious and critical than they would in an unobserved, high-pressure enterprise setting. This could potentially skew the true behavioral threshold at which automation bias occurs.

Furthermore, another limitation exists within the design of the user study's trap warnings. The triage experiment solely investigated automation bias by presenting participants with non-actionable warnings assigned artificially high probability scores (such as 98% and 75%). The study failed to test the inverse scenario: an actionable warning accompanied by an artificially low probability score. This omission presents a significant behavioral threat to the comprehensiveness of the findings. Without testing this inverse scenario, it remains entirely unknown whether developers would blindly trust a low probability score and subsequently dismiss an actionable warning without manual inspection. While incorrectly trusting a high score on a non-actionable warning merely results in wasted triage time, incorrectly trusting a low score on an actionable defect could lead to catastrophic security flaws being deployed into production. Unfortunately, this oversight in the experimental de-

sign was discovered too late in the research process to rectify and include in the participant evaluations.

6.4.3 External Validity

External validity assesses the extent to which the study's findings can be generalized to broader contexts, tools, and populations [51]. The primary external limitation is the strict technical scope of the dataset. The predictive models were trained and evaluated exclusively on the Java programming language, utilizing warnings generated specifically by PMD and SpotBugs. Consequently, the established performance trade-offs, optimal LSA dimensionality, and feature importance rankings cannot be automatically generalized to other programming languages, or alternative ASATs.

Similarly, the LLM evaluation was constrained to one company. Due to institutional access at the collaborating company, the LLM experiments exclusively utilized Anthropic's Claude 4.x models (Haiku, Sonnet, and Opus). It is entirely possible that other state-of-the-art foundational models, such as OpenAI's GPT-4 or open-source alternatives like Meta's Llama, might exhibit different baseline probability calibration or zero-shot reasoning capabilities.

Finally, the generalizability of the software engineering findings is limited by the constraints of the qualitative user study. The participant pool consists of 15 professional developers and consultants sourced exclusively from a single organization (SIG). While prioritizing a qualitative methodology provides deep, nuanced insights into the cognitive processes behind triaging warnings, this specific organizational context and small sample size lack the statistical power necessary to draw broadly generalized conclusions about human-AI interaction across the wider global software engineering industry.

6.5 Gen AI & Ethical Considerations

6.5.1 Use of Generative AI in the Context of this Research

Generative AI tools were utilized throughout this research. It was used for academic drafting and structuring, for brainstorming research ideas, testing counterarguments, and improving readability, grammar, and text organization. It also provided formatting assistance (such as LaTeX conversion for citations) and code troubleshooting support for boilerplate generation and debugging. In the empirical phase of the research, LLMs were also directly integrated into the study's framework. Multiple models were prompted to generate the automated probability scores evaluated.

6.5.2 Ethical Considerations for this Research

The integration of predictive ML into professional software engineering workflows, alongside the execution of a human-centric user study, introduces several critical ethical considerations that must be addressed.

First, the qualitative user study was conducted in strict adherence to ethical guidelines regarding human subjects and data privacy. Because the participant pool consisted of a specialized cohort from a single organization (SIG), there was an inherent risk of indirect re-identification if professional attributes, such as specific job roles and exact years of experience, were combined with triage performance metrics. To mitigate this ethical risk, all collected research data was strictly pseudonymized and stored on secure university servers, and all direct identifiers were removed. Additionally, demographic data was generalized into broader categories to ensure anonymity in the aggregate reporting. Furthermore, participation was entirely voluntary. Participants were fully informed of the study's objectives, retained the right to withdraw at any time without providing a reason, and were assured that all personal data would be explicitly destroyed three months after the completion of the study.

Second, the deployment of actionability probability scores in ASATs carries significant ethical implications regarding AI safety and automation bias. As demonstrated during the triage experiment's trap warnings, assigning artificially high probability scores (e.g., 98%) to non-actionable warnings overrode the critical validation processes of several developers, leading to blind deference. Ethically, deploying predictive models as black boxes in high-stakes enterprise environments introduces the severe risk of creating a false sense of security. If a system's probability estimates are structurally miscalibrated or lack transparency, users might blindly trust a low probability score and inadvertently allow critical security vulnerabilities to be deployed into production. Therefore, it is an ethical imperative that future integrations do not present probability scores as absolute truths, but rather tightly couple them with transparent, structural evidence to maintain human-in-the-loop oversight.

Finally, the core conclusion of this thesis carries an ethical dimension regarding computational sustainability. We conclude that deploying LLMs for warning triage represents an unnecessary over-allocation of resources compared to optimized classical ML models. So, in an era where the environmental and economic costs of training and querying foundational AI models are under intense scrutiny, defaulting to computationally expensive LLMs when a lightweight, highly efficient RF model achieves superior predictive robustness aligns better with the ethical principle of data parsimony and sustainable engineering practices.

Chapter 7

Conclusions and Future Work

This final chapter combines the empirical and qualitative findings of this thesis. Section 7.1 presents the overarching conclusions drawn from this research, systematically answering the main research question by comparing the capabilities of classical ML and LLM. Following this, Section 7.2 outlines the primary theoretical and practical contributions this research makes, spanning from model evaluation and structural system maintenance to behavioral insights into human trust. Finally, Section 7.3 explores potential avenues for future work. It discusses necessary pipeline refinements, dataset expansions, and the practical steps required to fully integrate these predictive models into the Sigrid environment.

7.1 Conclusions

To reiterate, this research seeks to compare classical ML with LLMs in the context of generating actionability probability scores for ASAT warnings. To achieve this, the study systematically evaluates the capabilities of classical ML versus LLMs by addressing two primary sub-question groups and one main research question.

7.1.1 Model Selection and Feature Engineering (RQ1)

To answer RQ1.1 regarding the optimal classical ML model, the study establishes that the optimized RF achieves the best trade-off between predictive performance and training efficiency. While LR is highly resistant to overfitting, its linear assumptions fail to capture the non-linear boundaries of the dataset. This ultimately results in high false positive rates.

