

How much model does warning triage need? A cost-aware comparison from rules to locally run language models

Peter Sullivan^{1,*} and Bogdan Mikhaylov² 

¹Software Engineering Manager, Los Angeles, United States

²Senior Software Developer

* Corresponding author: Peter Sullivan, peter.s.034512@gmail.com

OPEN ACCESS

Citation:

Peter Sullivan, Bogdan Mikhaylov (2026).
How much model does warning triage need?
A cost-aware comparison from rules to
locally run language models. *Am. Impact
Rev.* e80134906.

[10.66308/air.e80134906](https://doi.org/10.66308/air.e80134906)

Received: October 2, 2026

Accepted: October 28, 2026

Published: October 29, 2026

DOI:

[10.66308/air.e80134906](https://doi.org/10.66308/air.e80134906)

ISSN: 3071-124X

Copyright:

© 2026 Peter Sullivan. This is an open
access article distributed under the terms of
the Creative Commons Attribution License
(CC BY 4.0).

Abstract

Static analysis tools report far more warnings than developers act on, and machine learning is widely proposed to identify the actionable ones. Published comparisons rarely include a simple label-based reference, and rarely report quality on unseen projects together with compute time. This study compares five levels of model complexity on two open datasets, NASCAR (Java) and AW4C (C): a warning-type prior, a linear model on term frequency-inverse document frequency (TF-IDF) features, a frozen code encoder, fine-tuned code encoders, and open-weight large language models (LLMs) run locally. All models were scored on the same 1,000 warnings per dataset and protocol, using the Matthews correlation coefficient (MCC) with project-level bootstrap intervals. Three findings emerged. First, in the original AW4C release the number of lines in the supplied code context predicted the label with an MCC of 0.91 on unseen projects, and fine-tuned CodeBERT reached 0.81; in a variant rebuilt with a uniform context window, which also differs in composition, the corresponding values were 0.12 and 0.30. Second, on unseen NASCAR projects an advantage of fine-tuned encoders over the warning-type prior was not established: the paired difference in MCC was -0.08 (95% CI -0.21 to 0.02) for CodeBERT. Third, locally run LLMs without task-specific training scored near zero on NASCAR with the default threshold and required about 47 times the mean processing time of a fine-tuned encoder. Model complexity should be assessed against a simple reference under cross-project evaluation.

Keywords: static analysis, actionable warnings, code language models, large language models, cross-project evaluation, shortcut learning

1. Introduction

Static analysis tools inspect source code without running it and report potential defects, security weaknesses, and violations of coding rules. Their practical weakness is volume. A project can receive thousands of warnings, and reported shares of warnings on which developers do not act range from 35% to 91% [1, 2]. Warnings that are followed by a change of the code are called actionable. The label records observed developer behaviour; a non-actionable warning is not necessarily false or useless. Automatic identification of actionable warnings has therefore been studied for more than a decade, first with hand-crafted features and classifiers ranging from support vector machines to feed-forward networks [3, 4], later with neural models trained on

code [5, 6], and most recently with large language models (LLMs) [7, 8, 9, 10].

An engineer who has to deploy such a filter faces a choice that the literature does not settle. A rule, a linear model, a fine-tuned compact encoder, and an LLM differ by orders of magnitude in compute time and engineering effort. The more complex option is justified only if it is more accurate in the conditions of deployment. Three gaps make this judgement difficult.

The first gap concerns evaluation on new projects. A filter is most valuable when a project has no labelled history of its own, which means that the model must work on code it has not seen. Many studies split warnings at random, so that training and test data come from the same projects. Cross-project results have been reported for fine-tuned encoders [6] and for a sentence-transformer model [11], but without a rule-based reference measured on the same items. Kang et al. showed that earlier, very high scores for actionable warning identification were produced by data leakage and duplicates [2]. Similar problems have been documented for vulnerability detection datasets [12, 13, 14].

The second gap concerns the reference point. Comparisons usually set one learned model against another. A prior that uses only the type of the warning is learned from labels, does not read code, and runs in a negligible fraction of the time of any other model; it is rarely reported. Without it, a reader cannot tell how much of a model's score depends on reading code.

The third gap concerns cost. Studies of LLM-based triage report the price of LLM calls [8, 9, 10], and studies of compact models report accuracy [6], but the two are seldom measured on the same items and the same hardware. The closest study, from the group that built the NASCAR dataset, compared a sentence-transformer model with a commercial LLM, which was evaluated with few-shot prompts on 100 warnings, and did not report cost [11].

This study addresses the three gaps with a single controlled comparison. It asks four questions. RQ1: how does quality change along a ladder of model complexity, from a rule to a locally run LLM? RQ2: how much quality is lost on projects that were absent from training, and does greater complexity reduce the loss? RQ3: how much compute time does each rung require per warning, and at what volume does the total compute time of a fine-tuned encoder fall below that of an LLM? RQ4: does a cascade, in which an encoder passes its least confident cases to an LLM, improve the balance of quality and cost?

The study makes three contributions. It documents that the length of the code context distributed with the AW4C dataset predicts the label, and it describes a variant of the dataset with uniformly extracted context. It reports a comparison of five levels of model complexity under within-project and cross-project evaluation on identical test items, with a simple label-based reference and paired project-level uncertainty. It reports mean processing time for every rung on one machine and the volume at which the compute time of a fine-tuned encoder and an LLM is equal. All experiments used open data, open-weight models, and one consumer-grade computer.

2. Related work

Actionable warning identification

A survey by Ge et al. organizes the field along a workflow of dataset preparation, preprocessing, model construction, and evaluation [1]. Yang et al. reported that the task appeared intrinsically easy, since simple learners reached very high scores [3]. Kang et al. traced those scores to leaked features and duplicated records, and after correction the same learners performed far worse [2]. Yedida et al. revisited the corrected data with tuned learners and again reported high scores [4]. Xue et al. argued that an actionable label inferred from the disappearance of a warning is itself noisy [15]. This history motivates the checks for shortcuts applied in the

present study.

Datasets

NASCAR contains more than one million warnings raised by PMD and SpotBugs on Java projects selected from GitHub; its authors report 102 selected projects, and 92 projects are present in the released data by our count. A warning is labelled actionable when its context was changed in the following commit and the warning no longer appears there [16]. AW4C contains warnings raised by Cppcheck on C projects, labelled by commit-aware tracking [17]. D2A labels warnings of the Infer analyzer by differential analysis of bug-fixing commits [18]. SastBench targets agentic triage of security findings [19].

Compact encoders for code

CodeBERT [20], GraphCodeBERT [21], CodeT5 [22], and UniXcoder [23] are pre-trained models whose released base versions have roughly 125 to 220 million parameters. Ge et al. fine-tuned such models for actionable warning identification on Java warnings and found them feasible, without an LLM comparison or cost measurements [6].

LLMs for warning triage

Several systems pair a static analyzer with an LLM that confirms or rejects its findings, and supply code context obtained by slicing, data-flow extraction, or retrieval of definitions on request [8, 24, 25, 26, 27]. Wen et al. inspected thousands of warnings with an LLM [7]. Du et al. reported an industrial study with a cost between 0.0011 and 0.12 US dollars per warning on a proprietary dataset [9]. Xiong and Zhang compared LLM agents on the OWASP Benchmark and on alerts from real-world projects [10].

