False-Positive Bug Reports in Deep Learning Compilers: Stages, Root Causes, and Mitigation

LILI HUANG, Tianjin University, China
QINGCHAO SHEN, Tianjin University, China
DONG WANG*, Tianjin University, China
YUNPING WU, Tianjin University, China
MENG WANG, University of Bristol, UK
JUNJIE CHEN*, Tianjin University, China

Deep learning (DL) compilers are the essential infrastructure to optimize DL models for efficient execution across heterogeneous hardware. Like traditional compilers, they are also bug-prone. However, not all bug reports submitted to DL compiler repositories reflect genuine bugs. Many are false-positive bug reports caused by incorrect configurations or user misunderstandings. These reports can mislead developers, waste debugging resources, and delay critical bug fixes. This paper presents the first comprehensive study of false-positive bug reports in DL compilers, analyzing 1,075 closed issues and discussions from two representative systems: TVM and OpenVINO. We find that false-positive bug reports demand substantial developer effort, occur throughout the compiler workflow, especially during the build and import and IR transformation stages, and frequently result from incorrect environment configuration, incorrect usage, or misunderstanding of compiler features or limitations. To address this challenge, we further investigate the potential of large language models (LLMs) to automatically mitigate false-positive bug reports. Through extensive experiments, we find that few-shot prompting achieves promising performance, with strong accuracy and explanation quality. Our study sheds light on an overlooked yet important category of compiler issues and demonstrates the potential of LLMs in supporting more efficient bug report triage in DL compilers.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**; **Software defect analysis**.

Additional Key Words and Phrases: DL compilers, DL bugs, Bug report analysis, Empirical study

ACM Reference Format:
Lili Huang, Qingchao Shen, Dong Wang, Yunping Wu, Meng Wang, and Junjie Chen. 2018. False-Positive Bug Reports in Deep Learning Compilers: Stages, Root Causes, and Mitigation. *J. ACM* 37, 4, Article 111 (August 2018), 32 pages. https://doi.org/XXXXXXX.XXXXXXX

*Corresponding Author.

Authors' addresses: Lili Huang, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, huangll@tju.edu.cn; Qingchao Shen, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, qingchao@tju.edu.cn; Dong Wang, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, dong_w@tju.edu.cn; Yunping Wu, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, yunping_wu@tju.edu.cn; Meng Wang, Department of Computer Science, University of Bristol, Bristol, UK, BS8 1UB, meng.wang@bristol.ac.uk; Junjie Chen, College of Intelligence and Computing, Tianjin University, Tianjin, China, 300350, junjiechen@tju.edu.cn.

0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Deep learning (DL) compilers play a critical role in bridging high-level DL models and low-level hardware execution. They transform DL models, constructed from various DL frameworks (e.g., TensorFlow [11] and PyTorch [9]) into optimized and executable code for deployment on diverse hardware platforms (e.g., NVIDIA GPUs). Modern DL compilers, such as TVM [16], OpenVINO [26], and XLA [1], are extremely complex, involving sophisticated optimizations, hardware-specific code generation, and intricate interactions with DL frameworks and runtimes. Due to their complexity, bugs in DL compilers can stem from a wide range of root causes [55], making them particularly time-consuming and challenging to diagnose.

The challenge of diagnosing DL compiler bugs is further exacerbated by the high incidence of false-positive bug reports, which do not stem from actual compiler bugs but rather from incorrect usage or misunderstandings by users. Such reports often arise because DL compilers operate in a rapidly evolving ecosystem, involving tightly coupled frameworks, intricate APIs, and heterogeneous execution environments, all of which complicate root cause analysis. For instance, a user reported a compilation failure in TVM's Hexagon runtime[1], encountering multiple "invalid instruction" errors. Although the issue initially appeared to stem from incorrect inline assembly, it was ultimately traced to the user's misconfiguration: the architecture flag `-DUSE_HEXAGON_ARCH=v65` specified an older version not supported by the installed Hexagon SDK/toolchain, which required a newer version (e.g., `v68`). Updating the flag resolved the issue, confirming that it was caused by incorrect user-side configuration rather than a compiler bug. Moreover, prior studies have shown that even experienced developers struggle to reliably distinguish genuine compiler bugs from user-side errors [27, 39, 41]. These false-positive bug reports not only consume developers' effort but also hinder the timely identification and resolution of genuine compiler bugs.

Existing studies [13, 41, 60] have investigated the invalid bug reports in traditional software (e.g., Eclipse and Firefox), which broadly include reports that are false-positive, non-reproducible, or contain insufficient information. Prior work has typically attributed invalid bug reports to causes such as testing errors, misunderstandings of software functionality, or faults originating from external libraries. However, such taxonomies fall short in capturing the unique characteristics of false-positive bug reports specific to DL compilers. DL compilers differ from traditional software in several critical ways: (1) *Framework and Hardware Dependencies*: They rely heavily on diverse and rapidly evolving DL frameworks (e.g., TensorFlow and PyTorch) and heterogeneous hardware platforms (e.g., CPU, GPU, and NPU). This integration introduces complex compatibility challenges, such as unsupported operations, version mismatches, or backend-specific constraints, that users often misinterpret as compiler bugs. (2) *Opaque Internal Transformations*: DL compilers often employ sophisticated internal mechanisms for graph-level optimizations and hardware-specific code generation. These mechanisms are often opaque to users, leading them to misinterpret expected behaviors as bugs and contributing to a higher incidence of false-positive bug reports.

To mitigate the problem of false-positive bug reports, several automatic invalid bug reports identification techniques were proposed, which employ diverse machine learning or DL algorithms to build binary or multi-class classifiers [23, 27, 40, 68]. While these techniques can effectively filter some invalid bug reports, they are primarily trained on traditional software bug reports and target different categories of invalidity. As such, they are not directly applicable to our scenario, which involves distinct failure patterns and debugging practices. Moreover, they often lack explanatory power, an essential capability for diagnosing false-positive bug reports in DL compilers, which often stem from subtle configuration errors or misunderstanding of semantics. As highlighted by Laiq et al. [40], practitioners not only seek automated detection but also emphasize the need

[1]https://github.com/apache/tvm/issues/14407

for explainability to support practical debugging and reduce triage overhead. These limitations underscore the need for a comprehensive understanding of false-positive bug reports and the development of an effective, explainable approach to mitigate them in DL compilers.

To address the aforementioned challenges, we conducted an empirical study of 448 false-positive bug reports from two widely used open-source DL compilers, Apache TVM and Intel OpenVINO, to comprehensively understand their effects and characteristics. Furthermore, we propose an LLM-based approach to effectively mitigate false-positive bug reports in DL compilers. We formulate the following four research questions to guide our work:

- **RQ1: To what extent do false-positive bug reports in DL compilers consume developer time and effort?** It remains unclear whether developers expend substantial effort on false-positive bug reports. This uncertainty is the fundamental motivation of our study, as it provides the empirical foundation for our subsequent questions, which examine how to reduce these costs and provide practical guidance for developers. Accordingly, RQ1 measures the extent of developer effort devoted to such reports in DL compilers. By quantifying discussion length and response time, we surface the hidden costs of false-positive bug reports and test whether they are comparable to that of resolving genuine bugs.
  Results: We find that false-positive bug reports require substantial developer effort comparable to that of genuine bugs in terms of the discussion process and the resolution lifecycle. These results further strengthen our motivation to minimize the effort involved in mitigating false-positive bug reports, thereby improving the efficiency of DL compiler maintenance.
- **RQ2: How are false-positive bug reports distributed across different DL compiler stages?** DL compilers typically follow a multi-stage workflow, generally including the following six stages: build and import, model loading, IR transformation, user-customized optimization, inference execution, and performance evaluation. Each stage involves different components, dependencies, and potential failure points. Understanding how false-positive bug reports are distributed across these stages is essential for identifying which parts of the compiler are more prone to user confusion or misconfiguration.
  Results: We observe that false-positive bug reports occur across all six stages of the DL compiler workflow. The Build and Import and IR Transformation stages contribute to the highest proportions (23.88% and 23.66%), together accounting for nearly half of all false-positive bug reports, suggesting a need for improved documentation, stage-specific diagnostics, and automated tool support to assist users in correctly configuring and using DL compilers.
- **RQ3: What are the root causes of the false-positive bug reports in DL compilers?** This RQ aims to systematically identify and categorize the underlying causes of false-positive bug reports, such as incorrect environment configuration, misunderstanding of intended design, or limitations. Understanding the root causes is fundamental for both researchers and practitioners to develop effective mitigation strategies that reduce wasted engineering effort and improve trust in automated validation pipelines.
  Results: The majority of false-positive bug reports in DL compilers originate from three primary root causes: Incorrect Environment Configuration (34.38%), Incorrect Usage (29.69%), and Misunderstanding of Features or Limitations (22.77%). Furthermore, our stage-wise correlation analysis reveals that each stage of the DL compiler workflow is typically associated with specific dominant root causes.
- **RQ4: Can LLMs help mitigate false-positive bug reports in DL compilers?** Given the demonstrated potential of LLMs in code understanding and automated debugging, this RQ explores whether LLMs can effectively assist in identifying and interpreting false-positive bug reports. Specifically, building on insights from root cause analysis, we investigate the

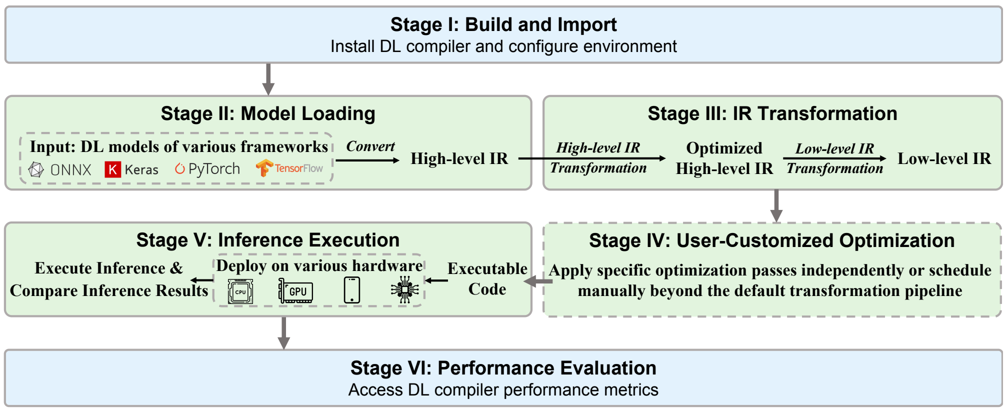

Stage I: Build and Import
Install DL compiler and configure environment
Stage II: Model Loading
Input: DL models of various frameworks
ONNX
Keras
PyTorch
TensorFlow
Convert
High-level IR
Stage III: IR Transformation
High-level IR Transformation
Optimized High-level IR
Low-level IR Transformation
Low-level IR
Stage IV: User-Customized Optimization
Apply specific optimization passes independently or schedule manually beyond the default transformation pipeline
Stage V: Inference Execution
Execute Inference & Compare Inference Results
Deploy on various hardware
CPU
GPU
Executable Code
Stage VI: Performance Evaluation
Access DL compiler performance metrics

Fig. 1. User Workflow over DL Compiler Architecture

extent to which LLMs can accurately classify bug reports as either genuine bugs or false-positive bug reports and generate meaningful interpretations by leveraging various prompt engineering strategies, including zero-shot learning, few-shot learning, chain-of-thought (CoT) prompting, and retrieval-augmented generation (RAG).
Results: LLMs, particularly under few-shot prompting, demonstrate strong effectiveness in identifying false-positive bug reports in DL compilers, with a precision of 86%. In terms of coverage, the model correctly identifies 42% of false-positive bug reports, indicating its potential to partially mitigate this issue in practice. Moreover, the generated rationales for correctly classified cases are accurate in 85.2% of instances, reinforcing the interpretable capability of LLMs.

This work makes the following major contributions: (1) We present the first comprehensive study of false-positive bug reports in well-established DL compilers (TVM and OpenVINO) and develop a taxonomy of root causes underlying false-positive bug reports. (2) We conduct an extensive evaluation of four LLM-based strategies (i.e., prompt-based zero-shot, few-shot, CoT, and RAG) for identifying false-positive bug reports in DL compilers. Our findings demonstrate promising performance and potential applicability of LLMs in automating false-positive bug report mitigation. (3) Based on our findings, we provide a set of actionable recommendations for both practitioners and researchers to improve DL compiler maintenance and to design more effective automated approaches for handling false-positive bug reports. (4) We release a novel dataset of 448 false-positive bug reports from the two studied compilers, each manually annotated with the root cause and the specific compiler workflow stage in which the false-positive error occurred. The dataset is publicly available [2] to support future research.