For RQ1.2, which investigated the most effective LLMs, Claude Opus 4.6 demonstrates the strongest capability for actionability prediction. Utilizing a single contextual example (one-shot prompting), Opus 4.6 improves its classification balance and significantly reduces its forecast error (the Brier score) compared to both its zero-shot baseline and the smaller Haiku and Sonnet models.

Finally, addressing RQ1.3 on the influence of features, the analysis reveals a distinct disconnect between model predictors and human validation. The classical RF model relies heavily on the age of a warning as its most powerful predictor. However, the user study confirms that users rate age as the least useful piece of information, relying instead on

raw Source Code Context and severity indicators (CVSS or CWE-ID) to establish trust and validate predictions. Furthermore, the user study indicates that participants require the model to have a sustained and accurate track record to establish trust. Finally, it shows that automation bias can be a risk, especially for high probability scores (such as 98%). Here manual validation can be overridden by the score, leading to blind deference to the model.

7.1.2 Comparative Performance and Robustness (RQ2)

When comparing the models across standard metrics (RQ2.1), the optimized classical RF significantly outperforms the LLMs. The RF achieves a test F1-score of 76.85% and an AUC of 0.87, whereas the best performing Opus 4.6 configuration peaks at an F1-score of 69.4% and an AUC of 0.65. This best performing Opus 4.6 was run with a smaller sample size of 1000 warnings. This same model with the full test set (50,000 warnings) only achieved an F1-score of 66.7% and an AUC of 0.60.

Furthermore, in addressing RQ2.2 regarding model calibration, the Brier score evaluations confirm that the RF model produces highly reliable and mathematically dependable probability estimates. It achieves a test score of 0.1549, which is well below the 0.25 random-guessing baseline. Conversely, all tested LLMs struggle significantly with miscalibration. Even with one-shot contextual improvements, the best LLM Brier score (0.269 for Opus 4.6) remains mathematically worse than chance. This indicates structural overconfidence in their absolute probability predictions.

7.1.3 Main Research Question

Ultimately, addressing the Main Research Question regarding the performance and resource trade-offs, this study concludes that an optimized RF model provides a substantially more effective and viable framework for generating actionability probability scores for ASAT warnings compared to zero/one-shot LLMs. Because the vocabulary and semantic structure of ASAT warnings are inherently dense and standardized, classical feature engineering combined with a RF model effectively capture the necessary semantic patterns without requiring massive transformer networks. While LLMs like Opus 4.6 offer deep contextual reasoning, deploying them at scale introduces severe network latency and prohibitive, compounding API costs. Therefore, the optimized RF model delivers superior performance metrics (such as F1 and AUC), well-calibrated probability scores (Brier scores well below the random guessing 0.25 baseline), and high computational efficiency with minimal ongoing operational overhead.

7.2 Contributions

The primary focus of this research is to comprehensively compare classical ML models with LLMs in the context of generating actionability probability scores for ASAT warnings. By evaluating the predictive performance and resource trade-offs of these architectures, and supplementing this with a qualitative user study to investigate what these scores actually

mean for the users using them, this thesis provides several key contributions that are listed in the following sections.

7.2.1 Large-Scale Comparison of Classical ML vs. LLMs

The core contribution of this research is a rigorous, large-scale empirical comparison between optimized classical ML models (specifically RF and LR) and state-of-the-art LLMs (the Claude 4.x family) using the NASCAR dataset. The results demonstrate that because the vocabulary and semantic structure of ASAT warning messages are intrinsically dense and standardized, deploying computationally expensive LLMs is an unnecessary over-allocation of resources. The research shows that classical feature engineering, combined with an optimized RF, yields superior predictive performance (achieving a peak F1-score of 76.85% and an AUC of 0.87) while operating with near-instantaneous inference times and negligible financial overhead compared to zero- and one-shot LLM experiments.

7.2.2 Evaluation of Probabilistic Calibration

This thesis also contributes a critical evaluation of probabilistic reliability. It moves beyond standard discrete classification metrics (such as accuracy or F1-scores), and evaluates the probabilistic calibration of all models. By incorporating Brier score evaluations, the research reveals a severe limitation in LLMs. Notably, regardless of model scale or the use of one-shot contextual prompting, LLMs consistently exhibit structural miscalibration and overconfidence. Conversely, the study shows that an optimized RF model provides highly reliable uncertainty signals that can be genuinely trusted to generate actionability probability scores.

7.2.3 Structural Validation for System Maintenance

From a systems engineering perspective, this thesis contributes a validated, maintainable preprocessing architecture. Through comprehensive feature correlation and hierarchical clustering analysis, the research shows that the engineered LSA semantic embeddings and the derived historical metadata function as orthogonal and non-redundant inputs. This structural independence guarantees that individual features can be updated or isolated in future development cycles without the risk of inducing severe multicollinearity errors. This ensures long-term system stability and predictable decision-making logic.

7.2.4 Behavioral Insights into Automation Bias and Conditional Trust

As an added component to the model comparison, the qualitative user study provides deep behavioral insights into what these generated probability scores actually mean for the practitioners using them. The research empirically demonstrates the precise dangers of automation bias, proving that exceptionally high probability scores (such as a 98% probability score) can successfully override a developer's analytical validation process and encourage blind deference. Furthermore, the study contributes a nuanced understanding of conditional reliance, revealing that developers demand near-absolute certainty (strict thresholds

between 75% and 95%) before bypassing manual reviews. Additionally, this threshold instantly collapses to zero when dealing with high risk categories like Security.

7.2.5 Statistical Predictors vs. Human Validation Criteria

Finally, by comparing the feature importance mapping with the supplementary qualitative user study, this research identifies a fundamental operational disconnect between the statistical drivers of predictive models and the specific criteria humans require to build trust. The findings highlight that while historical metadata, specifically the age of a warning, serves as the single most powerful predictor for the RF model, participants of the user study actively disregard it during manual validation. Instead, they universally demand source code context and severity indicators (such as CVSS or CWE-ID) to validate a model's output. This showcases a critical design constraint for how probability scores must be presented in future ASAT platforms.