Compact models against LLMs

In vulnerability detection, fine-tuned compact models exceeded prompted LLMs in several studies [28, 29]. A multi-language benchmark found that compact models were better for Java and JavaScript and worse for Python [30], and an evaluation on recent vulnerabilities found both trained models and prompted LLMs unreliable [31]. In the detection of self-admitted technical debt, ChatGPT achieved higher recall and small models higher precision, and their combination performed best [32]. The superiority of fine-tuned models is therefore not claimed as new here. The present study differs in the task, in the inclusion of a rule-based reference, in cross-project evaluation, and in the measurement of compute time.

Cost-aware model selection

FrugalGPT, RouteLLM, and Hybrid LLM route queries between cheaper and more expensive LLMs [33, 34, 35]. FrugalGPT cascades LLMs, whereas RouteLLM and Hybrid LLM select one LLM before the query is answered; in all three, every candidate is an LLM. The cascade examined here starts from a fine-tuned encoder.

3. Materials and methods

3.1. Study design

Figure 1 summarizes the design. Two datasets were prepared with two evaluation protocols each. Five rungs of model complexity were scored on the same test warnings. Rungs 0 to 3 were trained on the same sample of training warnings; the conditions that differed between rungs are listed in Table 2. The primary outcome was the Matthews correlation coefficient (MCC) on unseen projects.

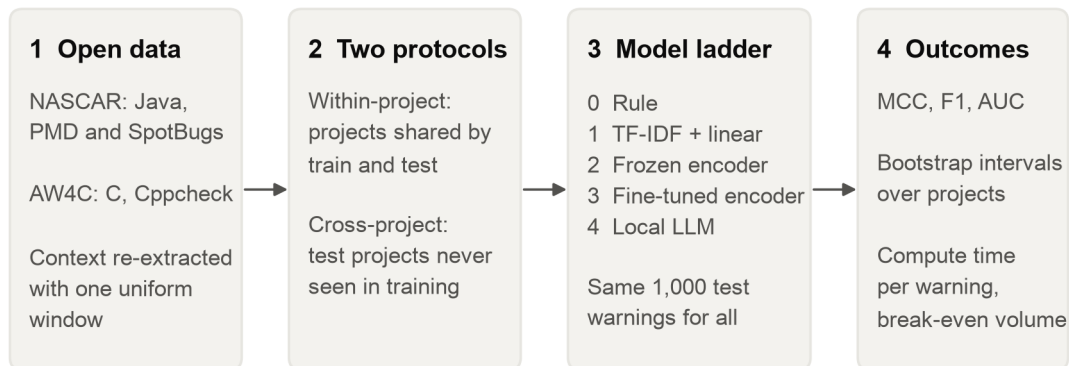


Figure 1. Study design. Two open datasets, two evaluation protocols, five rungs of model complexity, and the outcomes measured for every rung.

3.2. Datasets

NASCAR

The deduplicated release contains 1,083,073 warnings [16]. A small number of projects contribute most records, so at most 3,000 warnings per project were sampled, which left 246,583 warnings. The release stores complete source files. For every warning, the warned lines together with five lines before and five lines after were extracted. Records with identical warning type, message, and context were merged, and 94 records with contradictory labels were removed. The final set contains 246,475 warnings.

AW4C

The release contains 85,640 warnings with a code context field [17]. Removing records with identical type, message, and context left 71,869 warnings. This set is referred to as AW4C original. Section 4.1 shows that its context field leaks the label. A variant with uniformly extracted context, AW4C-U, was therefore built. It is not a corrected benchmark: it differs from the original in the warnings and projects retained and in the data split. Up to 150 warnings per project were sampled (34,481 warnings in 15,931 files). Each file was downloaded from its public repository at the commit in which the warning was reported; when the file could not be retrieved at that commit, the child commit recorded in the dataset was used. The warned line with five lines before and five lines after was extracted. The reporting commit supplied the code for all actionable warnings and for 43.3% of non-actionable warnings; the child commit supplied the rest. The extracted window contained at least one line of the originally distributed context for 99.8% of actionable and 77.8% of non-actionable warnings. Warnings that came from the reporting commit and whose window

overlapped the original context are called verified (19,700 warnings); they were used in a sensitivity analysis. Source code was retrieved for 99.1% of actionable and 93.2% of non-actionable warnings. After removal of duplicates the set contains 32,632 warnings from 561 projects. Table 1 describes the three sets. The share of actionable warnings varied widely between projects: among projects with at least 100 warnings the median share was 6% in the NASCAR release and 53% in the AW4C release. The D2A dataset [18] was inspected but not used, because only a reduced release with hidden test labels could be obtained.

Table 1. Datasets after preparation. Train and test sizes refer to the cross-project protocol.

Dataset	Language	Analyzer	Warnings	Projects	Warning types	Actionable (%)	Train / test	Test projects
NASCAR	Java	PMD, SpotBugs	246,475	92	406	16.5	172,774 / 47,277	16
AW4C original	C	Cppcheck	71,869	568	229	45.8	50,369 / 14,311	108
AW4C-U	C	Cppcheck	32,632	561	210	50.0	22,857 / 6,379	112

3.3. Evaluation protocols

In the within-project protocol, the warnings of every project were divided at random into training (70%), development (10%), and test (20%) parts, stratified by label. In the cross-project protocol, whole projects were assigned to the three parts in the same proportions, so that no test project contributed any training example. For each dataset and protocol, 3,000 test warnings were drawn at random, and 1,000 of them formed the core evaluation set. Every model was scored on these 1,000 warnings. On NASCAR the core set covered 16 projects under the cross-project protocol and 92 under the within-project protocol, and 17% of its warnings were actionable. On AW4C-U it covered 89 and 349 projects, with 54% actionable. The core sets of the two protocols contain different warnings, so differences between protocols are descriptive. The within-project split is random and does not follow time. The random seed for all splits was fixed before any model was trained.

3.4. Model ladder

Rung 0, warning-type prior

The score of a warning is the smoothed share of actionable warnings among training warnings of the same type. The prior is estimated from labelled data and does not read code. It is referred to below as the rule.

Rung 1, linear model

Unigram and bigram TF-IDF features of the tool name, warning type, severity, message, and code context were passed to logistic regression with balanced class weights, implemented in scikit-learn [36]. A variant without the code context was fitted as well.

Rung 2, frozen encoder

UniXcoder [23] produced mean-pooled embeddings without any update of its weights. Logistic regression was fitted on standardized embeddings.

Rung 3, fine-tuned encoder

CodeBERT [20] and UniXcoder were fine-tuned with a linear classification layer on mean-pooled output for two epochs, with a batch size of 32, a learning rate of 2e-5, linear warm-up over 6% of steps, class-weighted cross-entropy, and a maximum input length of 200 tokens.