Paper Organization. The remainder of this paper is organized as follows. Section 2 provides background knowledge on DL compilers. Section 3 describes our data collection and preparation process. Sections 4 to 7 present our four research questions (RQ1–RQ4) and their corresponding approaches and results analysis. Section 8 discusses the implications and limitations of our findings. Section 9 reviews related work, and Section 10 concludes the paper.

2 BACKGROUND

DL compilers play a critical role in efficiently deploying DL models across diverse hardware platforms (e.g., NVIDIA GPUs). Following existing studies [42, 55], the architecture of DL compilers is commonly divided into three phases: the frontend (model loading), the middle-end (high-level

IR transformation), and the backend (low-level IR transformation and code generation). From the perspective of DL compiler users, we further refine this workflow into six stages to better reflect the practical usage process, as shown in Figure 1. Specifically, Stage II corresponds to the frontend, Stages III–IV cover the middle-end and backend (low-level IR transformation), and Stage V corresponds to the backend (code generation). Stages I and VI capture environment setup and performance evaluation, which are often omitted in compiler-centric descriptions but are essential in real-world deployment. More details are shown below:

Stage I - Build and Import. This preliminary phase takes place before the main compilation process begins. To utilize DL compilers, users must first install the DL compiler toolchain and import necessary libraries or Python modules for later compilation and deployment (such as backend runtimes and hardware-specific toolchains).

Stage II - Model Loading. This stage corresponds to the **frontend**. In this stage, the compiler loads DL models from diverse DL frameworks (e.g., PyTorch [9] and TensorFlow [11]). These models are then transformed into a high-level IR to facilitate subsequent compiler-level analysis and optimizations. This stage enables cross-framework support and unifies model semantics within the compiler's internal representation.

Stage III - IR Transformation. This stage constitutes the core of the transformation process, in which the model undergoes a series of IR transformations for diverse optimizations and lowering and it spans both the middle-end and the backend. The **middle-end** corresponds to high-level IR transformation, where hardware-independent optimizations are applied (e.g., operator fusion and constant folding). Afterwards, the optimized high-level IR is lowered into a low-level IR, marking the beginning of the **backend**. In this phase, hardware-aware optimizations are executed (e.g., memory allocation and instruction selection), and the low-level IR is eventually translated into backend-specific executable code (e.g., LLVM IR, CUDA, or SPIR-V), ready for deployment on the designated hardware target. These transformations are typically invoked automatically by the compiler and form a complete transformation path from abstract model representation to optimized, hardware-specific code implementation. For example, in TVM, the entire IR transformation stage can be triggered by calling `relay.build()`.

Stage IV - User-Customized Optimization. This optional stage allows advanced users to customize the compilation pipeline. Unlike Stage III, where optimizations are applied automatically, here users may manually insert optimization passes, adjust pass order, or apply scheduling strategies (e.g., auto-scheduling). Depending on the level of intervention, this stage may operate at the **middle-end** (e.g., adding additional high-level passes such as `relay.transform.FuseOps()` in TVM) or at the **backend** (e.g., customizing scheduling or tensorization). Because such functionality is compiler-dependent, users can skip this stage entirely and proceed directly to Stage V.

Stage V - Inference Execution. This stage corresponds to the **backend**. Once low-level IR is generated and lowered into target-specific code (e.g., LLVM IR, CUDA, or SPIR-V), the model can be deployed and executed on the designated hardware. This stage involves running inference, collecting output results, and possibly conducting inference results consistency validation with reference frameworks to ensure correctness.

Stage VI - Performance Evaluation. This final stage returns to the user domain. Users evaluate the performance of the deployed model using metrics such as latency and throughput. The feedback loop formed here is crucial for iterative tuning, enabling users to refine compilation settings or schedules based on observed deployment performance.

3 DATA PREPARATION

In this section, we first introduce the studied DL compilers, then describe the data collection and annotation of the false-positive bug reports process illustrated in Figure 2.

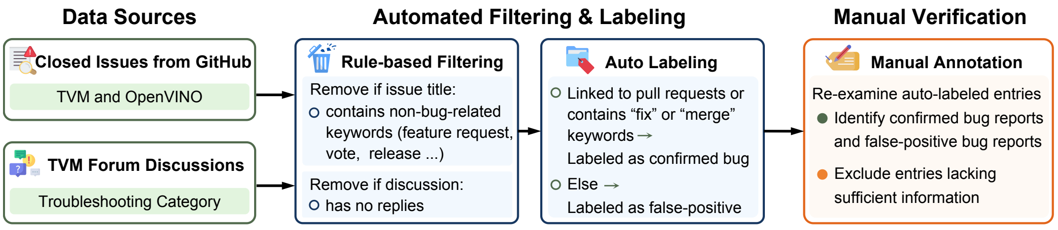

Data Sources
Automated Filtering & Labeling
Manual Verification
Closed Issues from GitHub
TVM and OpenVINO
TVM Forum Discussions
Troubleshooting Category
Rule-based Filtering
Remove if issue title:
contains non-bug-related keywords (feature request, vote, release ...)
Remove if discussion:
has no replies
Auto Labeling
Linked to pull requests or contains "fix" or "merge" keywords → Labeled as confirmed bug
Else → Labeled as false-positive
Manual Annotation
Re-examine auto-labeled entries
Identify confirmed bug reports and false-positive bug reports
Exclude entries lacking sufficient information

Fig. 2. The Dataset Construction Process for DL Compiler False-Positive Bug Reports

Studied DL Compilers. In this work, we chose the two most widely used open-source DL compilers, *Apache TVM* [16] and *Intel OpenVINO* [26], based on their popularity as indicated by GitHub repository star counts. We did not include other DL compilers such as Glow [3], nGraph [6], and XLA [1] due to various limitations in their current development or community status. For example, the repositories of nGraph and Glow have been archived and are no longer maintained, while XLA has a relatively small number of publicly reported issues (only 252 closed issues at the time of writing), indicating limited community engagement. Similarly, other widely used compilers such as NVIDIA TensorRT [7] and OpenAI Triton [61] were excluded because they fall outside the scope of our study. TensorRT, although popular in industry, is not a fully open-source end-to-end DL compiler: only its ONNX parser, plugin library, and sample applications are open sourced, while the core optimization and runtime remain proprietary [12]. Triton, in contrast, is a domain-specific compiler designed to generate efficient GPU kernels from a Python-like DSL, but it does not function as a complete model-level compiler pipeline. Since our work focuses on open-source, end-to-end DL compilers that provide comprehensive model optimization and deployment pipelines, we selected TVM and OpenVINO as representative cases. Apache TVM is an end-to-end DL compiler designed to optimize and deploy DL models across diverse hardware backends, such as CPUs, GPUs, and specialized accelerators. It is renowned for its adaptable intermediate representation (IR), automated tuning capabilities, and extensive support for customizable optimization passes, establishing it as a leading framework in the DL compiler domain. Intel OpenVINO, another widely used DL compiler, specializes in the efficient deployment of DL models on Intel hardware. It supports multiple frontends, including TensorFlow, ONNX [8], and PyTorch, and provides a streamlined toolchain for model conversion, IR transformation, and inference execution. Unlike TVM, OpenVINO follows a more predefined and hardware-specific compilation approach.

Data Collection and Cleaning. We collected data from two primary sources, GitHub repositories (for both studied compilers) and community discussion forums (TVM only). (1) Github Issues provides structured, version-controlled records closely tied to the development lifecycle. We restrict our analysis to closed issues only, as unresolved (open) issues often lack sufficient diagnostic context or resolution, making it difficult to determine whether they reflect genuine bugs or false-positive bug reports. In total, we collected **1,594 closed issues** from TVM and **2,178** from OpenVINO. To ensure relevance, we applied a rule-based filtering process to all collected GitHub issues. Specifically, we removed issues with non-bug-related keywords in their titles (e.g., *vote*, *announcement*, *release*, *question*, and *feature request*). This excluded 224 issues from TVM and 376 from OpenVINO, resulting in **1,370 and 1,802 candidate issues**, respectively. (2) Discussion forums capture a broader range of user-reported errors, including those never formally filed as issues. For TVM, we focus on the "Troubleshooting" category, which includes real-world problems encountered during usage that users cannot resolve on their own and require developer responses to clarify, diagnose, or resolve user confusion, aligning well with our study of false-positive bug reports. We collected **1,246 threads** from this category, excluding 374 unreplied entries due to lack of diagnostic context.

Since OpenVINO's discussion forum does not have clearly defined categories like troubleshooting, accurately extracting bug-related posts from these discussions would be challenging and may introduce bias or inaccuracies, so we did not collect discussion data for OpenVINO.

Annotation of False-Positive Bug Reports. We conducted a two-stage labeling process comprising automatic rule-based labeling and subsequent manual verification, in order to distinguish false-positive bug reports from genuine bugs. (1) Auto Labeling. For all candidate GitHub issues, we applied rules to identify genuine bug reports. Specifically, issues or discussions explicitly linked to pull requests or containing commit-related keywords (e.g., *fix* and *merge*) were labeled as *confirmed bugs*, assuming they led to actual code changes. This step identified **482 confirmed bugs** in TVM issues, **75 confirmed bugs** in TVM discussions, and **221** in OpenVINO issues. The remaining issues, those not removed and not matched by auto-confirmation criteria, were initially labeled as *false-positive bug reports*, resulting in **888** potential false-positive bug reports in TVM issues, **797** in TVM discussions, and **1,581** in OpenVINO issues. (2) Manual Verification. To ensure the reliability of automatic labeling, all auto-labeled entries (both confirmed and false-positive) were subjected to a structured manual validation process. Specifically, two authors independently reviewed and labeled each entry, following a systematic annotation process inspired by prior work [62]. We examined each issue or thread's content, developer responses, error traces, and resolution status. Reports were labeled as *confirmed bugs* if they described a compiler bug that was acknowledged and addressed by developers. In contrast, they were labeled as *false-positive bug reports* if the reported issue stemmed from user misuse, invalid model input, misconfiguration, or misunderstanding of compiler behavior, without any actual compiler-side fix. Notably, our classification follows the maintainers' perspective: an issue was considered a false-positive bug report when developers clarified that it originated from user-side misuse rather than compiler bugs. Entries lacking sufficient discussion, diagnostic detail, or a clear resolution were excluded from the final dataset. This validation step ensures that the dataset only includes cases with a clear and defensible classification. To ensure labeling consistency, we measured inter-rater agreement using Cohen's Kappa coefficient [19, 47]. An initial calibration round on 3% of the data (120 bug reports) resulted in moderate agreement (Cohen's Kappa = 0.42), prompting a discussion to refine labeling criteria and clarify ambiguous cases. After alignment, inter-rater agreement improved substantially, reaching a Cohen's Kappa of 0.82 on the next 10% of the data (400 bug reports). For the remaining entries, the raters continued to annotate independently, with disagreements resolved through discussion with a third author. The final round achieved strong inter-rater agreement (Cohen's Kappa > 0.95), ensuring high consistency in all labels.

This rigorous multi-stage process resulted in a reliable dataset for the subsequent analysis. Our final dataset spans five years, from October 2019 to October 2024, and includes both genuine bugs and false-positive bug reports. As shown in Table 1, we collected **448** false-positive bug reports and **627** genuine bug reports, with an overall false-positive proportion of **41.67%** (specifically 34.92% for TVM and 52.13% for OpenVINO, respectively), underscoring the prevalence of false-positive bug reports in DL compiler development. To examine whether this proportion changes over time, we further divided the dataset into yearly slices (2019–2024). The proportion of false-positive bug reports remained relatively stable across years, indicating that such reports pose a persistent issue rather than a transient phenomenon.