7.3 Future work

While this research establishes a foundation for prioritizing Static Code Analysis (SCA) warnings, several avenues remain for future exploration and pipeline refinement. First, the generalizability of the optimized models should be validated by expanding the research scope. Future work should incorporate additional Java datasets generated by a wider variety of static analysis tools beyond PMD and SpotBugs, and subsequently extend the methodology to entirely different programming languages. Furthermore, subsequent studies should test a broader diversity of state-of-the-art LLMs from various providers to determine if alternative architectures can resolve the probability miscalibration and cost-efficiency bottlenecks observed in this study.

Methodologically, future data processing must address the context fragmentation caused by purely random sampling. By cutting datasets along project boundaries rather than at the individual warning level, models can be trained to better capture localized architectural styles and project-specific warning clusters. Additionally, the classification pipeline itself should be expanded. While this research utilizes binary classification (actionable versus non-actionable warnings), enterprise tools like Sigrid utilize a wider spectrum of triage states. Future iterations of this pipeline should transition to multi-class classification, explicitly separating non-actionable warnings into: genuine false positives, non-actionable but technically valid code smells, and unreviewed states. This expansion will align the model's outputs more directly with the real-world enterprise workflow of the collaborating company.

To further optimize precision, future research should explore replacing the general classifier with category-specific models. Given the distinct semantic structures of different warning types, training dedicated ML pipelines for isolated categories, such as Security, Performance, Multi-threading, and Code Quality, could yield higher information density and more accurate probability calibration than a generalized model. Furthermore, computational resources should be scaled to allow for interpretability analysis on larger validation samples, moving beyond the 200-instance cap to better map the correlation of sparse categorical features.

Finally, the ultimate objective for this pipeline is its full integration into the current Sigrid platform at SIG. Deploying the optimized RF model into this live production environment will allow Sigrid to actively prioritize warnings in real-time workflows. This industrial integration, coupled with an expansion of the user study to include more developers and consultants in- and potentially outside of SIG (clients that use the Sigrid platform), will enable the continuous collection of feedback. Specifically, the integrated interface should feature a mechanism allowing users to explicitly flag when the model's actionability prediction is incorrect. These manual developer corrections will be aggregated into a gold-standard dataset[1]. By retraining the model on this gold-standard dataset during subsequent training sessions, the pipeline can actively learn from its edge cases and mistakes. This closed-loop approach allows the system to iteratively refine its performance and actively build developer trust at scale.

[1] A gold-standard dataset is a rigorously curated, high-quality repository of human-verified labels that serve as the absolute ground truth for the model [77].

Bibliography

[1] Vaibhav Agrawal and Kiarash Ahi. Llm-driven sast-genius: A hybrid static analysis framework for comprehensive and actionable security, 2025. URL https://arxiv.org/abs/2509.15433.

[2] Enas A. Alikhashashneh, Rajeev R. Raje, and James H. Hill. Using machine learning techniques to classify and predict static code analysis tool warnings. In *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–8, 2018. doi: 10.1109/AICCSA.2018.8612819.

[3] Anthropic. Introducing Claude 4. https://www.anthropic.com/news/claude-4?c=6709, 2025. Accessed: 30-04-2026.

[4] Anthropic. 2026 agentic coding trends report: How coding agents are reshaping software development. Research report, Anthropic Research, 2026. URL https://resources.anthropic.com/hubfs/2026%20Agentic%20Coding%20Trends%20Report.pdf. Accessed: 29-05-2026.

[5] Anthropic. Pricing - claude api docs. https://claude.com/pricing, 2026. Accessed: 20-05-2026.

[6] Anthropic. Claude Haiku 4.5. https://www.anthropic.com/claude/haiku, 2026. Accessed: 30-04-2026.

[7] Anthropic. Introducing Claude Opus 4.6. https://www.anthropic.com/news/claude-opus-4-6, 2026. Accessed: 30-04-2026.

[8] Anthropic. Claude Sonnet 4.6. https://www.anthropic.com/claude/sonnet, 2026. Accessed: 30-04-2026.

[9] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008. doi: 10.1109/MS.2008.130.

[10] J.A. Barten, M.F. Pisters, P.A. Huisman, T. Takken, and C. Veenhof. Measurement properties of patient-specific instruments measuring physical function. *Journal of Clinical Epidemiology*, 65(6):590–601, 2012. ISSN 0895-4356. doi: https://doi.org/10.1016/j.jclinepi.2011.12.005. URL https://www.sciencedirect.com/science/article/pii/S0895435611003842.

[11] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 470–481, United States, 2016. IEEE. doi: 10.1109/SANER.2016.105.

[12] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8): 1798–1828, August 2013. ISSN 0162-8828. doi: 10.1109/TPAMI.2013.50. URL https://doi.org/10.1109/TPAMI.2013.50.

[13] Vikram Bhutani, Farshad Ghassemi Toosi, and Jim Buckley. Analysing the analysers: An investigation of source code analysis tools. *Applied Computer Systems*, 29(1):98–111, 2024. doi: 10.2478/acss-2024-0013. URL https://doi.org/10.2478/acss-2024-0013.

[14] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006. Discusses the 1-of-K coding scheme (One-Hot Encoding) for categorical variables.

[15] Tegawendé F Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillere. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th annual computer software and applications conference*, pages 303–312. IEEE, 2013.

[16] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. doi: 10.1191/1478088706qp063oa.

[17] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. doi: 10.1023/A:1010933404324.

[18] Glenn W. Brier. Verification of Forecasts Expressed in Terms of Probability. *Monthly Weather Review*, 78(1):1, January 1950. doi: 10.1175/1520-0493(1950)078⟨0001:VOFEIT⟩2.0.CO;2.

[19] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario

Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

[20] Dan's Tools. Unix time stamp - epoch converter. https://www.unixtimestamp.com/, 2026. Accessed: 20-04-2026.

[21] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[22] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 27, 2014. URL https://papers.nips.cc/paper/2014/hash/ede7e2b6d13a41ddf9f4bdef84fdc737-Abstract.html.