Rung 4, local LLM

Qwen2.5-Coder [37] with 7 and 14 billion parameters was run locally through Ollama in 4-bit quantized form. The prompt defined the two classes and supplied the tool, language, rule, message, and code context. The model returned a single letter. The probability of the actionable class was computed from the log-probabilities of the first generated token. The system prompt read: “You triage warnings produced by a static code analysis tool. A warning is ACTIONABLE if developers are likely to change the code so that the warning disappears. It is NON-ACTIONABLE if developers are likely to leave the code as it is. Answer with exactly one letter: A for actionable, N for non-actionable.” Generation used a temperature of zero and a context window of 4,096 tokens. The zero-shot variant contained no examples. The few-shot variant contained three actionable and three non-actionable training examples, the same for every test warning. The few-shot examples were drawn from the complete training part. The main comparison uses the default decision threshold of 0.5, which requires no labels. In a separate analysis the threshold was adapted by two-fold cross-fitting on the core set: the threshold that maximized MCC on one half was applied to the other half. Both halves contain warnings of the test projects, so this analysis represents adaptation with 500 labelled warnings of the target projects and is not a test of transfer to projects without labels. Its results are marked in Table 3.

All rungs received the same fields: the tool name, warning type, severity, and message, followed by the code context. NASCAR does not record severity, so the field was empty. Encoder inputs were truncated to 200 tokens, which shortened the code context in about 45% of NASCAR inputs and 27% of AW4C-U inputs; the linear model and the LLMs read the complete window. Rungs 0 to 3 were trained on the same 16,000 training warnings per dataset and protocol, drawn at random from the training part. For rungs 0 to 3, the decision threshold maximized MCC on the development part. The rule and the linear models used the complete development part, the fine-tuned encoders at most 5,000 of its warnings, and the frozen encoder at most 3,000. Fine-tuning used one random seed. Table 2 lists the rungs and the conditions of each.

Table 2. Model ladder, training conditions, and training time. Training time was measured on NASCAR under the cross-project protocol. Parameter counts were obtained from the fitted or released models.

Rung	Model	Parameters updated	Training warnings	Warnings for threshold selection	Training time (s)
0	Warning-type prior	One value per warning type	16,000	Complete development part	0.004
1	TF-IDF + logistic regression, message only	8,334	16,000	Complete development part	0.26
1	TF-IDF + logistic regression, message and code	102,096	16,000	Complete development part	1.27
2	Frozen UniXcoder + linear layer	769 of 126 million	16,000	At most 3,000	139
3	Fine-tuned CodeBERT	125 million	16,000	At most 5,000	1082
3	Fine-tuned UniXcoder	126 million	16,000	At most 5,000	1094
4	Qwen2.5-Coder 7B and 14B	None	0, or 6 examples in the prompt	None (threshold 0.5)	0

3.5. Shortcut probe

To test whether the form of the code context reveals the label, a decision tree of depth four was trained on a single feature, the number of lines in the context. A probe that performs well indicates that a model can score highly without reading the code [38].

3.6. Outcomes and uncertainty

The primary outcome was MCC, which remains informative under class imbalance [39]. The F1 score and the area under the receiver operating characteristic curve (AUC) were secondary outcomes. Uncertainty was estimated with a percentile bootstrap of 2,000 draws in which whole projects were resampled with replacement, because warnings of one project are not independent. Differences between two models were estimated on the same resampled projects, and the interval of the paired difference is reported; overlap of the intervals of two separate models was not used as a test. All intervals are conditional on the fitted models and on the single data split.

3.7. Compute time and cascade

All computation ran on one Apple M4 Pro computer with 24 GB of memory. Encoders and LLMs used the graphics processor; rungs 0 and 1 used the central processor. For rungs 0 and 1, prediction of the 1,000 core warnings, including feature extraction, was repeated 21 times and the median was taken. For the encoders, time was recorded during the evaluation pass over up to 15,000 warnings, in batches of 32 for fine-tuned encoders and 64 for the frozen encoder. The LLMs processed the 1,000 core warnings one at a time. The reported quantity is the mean processing time per warning in the stated mode on this computer; for batched models it is not the latency of a single request. With fixed training time F and inference time v per warning, the total time for n warnings is $F + vn$. The break-even volume between a fine-tuned encoder and an LLM is the n at which the two totals are equal. The break-even volume refers to compute time only. It does not include development, labelling, maintenance, or the cost of errors, and it is not an estimate of financial return.

In the cascade, fine-tuned CodeBERT scored every warning. A fraction f of warnings with scores closest to the decision threshold was passed to the few-shot 7B LLM with calibrated threshold, whose decision replaced that of the encoder. The time of the cascade was calculated as the encoder time plus the deferred fraction multiplied by the mean LLM time; it was not measured end to end.

4. Results

4.1. Context length predicts the label in AW4C

In AW4C original, 98.3% of non-actionable warnings had a context of exactly five lines, against 10.5% of actionable warnings, whose context ranged from one line to 5,155 lines (Figure 2a). The shortcut probe, which sees only the number of lines, reached an MCC of 0.91 (95% CI 0.85 to 0.94) on unseen projects. Fine-tuned CodeBERT reached 0.81 (95% CI 0.75 to 0.86) and fine-tuned UniXcoder 0.85 (95% CI 0.80 to 0.90). Neither encoder exceeded the probe. The linear model, whose tokenizer discards line breaks, reached 0.25. The dataset paper states that the context is derived from the changes around the warning in the commit difference [17], which may explain why its form depends on whether the code was later changed.

In AW4C-U, the mean context length was 10.8 lines for non-actionable and 10.9 lines for actionable warnings. On AW4C-U the probe scored 0.12 (95% CI 0.06 to 0.17) and fine-tuned CodeBERT 0.30 (95% CI 0.22 to 0.38) (Figure 2b). The two datasets differ in more than the context window: the cross-project core sets share

only a small number of warnings, and the split was drawn anew. The lower scores on AW4C-U therefore cannot be attributed to context length alone, and the experiment does not show which property of the original context the encoders used. On NASCAR, where context was extracted uniformly from the start, the probe scored -0.02 across projects and 0.10 within projects. The result for AW4C original shows that the form of the context field is sufficient to predict the label.

Uniform extraction reduced the predictive value of length, but it did not make the two classes fully comparable, because the source commit differed between classes (Section 3.2) and the line number of the reporting commit was applied to the child commit without alignment. On the 648 verified warnings of the cross-project core set, MCC was 0.22 for the rule, 0.23 for the linear model, 0.24 for CodeBERT, and 0.40 for the few-shot LLM, against 0.25 , 0.30 , 0.30 , and 0.29 on the complete core set. The LLM values refer to the adapted threshold. The verified subset contains 83% actionable warnings against 54% in the complete core set, and overlap of a line does not prove that the window is centred on the warning, so this check is indicative only. No trained model approached the scores obtained on AW4C original. Results on AW4C-U are treated as secondary evidence, and all further results for C code refer to AW4C-U.