4 RQ1: DEVELOPER EFFORT OVERHEAD OF FALSE-POSITIVE BUG REPORTS

4.1 Approach

To address RQ1, we conduct a quantitative analysis to compare the developer effort involved in handling false-positive versus confirmed bug reports. Drawing inspiration from prior work on

Table 1. Statistical Information on Datasets

Source	Duration	#Bug	#False-Positive	#Total	False-Positive Ratio
TVM	2019-10 ~ 2024-10	425	228	653	34.92%
OpenVINO	2019-09 ~ 2024-09	202	220	422	52.13%
Overall	2019-09 ~ 2024-10	627	448	1,075	41.67%

issue report analysis [38, 69], we define a set of metrics to capture the effort from two perspectives: the discussion process and the resolution lifecycle. While these metrics are widely adopted in issue report studies [38, 69], they may not fully reflect the actual cognitive or time investment of developers, as they can be influenced by external factors (e.g., batch processing of issues delaying the first response). To mitigate this, we employ the multi-metric design, ensuring the robustness of our findings. Based on this design, we extracted metadata from both GitHub issues and forum discussions, including the timestamp of issue or discussion creation (*open_time*), timestamp when the issue was closed (*close_time*), timestamp of the last reply of discussion (*last_comment_time*), time of the first comment from developers other than the report author (*first_non_author_comment_time*), the total number of comments, and the number of unique participants involved in the thread of each bug report in our dataset. Specifically, we selected four metrics, defined as follows:

- **First Response Time**: refers to the duration (hours) from when the author submits an issue report until when the first comment is made. It is computed as the time interval between the metadata fields *open_time* and *first_non_author_comment_time.*
- **Time to Close**: represents the duration (hours) from when the author submits an issue report until when the issue is finally closed. It is calculated as the time interval between the *open_time* and *close_time* for issues or the duration between the *open_time* and *last_comment_time* for forum discussions.
- **Number of Comments**: represents the quantity of comments made within an issue report or a discussion.
- **Number of Participants**: denotes the number of unique participants involved in the issues or discussion thread.

To determine whether developer effort differs significantly between false-positive and confirmed bug reports across these metrics, we applied the Mann-Whitney U test [48], a non-parametric statistical test used to assess whether two independent samples originate from the same distribution. This test is particularly suitable for comparing ordinal data or variables that do not follow a normal distribution. We report p-values to evaluate the presence of statistically significant differences, with $p < 0.05$ indicating a statistically significant result. Moreover, we measured the effect size using *Cliff's Delta* [18], a non-parametric metric that quantifies the degree of overlap between two distributions. Effect size is analyzed as follows: (1) $|\delta| < 0.147$ as Negligible, (2) $0.147 \leq |\delta| < 0.33$ as Small, (3) $0.33 \leq |\delta| < 0.474$ as Medium, or (4) $0.474 \leq |\delta|$ as Large [51]. These metrics together allow us to assess not only whether the differences are statistically significant but also whether they are practically meaningful.

4.2 Results and Analysis

Figure 3 presents a comparative boxplot of false-positive and confirmed bug reports across the four metrics for TVM and OpenVINO. The corresponding statistical results are summarized in Table 2.

Finding 1: False-Positive bug reports exhibit comparable metrics to confirmed bugs in terms of first response, resolution time, and discussion overhead, indicating that they demand substantial

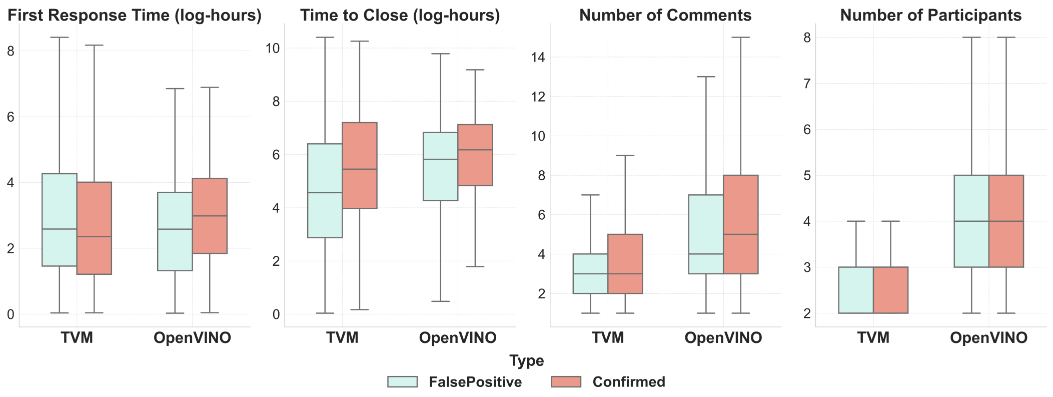


Fig. 3. Comparison of False-Positive and Confirmed Bug Reports in DL Compilers (TVM vs. OpenVINO)

Table 2. Statistical Test Results Comparing False-Positive and Confirmed Bug Reports

Metric	**Project**	**Mann-Whitney U p-value**	**Cliff's Delta**	**Magnitude**
First Response Time (hours)	TVM	9.00E-05	0.186	small
	OpenVINO	0.006848	-0.152	small
Time to Close (hours)	TVM	3.00E-05	-0.198	small
	OpenVINO	0.028438	-0.123	negligible
Number of Comments	TVM	0.602885	-0.024	negligible
	OpenVINO	0.529345	-0.035	negligible
Number of Participants	TVM	0.407571	-0.036	negligible
	OpenVINO	0.012724	-0.138	negligible

developer effort and are non-trivial to triage. For both TVM and OpenVINO, the *First Response Time* exhibits statistically significant differences between false-positive and confirmed bug reports (TVM: $p < 0.001$, $\delta = 0.186$; OpenVINO: $p < 0.01$, $\delta = -0.152$), though the effect sizes are small. In TVM, false-positive bug reports are responded to slightly slower on average (mean: 43.63h vs. 39.47h), but the median time is shorter (6.92h vs. 8.48h). OpenVINO shows a similar pattern, with confirmed bugs having a slightly higher mean time (49.28h vs. 44.16h) and a longer median time (15.39h vs. 12.28h), suggesting comparable responsiveness across both categories. For *Time to Close*, TVM shows statistically significant differences ($p < 0.001$, $\delta = -0.198$), while OpenVINO exhibits marginal significance ($p < 0.05$, $\delta = -0.123$), both with small or negligible effect sizes. The median resolution time is slightly lower for false-positive bug reports in both projects (TVM: 91.12h vs. 100.93h; OpenVINO: 136.26h vs. 146.39h), with mean values remaining close. For both *Number of Comments* and *Number of Participants*, no statistically significant differences were observed (all $p > 0.4$, $|\delta| < 0.04$). On average, false-positive bug reports in OpenVINO receive 6.55 comments from 4.28 participants, while confirmed bugs receive 7.59 comments from 4.53 participants. TVM shows similar engagement levels (false-positive: 3.91 comments / 3.12 participants; confirmed: 4.14 comments / 3.13 participants), reinforcing the observation that discussion activities on false-positive bug reports are as active and collaborative as those on confirmed bugs.

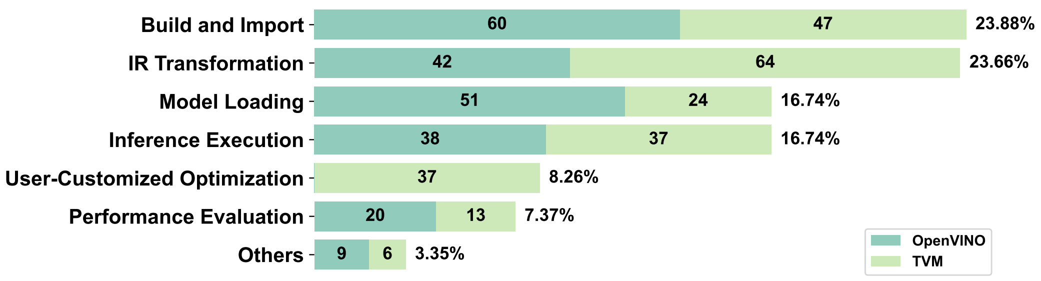


Fig. 4. False-Positive Bug Reports Distribution by Stages

RQ1 Summary

False-Positive bug reports demand a considerable amount of developer effort, comparable to that of confirmed bugs, in terms of first response time, resolution duration, and discussion overhead. These findings support our core motivation: reducing the effort spent on identifying and closing false-positive bug reports can lead to more efficient resource allocation in DL compiler maintenance.

5 RQ2: STAGES OF FALSE-POSITIVE BUG REPORTS

5.1 Approach

To investigate at which stages false-positive bug reports occur, we defined a stage-oriented taxonomy grounded in the practical workflow of DL compiler usage. Specifically, we identified seven stages based on typical user interactions and system functionality: (1) Build and Import, (2) Model Loading, (3) IR Transformation, (4) User-Customized Optimization, (5) Inference Execution, (6) Performance Evaluation, and (7) Others. Among them, Stages 2 to 5 correspond directly to core components of DL compiler pipelines, while Stage 1 captures environment setup or import, and Stage 6 reflects compiled model performance evaluation. More details for each stage are presented in Section 2.

Our analysis was conducted on the set of false-positive bug reports annotated in Section 3. The dataset comprises 448 false-positive bug reports in total, with 228 reports from TVM and 220 from OpenVINO. We annotated the responsible stage for each false-positive bug report based on the user issue report description and developer discussions. In particular, we examined explicit runtime errors, stack traces, referenced APIs or modules, and user commands or reproduction scripts to determine when the failure occurred. For example, reports referencing `tvm.relay.build()` failures were mapped to IR Transformation, while cases involving inconsistent inference results were assigned to Inference Execution. Performance-related complaints, such as "unexpectedly slow inference on GPU," were classified under Performance Evaluation.

To ensure annotation reliability for labeling the compiler stage where the error occurred, two authors independently annotated all false-positive bug reports. An initial calibration on 10% of the data (45 false-positive bug reports) yielded moderate agreement (Cohen's Kappa = 0.45), leading to a refinement of labeling criteria. Subsequent rounds showed substantial improvement (Kappa = 0.86), and after resolving remaining disagreements with a third author, the final round achieved near-perfect agreement (Kappa > 0.98).

5.2 Results and Analysis

To explore which stage is most vulnerable to false-positive bug reports in DL compilers from a user standpoint, we summarized their distribution by stages, shown in Figure 4.

Finding 2: Build and Import is the most frequent stage for false-positive bug reports overall, especially in OpenVINO, highlighting persistent challenges in environment setup and dependency management, suggesting a need for better documentation and automated tools. The figure reveals that false-positive bug reports are not confined to specific stages but occur across all six stages of the DL compiler workflow. Among these stages, we find that the Build and Import stage accounts for the highest proportion of false-positive bug reports (23.88% for total), suggesting that users frequently encounter challenges during initial setup, such as dependency conflicts, hardware compatibility issues, or incorrect environment configurations. To mitigate these issues, improving documentation with step-by-step installation guides and providing automated dependency resolution tools could be beneficial. On the other hand, the Performance Evaluation stage is the least prone to false-positive bug reports for both studied DL compilers, with only 7.37% of cases identified in this stage (20 cases and 13 cases for OpenVINO and TVM, respectively).

Finding 3: TVM's flexible optimization system leads to more false-positive bug reports in the transformation stage than OpenVINO, and uniquely exhibits false-positive bug reports in the User-Customized Optimization stage, necessitating better-encapsulated interfaces and debugging tools. The IR Transformation stage also accounts for a substantial portion of false-positive bug reports (23.66%). This stage encompasses both high-level and low-level IR transformations, the core compilation functionality of DL compilers, which involves complex optimization logic and code generation. Consequently, it is susceptible to user misuse of APIs and misunderstandings of compiler semantics, assumptions, or limitations. Notably, TVM reports significantly more false-positive bug reports in this stage than OpenVINO (64 cases in TVM compared to 42 in OpenVINO), likely due to its highly customizable and fine-grained optimization pipeline. This flexibility also contributes to the presence of false-positive bug reports in the User-Customized Optimization stage, which is unique to TVM (37 cases in TVM, while none in OpenVINO). In this stage, users engage in auto-tuning, schedule rewriting, and independent optimization pass development, capabilities not exposed in OpenVINO, whose optimization strategies are predefined and internally encapsulated.

Finding 4: OpenVINO exhibits a high concentration of false-positive bug reports in the Model Loading stage, reflecting its strict model format and version requirements. Among all stages, Model Loading accounts for the second largest share of OpenVINO's false-positive bug reports, comprising 23.18% (51 out of 220) of its total. This proportion is significantly higher than that observed in TVM, where Model Loading represents only 16.8% of false-positive bug reports. These issues often stem from importing models from external frameworks like ONNX or TensorFlow, with unsupported operators, dynamic shapes, or mismatched versions. In contrast, this stage is less problematic in TVM, likely due to its broader support for diverse model formats (e.g., model with dynamic tensor shapes) and operator sets. To reduce such false-positive bug reports, OpenVINO could benefit from enhanced model validation feedback and clearer documentation on supported formats.