[23] Richard A. Dubniczky, Krisztofer Zoltan Horvát, Tamás Bisztray, Mohamed Amine Ferrag, Lucas C. Cordeiro, and Norbert Tihanyi. Castle: Benchmarking dataset fornbsp;static code analyzers andnbsp;llms towards cwe detection. In *Theoretical Aspects of Software Engineering: 19th International Symposium, TASE 2025, Limassol, Cyprus, July 14–16, 2025, Proceedings*, page 253–272, Berlin, Heidelberg, 2025. Springer-Verlag. ISBN 978-3-031-98207-1. doi: 10.1007/978-3-031-98208-8_15. URL https://doi.org/10.1007/978-3-031-98208-8_15.

[24] Matteo Esposito, Valentina Falaschi, and Davide Falessi. An extensive comparison of static application security testing tools. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, page 69–78, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400717017. doi: 10.1145/3661167.3661199. URL https://doi.org/10.1145/3661167.3661199.

[25] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020. URL https://arxiv.org/abs/2002.08155.

[26] Xiuting Ge, Chunrong Fang, Quanjun Zhang, Daoyuan Wu, Bowen Yu, Qirui Zheng, An Guo, Shangwei Lin, Zhihong Zhao, Yang Liu, and Zhenyu Chen. Pre-trained model-based actionable warning identification: A feasibility study. *ACM Trans. Softw. Eng. Methodol.*, November 2025. ISSN 1049-331X. doi: 10.1145/3777369. URL https://doi.org/10.1145/3777369. Just Accepted.

[27] Software Improvement Group. Sigrid — software assurance platform — documentation. https://docs.sigrid-says.com/, 2025. Accessed: 17-11-2025.

[28] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. Mitigating false positive static analysis warnings: Progress, challenges, and opportunities. *IEEE Transactions on Software Engineering*, 49(12):5154–5188, 2023. doi: 10.1109/TSE.2023.3329667.

[29] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Science & Business Media, 2nd edition, 2009.

[30] Sarah Heckman and Laurie Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4):363–387, 2011. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2010.12.007. URL https://www.sciencedirect.com/science/article/pii/S0950584910002235. Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing.

[31] Sarah Smith Heckman. *A Systematic Model Building Process for Predicting Actionable Static Analysis Alerts*. PhD thesis, North Carolina State University, Raleigh, NC, USA, 2009. Committee Chair: Laurie Williams.

[32] Péter Hegedűs and Rudolf Ferenc. Static code analysis alarms filtering reloaded: A new real-world dataset and its ML-based utilization. *IEEE Access*, 10:55090–55101, 2022. doi: 10.1109/ACCESS.2022.3176865. URL https://doi.org/10.1109/ACCESS.2022.3176865.

[33] David W. Hosmer Jr, Stanley Lemeshow, and Rodney X. Sturdivant. *Applied Logistic Regression*. John Wiley & Sons, Hoboken, New Jersey, 2013.

[34] Huimin Hu, Yingying Wang, Julia Rubin, and Michael Pradel. An empirical study of suppressed static analysis warnings. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi: 10.1145/3715729. URL https://doi.org/10.1145/3715729.

[35] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013. doi: 10.1109/ICSE.2013.6606613.

[36] Pascal Joos, Islem Bouzenia, and Michael Pradel. Codecureagent: Automatic classification and repair of static analysis warnings, 2026. URL https://arxiv.org/abs/2509.11787.

[37] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2017, pages 35–42, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350716. doi: 10.1145/3088525.3088675. URL https://doi.org/10.1145/3088525.3088675.

[38] Ugur Koc, Shiyi Wei, Jeffrey S. Foster, Marine Carpuat, and Adam A. Porter. An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 288–299, 2019. doi: 10.1109/ICST.2019.00036.

[39] Tom Krantz and Alexandra Jonker. What is alert fatigue? URL https://www.ibm.com/think/topics/alert-fatigue. Accessed: 11-03-2026.

[40] Ted Kremenek and Dawson Engler. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, page 295–315, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3540403256.

[41] Dávid Kószó, Tamás Aladics, Rudolf Ferenc, and Péter Hegedűs. A large-scale collection of (non-)actionable static code analysis reports, 2025. URL https://arxiv.org/abs/2511.10323.

[42] LangChain. State of agent engineering. https://www.langchain.com/state-of-agent-engineering, 2026. Accessed: 29-05-2026.

[43] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim, and Shin Yoo. Classifying false positive static checker alarms in continuous integration using convolutional neural networks. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 391–401, 2019. doi: 10.1109/ICST.2019.00048.

[44] Gilles Louppe. Understanding random forests: From theory to practice, 2015. URL https://arxiv.org/abs/1407.7502.

[45] P. Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, 2006. doi: 10.1109/MS.2006.114.

[46] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008. ISBN 0521865719.

[47] M.M. Mark and C.S. Reichardt. Internal validity. In Neil J. Smelser and Paul B. Baltes, editors, *International Encyclopedia of the Social Behavioral Sciences*, pages 7749–7752. Pergamon, Oxford, 2001. ISBN 978-0-08-043076-8. doi: https://doi.org/10.1016/B0-08-043076-7/00729-4. URL https://www.sciencedirect.com/science/article/pii/B0080430767007294.

[48] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *International Conference on Learning Representations*, 2013. URL https://api.semanticscholar.org/CorpusID:5959482.

[49] MITRE Corporation. Common weakness enumeration (CWE). https://cwe.mitre.org/data/index.html, 2026. Accessed: 2026-05-24.

[50] Tukaram Muske and Alexander Serebrenik. Survey of approaches for postprocessing of static analysis alarms. *ACM Comput. Surv.*, 55(3), February 2022. ISSN 0360-0300. doi: 10.1145/3494521. URL https://doi.org/10.1145/3494521.

[51] Marc Oriol, Jordi Marco, and Xavier Franch. Quality models for web services: A systematic mapping. *Information and Software Technology*, 56(10):1167–1182, 2014. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2014.03.012. URL https://www.sciencedirect.com/science/article/pii/S0950584914000822.

[52] Julie Peterson. Stopping alert fatigue in 3 simple steps, 2025. URL https://cycode.com/blog/stopping-alert-fatigue-3-simple-steps/. Accessed: 11-03-2026.