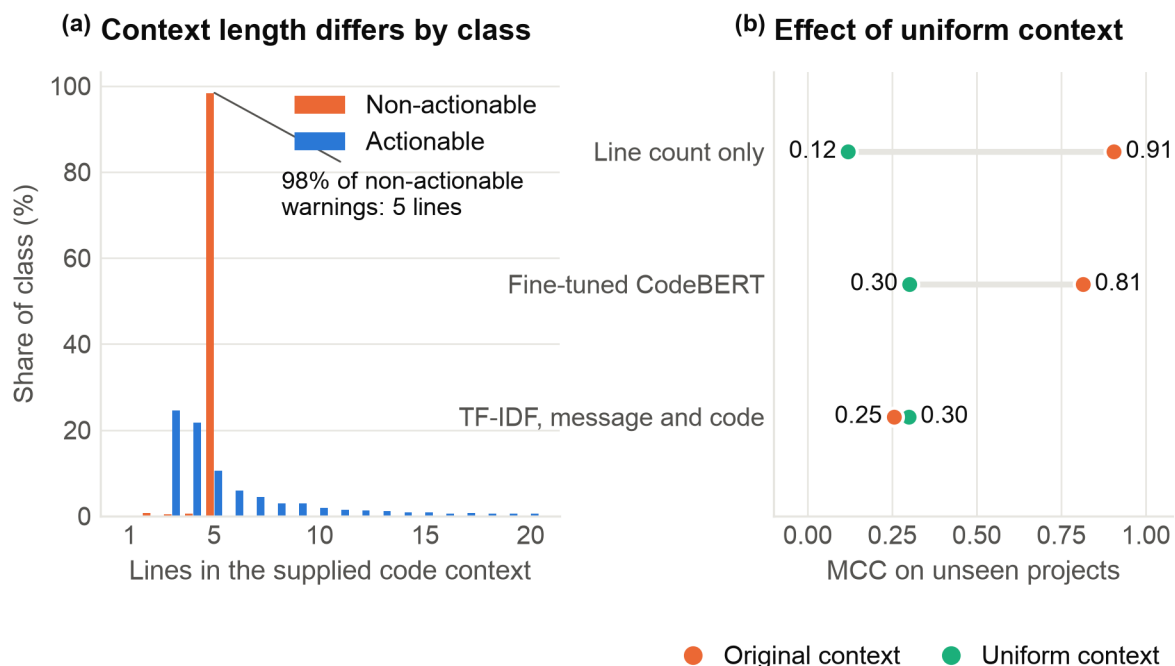


Figure 2. Context length and the label in AW4C. (a) Distribution of the number of context lines by class in the original release; the horizontal axis is truncated at 20 lines. (b) MCC on unseen projects for AW4C original and for AW4C-U.

The two datasets differ in composition and split, so panel (b) is a comparison of two dataset variants and not a controlled before-and-after experiment.

4.2. Quality along the ladder (RQ1)

Table 3 and Figure 3 report all rungs. Under the within-project protocol, quality on NASCAR rose from the rule (0.34) and the message-only linear model (0.37) to the linear model with code (0.46) and to the fine-tuned encoders (0.51 for CodeBERT and 0.58 for UniXcoder). The frozen encoder scored 0.35 , a paired difference from the rule of 0.01 (95% CI -0.11 to 0.14). On AW4C-U, the within-project MCC was 0.17 for the rule, 0.44 for the linear model, 0.30 for the frozen encoder, and 0.40 for fine-tuned CodeBERT.

The locally run LLMs had the lowest scores on NASCAR. With the default threshold, the zero-shot models labelled 98% (7B) and 99% (14B) of warnings actionable, and MCC on unseen projects was -0.04 (95% CI -0.11 to 0.01) and -0.08 (95% CI -0.17 to 0.01); the few-shot 7B model reached 0.09 (95% CI 0.04 to 0.14). The AUC of the LLMs on NASCAR ranged from 0.45 to 0.62 . With the threshold adapted on warnings of the test projects, MCC on NASCAR did not exceed 0.14 . On AW4C-U the 7B model reached 0.19 (95% CI 0.10 to 0.27) zero-shot and 0.16 (95% CI 0.08 to 0.24) few-shot on unseen projects with the default threshold, and 0.29 (95% CI 0.20 to 0.37) few-shot with the adapted threshold. Every LLM answer was a valid class letter.

Table 3. Results on 1,000 test warnings per protocol. Mean time per warning refers to the cross-project protocol and to the mode of operation described in Section 3.7. Intervals describe single models and are not intervals of differences. Rows marked † use an LLM threshold adapted on labelled warnings of the test projects and are not comparable with the other rows as a test of transfer. (a) NASCAR (Java).

Rung	Model	Within: MCC (95% CI)	Within: AUC	Cross: MCC (95% CI)	Cross: AUC	Mean time per warning (ms)
0	Warning-type prior	0.34 (0.23 to 0.44)	0.71	0.34 (0.24 to 0.44)	0.69	0.00006
1	TF-IDF + logistic regression, message only	0.37 (0.26 to 0.47)	0.72	0.36 (0.24 to 0.46)	0.69	0.009
1	TF-IDF + logistic regression, message and code	0.46 (0.34 to 0.54)	0.83	0.25 (0.16 to 0.35)	0.69	0.048
2	Frozen UniXcoder + linear layer	0.35 (0.23 to 0.43)	0.77	0.15 (0.05 to 0.25)	0.65	9
3	Fine-tuned CodeBERT	0.51 (0.38 to 0.59)	0.86	0.26 (0.11 to 0.38)	0.69	8
3	Fine-tuned UniXcoder	0.58 (0.43 to 0.67)	0.89	0.25 (0.12 to 0.38)	0.69	9
4	Qwen2.5-Coder 7B, zero-shot	-0.01 (-0.07 to 0.04)	0.47	-0.04 (-0.11 to 0.01)	0.50	399
4	Qwen2.5-Coder 7B, few-shot	0.00 (-0.06 to 0.06)	0.45	0.09 (0.04 to 0.14)	0.62	425
4	Qwen2.5-Coder 14B, zero-shot	-0.02 (-0.10 to 0.05)	0.51	-0.08 (-0.17 to 0.01)	0.55	837
4†	Qwen2.5-Coder 7B, zero-shot, adapted threshold	0.01 (-0.05 to 0.07)	0.47	0.07 (0.00 to 0.13)	0.50	399
4†	Qwen2.5-Coder 7B, few-shot, adapted threshold	-0.08 (-0.16 to 0.00)	0.45	0.09 (0.01 to 0.16)	0.62	425
4†	Qwen2.5-Coder 14B, zero-shot, adapted threshold	0.07 (0.01 to 0.13)	0.51	0.14 (0.04 to 0.22)	0.55	837

(b) AW4C-U (C, uniform context).

Rung	Model	Within: MCC (95% CI)	Within: AUC	Cross: MCC (95% CI)	Cross: AUC	Mean time per warning (ms)
0	Warning-type prior	0.17 (0.10 to 0.23)	0.64	0.25 (0.16 to 0.35)	0.66	0.00006
1	TF-IDF + logistic regression, message only	0.33 (0.28 to 0.39)	0.73	0.25 (0.17 to 0.32)	0.68	0.009
1	TF-IDF + logistic regression, message and code	0.44 (0.38 to 0.50)	0.81	0.30 (0.22 to 0.38)	0.69	0.044
2	Frozen UniXcoder + linear layer	0.30 (0.24 to 0.36)	0.71	0.17 (0.07 to 0.27)	0.62	10
3	Fine-tuned CodeBERT	0.40 (0.34 to 0.46)	0.77	0.30 (0.22 to 0.38)	0.70	12
4	Qwen2.5-Coder 7B, zero-shot	0.12 (0.05 to 0.19)	0.54	0.19 (0.10 to 0.27)	0.59	343
4	Qwen2.5-Coder 7B, few-shot	0.09 (0.03 to 0.16)	0.58	0.16 (0.08 to 0.24)	0.64	464
4†	Qwen2.5-Coder 7B, zero-shot, adapted threshold	0.09 (0.03 to 0.15)	0.54	0.18 (0.10 to 0.26)	0.59	343
4†	Qwen2.5-Coder 7B, few-shot, adapted threshold	0.10 (0.04 to 0.16)	0.58	0.29 (0.20 to 0.37)	0.64	464