RQ2 Summary

False-positive bug reports are across all stages of DL compiler workflows, indicating that such issues can arise at any point in the compilation and execution process. Build and Import and IR Transformation stages account for the largest proportions (23.88% and 23.66%, respectively), together contributing about half of all false-positive bug reports.

6 RQ3: ROOT CAUSES OF FALSE-POSITIVE BUG REPORTS

6.1 Approach

To identify the root causes of false-positive bug reports, we manually analyzed a dataset of 448 false-positive bug reports (228 from TVM and 220 from OpenVINO) as constructed in Section 3. We initially referred to taxonomies from prior work [40, 41] that focus on false-positive bug reports in traditional software as a starting point. We then followed an open-coding approach [20] to iteratively refine, expand, or discard categories based on actual DL compiler issues. This allowed us to develop a taxonomy that incorporates domain-specific patterns (e.g., invalid model input) and better reflects real-world DL compiler false-positive bug reports. For each entry, two authors independently analyzed descriptions and developer discussions to determine root causes. To ensure labeling consistency, we measured inter-rater agreement using Cohen's Kappa coefficient. An initial round on 10% of the data (45 false-positive bug reports) yielded low agreement (32%), prompting a calibration discussion. Agreement improved to 83% in the next 10%, and subsequent rounds exceeded 95% after resolving inconsistencies. Remaining disagreements were adjudicated by a third author to ensure accurate labeling of all false-positive bug reports. Drawing from an in-depth analysis of false-positive bug reports of DL compilers, we devise a taxonomy of root causes behind them, as outlined below:

(1) Incorrect Environment Configuration. This root cause refers to false-positive bug reports triggered by incorrect or incomplete environment setups, rather than bugs in the compiler itself. These setups include (i) DL-specific configuration errors such as improper ONNX operator sets, (ii) general software stack issues like mismatched Python or operating system versions, and (iii) hardware-related misconfiguration, including unavailable GPUs or unsupported instruction sets. Incorrect environment configuration often follows recurring symptoms, including missing configurations, conflicting configurations, unavailable resources, and missing dependencies.

(2) Invalid Model or Model Input. This category includes issues arising from invalid usage of DL models or model inputs that violate the input specifications. These problems typically occur when the user provides a model or input data that does not meet the required format (e.g., supplying models in unsupported formats such as Caffe [34]), structure (e.g., mismatched input-output tensor shapes), or other constraints of compilers. Specifically, issues in this category can be triggered by unsupported operators or layers, invalid input due to incompatibilities across frameworks, and invalid inputs such as incorrectly shaped or typed tensors that may lead to undefined behavior. These issues lead to unexpected behaviors during compilation or execution, which are attributed to user-side input errors rather than compiler bugs.

(3) Incorrect Usage. This category encompasses false-positive bug reports that stem from users incorrectly using the compiler, such as API misuse, unsupported usage, or failing to follow required compilation procedures. The incorrect use of DL compilers deviates from the expected usage, resulting in invalid compilation scripts. For instance, users may forget to register a pass or invoke an API with incorrect parameters.

(4) Misunderstanding of Features or Limitations. This category includes false-positive bug reports where the user's usage is technically correct. That is, the compilation script constructed by users is valid, but the user misinterprets the output or behavior due to a misunderstanding of the compiler's features, optimizations, inherent constraints, or some known limitations. These misunderstandings typically arise when users assume incorrect behavior expectations. For example, misreading precision loss as a bug, or not recognizing hardware-specific fallbacks. In essence, these issues lie not in the execution results but in the misalignment between user expectations and the compiler's intended design.

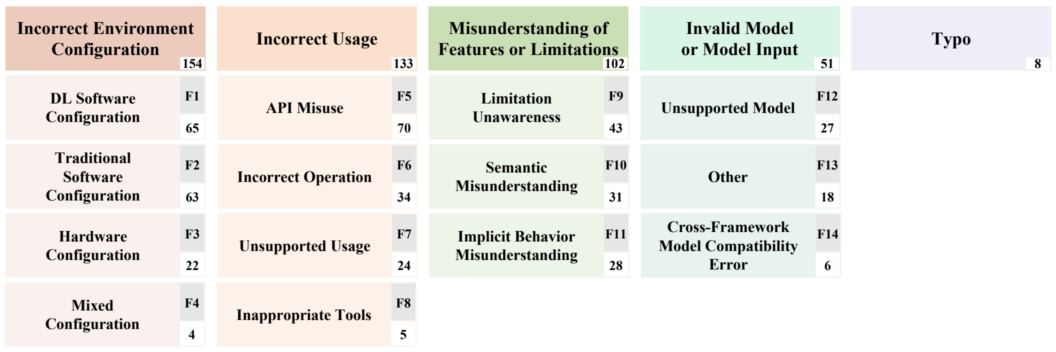

Incorrect Environment Configuration 154
Incorrect Usage 133
Misunderstanding of Features or Limitations 102
Invalid Model or Model Input 51
Typo 8
DL Software Configuration F1 65
Traditional Software Configuration F2 63
Hardware Configuration F3 22
Mixed Configuration F4 4
API Misuse F5 70
Incorrect Operation F6 34
Unsupported Usage F7 24
Inappropriate Tools F8 5
Limitation Unawareness F9 43
Semantic Misunderstanding F10 31
Implicit Behavior Misunderstanding F11 28
Unsupported Model F12 27
Other F13 18
Cross-Framework Model Compatibility Error F14 6

Fig. 5. False-Positive Bug Report Distribution by Root Causes

Table 3. Distribution of Incorrect Environment Configuration by Subcategories and Symptoms

Category Type	Subcategory / Symptoms	TVM	OpenVINO	Total
Subcategories	DL Software Configuration	24	41	65
	Traditional Software Configuration	28	35	63
	Hardware Configuration	13	9	22
	Mixed Configuration	4	0	4
Symptoms	Missing Configuration	14	2	16
	Conflict Configuration	45	76	121
	Missing Dependencies	7	4	11
	Unavailable Resource	3	3	6
Total	–	**69**	**85**	**154**

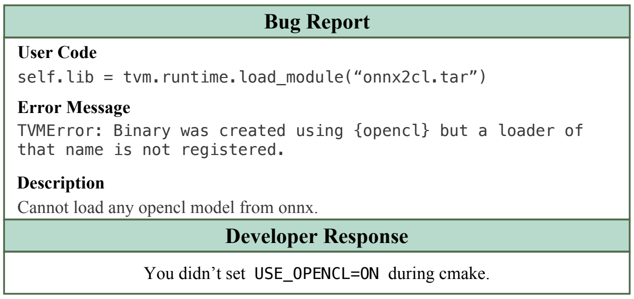
(5) Typo. False-Positive bug reports in this category arise from user typos in the user code, which inadvertently produce invalid commands or malformed syntax.

During the manual classification of root causes, we further annotated the observable sub-root causes associated with each category. To systematically identify and organize these subcategories, we employed a card sorting method [58], enabling us to uncover common root causes and improve our understanding of how different root causes manifest in false-positive bug reports.

6.2 Results and Analysis

Figure 5 presents the distribution of identified root causes and their associated subcategories for false-positive bug reports in DL compilers. Each root cause highlights distinct challenges related to compiler usability and ecosystem design. To complement this overview, Tables 3 through 6 provide detailed statistical summaries of each subcategory, along with the corresponding symptoms, offering deeper insight into how these issues manifest in practice.

Finding 5: Incorrect Environment Configuration is the most frequent root cause (34.38%), with over 80% involving software configuration errors. Conflict configuration dominates (78%), indicating that version mismatches and dependency incompatibilities are the primary symptoms of false-positive bug reports. As shown in Figure 5, the largest proportion of false-positive bug reports stems from Incorrect Environment Configuration, with 154 cases being identified (34.38%). This is primarily due to the inherent complexity of DL compilers, which are designed to interact with multiple DL frameworks, hardware platforms, and various third-party libraries. Additionally, DL compilers are typically implemented using multiple programming languages (C/C++ and Python), which further complicates the environment setup. Meanwhile, the rapid evolution of DL compiler



Bug Report

User Code

```
self.lib = tvm.runtime.load_module(“onnx2cl.tar”)
```

Error Message

```
TVMError: Binary was created using {opencl} but a loader of
that name is not registered.
```

Description

Cannot load any opencl model from onnx.

Developer Response

You didn’t set `USE_OPENCL=ON` during cmake.

Fig. 6. Incorrect Environment Configuration Example

features (e.g., IRs incompatible evaluation from `Tensor Expression (TE)` to `relay` to `relax` in TVM) and related third-party components leads to compatibility challenges, significantly increasing the number of configuration-related issues.

Additionally, our analysis also reveals several symptoms underlying these incorrect configurations. The most common symptom is *Conflict Configuration*, such as incompatible versions between drivers and libraries. We also observe *Missing Configuration*, *Missing Dependencies*, and *Unavailable Resource* (e.g., absent hardware drivers or device access failures). These symptoms often transcend specific categories and reflect common root causes of user confusion. Table 3 shows a closer inspection of the distribution. Notably, conflict configuration emerges as the dominant symptom (78% cases), highlighting the subtle nature of these issues and the limitations of current detection mechanisms. Furthermore, differences between compilers suggest that TVM is more prone to missing setups, whereas OpenVINO exhibits more version conflicts, possibly due to differences in packaging or documentation.

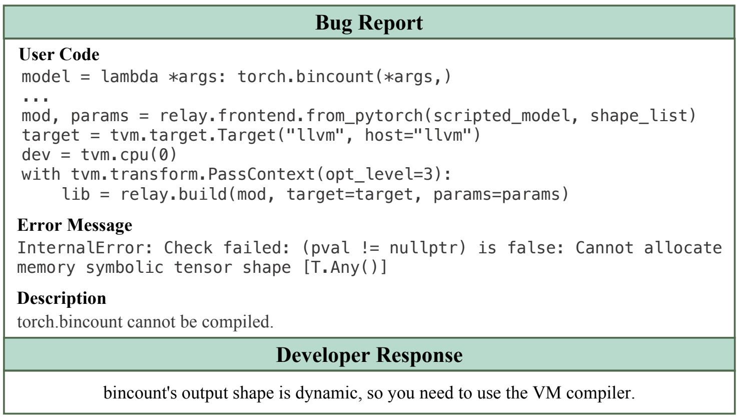
Illustrative Example. Figure 6 shows a false-positive bug report[2] caused by Incorrect Environment Configuration, a user reported a runtime error when using `tvm.runtime.load_module("onnx2cl.tar")` to attempt to load an OpenCL-compiled model. The error message stated that the OpenCL loader was not registered. Note that the issue title included the word “bug” and the user explicitly added a “bug” label, indicating the belief that the problem stemmed from a compiler bug. Upon inspection, the root cause was identified as Incorrect Environment Configuration: the user had not enabled the `USE_OPENCL=ON` flag during the TVM build process. This led to the absence of the necessary OpenCL runtime components. A TVM developer clarified that recompiling with the appropriate flag would resolve the issue. This case exemplifies a false-positive bug report resulting from *Incorrect Environment Configuration*, rather than a bug in the compiler itself. According to our classification, this case belongs to the *DL Software Configuration* subcategory and reflects the *Missing Configuration* symptom.

Finding 6: Incorrect Usage accounts for 29.69% of false-positive bug reports, primarily driven by API misuse and incorrect operation. These errors often stem from complex compiler usage requirements, insufficient or outdated documentation, and users’ limited familiarity with the DL compiler internals. Incorrect Usage constitutes the second most prevalent source of false-positive bug reports, accounting for 133 cases (29.69%) across our dataset. To understand its underlying patterns, we further categorized it into four subcategories: *API Misuse*, *Incorrect Operation*, *Unsupported Usage*, and *Inappropriate Tools*. Among them, API Misuse accounts for the largest share, accounting for 52.63% of Incorrect Usage issues. Further analysis found four symptoms behind the API Misuse subcategory, including *Argument Error*, *Wrong API*, *Missing Parameters*, and *Incorrect Return Value*

[2]https://github.com/apache/tvm/issues/13076

Table 4. Distribution of Incorrect Usage by Subcategories and API Misuse Symptoms

Category	Subcategory	Symptom	TVM	OpenVINO	Total
Incorrect Usage	API Misuse	Argument Error	20	9	29
		Wrong API	21	5	26
		Missing Parameters	4	8	12
		Incorrect Return Value Handling	3	0	3
		Subtotal	**48**	**22**	**70**
	Incorrect Operation	–	19	15	34
	Unsupported Usage	–	6	18	24
	Inappropriate Tools	–	0	5	5
Total	–	–	**73**	**60**	**133**


Bug Report

User Code
```
model = lambda *args: torch.bincount(*args,)
...
mod, params = relay.frontend.from_pytorch(scripted_model, shape_list)
target = tvm.target.Target("llvm", host="llvm")
dev = tvm.cpu(0)
with tvm.transform.PassContext(opt_level=3):
    lib = relay.build(mod, target=target, params=params)
```

Error Message
```
InternalError: Check failed: (pval != nullptr) is false: Cannot allocate
memory symbolic tensor shape [T.Any()]
```

Description

torch.bincount cannot be compiled.