[53] PMD Open Source Project. *PMD Documentation (v7.18.0)*. PMD Open Source Project, 2025. URL https://docs.pmd-code.org/latest/. Accessed: 11-03-2026.

[54] Associated Press. Recently uncovered software flaw 'most critical vulnerability of the last decade', dec 2021. URL https://www.theguardian.com/technology/2021/dec/10/software-flaw-most-critical-vulnerability-log-4-shell. Accessed: 22-01-2026.

[55] Philipp Probst, Marvin N. Wright, and Anne-Laure Boulesteix. Hyperparameters and tuning strategies for random forest. *WIREs Data Mining and Knowledge Discovery*, 9(3), January 2019. ISSN 1942-4795. doi: 10.1002/widm.1301. URL http://dx.doi.org/10.1002/widm.1301.

[56] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 341–350, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791. doi: 10.1145/1368088.1368135. URL https://doi.org/10.1145/1368088.1368135.

[57] Saurabh Sarkar and Chris Parnin. Characterizing and predicting mental fatigue during programming tasks. In *Proceedings of the 2nd International Workshop on Emotion Awareness in Software Engineering*, SEmotion '17, page 32–37. IEEE Press, 2017. ISBN 9781538627938.

[58] scikit-learn developers. 3.2.3. Searching for optimal parameters with successive halving. https://scikit-learn.org/stable/modules/grid_search.html#successive-halving-user-guide, 2026. Accessed: 20-04-2026.

[59] scikit-learn developers. OneHotEncoder. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html, 2026. Accessed: 20-04-2026.

[60] Scikit-learn Developers. Precision-recall documentation and visual examples. https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html, 2026. Accessed: 22-06-2026.

[61] Software Improvement Group. About us - SIG. https://www.softwareimprovementgroup.com/who-we-are/, 2026. Accessed: 29-05-2026.

[62] Software Improvement Group. Sigrid® — Continuous Software Portfolio Governance. https://www.softwareimprovementgroup.com/sigrid/, 2026. Accessed: 29-05-2026.

[63] SpotBugs Project. *SpotBugs Manual*, 2024. URL https://spotbugs.readthedocs.io/. Accessed: 11-03-2026.

[64] Daniela Steidl and Sebastian Eder. Prioritizing maintainability defects based on refactoring recommendations. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, page 168–176, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328791. doi: 10.1145/2597008.2597805. URL https://doi.org/10.1145/2597008.2597805.

[65] Yaodan Tan and Junfeng Tian. A method for processing static analysis alarms based on deep learning. *Applied Sciences*, 14(13), 2024. ISSN 2076-3417. doi: 10.3390/app14135542. URL https://www.mdpi.com/2076-3417/14/13/5542.

[66] The Joblib Developers. Joblib: running python functions as pipeline jobs. URL https://joblib.readthedocs.io. Accessed: 17-06-2026.

[67] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[68] TIOBE Software BV. Tiobe index for Java. https://www.tiobe.com/tiobe-index/, 2026. Accessed: 10-04-2026.

[69] TIOBE Software BV. Tiobe index for Java. https://www.tiobe.com/tiobe-index/java/, 2026. Accessed: 10-04-2026.

[70] Alexander Trautsch. Effects of automated static analysis tools: A multidimensional view on quality evolution. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 184–185, 2019. doi: 10.1109/ICSE-Companion.2019.00075.

[71] Floris van Leeuwen. Commit classification using large language models. Master's thesis, University of Amsterdam, Amsterdam, Netherlands, September 2024. URL https://scripties.uba.uva.nl/search?id=record_55281. 100 pages. Academic supervisor: Thomas van Binsbergen. Host organisation: Software Improvement Group.

[72] Cornelis J. Van Rijsbergen. *Information Retrieval*. Butterworths, London, 2nd edition, 1979. URL http://www.dcs.gla.ac.uk/Kepler/vjr/book.html.

[73] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25(2):1419–1457, 2020. doi: 10.1007/s10664-019-09750-5. URL https://doi.org/10.1007/s10664-019-09750-5.

[74] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

[75] Maja Vukovic, Rangeet Pan, Tin Kam Ho, Rahul Krishna, Raju Pavuluri, and Michele Merler. Usage, effects and requirements for AI coding assistants in the enterprise: An empirical study. In *Proceedings of the 3rd International Workshop on Large Language Models For Code (LLM4Code '26)*. ACM, 2026. doi: 10.1145/3786181.3788727. Co-located at ICSE 2026. Also available as arXiv preprint arXiv:2601.20112.

[76] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308, 2016. doi: 10.1145/2884781.2884804.

[77] Guo Xintong, Wang Hongzhi, Yangqiu Song, and Gao Hong. Brief survey of crowdsourcing for data mining. *Expert Systems with Applications*, 41(17):7987–7994, 2014. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2014.06.044. URL https://www.sciencedirect.com/science/article/pii/S0957417414003984.

[78] Xueqi Yang, Jianfeng Chen, Rahul Yedida, Zhe Yu, and Tim Menzies. Learning to recognize actionable static code warnings (is intrinsically easy). *Empirical Softw. Engg.*, 26(3), May 2021. ISSN 1382-3256. doi: 10.1007/s10664-021-09948-6. URL https://doi.org/10.1007/s10664-021-09948-6.

[79] Xueqi Yang, Zhe Yu, Junjie Wang, and Tim Menzies. Understanding static code warnings: An incremental ai approach. *Expert Syst. Appl.*, 167(C), April 2021. ISSN 0957-4174. doi: 10.1016/j.eswa.2020.114134. URL https://doi.org/10.1016/j.eswa.2020.114134.

[80] Jongwon Yoon, Minsik Jin, and Yungbum Jung. Reducing false alarms from an industrial-strength static analyzer by svm. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 2, pages 3–6, 2014. doi: 10.1109/APSEC.2014.81.