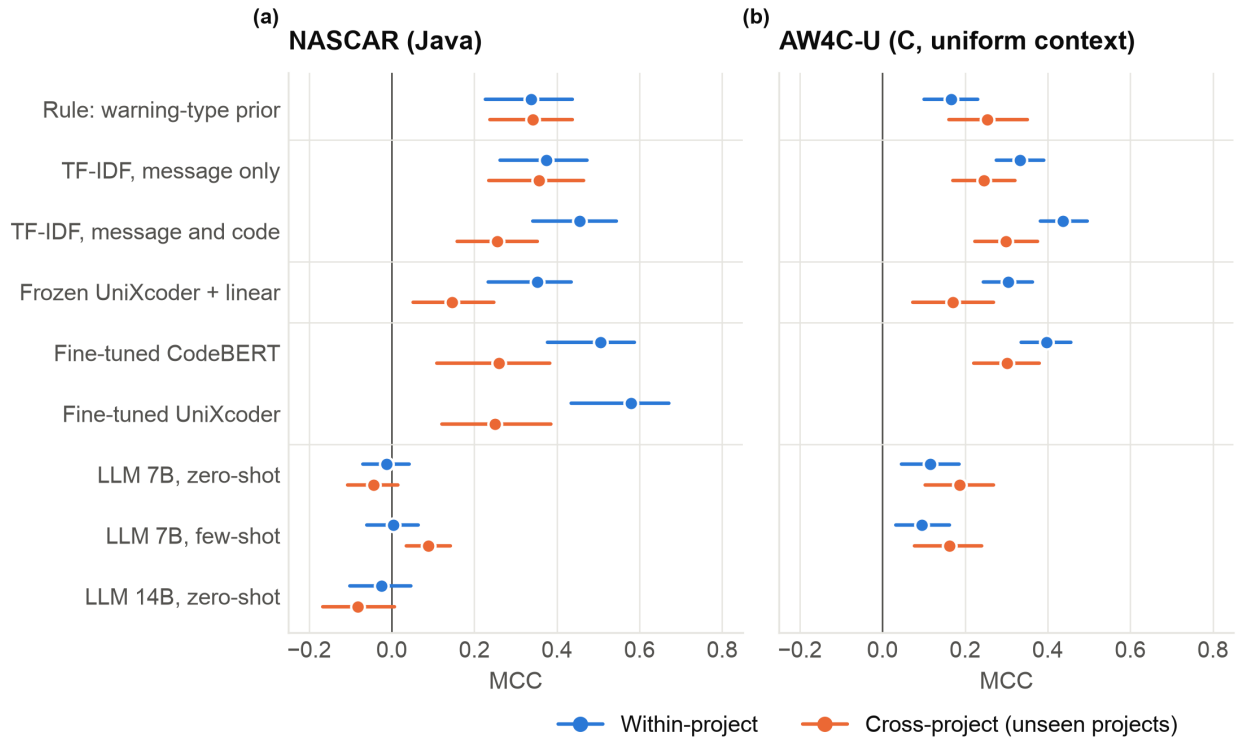


Figure 3. MCC along the model ladder under two protocols. Points are estimates on 1,000 test warnings; bars are 95% bootstrap intervals over projects for single models, not intervals of differences. LLMs use the default threshold of 0.5.

(a) NASCAR. (b) AW4C-U. A model absent from a panel was not run on that dataset.

4.3. Loss of quality on unseen projects (RQ2)

On NASCAR, the models that read code scored lower on unseen projects than within projects. The difference between the two protocols was 0.20 for the linear model with code, 0.21 for the frozen encoder, 0.25 for CodeBERT, and 0.33 for UniXcoder. The rule scored 0.34 within projects and 0.34 across projects, and the

message-only linear model 0.37 and 0.36. The two protocols were evaluated on different warnings, so these differences describe the protocols and do not isolate the effect of removing projects from training.

Table 4 reports paired differences from the rule on unseen projects. On NASCAR the difference was -0.08 (95% CI -0.21 to 0.02) for CodeBERT, -0.09 (95% CI -0.22 to 0.01) for UniXcoder, -0.09 (95% CI -0.17 to 0.01) for the linear model with code, and 0.02 (95% CI -0.07 to 0.09) for the message-only linear model. An advantage of the encoders over the rule was therefore not established in this experiment. The result does not demonstrate equivalence of the models or superiority of the rule. Within NASCAR projects, the paired difference from the rule was 0.17 (95% CI 0.05 to 0.28) for CodeBERT and 0.24 (95% CI 0.11 to 0.36) for UniXcoder.

On AW4C-U, the difference between protocols was 0.14 for the linear model with code, 0.13 for the frozen encoder, and 0.10 for CodeBERT. On unseen projects the paired difference from the rule was 0.05 (95% CI -0.07 to 0.17) for CodeBERT and 0.05 (95% CI -0.04 to 0.12) for the linear model with code.

Table 4. Paired differences in MCC on unseen projects. Each model is compared with the warning-type prior on the same 1,000 warnings, with 2,000 bootstrap draws of projects applied to both models. LLMs use the default threshold. Cascade rows compare the cascade with fine-tuned CodeBERT alone; the deferred warnings are decided by the few-shot 7B LLM with adapted threshold.

Model	NASCAR: difference from the prior (95% CI)	AW4C-U: difference from the prior (95% CI)
TF-IDF + logistic regression, message only	0.02 (-0.07 to 0.09)	-0.01 (-0.06 to 0.05)
TF-IDF + logistic regression, message and code	-0.09 (-0.17 to 0.01)	0.05 (-0.04 to 0.12)
Frozen UniXcoder + linear layer	-0.19 (-0.31 to -0.09)	-0.08 (-0.20 to 0.03)
Fine-tuned CodeBERT	-0.08 (-0.21 to 0.02)	0.05 (-0.07 to 0.17)
Fine-tuned UniXcoder	-0.09 (-0.22 to 0.01)	not run
Qwen2.5-Coder 7B, zero-shot	-0.38 (-0.50 to -0.27)	-0.07 (-0.17 to 0.05)
Qwen2.5-Coder 7B, few-shot	-0.25 (-0.37 to -0.15)	-0.09 (-0.22 to 0.04)
Qwen2.5-Coder 14B, zero-shot	-0.42 (-0.54 to -0.29)	not run
Cascade, 10% deferred, against CodeBERT alone	0.03 (-0.02 to 0.07)	0.01 (-0.02 to 0.04)
Cascade, 30% deferred, against CodeBERT alone	-0.10 (-0.22 to 0.04)	0.05 (-0.01 to 0.11)
Cascade, 50% deferred, against CodeBERT alone	-0.14 (-0.25 to -0.00)	0.04 (-0.02 to 0.11)

4.4. Compute time (RQ3)

Mean processing time per warning spanned about seven orders of magnitude (Figure 4a). On NASCAR the rule needed 0.00006 ms, the message-only linear model 0.009 ms, the linear model with code 0.048 ms, fine-tuned CodeBERT 8.4 ms, the zero-shot 7B LLM 399 ms, the few-shot 7B LLM 425 ms, and the 14B LLM 837 ms. CodeBERT needed about 177 times the time of the linear model with code. Relative to CodeBERT, the LLMs needed about 47 times (7B zero-shot), 51 times (7B few-shot), and 100 times (14B) the time. Fine-tuning an encoder took between 18 and 27 minutes across runs; one further run, interrupted by system sleep, was excluded from this range. With the NASCAR measurements, the total compute time of fine-tuned CodeBERT and of the zero-shot 7B LLM was equal at 2,768 warnings (Figure 4b), and with the longest training time at about 4,113. This break-even point compares two systems of different quality and covers compute time only.