Developer Response

bincount's output shape is dynamic, so you need to use the VM compiler.

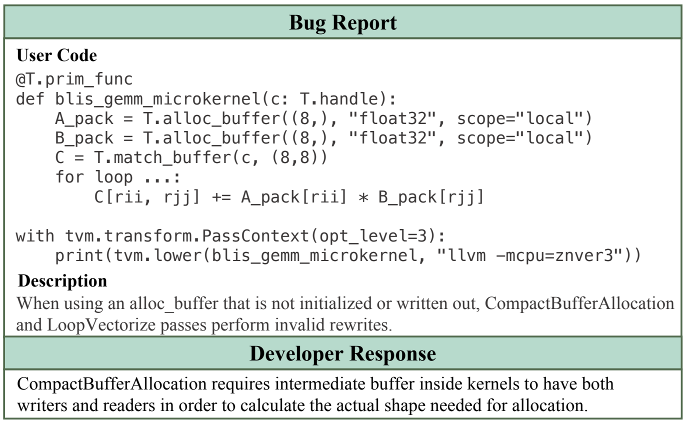
Fig. 7. Incorrect Usage Example

Handling; their distribution is shown in Table 4. We focus on symptom-level classification only on API Misuse due to its well-defined and recurring error symptoms. Other subcategories involve more diverse and less structured issues, which are not amenable to consistent symptom categorization. Notably, Argument Error and Wrong API together account for nearly 80% of all API Misuse cases, highlighting the prevalence of misunderstandings around API semantics and expected parameter configurations. Additionally, our manual annotation process revealed repeated patterns where API Misuse stemmed from insufficient or outdated documentation.

The Incorrect Operation subcategory is the second most frequent source of incorrect usage, accounting for 34 cases across TVM and OpenVINO. These issues typically arise from internal constraints in DL compilers, such as optimization passes ordering dependencies (e.g., certain transformations require others to be completed beforehand) and dynamic tensor dimensions or datatype mismatches. The Unsupported Usage subcategory accounts for 24 cases (6 in TVM and 18 in OpenVINO), where users attempt to use features or operations not supported by the current version of the DL compiler. For example, in TVM, certain dynamic control flow patterns (e.g., if statements with data-dependent conditions) may not be fully supported. Inappropriate Tools is a unique subcategory observed only in OpenVINO. It appears when users apply incompatible tools or plugins due to its modular architecture, comprising components like the Model Optimizer, each with specific usage constraints. These false-positive bug reports typically result from limited

Table 5. Distribution of Misunderstanding of Features or Limitations by Subcategories

Subcategory	**TVM**	**OpenVINO**	**Total**
Implicit Behavior Misunderstanding	12	16	28
Semantic Misunderstanding	25	6	31
Limitation Unawareness	30	13	43
Total	**67**	**35**	**102**



Bug Report

User Code

```
@T.prim_func
def blis_gemm_microkernel(c: T.handle):
    A_pack = T.alloc_buffer((8,), "float32", scope="local")
    B_pack = T.alloc_buffer((8,), "float32", scope="local")
    C = T.match_buffer(c, (8,8))
    for loop ...:
        C[rii, rjj] += A_pack[rii] * B_pack[rjj]

with tvm.transform.PassContext(opt_level=3):
    print(tvm.lower(blis_gemm_microkernel, "llvm -mcpu=znver3"))
```

Description

When using an alloc_buffer that is not initialized or written out, CompactBufferAllocation and LoopVectorize passes perform invalid rewrites.

Developer Response

CompactBufferAllocation requires intermediate buffer inside kernels to have both writers and readers in order to calculate the actual shape needed for allocation.

Fig. 8. Misunderstanding of Features or Limitations Example

user understanding. In contrast, TVM's unified compilation interface minimizes such decisions, explaining the absence of this category of false-positive bug reports in TVM.

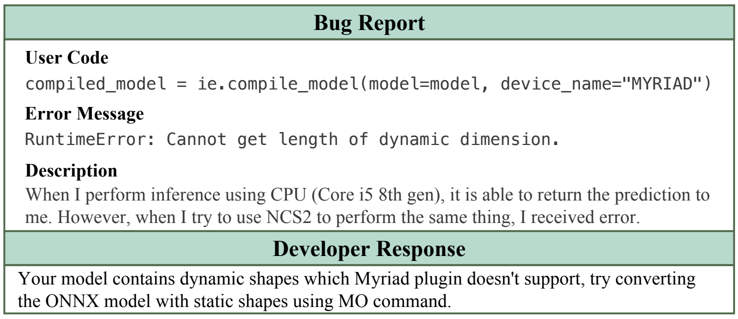
Illustrative Example. Figure 7 illustrates a false-positive bug report[3] in the TVM caused by *Incorrect Usage*, where compiling a PyTorch model containing `torch.bincount` in the static mode operator failed. The user misidentified this as a compiler bug, while the failure resulted from *API Misuse (Wrong API)* for compiling dynamic models. Specifically, `torch.bincount` returns a 1D tensor of length `max(input) + 1`, which introduces dynamic output shapes. In such cases, the correct API is `relay.vm.compile` rather than the static graph compiler `relay.build`. This issue has been repeatedly reported in six separate issues and two discussion threads, highlighting a recurring wrong usage. Developers acknowledged the problem, stating: "Unfortunately, this gotcha is common and not well documented." Yet no corresponding documentation updates were provided. One user even criticized the premature issue closure, citing poor user experience due to uninformative error messages and the lack of usage guidance. Another example[4] involves the misuse of the `LiftTransformParams` optimization. The user was unaware that this transformation, unlike most optimization passes, alters function signatures. A developer noted that such confusion "probably means that the API/documentation needs improvement." Since only the latest version documentation is maintained, it becomes more difficult for users to locate references matching the version they are using. In contrast, OpenVINO maintains versioned documentation, which can reduce such confusion.

Finding 7: Misunderstanding of Features or Limitations causes 22.77% of false-positive bug reports, revealing both a semantic gap between user expectations and compiler behavior, and

[3]https://github.com/apache/tvm/issues/15785
[4]https://github.com/apache/tvm/issues/17207

Table 6. Distribution of Invalid Model or Model Input by Subcategories

Subcategory	**TVM**	**OpenVINO**	**Total**
Unsupported Model	2	25	27
Cross-Framework Model Compatibility Error	5	1	6
Other	8	10	18
Total	**15**	**36**	**51**



Bug Report

User Code

```
compiled_model = ie.compile_model(model=model, device_name="MYRIAD")
```

Error Message

```
RuntimeError: Cannot get length of dynamic dimension.
```

Description

When I perform inference using CPU (Core i5 8th gen), it is able to return the prediction to me. However, when I try to use NCS2 to perform the same thing, I received error.

Developer Response

Your model contains dynamic shapes which Myriad plugin doesn't support, try converting the ONNX model with static shapes using MO command.

Fig. 9. Invalid Model or Model Input Example

the lack of transparency in compiler-side assumptions and constraints. The third most common root cause of false-positive bug reports involves the Misunderstanding of Compiler Features or Limitations, accounting for 102 cases (22.77%) across our dataset. This category of false-positive bug reports arises when users misinterpret the definitions, assumptions, or implicit behaviors embedded in DL compiler design, often due to divergence from conventional compiler or DL framework expectations, or inadequate documentation. Based on the content of the misunderstanding, we further classify this category into three subcategories: *Semantic Misunderstanding*, *Implicit Behavior Misunderstanding*, and *Limitation Unawareness*. Table 5 shows the subcategory distribution, with Semantic Misunderstanding (31 cases) and Implicit Behavior Misunderstanding (28 cases) being most common in TVM. Of these, 17 reports stem from optimization pass misunderstandings, likely due to TVM's fine-grained control over IR transformations. OpenVINO's constrained interface results in fewer such issues.

Limitation Unawareness (43 cases) is a prominent subcategory within Misunderstanding of Features or Limitations. It occurs when users lack sufficient knowledge about DL compiler constraints or hardware limitations, such as unsupported dynamic shapes or performance boundaries. These cases often highlight a disconnect between the compiler's implicit assumptions and user expectations of what should be valid or supported.

Illustrative Example. Figure 8 shows a false-positive bug report[5] where a user misinterpreted TVM's buffer optimization behavior. The user created a TIR function with two unwritten local buffers (allocated via `alloc_buffer`). When optimized with `CompactBufferAllocation` and `LoopVectorize`, the output appeared buggy (e.g., repeated buffer indices). However, this is expected—since the buffers were never written, the pass legally reuses or reshapes them to save memory. To enforce fixed buffer shapes, users can disable `tir.CompactBufferAllocation` and `tir.PlanAndUpdateBufferAllocationLocation`.

[5]https://github.com/apache/tvm/issues/10877

Table 7. Stage-wise Root Cause Proportions

Stage	Root Cause			
	Incorrect Environment Configuration	Incorrect Usage	Invalid Model or Model Input	Misunderstanding of Features or Limitations
Build and Import	**85.44%**	11.65%	0	2.91%
Model Loading	27.40%	**36.99%**	26.03%	9.59%
IR Transformation	21.21%	**45.45%**	20.20%	13.13%
User-Customized Optimization	18.92%	37.84%	0	**43.24%**
Inference Execution	20.90%	**40.30%**	14.93%	23.88%
Performance Evaluation	5.56%	27.78%	11.11%	**55.56%**

Finding 8: Invalid Model or Model Input account for 11.38% of false-positive bug reports, underscoring insufficient input validation. A non-negligible portion of false-positive bug reports stems from Invalid Model or Model Input, accounting for 51 cases (11.38%). It can be divided into three subcategories: *Unsupported Models*, *Cross-Framework Model Compatibility Error*, and *Other*. (1) Unsupported Models refer to model-built frameworks, operators in models, or model dynamic input features that were not supported by the DL compilers, leading to a crash issue. For instance, figure 9 depicts a crash issue[6] where an ONNX model with a dynamic batch dimension (`[None,20,64,64,3]`) failed on OpenVINO's MYRIAD plugin due to its lack of dynamic shape support. To address this, developers recommended using OpenVINO's Model Optimizer with an explicitly defined static input shape (e.g., `input_shape [1,20,64,64,3]`) to bypass runtime inference limitations. (2) Cross-Framework Model Compatibility Errors occur when users export models or provide inputs that are valid in the source framework (e.g., TensorFlow or PyTorch) but violate constraints of the target compiler. This may result from third-party conversion tools that introduce structural inconsistencies or unsupported semantics. For example, users may wrongly assume that input layouts like NHWC (used in TensorFlow) are directly transferable to frameworks expecting NCHW (e.g., TVM or ONNX). In TVM, issues such as passing a 1D tensor to a dense operator that requires 2D input, or using `torch.grid_sample` with `padding_mode="reflection"` and `align_corners=True` (which leads to division-by-zero due to truncation behavior), illustrate such mismatches. Similarly, TVM imposes stricter shape and stride compatibility than TensorFlow, leading to shape inference errors when models use padding or slicing with flexible input shapes. (3) Other cases include user errors unrelated to model semantics, such as incorrect model paths or corrupted model files. As Table 6 indicates, OpenVINO has more reports related to unsupported models, reflecting its stricter support boundaries. TVM has more issues with cross-framework input assumptions, highlighting the challenges in model portability across compilers.

6.3 Relationship between Root Causes and Stages

Due to the specific focus of each compiler workflow stage, a specific root cause may occur in a specific stage. Thus, we further investigate the relationship between root causes and stages. Table 7 presents a statistics of proportion analysis of root cause distributions across distinct stages of the DL compiler workflow, while Figure 10 illustrates the co-occurrence patterns. In this figure, each colored band connects a root cause of a false-positive bug report to the stage where it occurs. The width and color (as shown in the colorbar) of each band represent the number of cases. Wider bands and warmer colors indicate higher frequencies for the corresponding root cause-stage pairs.