[81] Ulas Yüksel and Hasan Sözer. Automated classification of static code analysis alerts: A case study. In *2013 IEEE International Conference on Software Maintenance*, pages 532–535, 2013. doi: 10.1109/ICSM.2013.89.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

ASAT: Automatic Static Analysis Tool.

SCA: Static Code Analysis.

SIG: Software Improvement Group.

ML: Machine Learning.

RF: Random Forest.

LR: Logistic Regression.

DL: Deep Learning.

LLM: Large Language Model.

AI: Artificial Intelligence.

TP: True Positive.

FP: False Positive.

NASCAR: Non-Actionable Static Code Analysis Reports.

LSA: Latent Semantic Analysis.

TF-IDF: Term Frequency-Inverse Document Frequency.

CWE: Common Weakness Enumeration.

Appendix B

Helper Functions

This chapter provides the functions referenced throughout the study. While these functions support the underlying technical framework, they are presented here in the appendix to maintain the narrative flow of the main text. The core analysis remains self-contained and comprehensible without these secondary scripts, but they are included here for the sake of reproducibility and technical completeness.

```
def helper_split_data(data, pos_split, neg_split):
    '''
    This function splits the data into a new dataframe with a specific
    amount of positive and negative samples.

    Args:
    data (DataFrame): The original dataframe.
    pos_split (int): The number of positive samples to include.
    neg_split (int): The number of negative samples to include.

    Returns:
    DataFrame: A new dataframe with the specified number of positive and
    negative samples.
    '''
    pos_data = data[data['label'] == 1].sample(n=pos_split, random_state
    =42)
    neg_data = data[data['label'] == 0].sample(n=neg_split, random_state
    =42)
    new_data = pd.concat([pos_data, neg_data]).reset_index(drop=True)

    return new_data
```

Listing B.1: The code for the helper_split function. It splits data into a new dataframe.

Appendix C

Informed Consent Form

This chapter presents the informed consent form provided to all participants prior to the user study phase of this research. The following form outlines the study's objectives, the nature of the tasks involving static analysis warning triage, and the data handling protocols used to maintain participant anonymity.

Participant Information

You are being invited to participate in a research study titled "Measuring ML-Driven Triage Efficiency and Developer Trust for SCA". This study is conducted by Vivian Ning from the TU Delft in collaboration with SIG, and is part of a master thesis.

The primary focus of this study is to evaluate how an ML-based ranking system alters developer behavior and decision-making across the Software Development Life Cycle (SDLC) for Static Code Analysis (SCA). By exploring the dimensions of workflow integration, trust calibration, and decision-making efficiency, this research investigates where a prioritized list of warnings provides the most significant reduction in manual triage time. Whether this is in the IDE or during the pull request phase. Crucially, the study examines the dynamics of developer trust to identify how the presentation of probability scores influences a user's willingness to investigate a warning and at what accuracy threshold the tool maintains credibility. Finally, this research seeks to determine if such a tool aligns with practical developer needs to improve code review efficiency and reduce the Mean Time to Detect (MTTD), or if the presence of ML-generated rankings introduces new forms of cognitive bias that could impact the reliability of the development process.

As with any online activity the risk of a breach is always possible. To the best of our ability your answers in this study will remain confidential. We will minimize any risks by fully informing you about the details of the experiment and its associated risks, providing you with the option to opt out at any time, and collecting only information that does not qualify as personal data under the GDPR.

Your participation in this study is entirely voluntary, and you can withdraw at any time. You are free to omit any questions. You can request to delete your experiment data such as demographic information and your questionnaire results at any time. You can find the contact details below.

The responsible researcher can be reached by email : vivian.ning@softwareimprovementgroup.com or phone number: +31 619274274.

24th March 2026

Explicit Consent points

PLEASE TICK THE APPROPRIATE BOXES	Yes	No
A: GENERAL AGREEMENT – RESEARCH GOALS, PARTICIPANT TASKS AND VOLUNTARY PARTICIPATION	☐	☐
1. I have read and understood the study information dated 24/03/2026, or it has been read to me. I have been able to ask questions about the study and my questions have been answered to my satisfaction.	☐	☐

2. I consent voluntarily to be a participant in this study and understand that I can refuse to answer questions and I can withdraw from the study at any time, without having to give a reason.	☐	☐
3. I understand that taking part in the study involves: - Triaging warnings manually. - Filling out a questionnaire to provide demographic information (such as job role and security experience), report current triage habits, and rank the features and probability thresholds that influence my trust in automated tools. - Answering questions from the interviewer about the triage experiment and questionnaire.	☐	☐
4. I understand that I will be compensated for my participation by home-made cookies.	☐	☐
5. I understand that the study will end in about 75 minutes.	☐	☐
B: POTENTIAL RISKS OF PARTICIPATING (INCLUDING DATA PROTECTION)		
6. I understand that taking part in the study involves the following risks: The study involves evaluating professional performance during code triage, which includes tracking my accuracy and speed in identifying security warnings. Because this research is conducted within a specific organizational context (SIG), there is a risk of indirect re-identification if my professional attributes (e.g., job role, experience level) are combined with my performance metrics. I understand that these will be mitigated by: All research data will be strictly pseudonymized and stored on secure university servers, with all direct identifiers removed. To prevent re-identification within a small participant pool, demographic data will be generalized into broad categories (e.g., experience ranges) and results will only be reported in aggregate form.	☐	☐
7. I understand that taking part in the study also involves collecting specific personally identifiable information (PII) and associated personally identifiable research data (PIRD) (Familiarity with	☐	☐

technology, years of experience, job position). I understand that measures (e.g., aggregation of data, no collection of personal names or other identifications) have been taken to make sure that re-identification of myself is not possible after I have completed the study.		
8. I understand that the following steps will be taken to minimize the threat of a data breach and protect my identity in the event of such a breach: The data collected is anonymized and cannot be traced back to individual participants. We will avoid storing any direct identifiers (e.g., names, email addresses) unless absolutely necessary.	☐	☐
9. I understand that the (identifiable) personal data I provide will be destroyed after the completion of the study. So the data will be destroyed 3 months after completion of this study, and I understand that I can ask for the destruction of my data at any moment after I have completed the study.	☐	☐
C: RESEARCH PUBLICATION, DISSEMINATION AND APPLICATION		
13. I agree that my responses, views or other input can be quoted anonymously in research outputs	☐	☐

Signatures

_____	_____	_____
Name of participant	Signature	Date

I, as legal representative, have witnessed the accurate reading of the consent form with the potential participant and the individual has had the opportunity to ask questions. I confirm that the individual has given consent freely.