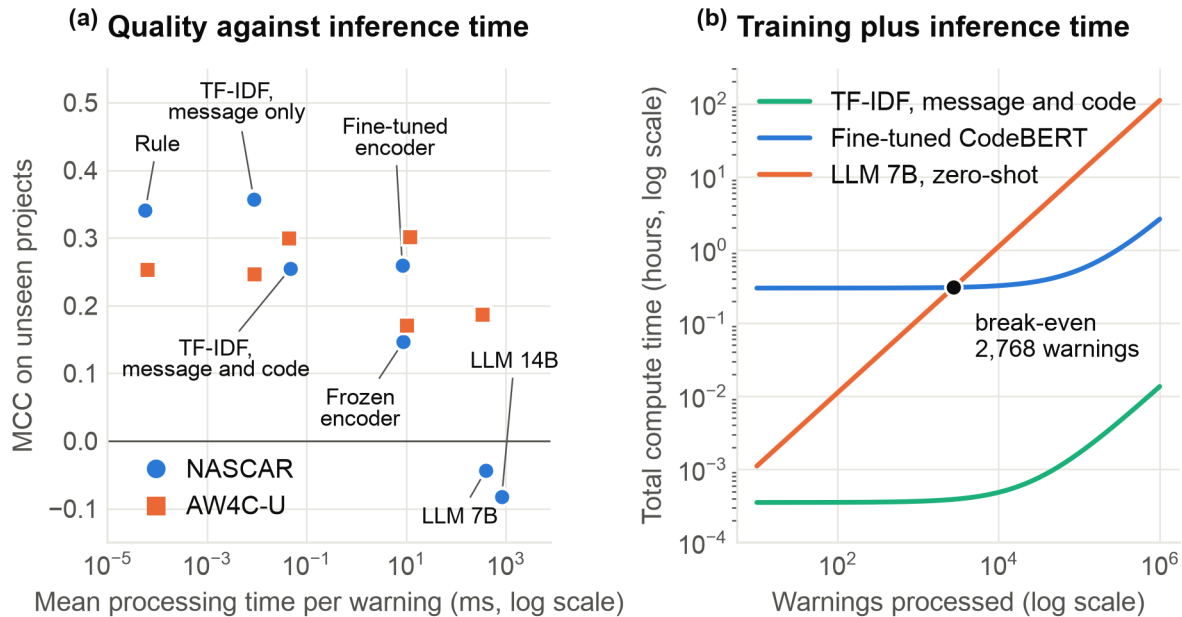


Figure 4. Quality and compute time. (a) MCC on unseen projects against mean processing time per warning on one computer. Encoders were timed in batches and LLMs one warning at a time. Shown are the rule, both linear models, the frozen encoder, fine-tuned CodeBERT, and the zero-shot LLMs with the default threshold; fine-tuned UniXcoder and the few-shot LLM are omitted, so the panel is not a complete frontier. Labels refer to NASCAR points. (b) Total compute time in hours, training plus processing, for the linear model with code, fine-tuned CodeBERT, and the zero-shot 7B LLM, from NASCAR measurements. The marked point is the equality of compute time under these conditions.

4.5. Cascade (RQ4)

A gain from passing the least confident warnings to the LLM was not established (Table 4). On NASCAR under the cross-project protocol, MCC was 0.26 with no deferral, 0.29 with 10% deferred, 0.15 with 30%, and 0.12 with 50%; the paired difference from the encoder alone was 0.03 (95% CI -0.02 to 0.07) at 10% and -0.14 (95% CI -0.25 to -0.00) at 50%. On AW4C-U the corresponding MCC values were 0.30, 0.31, 0.35, and 0.35, with a paired difference of 0.05 (95% CI -0.01 to 0.11) at 30%. Under the within-project protocol, the paired difference at 30% deferral was -0.22 (95% CI -0.29 to -0.14) on NASCAR and 0.01 (95% CI -0.04 to 0.05) on AW4C-U. Each deferred tenth of warnings added a calculated 43 ms on NASCAR and 46 ms on AW4C-U to the mean time per warning.

4.6. Sensitivity to the amount of training data

The main comparison fixed the training budget at 16,000 warnings. When the linear model was trained on the complete training part, its within-project MCC on NASCAR rose from 0.46 to 0.70, above both fine-tuned encoders at the fixed budget. On unseen projects the additional data, which came from other projects, did not help: MCC was 0.20 against 0.25 at the fixed budget. On AW4C-U the corresponding values were 0.50 against 0.44 within projects and 0.28 against 0.30 across projects. Training the linear model on the complete NASCAR training part (Table 1) took less than one minute. More training data therefore improved within-project quality and did not improve cross-project quality. The encoders were not trained on the complete training part, so this analysis does not compare the rungs at a larger budget.

5. Discussion

5.1. Principal findings

The study supports three conclusions. The first is methodological. A public dataset for actionable warning identification contains a field whose form predicts the label. A score of 0.81 on unseen projects would have looked like strong generalization, yet a single count of lines scored 0.91. The experiment does not prove that the encoders relied on length and not on another property of the context. This is an instance of shortcut learning [38]. It extends earlier reports of leakage in warning datasets [2] and of label and duplication problems in vulnerability datasets [13, 40, 41] to a dataset published after those reports. The practical lesson is to run a probe on the form of every input field before training a model on its content.

The second conclusion concerns proportionality between model and task. Within projects of NASCAR, fine-tuned encoders scored higher than the rule. On unseen NASCAR projects, an advantage of any model that read code over the rule was not established. One possible explanation is that the models learned conventions of particular projects, such as which warnings a team tends to fix; the study did not test this explanation. On AW4C-U the differences from the rule on unseen projects were small and their intervals included zero. For an organization that introduces a filter in a project without labelled history, the warning-type prior is a defensible first reference, and a more complex model should be preferred after it shows an advantage on held-out projects.

The third conclusion concerns LLMs. Without task-specific training and with the default threshold, open-weight models with 7 and 14 billion parameters scored near zero on NASCAR; adaptation of the threshold on labelled warnings of the test projects raised MCC to at most 0.14. PMD raised 99.6% of the warnings in the deduplicated release by our count, and PMD rules concern mainly code quality. Whether developers act on such a warning may depend on team policy more than on the code. The result is consistent with the low score of a commercial LLM reported on the same dataset [11]. On AW4C-U the LLM scores were closer to those of the trained models. The findings apply to the evaluated models, prompt, and datasets and should not be generalized to all LLMs or to all static analysis findings.