[6]https://github.com/openvinotoolkit/openvino/issues/17531

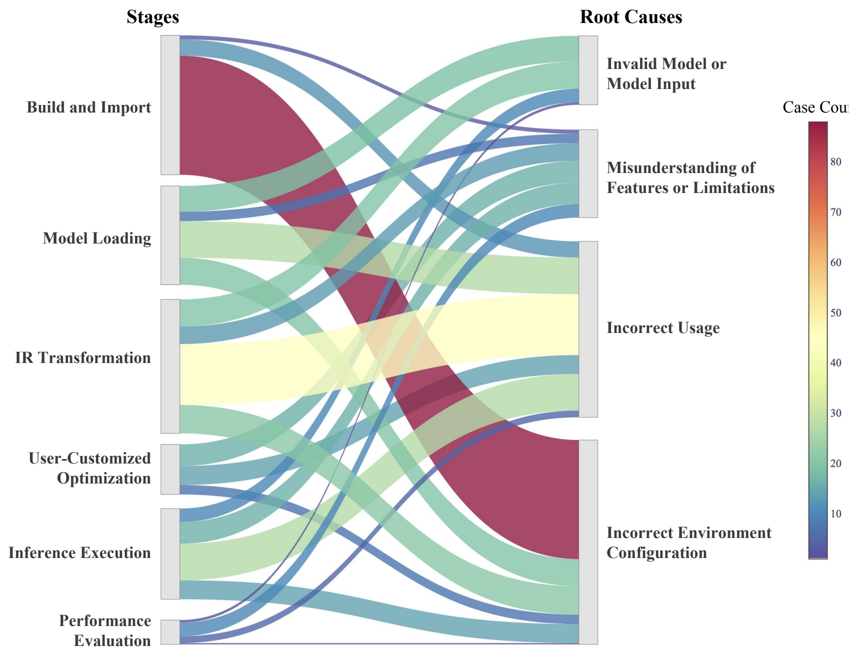


Fig. 10. Parallel Sets between Stages and Root Causes of DL Compiler False-Positive Bug Reports

Finding 9: False-Positive bug reports exhibit distinct, stage-specific root cause patterns, indicating that certain types of misunderstandings or misconfigurations are more likely to occur at specific points in the DL compiler workflow. Notably, the Build and Import stage is overwhelmingly associated with Incorrect Environment Configuration (85.44% of false-positive bug reports), underscoring persistent challenges in dependency management, driver compatibility, and hardware/software setup. In contrast, the Performance Evaluation stage is dominated by Misunderstanding of Features or Limitations false-positive bug reports (55.56%), suggesting frequent user misattribution of bottlenecks to compiler bugs rather than underlying system or algorithmic limitations.

The Model Loading and Inference Execution stages demonstrate more varied root cause distributions, indicative of their inherent complexity. During Model Loading, false-positive bug reports arise from a mix of Incorrect Usage (36.99%), Incorrect Environment Configuration (27.40%), and Invalid Model or Model Input (26.03%), highlighting issues such as unsupported operators, format mismatches, or API Misuse. Similarly, Inference Execution sees comparable trends, with Incorrect Usage (40.30%), Misunderstanding of Features or Limitations (23.88%), and Incorrect Environment Configuration (20.90%) as leading factors. These patterns emphasize that user errors and misunderstandings persist even in later stages, often due to subtle operational or conceptual gaps.

The IR Transformation and User-Customized Optimization stages exhibit distinct but revealing trends. IR transformations are primarily affected by Incorrect Usage (45.45%) and Incorrect Environment Configurations (21.21%), alongside non-trivial contributions from Invalid Model or Model Input (20.20%) and Misunderstanding of Features or Limitations (13.13%). This suggests that IR Transformation phases expose user unfamiliarity with API constraints or transformation semantics. Meanwhile, User-Customized Optimization shows a pronounced prevalence of Misunderstanding of Features or Limitations (43.24%), as users frequently mistake optimization behaviors for bugs. Incorrect Usage (37.84%) and Incorrect Environment Configuration (18.92%) remain significant, further illustrating the challenges of debugging user-customized optimization runtime behaviors.

RQ3 Summary

The majority of false-positive bug reports in DL compilers stem from three primary root causes: Incorrect Environment Configuration (34.38%), Incorrect Usage (29.69%), and Misunderstanding of Features or Limitations (22.77%). Furthermore, our stage-wise correlation analysis reveals that each compiler stage tends to be associated with specific dominant root causes. For instance, at the Build and Import stage, the most prevalent root cause is Incorrect Environment Configuration.

7 RQ4: CAN LLMS HELP MITIGATE FALSE-POSITIVE BUG REPORTS IN DL COMPILERS?

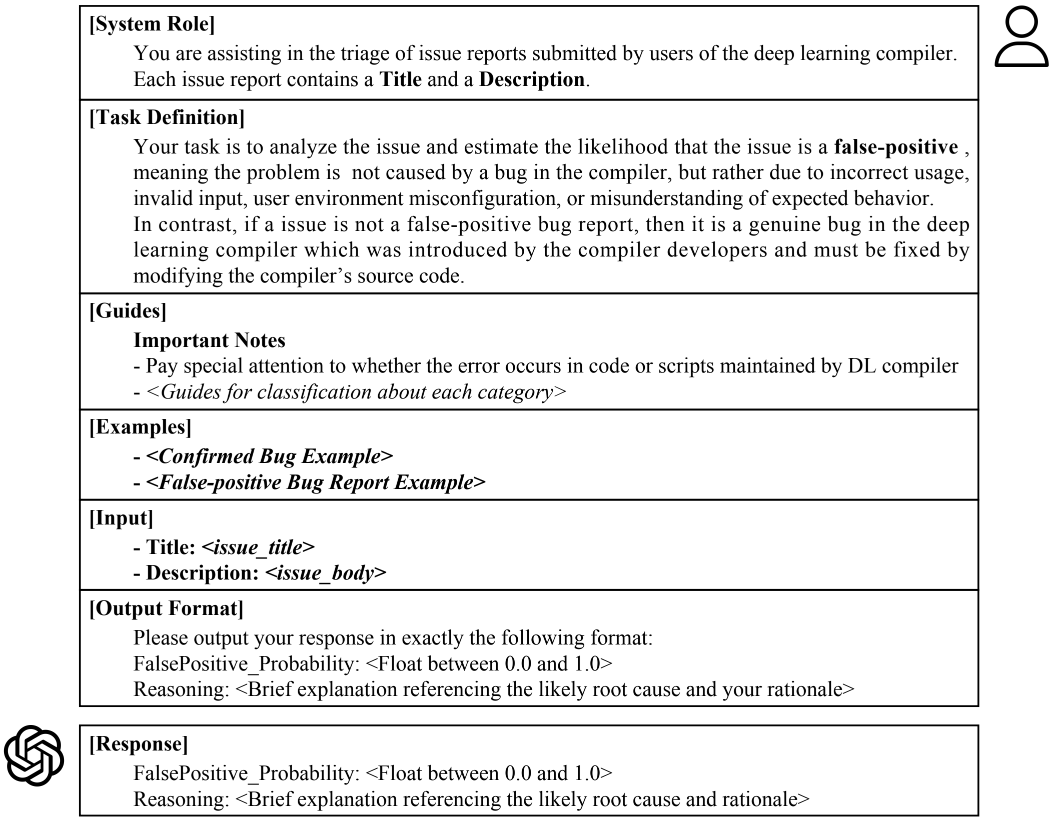
7.1 Approach

To explore the LLM's ability to automatically distinguish between genuine bugs and false-positive bug reports in DL compiler bug reports, we designed four LLM-based approaches for evaluations: zero-shot (including basic zero-shot and enhanced zero-shot) [37], few-shot learning [36], chain-of-thought (CoT) [64], and retrieval-augmented generation (RAG) [25]. We formulate this as a classification-by-simulation task: rather than labeling directly, the LLM estimates the probability that a report is a false-positive bug report. This probabilistic framing encourages nuanced reasoning and calibrated predictions [35]. Each strategy builds upon our prior insights from common root causes of false-positive bug reports and where false-positive bug reports typically arise. Below, we provide detailed descriptions of each approach.

Zero-Shot. We developed two zero-shot approaches that differed in the level of details provided in their definitions and contextual information. (1) Basic zero-shot: provides only the definitions of genuine and false-positive bug reports, plus the issue title and description, serving as a baseline to assess raw model capability. (2) Enhanced zero-shot: adds explicit root cause categories and refined definitions to reduce ambiguity and improve precision. Both approaches establish key baselines for evaluating the LLM's capabilities before advancing to more complex techniques like few-shot learning, CoT prompting, and RAG prompting.

Few-shot. Figure 11 clearly illustrates the structure of the few-shot prompt used for automated false-positive bug report identification. The LLM is tasked with triaging DL compiler bug reports using definitions of root causes of false-positive bug reports (e.g., incorrect usage, invalid model or model input, or incorrect environment configuration) and genuine bugs (issues requiring code fixes). We embed heuristic notes derived from prior findings. For instance, environment-related errors are false-positive bug reports only when caused by user misconfiguration. To enhance adaptability, we adopt a *dynamic few-shot* strategy: from a balanced seed pool of 8 real-world examples, 4 genuine bugs and 4 false-positive bug reports. We sample each example to represent a distinct root cause. Specifically, the false-positive bug reports correspond to *Incorrect Environment Configuration*, *Incorrect Usage*, *Invalid Model or Model Input*, and *Misunderstanding of Features or Limitations*. The confirmed bug pool similarly spans multiple root cause types to ensure representative diversity and comparability; they are selected to reflect corresponding categories based on the taxonomy proposed by Shen et al. [55], including misconfiguration, incompatibility, incorrect code logic, and API misuse root causes. Then the model selects one example from the false-positive bug reports and one from the confirmed bugs. This ensures context-aligned reasoning and avoids overfitting to fixed examples.

Chain-of-Thought. We further introduce CoT prompting to emulate the human reasoning process when analyzing bug reports and to guide the model through stepwise reasoning before final



[System Role]

You are assisting in the triage of issue reports submitted by users of the deep learning compiler. Each issue report contains a **Title** and a **Description**.

[Task Definition]

Your task is to analyze the issue and estimate the likelihood that the issue is a **false-positive** , meaning the problem is not caused by a bug in the compiler, but rather due to incorrect usage, invalid input, user environment misconfiguration, or misunderstanding of expected behavior.
In contrast, if a issue is not a false-positive bug report, then it is a genuine bug in the deep learning compiler which was introduced by the compiler developers and must be fixed by modifying the compiler's source code.

[Guides]

Important Notes
- Pay special attention to whether the error occurs in code or scripts maintained by DL compiler
- <*Guides for classification about each category*>

[Examples]

- ***<Confirmed Bug Example>***
- ***<False-positive Bug Report Example>***

[Input]

- **Title:** ***<issue_title>***
- **Description:** ***<issue_body>***

[Output Format]

Please output your response in exactly the following format:
FalsePositive_Probability: <Float between 0.0 and 1.0>
Reasoning: <Brief explanation referencing the likely root cause and your rationale>

[Response]

FalsePositive_Probability: <Float between 0.0 and 1.0>
Reasoning: <Brief explanation referencing the likely root cause and rationale>

Fig. 11. Structure of the Few-shot Prompt for False-Positive Bug Report Identification

classification. Following the common CoT practice [64], we decompose the reasoning process into five intuitive stages: (1) Error Location: determine where the issue originates, especially whether it arises from compiler-maintained code or user's scripts. (2) Environment Check: verify whether the error results from incorrect environment configurations. (3) Usage Check: identify possible user misuse or invalid inputs that could trigger the issue. (4) Diagnostic Depth: examine the degree of technical analysis provided in the report itself, which often indicates the reporter's understanding of the system and the plausibility of the diagnosis. (5) Similarity to Known Cases: compare the current report with one known false-positive and one confirmed bug case, selected following the same few-shot sampling strategy, to encourage analogical reasoning and context-aware judgment. This structured reasoning path provides a transparent and interpretable process for decision-making, allowing us to analyze how LLMs align with expert-level triage heuristics.