_____	_____	_____
Name of witness	Signature	Date

I, as researcher, have accurately read out the information sheet to the potential participant and, to the best of my ability, ensured that the participant understands to what they are freely consenting.

_____	_____	_____
Researcher name	Signature	Date

Study contact details for further information:

Name: *Vivian Ning,*

Phone number: +31 619247247,

Email address: vivian.ning@softwareimprovementgroup.com]

Appendix D

The User Study Questions

This chapter details the interview guide used during the semi-structured phase of the user study. These core questions served as a flexible framework rather than a rigid script. They were employed selectively to probe deeper into specific topics or to clarify insights that emerged during the preceding experimental phases.

- "You started with/without the scores. Did you find yourself triaging different? If so why?"
- "You ranked A as more important than B for trusting the score. Why is A a better validator of the model's logic than B?"
- "Given your expertise, do you find yourself looking for reasons to disprove the ML score, or do you see it as a teammate that catches things you might miss?"
- "You indicated that X% is your 'trust threshold'. What is the mental difference for you between an $X - 10\%$ score and an X% score?"
- "If these scores appeared in your workflow tomorrow, would it actually reduce your workload or help you during the process? If so, how (not)?"

Appendix E

LSA Components

This chapter provides the tables listing the LSA components for the Warning Message and Source Code features.

Table E.1: Keywords across LSA Components for Warning Message Feature.

Component	Keywords
Component 1	final, declared, local, variable, parameter, assigned, names, like, excessively, long
Component 2	names, like, avoid, excessively, long, variable, short, variables, local, id
Component 3	parameter, assigned, names, like, avoid, short, variables, excessively, long, final
Component 4	digit, underscore, separate, number, high, indicate, imports, coupling, object, degree
Component 5	match, doesn, field, constant, za, method, public, access, modifier, missing
Component 6	line, file, occurrence, appears, literal, string, times, long, excessively, used
Component 7	variables, short, local, file, string, appears, occurrence, literal, times, line
Component 8	access, field, modifier, missing, commented, default, method, value, foreign, degree
Component 9	method, statement, exit, point, complexity, class, cyclomatic, junit, threshold, current
Component 10	value, method, field, foreign, degree, used, overwritten, statement, exit, point
Component 11	tests, junit, private, package, assert, unit, contain, class, test, constructor
Component 12	class, used, overwritten, constructor, declare, value, line, variable, assigned, constant

Table E.1: Keywords across LSA Components for Warning Message Feature.

Component	**Keywords**
Component 13	class, field, declare, constructor, test, final, za, foreign, degree, declared
Component 14	foreign, degree, constant, public, value, access, statement, point, exit, local
Component 15	za, test, junit, foreign, degree, value, access, instance, match, doesn
Component 16	assert, contain, unit, tests, za, fail, include, declare, test, short
Component 17	complexity, cyclomatic, current, threshold, cognitive, method, string, npath, unnecessary, public
Component 18	unused, import, avoid, using, use, test, org, za, java, assert
Component 19	unused, import, constructor, declare, java, org, field, lang, util, com
Component 20	constructor, declare, using, complexity, avoid, redundant, threshold, literals, initializer, current
Component 21	type, unnecessary, new, explicit, constructor, replaced, use, diamond, arguments, declare
Component 22	new, braces, statement, complexity, replaced, inside, instantiating, objects, loops, diamond
Component 23	use, braces, statement, implementation, run, concurrenthashmap, concurrent, newer, complexity, field
Component 24	exception, block, catch, nullpointerexception, exceptions, catching, generic, runtimeexception, try, lang
Component 25	braces, test, statement, unnecessary, scope, qualifier, java, using, lang, initializer
Component 26	usage, err, qualified, useless, java, lang, braces, statements, unnecessary, element
Component 27	test, cases, contains, public, field, constant, inside, instantiating, loops, objects
Component 28	logger, level, log, calls, guards, surrounded, braces, unnecessary, size, fields
Component 29	fields, constructors, inner, classes, initializers, declarations, private, braces, methods, unnecessary
Component 30	objects, instantiating, loops, inside, java, lang, scope, qualifier, unnecessary, new
Component 31	return, linguistics, linguistically, antipattern, void, setter, instantiating, objects, loops, inside
Component 32	cyclomatic, total, highest, complexity, class, initializer, implementation, concurrent, run, newer

Table E.1: Keywords across LSA Components for Warning Message Feature.

Component	Keywords
Component 33	initializer, java, lang, redundant, concurrent, run, newer, concurrenthashmap, access, implementation
Component 34	consider, methods, refactoring, current, threshold, object, data, exit, point, class
Component 35	consider, practice, java, refactoring, methods, lang, object, super, good, constants

Table E.2: Keywords across LSA Components for Source Code Feature.

Component	Keywords
Component 1	final, int, static, public, string, private, class, void, new, return
Component 2	string, private, void, new, null, return, class, map, list, boolean
Component 3	private, final, string, static, map, logger, volatile, getlogger, loggerfactory, hashmap
Component 4	string, int, map, final, hashmap, protected, key, entry, length, format
Component 5	return, int, null, new, boolean, private, false, list, long, object
Component 6	return, class, null, static, import, public, extends, boolean, final, false
Component 7	new, import, list, org, arraylist, static, map, final, class, java
Component 8	import, int, org, java, string, util, class, com, private, junit
Component 9	class, int, extends, author, implements, string, map, new, public, logger
Component 10	null, void, test, static, throws, exception, private, try, file, logger
Component 11	boolean, null, protected, class, object, false, map, true, extends, int
Component 12	long, list, object, map, protected, boolean, value, null, jsonproperty, override
Component 13	list, public, arraylist, object, private, map, static, void, integer, override
Component 14	object, map, value, hashmap, test, return, key, throws, entry, exception
Component 15	append, sb, stringbuilder, object, public, override, value, toindentedstring, tostring, private
Component 16	list, test, append, protected, final, exception, throws, sb, return, class
Component 17	protected, override, void, final, throws, import, ioexception, logger, object, extends

Table E.2: Keywords across LSA Components for Source Code Feature.