The sensitivity analysis adds a qualification. With enough labelled history of the same projects, the cheapest learned model reached the highest within-project score in the study. Because the encoders were trained at the smaller budget only, the study does not show whether they would gain as much from the same data. This observation is compatible with earlier high scores obtained when training and test data came from consecutive releases of the same projects [3], although those scores were later attributed largely to leakage and duplication [2].

5.2. Implications for practice

The measurements suggest an order of evaluation. Start with the warning-type prior, which needs labelled warnings and negligible compute. Add a linear model when labelled history of the same project exists. Consider a fine-tuned encoder when cross-project validation shows a gain and the volume is large enough for its training time to be recovered. Verify the benefit of an LLM on labelled data of the target setting before use. The study predicts observed developer behaviour; it does not establish that warnings predicted as non-actionable can be suppressed safely.

5.3. Limitations

The actionable label records that a warning disappeared, not that it was a true defect [15]. Conclusions about false positives of security analyzers cannot be drawn from these data. NASCAR consists almost entirely of

PMD warnings. Fine-tuning used 16,000 training warnings, two epochs, and one seed; larger budgets may raise encoder scores, and seed variance was not measured. The intervals reflect sampling of projects and test items only. The cross-project protocol used one assignment of projects, with 16 NASCAR test projects. Commercial LLMs, larger open models, fine-tuned LLMs, and agents with program analysis tools were outside the scope, so the results describe locally run models of moderate size with a plain prompt. Timing was recorded on a consumer computer that was not dedicated to the experiment, the rungs were timed in passes of different size and batching, and training time of the same model varied between runs, so the break-even volume is approximate. The intervals for the adapted LLM threshold do not include the uncertainty of threshold selection. Under the within-project protocol, the code window of 2.1% of NASCAR core warnings and 10.5% of AW4C-U core warnings also occurred in the 16,000-warning training sample with a different warning. The amount of development data used for threshold selection differed between rungs (Table 2). Context for 6.8% of non-actionable AW4C warnings could not be retrieved, which may bias AW4C-U if missing files differ systematically. On AW4C-U the probe remained slightly above zero, because windows near the start or end of a file are shorter. The source commit of the code differed between classes in AW4C-U, and about a fifth of non-actionable windows did not overlap the originally distributed context; a model could in principle exploit such differences, although the scores on the verified subset gave no sign of it. Pre-training data of the models may include the evaluated projects.

5.4. Future work

Three extensions follow directly: repeated cross-project assignments with several seeds, fine-tuning of compact LLMs with parameter-efficient methods, and evaluation on warnings with verified true-positive labels.

6. Conclusions

On the NASCAR dataset, fine-tuned code encoders scored higher than a warning-type prior within projects, and an advantage over the prior was not established on unseen projects. Locally run LLMs without task-specific training scored near zero on NASCAR with the default threshold, at about 47 times the mean processing time of a fine-tuned encoder for the 7B zero-shot model. In the original AW4C release, the length of the supplied code context predicts the label, which can inflate the scores of models that read it; a variant with uniformly extracted context gave much lower scores, although the two datasets differ in more than context length. Comparisons of triage models should include a simple label-based reference, cross-project evaluation with paired differences, a probe of the form of input fields, and compute time measured in a stated mode on the same hardware.

Declarations

Acknowledgments. The authors thank the creators of the NASCAR and AW4C datasets for making their data publicly available.

Author contributions. P.S.: Conceptualization, Methodology, Software, Validation, Formal Analysis, Investigation, Data Curation, Writing (Original Draft), Writing (Review & Editing), Visualization. B.M.: Conceptualization, Validation, Writing (Review & Editing), Supervision. All authors have read and agreed to the published version of the manuscript.

Funding. This research received no external funding.

Data availability. The source datasets are publicly available: NASCAR at <https://doi.org/10.5281/zenodo.17079912> and AW4C at <https://doi.org/10.5281/zenodo.15016587>, both under the CC BY 4.0 license. The procedures for sampling, context extraction, data splitting, model training, and evaluation are described in Section 3. The derived data, including the corrected AW4C-U set, the identifiers of the evaluated warnings, and the predictions of all models, together with random seeds, software versions, and the identifiers of the few-shot examples, are available from the corresponding author on reasonable request. The analysis code has not been publicly released.

Conflicts of interest. The authors declare no competing interests related to this study.

References

- [1] Xiuting Ge, Chunrong Fang, Xuanye Li, Weisong Sun, Daoyuan Wu, Juan Zhai, et al. Machine Learning for Actionable Warning Identification: A Comprehensive Survey. *ACM Computing Surveys*, 57(2), 1-35, 2024. [doi:10.1145/3696352](https://doi.org/10.1145/3696352).
- [2] Hong Jin Kang, Khai Loong Aw, David Lo. Detecting false alarms from automatic static analysis tools: how far are we? *Proceedings of the 44th International Conference on Software Engineering*, 698-709, 2022. [doi:10.1145/3510003.3510214](https://doi.org/10.1145/3510003.3510214).
- [3] Xueqi Yang, Jianfeng Chen, Rahul Yedida, Zhe Yu, Tim Menzies. Learning to recognize actionable static code warnings (is intrinsically easy). *Empirical Software Engineering*, 26(3), 56, 2021. [doi:10.1007/s10664-021-09948-6](https://doi.org/10.1007/s10664-021-09948-6).
- [4] Rahul Yedida, Hong Jin Kang, Huy Tu, Xueqi Yang, David Lo, Tim Menzies. How to Find Actionable Static Analysis Warnings: A Case Study With FindBugs. *IEEE Transactions on Software Engineering*, 49(4), 2856-2872, 2023. [doi:10.1109/tse.2023.3234206](https://doi.org/10.1109/tse.2023.3234206).
- [5] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin Clement, et al. Learning to reduce false positives in analytic bug detectors. *Proceedings of the 44th International Conference on Software Engineering*, 1307-1316, 2022. [doi:10.1145/3510003.3510153](https://doi.org/10.1145/3510003.3510153).
- [6] Xiuting Ge, Chunrong Fang, Qunjun Zhang, Daoyuan Wu, Bowen Yu, Qirui Zheng, et al. Pre-trained Model-based Actionable Warning Identification: A Feasibility Study. *ACM Transactions on Software Engineering and Methodology*, 35(8), 1-29, 2026. [doi:10.1145/3777369](https://doi.org/10.1145/3777369).
- [7] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, et al. Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We? *ACM Transactions on Knowledge Discovery from Data*, 18(7), 1-34, 2024. [doi:10.1145/3653718](https://doi.org/10.1145/3653718).