RAG. To integrate domain knowledge into the classification of false-positive bug reports, we employ RAG implemented via the RAGFlow framework[10], an open-source tool designed to seamlessly combine retrieval with LLM inference. Following the widely used RAG practice [21], intuitively, we first construct separate knowledge bases for TVM and OpenVINO by indexing their official documentation.[7] Given a bug report, RAGFlow retrieves the most relevant textual snippets from the corresponding compiler's documentation. These retrieved documentations are then combined with the original prompt and fed into the LLM for prediction. This RAG setup allows

[7]TVM documentation: https://tvm.apache.org/docs/; OpenVINO documentation: https://docs.openvino.ai/

the model to access up-to-date, compiler-specific technical references (e.g., supported operators, known limitations, usage constraints), thereby likely enhancing the accuracy and reliability of its classification decisions.

Implementation Details. In this work, we used OpenAI GPT-4o [5] (specifically, the gpt-4o-2024-05-13 version) as the underlying language model, as it has demonstrated strong performance across a variety of natural language understanding, code understanding, and reasoning tasks [4, 30, 53]. We set the temperature to 0.8 to allow for moderate output diversity, which is beneficial when generating natural language explanations. All other hyperparameters (e.g., top-p, max tokens, stop sequences) were kept at their default values provided by the API. These settings were applied consistently across all prompting strategies.

7.2 Results and Analysis

Unlike conventional classification methods, our LLM-based approach generates a soft probability score (ranging from 0.0 to 1.0) instead of a binary label. In our setting, the positive class is false-positive bug reports; thus, the principal risk is misclassifying a genuine bug as a false-positive bug report, which could lead developers to ignore a real bug and delay critical fixes. Accordingly, we prioritize precision over recall, in contrast to conventional bug-detection tasks that typically emphasize recall. To operationalize this trade-off, we experimented with different thresholds for converting probability scores into discrete predictions. Following prior works[50, 52], we evaluated thresholds between 0.7 and 0.9. As expected, lower thresholds (e.g., 0.7) improve recall but introduce more incorrect positive predictions (i.e., genuine bugs mislabeled as false-positive bug reports), whereas higher thresholds (e.g., 0.9) reduce such misclassifications but significantly lower recall by missing valid false-positive bug reports. We found that a threshold of 0.8 offers the best balance between minimizing the misclassification of genuine bugs and maintaining reasonable coverage of false-positive bug reports. Detailed results for different threshold values are provided in our supplementary repository for transparency. Under this setting, our best-performing few-shot prompt achieves a precision of 86% and a recall of 42%. All subsequent results are reported using this threshold. Table 8 presents the evaluation metrics across different prompt strategies and compilers (TVM, OpenVINO, and combined). We report standard metrics: accuracy, precision, recall, F1-score, and $F_{0.5}$-score. $F_{0.5}$-score is defined as:

$$F_{0.5}\text{-score} = (1 + 0.5^2) \times \frac{\text{precision} \cdot \text{recall}}{0.5^2 \cdot \text{precision} + \text{recall}}$$

This metric emphasizes precision over recall, making it particularly suitable for our objective of minimizing the misclassification of genuine bugs as false-positive bug reports. The use of $F_{0.5}$-score is also consistent with prior work in bug report classification [24]. Table 8 summarizes the performance of the different strategies studied on the identification of false-positive bug reports.

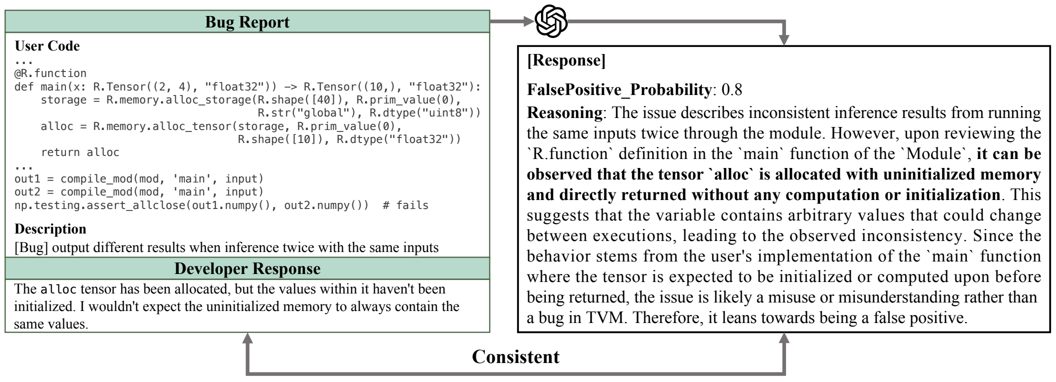
Finding 10: Prompting strategies that embed domain knowledge (i.e., root causes of false-positive bug reports) can improve the usability of LLM-based approaches in false-positive bug report classification. Table 8 shows that the two zero-shot strategies affirm the benefit of incorporating domain knowledge into the prompt. The enhanced zero-shot prompt markedly boosts recall compared to the basic zero-shot baseline, increasing it from 0.33 to 0.68. This indicates that even minimal compiler-specific context (e.g., hints about common false-positive root causes or pitfalls) enables GPT-4o to detect subtle indicators of false-positive bug reports more effectively. However, the enhanced zero-shot approach exhibits marginally lower precision than the basic prompt (0.55 versus 0.63), likely due to its broader context occasionally prompting the model to over-identify false-positive bug reports, including some false positives.

Table 8. Performance of Different Prompting Strategies on False-Positive Bug Reports Identification

Prompt Strategy	Compiler	Accuracy	Precision	Recall	F1-score	$F_{0.5}$-score
Basic Zero-shot	TVM	0.69	0.56	0.40	0.47	0.52
	OpenVINO	0.57	0.77	0.25	0.38	0.54
	Overall	0.64	0.63	0.33	0.43	0.53
Enhanced Zero-shot	TVM	0.64	0.47	**0.66**	**0.55**	0.50
	OpenVINO	0.65	0.65	**0.71**	**0.68**	0.66
	Overall	0.64	0.55	**0.68**	**0.61**	0.57
Few-shot	TVM	**0.76**	**0.85**	0.41	**0.55**	**0.70**
	OpenVINO	**0.67**	**0.86**	0.43	0.57	**0.72**
	Overall	**0.73**	**0.86**	0.42	0.56	**0.71**
CoT	TVM	0.67	0.55	0.38	0.45	0.50
	OpenVINO	0.63	0.68	0.55	0.61	0.65
	Overall	0.65	0.62	0.46	0.53	0.58
RAG	TVM	0.64	0.48	0.27	0.35	0.42
	OpenVINO	0.60	0.71	0.38	0.50	0.60
	Overall	0.62	0.59	0.32	0.41	0.50

Finding 11: Few-shot prompting substantially outperforms other strategies across most metrics, demonstrating the importance of contextual examples in LLM-based false-positive bug report classification. As highlighted in the table, few-shot prompting emerges as the most effective strategy, achieving the highest overall performance across both compilers, with an accuracy of 0.73, a precision of 0.86, and an $F_{0.5}$-score of 0.71. This reflects the significant benefits of including contextually relevant examples, enabling the model to better distinguish subtle patterns of false-positive reports. In contrast, enhanced zero-shot achieves the best recall and F1-score, particularly in OpenVINO, indicating its advantage in capturing a broader range of false-positive bug reports. However, this comes at the cost of precision, leading to more false positives.

Finding 12: The generally-studied step-wise reasoning (CoT) strategy limits LLM adaptability in domain-specific tasks, while the RAG strategy relies heavily on up-to-date and version-aligned documentation to ensure contextual accuracy. Simply adapting the general CoT and RAG strategies to our scenario yielded lower performance compared to simpler prompting strategies such as zero-shot and few-shot classification. For instance, the CoT strategy achieved lower precision and recall (0.62 and 0.46, respectively), potentially due to overly constrained reasoning that limits the model's flexibility in interpreting noisy or ambiguous bug reports. Similarly, the RAG strategy struggled to provide performance gains, likely due to challenges in retrieving contextually relevant and version-aligned documentation. In practice, many compiler bug reports emerge after users have exhausted available documentation, reducing the likelihood of useful retrieval hits. Moreover, mismatches between retrieved content and the specific compiler version often introduce distracting or misleading information. These findings do not undermine the potential of CoT and RAG, but rather reveal clear directions for future improvements. Enhanced CoT strategies that allow for more flexible or adaptive reasoning, and RAG pipelines that integrate version-aware filtering or structured retrieval from curated technical sources, could substantially boost effectiveness. Our results provide an early exploration of these approaches in a specialized domain, and we encourage further research into tailoring LLM-based reasoning and retrieval strategies for developer-facing tasks such as bug triage and compiler diagnosis.


Bug Report

User Code

```
...
@R.function
def main(x: R.Tensor((2, 4), "float32")) -> R.Tensor((10,), "float32"):
    storage = R.memory.alloc_storage(R.shape([40]), R.prim_value(0),
                                     R.str("global"), R.dtype("uint8"))
    alloc = R.memory.alloc_tensor(storage, R.prim_value(0),
                                  R.shape([10]), R.dtype("float32"))
    return alloc
...
out1 = compile_mod(mod, 'main', input)
out2 = compile_mod(mod, 'main', input)
np.testing.assert_allclose(out1.numpy(), out2.numpy())  # fails
```

Description

[Bug] output different results when inference twice with the same inputs

Developer Response

The `alloc` tensor has been allocated, but the values within it haven't been initialized. I wouldn't expect the uninitialized memory to always contain the same values.

[Response]

FalsePositive_Probability: 0.8

Reasoning: The issue describes inconsistent inference results from running the same inputs twice through the module. However, upon reviewing the \`R.function\` definition in the \`main\` function of the \`Module\`, **it can be observed that the tensor \`alloc\` is allocated with uninitialized memory and directly returned without any computation or initialization**. This suggests that the variable contains arbitrary values that could change between executions, leading to the observed inconsistency. Since the behavior stems from the user's implementation of the \`main\` function where the tensor is expected to be initialized or computed upon before being returned, the issue is likely a misuse or misunderstanding rather than a bug in TVM. Therefore, it leans towards being a false positive.

Fig. 12. Example of LLM Explanation

Finding 13: LLMs' dual utility in accurately classifying false-positive bug reports while generating human acceptable explanations, offering practical value for DL compiler issue triage, especially for the few-shot learning strategy. Our evaluation extended beyond classification accuracy to assess the explanatory capabilities of LLMs by examining the justifications generated for correctly identified false-positive bug reports. Specifically, we conducted a manual evaluation in which the first author reviewed the explanations produced by the few-shot prompting approach and assessed whether they were both plausible and factually accurate. Overall, 85.2% of these justifications were deemed satisfactory. Figure 12 presents a representative example. In this case[8], the user reported inconsistent inference outputs when running the same inputs twice. The LLM not only classified the issue as a likely false-positive bug report with high probability (0.8) but also provided a technically sound explanation: it correctly identified that the returned tensor was allocated from uninitialized memory in the `main` function, leading to nondeterministic outputs. This explanation is consistent with the official maintainer's follow-up comment, which attributed the behavior to undefined memory initialization rather than a genuine bug. Such outputs demonstrate how LLMs go beyond simple classification by offering actionable reasoning that helps developers quickly understand why a report is a false-positive bug report, thereby increasing trust and reducing manual effort.

Beyond the representative case in Figure 12, we observed that LLMs often provided useful explanations in other recurring scenarios. For instance, in some reports the model correctly pointed out that the failure was caused by invalid input shapes rather than a compiler bug; in others, it highlighted configuration or dependency issues (e.g., mismatched CUDA versions). These indicate that the explanatory ability of LLMs is not limited to one specific type of false-positive bug report but can generalize across diverse contexts, thereby further increasing their practical value for issue triage. At the same time, we also observed a small fraction of explanations that, while plausible, were overly generic (e.g., attributing failures to "user errors" without technical detail). Such observations suggest both the practical promise of LLM explanations in guiding developers during triage and the importance of future work to ensure faithfulness and avoid misleading rationales.

Data Leakage Analysis. To assess whether our results were affected by potential data leakage during model training, we conducted a controlled evaluation based on GPT-4o's official training cutoff date of October 1, 2023.[9] We divided the dataset into two subsets: (1) a *leak-prone* subset consisting of bug reports publicly available before the cutoff (totaling 915 entries), and (2) a *no-leak* subset containing 160 newer reports posted after the cutoff.