Component	Keywords
Component 18	value, protected, jsonproperty, jsoninclude, override, key, throws, void, exception, byte
Component 19	static, throws, exception, catch, ioexception, file, try, error, logger, throw
Component 20	protected, static, test, void, return, map, value, append, float, hashmap
Component 21	byte, object, override, static, test, bytes, string, length, key, assertequals
Component 22	jsonproperty, object, bigdecimal, id, test, type, static, jsoninclude, protected, string
Component 23	byte, jsonproperty, map, bigdecimal, public, id, bytes, throws, test, protected
Component 24	void, byte, float, type, logger, false, true, java, assertequals, class
Component 25	file, java, util, lang, path, ioexception, files, com, google, exists
Component 26	assertequals, override, file, org, testresult, jsonobject, assert, id, result, logger
Component 27	float, file, public, throws, org, test, double, assertequals, code, color
Component 28	override, type, java, throws, static, com, assertequals, id, test, lang
Component 29	assertequals, java, testresult, jsonobject, catch, public, logger, assert, com, lang
Component 30	type, boolean, return, org, builder, method, file, case, final, state
Component 31	override, jsonproperty, boolean, logger, static, test, void, class, return, java
Component 32	id, logger, test, key, float, java, error, catch, value, getlogger
Component 33	integer, id, throws, void, assertequals, class, return, ioexception, object, final
Component 34	integer, test, logger, type, override, catch, java, true, public, error
Component 35	com, model, alibaba, google, builder, protobuf, assertthat, nacos, result, istio
Component 36	extends, catch, result, try, throw, super, key, java, void, exception
Component 37	key, result, assertthat, config, isequalto, size, properties, context, true, set
Component 38	result, false, logger, key, extends, size, equals, entry, getlogger, throws
Component 39	key, config, extends, properties, super, jsonproperty, com, integer, void, index
Component 40	true, result, logger, code, extends, throws, getlogger, add, field, set

Table E.2: Keywords across LSA Components for Source Code Feature.

Component	Keywords
Component 41	extends, assertthat, logger, super, isequalto, size, throws, getlogger, loggerfactory, task
Component 42	config, super, extends, properties, context, result, value, builder, false, java
Component 43	builder, method, build, false, code, request, org, key, super, add
Component 44	size, set, assert, double, assertequals, add, exception, entry, test, builder
Component 45	context, properties, super, size, message, try, var, source, target, cache
Component 46	properties, assertthat, builder, isequalto, assertequals, override, double, void, assert, hashmap
Component 47	set, add, properties, hashset, jsonobject, testresult, field, ok, entry, getjsonobject
Component 48	double, super, entry, equals, add, bigdecimal, model, message, field, code
Component 49	super, message, try, equals, ioexception, request, data, method, throwable, exchange
Component 50	try, ioexception, size, jsonobject, testresult, false, data, println, url, double
Component 51	equals, entry, try, bigdecimal, extends, ioexception, add, url, objects, author
Component 52	size, entry, equals, jsonobject, testresult, exception, method, add, ok, super
Component 53	data, add, message, length, bigdecimal, code, node, arraylist, println, properties
Component 54	entry, bigdecimal, instance, super, getvalue, entryset, getkey, add, assertequals, assert
Component 55	field, bigdecimal, method, code, length, add, index, false, throw, hashmap
Component 56	method, request, add, jsoninclude, code, field, entry, path, include, assertequals
Component 57	add, try, arraylist, model, request, exception, hashmap, byte, size, bigdecimal
Component 58	length, add, jsoninclude, arraylist, super, try, char, include, args, stringbuilder
Component 59	instance, field, code, message, com, equals, service, add, author, ioexception
Component 60	request, instance, bigdecimal, length, method, exchange, response, true, throw, println

Table E.2: Keywords across LSA Components for Source Code Feature.

Component	Keywords
Component 61	instance, model, length, data, field, try, exception, super, node, org
Component 62	ioexception, model, throw, add, error, catch, case, org, jsoninclude, runtimeexception
Component 63	data, method, node, state, url, bigdecimal, visit, instance, bean, apigetfunctionaddress
Component 64	model, message, println, method, instance, args, extends, objectmapper, throwable, length
Component 65	println, case, field, implements, thread, author, exception, millis, break, start
Component 66	url, model, author, path, message, implements, println, instance, method, exception
Component 67	author, method, case, implements, state, node, try, event, args, code
Component 68	case, url, field, break, message, switch, objectmapper, state, args, event
Component 69	field, author, implements, model, message, ioexception, path, service, error, true
Component 70	task, thread, jsonnode, args, getid, instanceof, org, lang, implements, state
Component 71	event, apigetfunctionaddress, index, stringbuilder, implements, char, task, service, objectmapper, state
Component 72	state, tree, index, visitorstate, description, log, println, date, char, jsonproperty
Component 73	event, org, author, lang, google, protobuf, str, iterator, stringbuilder, java
Component 74	path, log, stringbuilder, org, lang, protobuf, google, jsonproperty, input, event
Component 75	node, stringbuilder, author, index, char, path, instanceof, param, date, offset
Component 76	implements, objectmapper, bean, jsonproperty, mapper, str, stringbuilder, json, input, add
Component 77	stringbuilder, bigdecimal, jsoninclude, implements, service, log, include, annotation, instanceof, node
Component 78	objectmapper, index, path, jsonnode, arraylist, source, mapper, target, objectnode, iterator
Component 79	service, bean, node, path, net, code, iterator, sip, communicator, thread

Table E.2: Keywords across LSA Components for Source Code Feature.

Component	**Keywords**
Component 80	arraylist, date, node, offset, event, bytes, hashmap, log, thread, char