- [8] Jinbao Chen, Hongjing Xiang, Zuohong Zhao, Luhao Li, Yu Zhang, Boyao Ding, et al. Utilizing Precise and Complete Code Context to Guide LLM in Automatic False Positive Mitigation. arXiv preprint arXiv:2411.03079, 2024. doi:10.48550/arXiv.2411.03079.
- [9] Xueying Du, Jiayi Feng, Yi Zou, Wei Xu, Jie Ma, Wei Zhang, et al. Reducing False Positives in Static Bug Detection with LLMs: An Empirical Study in Industry. Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice, 716-727, 2026. doi:10.1145/3786583.3786910.
- [10] Yunpeng Xiong, Ting Zhang. Sifting the Noise: A Comparative Study of LLM Agents in Vulnerability False Positive Filtering. arXiv preprint arXiv:2601.22952, 2026. doi:10.48550/arXiv.2601.22952.
- [11] Tamás Aladics, Norbert Vándor, Rudolf Ferenc, Péter Hegedűs. Towards Better Static Code Analysis Reports: Sentence Transformer-based Filtering of Non-Actionable Alerts. arXiv preprint arXiv:2604.18525, 2026. doi:10.48550/arXiv.2604.18525.
- [12] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, Baishakhi Ray. Deep Learning Based Vulnerability Detection: Are We There Yet? IEEE Transactions on Software Engineering, 48(9), 3280-3296, 2022. doi:10.1109/tse.2021.3087402.
- [13] Roland Croft, M. Ali Babar, M. Mehdi Kholoosi. Data Quality for Software Vulnerability Datasets. 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 121-133, 2023. doi:10.1109/icse48619.2023.00022.
- [14] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alo-mair, et al. Vulnerability Detection with Code Language Models: How Far are We? 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), 1729-1741, 2025. doi:10.1109/icse55347.2025.00038.
- [15] Zhipeng Xue, Zhipeng Gao, Tongtong Xu, Xing Hu, Xin Xia, Shanping Li. Actionable Warning Is Not Enough: Recommending Valid Actionable Warnings with Weak Supervision. Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering, 2198-2210, 2026. doi:10.1145/3744916.3773133.
- [16] Dávid Kószó, Tamás Aladics, Rudolf Ferenc, Péter Hegedűs. A Large-Scale Collection Of (Non-)Actionable Static Code Analysis Reports. Scientific Data, 12(1), 1884, 2025. doi:10.1038/s41597-025-06154-7.
- [17] Zhipeng Liu, Meng Yan, Zhipeng Gao, Dong Li, Xiaohong Zhang, Dan Yang. AW4C: A Commit-Aware C Dataset for Actionable Warning Identification. Proceedings of the 21st International Conference on Mining Software Repositories, 133-137, 2024. doi:10.1145/3643991.3644885.
- [18] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, et al. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 111-120, 2021. doi:10.1109/icse-seip52600.2021.00020.
- [19] Jake Feiglin, Guy Dar. SastBench: A Benchmark for Testing Agentic SAST Triage. arXiv preprint arXiv:2601.02941, 2026. doi:10.48550/arXiv.2601.02941.
- [20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. Findings of the Association for Computational Linguistics: EMNLP 2020, 1536-1547, 2020. doi:10.18653/v1/2020.findings-emnlp.139.

- [21] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. arXiv preprint arXiv:2009.08366, 2020. doi:10.48550/arXiv.2009.08366.
- [22] Yue Wang, Weishi Wang, Shafiq Joty, Steven C. H. Hoi. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 8696-8708, 2021. doi:10.18653/v1/2021.emnlp-main.685.
- [23] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, Jian Yin. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 7212-7225, 2022. doi:10.18653/v1/2022.acl-long.499.
- [24] Haonan Li, Yu Hao, Yizhuo Zhai, Zhiyun Qian. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. Proceedings of the ACM on Programming Languages, 8(OOPSLA1), 474-499, 2024. doi:10.1145/3649828.
- [25] Ziyang Li, Saikat Dutta, Mayur Naik. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. arXiv preprint arXiv:2405.17238, 2024. doi:10.48550/arXiv.2405.17238.
- [26] Mohsen Iranmanesh, Sina Moradi Sabet, Sina Marefat, Ali Javidi Ghasr, Allison Wilson, Iman Sharafaldin, et al. ZeroFalse: Improving Precision in Static Analysis with LLMs. arXiv preprint arXiv:2510.02534, 2025. doi:10.48550/arXiv.2510.02534.
- [27] William Klieber, David Svoboda, Lori Flynn, Ruben Martins. Using LLMs to Adjudicate Static-Analysis Alerts with Error Reduction Techniques. arXiv preprint arXiv:2607.09979, 2026. doi:10.48550/arXiv.2607.09979.
- [28] Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, Trung Le. ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We? 2023 30th Asia-Pacific Software Engineering Conference (APSEC), 632-636, 2023. doi:10.1109/apsec60848.2023.00085.
- [29] Fouad Trad, Ali Chehab. Retrieval-Augmented Few-Shot Prompting Versus Fine-Tuning for Code Vulnerability Detection. 2025 3rd International Conference on Foundation and Large Language Models (FLLM), 615-622, 2025. doi:10.1109/fllm67465.2025.11391248.
- [30] Ting Zhang, Chengran Yang, Yindu Su, Martin Weyssow, Hung Nguyen, Tan Bui, et al. Benchmarking Large Language Models for Multi-Language Software Vulnerability Detection. arXiv preprint arXiv:2503.01449, 2025. doi:10.48550/arXiv.2503.01449.
- [31] Chaomeng Lu, Bert Lagaisse. From Lab to Reality: A Practical Evaluation of Deep Learning Models and LLMs for Vulnerability Detection. arXiv preprint arXiv:2512.10485, 2025. doi:10.48550/arXiv.2512.10485.
- [32] Jun Li, Lixian Li, Jin Liu, Xiao Yu, Xiao Liu, Jacky Wai Keung. Large language model ChatGPT versus small deep learning models for self-admitted technical debt detection: Why not together? Software: Practice and Experience, 55(1), 3-28, 2025. doi:10.1002/spe.3360.
- [33] Lingjiao Chen, Matei Zaharia, James Zou. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. arXiv preprint arXiv:2305.05176, 2023. doi:10.48550/arXiv.2305.05176.

- [34] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, et al. RouteLLM: Learning to Route LLMs with Preference Data. arXiv preprint arXiv:2406.18665, 2024. doi:10.48550/arXiv.2406.18665.
- [35] Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Ruhle, et al. Hybrid LLM: Cost-Efficient and Quality-Aware Query Routing. arXiv preprint arXiv:2404.14618, 2024. doi:10.48550/arXiv.2404.14618.
- [36] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, et al. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12(85), 2825-2830, 2011. Available: <https://jmlr.org/papers/v12/pedregosa11a.html>. Accessed: September 28, 2026.
- [37] Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, et al. Qwen2.5-Coder Technical Report. arXiv preprint arXiv:2409.12186, 2024. doi:10.48550/arXiv.2409.12186.
- [38] Robert Geirhos, Jörn-Henrik Jacobsen, Claudio Michaelis, Richard Zemel, Wieland Brendel, Matthias Bethge, et al. Shortcut learning in deep neural networks. Nature Machine Intelligence, 2(11), 665-673, 2020. doi:10.1038/s42256-020-00257-z.
- [39] Davide Chicco, Giuseppe Jurman. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. BMC Genomics, 21(1), 6, 2020. doi:10.1186/s12864-019-6413-7.
- [40] Xu Nie, Ningke Li, Kailong Wang, Shangguang Wang, Xiapu Luo, Haoyu Wang. Understanding and Tackling Label Errors in Deep Learning-Based Vulnerability Detection (Experience Paper). Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 52-63, 2023. doi:10.1145/3597926.3598037.
- [41] Yuejun Guo, Seifeddine Bettaieb. An Investigation of Quality Issues in Vulnerability Detection Datasets. 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), 29-33, 2023. doi:10.1109/eurospw59978.2023.00008.