[8]https://github.com/apache/tvm/issues/17244
[9]https://platform.openai.com/docs/models/gpt-4o?snapshot=gpt-4o-2024-05-13

Table 9. Performance Comparison on Leak-Prone vs. No-Leak Subsets

Subset	Compiler	Accuracy	Precision	Recall	F1-score	$F_{0.5}$-score
Leak-Prone	TVM	0.76	0.82	0.37	0.51	0.66
	OpenVINO	0.67	0.85	0.47	0.61	0.73
	Overall	0.73	0.84	0.42	0.56	0.70
No-Leak	TVM	0.81	1.00	0.67	0.80	0.91
	OpenVINO	0.67	0.93	0.30	0.45	0.65
	Overall	0.73	0.97	0.46	0.62	0.79

Table 9 presents the comparative performance of the leak-prone and no-leak subsets under the few-shot prompting strategy. Interestingly, the subset of data with no risk of training-time leakage exhibited slightly better performance compared to the potentially leaked subset. Specifically, it achieves a precision of 0.97, an F1-score of 0.62, and an $F_{0.5}$-score of 0.79, compared to 0.84, 0.56, and 0.70 in the leak-prone set. This suggests that our results are unlikely to be the result of data leakage. One possible explanation is that the no-leak subset consists of more recent, consistently structured reports with clearer semantics, making them easier for LLMs to generalize to despite not having been seen during training. In contrast, the leak-prone subset contains older reports with more varied formats and ambiguous content, which may introduce noise. These findings support the robustness and generalizability of our LLM-based approach, regardless of training-time exposure.

RQ4 Summary

LLMs, especially under few-shot prompting, are effective in identifying false-positive bug reports in DL compilers, achieving a precision of 86% and a recall of 42%. Furthermore, the generated rationales for correctly classified cases are accurate in 85.2% of instances, highlighting LLMs' potential for transparent decision-making.

8 DISCUSSION

8.1 Implications

Our findings yield significant insights with practical implications for DL compiler repository maintainers, users, and researchers. Below, we outline the most salient recommendations emerging from our study.

(1) Repository Maintainers: (i) Automated Triage Tools. Findings 1-4 underscore the practical burden posed by false-positive bug reports across all stages of the DL compiler workflow. This highlights the need for automated triage tools, such as LLM-based classifiers, to help identify likely false-positive bug reports early and reduce unnecessary developer effort. (ii) Environment and Dependency Validation. Guided by Finding 2 and Finding 5, developers could integrate automated configuration validators and dependency conflict checkers directly into the toolchain, reducing the high prevalence of environment-related false-positive bug reports. (iii) Stage Coverage. In light of Finding 3 and Finding 4, such tools should not focus solely on early-stage issues like installation or configuration but should offer comprehensive stage-level coverage, especially in transformation and model loading stages, where false-positive bug reports are concentrated. Moreover, Finding 9 suggests that false-positive bug reports are likely to exhibit distinct, stage-specific root cause patterns. For instance, enhanced environment validation and setup documentation could mitigate early-stage false-positive bug reports, while clearer error messages and compatibility checks during model loading would help users distinguish unsupported features from genuine bugs. (iv) Robust Error Reporting. As highlighted by Finding 4 and Finding 8, developers should design error messages that not only indicate failure but also provide actionable guidance (e.g., suggesting compatible model versions or pointing to missing input validations), to minimize ambiguity and avoid user misinterpretation. (v) Enhancing Documentation Quality and Feature Transparency. As evident by Finding 6 and Finding 7, some false-positive bug reports arise from vague or incomplete documentation, particularly regarding unsupported features, version dependencies, or usage constraints. To mitigate this, projects should maintain version-aligned documentation and explicitly state assumptions and limitations. This also supports RAG-based tools, which rely on textual alignment between bug reports and documentation (Finding 12). (vi) Defining Bug versus False-Positive Bug Report. Our analysis reveals that different compiler communities may adopt different practices in classifying internal errors. For example, in some cases an internal compiler error is treated as a false-positive bug report if it stems from user misuse or a missing compilation option, whereas other communities may classify similar cases as bugs that require improved error handling. This inconsistency creates ambiguity for both users and researchers. We suggest that repository maintainers establish clearer criteria for distinguishing between false-positive bug reports and actual bugs, particularly for internal errors and unexpected crashes. Such definitions would not only improve transparency and reliability in issue management, but also help ensure that future datasets and research based on these reports can achieve greater consistency and validity.

(2) DL Compiler Users: Our analysis (Finding 6) shows that many false-positive bug reports originate from misunderstandings that could be resolved by consulting existing resources or improved documentation. To reduce redundancy and improve report quality, users are encouraged to perform pre-submission checks, consulting relevant documentation and searching existing issues before submitting a new bug report. Integrating documentation search or similar-issue retrieval tools into the issue submission interface may help catch common pitfalls early.

(3) Researchers: Our LLM-based experiments (RQ4, Finding 10-13) reveal that LLMs can assist in identifying false-positive bug reports, but their effectiveness hinges on prompt design and contextual grounding. (i) High-quality Example Pool. In our current setup, we selected four false-positive and four true bug reports that represent the major root causes observed in our dataset. However, these examples do not fully capture the diversity of subcategories or compiler stages. Future work could expand and refine the example pool, both in size and coverage, by incorporating more fine-grained root cause types and examples from a broader range of compiler stages. This would support better generalization and improve the robustness of LLM-based classification. (ii) Stage-aware Prompting. Stage-aware prompting could be leveraged to improve contextual relevance. Given the strong association between root causes and compilation stages (as shown in Finding 9), explicitly conditioning prompts on stage-level priors has the potential to enhance classification accuracy by aligning model reasoning with stage-specific patterns. (iii) Model Ensembles. Existing work [17] demonstrates that ensemble-after-inference and voting-based strategies tend to outperform individual models across a range of reasoning and code-related benchmarks. Researchers could employ multiple LLMs and aggregate their false-positive bug reports classification through voting or confidence-based ranking to mitigate inconsistencies observed in ambiguous cases, thereby offering a more robust and reliable decision-making mechanism. (iv) Structured Bug Reports. We observe that more recent bug reports are often written in a more structured format, including clear reproduction steps, environment details, and root-cause hints. Leveraging this structure before classification (e.g., by normalizing reports into structured fields and then feeding them to LLMs) may improve classification accuracy. This suggests a promising research direction for integrating automated report structuring with LLM-based classification.

8.2 Threat to Validity

Several threats to validity may affect the generalizability and robustness of our conclusions. We discuss the key threats below across three dimensions: external, internal, and construct validity.

The **external** threat to validity primarily stems from the studied DL compilers in our work. To mitigate this concern, we chose the two most widely adopted and diverse open-source DL compilers (i.e., TVM and OpenVINO) based on their GitHub repository popularity (measured by star count). These compilers vary in architecture and user base, allowing us to examine a broad range of error scenarios. Notably, our key findings remain consistent across both systems, indicating potential generalizability.

The **internal** threat to validity primarily arises from the manual labeling of DL compiler false-positive bug reports, which may introduce subjective bias or human error. To mitigate this risk, we limited our analysis to closed issues with clearly documented developer responses, enabling us to trace the resolution process and more accurately identify root causes. Furthermore, two researchers collaboratively performed all annotations, achieving near-perfect inter-rater agreement. Any disagreements were resolved through round-table discussions with a third experienced researcher, ensuring consistency and reliability in the labeling process. Another internal threat stems from the evaluation of LLM performance in identifying false-positive bug reports. The configuration of the LLM (e.g., model version, temperature, and prompt structure) may influence its performance and output consistency. While we fixed key parameters and used a consistent prompting framework across all experiments, some variation in output may still occur due to the inherent nondeterminism of LLMs, potentially impacting the reproducibility and interpretation of our results.

The **construct** threat to validity concerns whether our measurements accurately capture the nature of false-positive bug reports in DL compilers. First, our identification of false-positive bug reports relies on manual analysis of issue discussions and resolution traces. Although we relied on explicit developer feedback to identify false-positive bug reports, this process inherently reflects the maintainers' perspective and may not fully align with users' expectations. In particular, some reports marked as false-positive bug reports could still indicate poor error handling or ambiguous diagnostics. To mitigate this, we focused on closed issues with clear analysis and applied a consistent labeling process, but ambiguity in such cases remains a potential threat. Second, we introduced several metrics (e.g., First Response Time and Number of Comments) to estimate developers' effort in addressing false-positive bug reports. Although our metrics effectively approximate developer effort, they do not capture the specific roles or responsibilities of participants (e.g., maintainers vs. users). We explored a role-based split but found that only TVM provides explicit "Member" tags, whereas OpenVINO lacks comparable metadata, preventing consistent cross-project analysis. Therefore, a more fine-grained, role-based analysis remains a clear direction for future work.

9 RELATED WORK

9.1 Empirical Studies on Bugs in DL Systems and Compilers

There are a number of empirical studies on bugs in DL systems [15, 46, 55] and related works have also developed techniques for detecting bugs in DL systems [54, 57, 63, 65]. Among these, the work most closely related to ours is by Shen et al. [55], who conducted a comprehensive investigation of genuine bugs in DL compilers. Their study analyzed resolved issues in TVM, Glow, and nGraph, identifying root causes, failure symptoms, and bug locations. Beyond compilers, several studies have investigated bugs in DL frameworks. Chen et al. [15] performed a large-scale analysis of 1,000 bugs from four popular frameworks (Tensorflow, PyTorch, MXNet, and DL4J), categorizing their causes and evaluating the effectiveness of testing techniques. Jia et al. [32] explored the characteristics of TensorFlow bugs, examining their distribution and impact on different components. More recently, Jia et al. [33] extended this line of work by analyzing symptoms, causes, and repair patterns of bugs in DL libraries, providing actionable insights for improving debugging support. At the application level, Humbatova et al. [29] proposed a taxonomy of bugs in DL programs, classifying them into five top categories (i.e., model, tensors or inputs, training, GPU usage, and API). Islam et al. [31] and Zhang et al. [70] conducted empirical studies on DL programs built using frameworks such as TensorFlow, revealing common bug patterns and debugging challenges.

Similar empirical efforts have also been carried out in the domain of traditional compilers. Large-scale studies of GCC and LLVM have investigated compiler bugs in depth, examining their distributions, lifetimes, and repair complexity [59, 72]. These works show that compiler-specific concerns, such as optimization logic and language frontend design, heavily influence bug manifestations and triage. Other studies have explored how real-world compiler bugs are discovered and reported, often using testing or fuzzing techniques [67]. While informative, these studies typically concentrate on confirmed bugs, rather than false-positive bug reports.

In contrast to their research, our focus is on false-positive bug reports, those that fail to identify genuine bugs in DL compilers. We conducted an in-depth analysis of the root causes and stages at which false-positive bug reports occur. Additionally, based on the findings from this study, we devised an automated LLM-based false-positive bug reports identification tool to determine whether a reported bug represents a genuine bug in DL compilers.

9.2 Bug Report Classification Studies

Several studies have specifically addressed the classification of software bug reports, aiming to streamline bug report triage and maintenance efforts by distinguishing software bugs from other types of user-submitted issues, such as feature requests or general questions. Du et al.[22] conducted an empirical analysis on DL compiler bug reports, classifying them broadly into "Bug" and "Non-bug" categories, with the latter including feature requests, misunderstandings, and usage questions. Similarly, Herzig et al.[28] provided a foundational study on bug classification, exploring whether issues labeled as bugs in software repositories truly represent software bugs or other issue types, such as improvements or feature enhancements. In addition, researchers have proposed several automated bug report classification approaches. Zhou et al.[71] combine textual and structural features of bug reports to automatically identify and separate bug reports from feature requests with higher accuracy. Pandey et al. [49] utilized neural network-based models to automatically classify GitHub issue reports, achieving substantial improvement over traditional methods. More recently, LLM-based techniques have been adopted. For example, the study by Aracena et al. [14] introduced a GPT-based model that prioritizes and classifies issue reports with high accuracy under few-shot learning settings, achieving up to 93.2% precision and 95% recall in labeling issue types.

While these studies make significant progress in general issue classification, they mainly focus on distinguishing high-level categories such as bug reports, feature requests, and questions. Existing research has demonstrated the importance of bug report classification and shown promising results using automated (e.g., machine learning-based and LLM-based) approaches. However, these studies typically employ broad categorization schemes that do not specifically isolate false-positive bug reports. Unlike feature requests or general questions, false-positive bug reports represent a unique category where users mistakenly identify non-bugs as compiler bugs. These cases require distinct investigative efforts from developers, as they involve clarifying misconceptions rather than addressing system bugs. This distinction requires deeper semantic understanding of the report content, developer discussions, and resolution traces, making it substantially more complex than prior classification tasks. By explicitly defining and analyzing false-positive bug reports in the context of DL compilers, our study addresses this critical gap in the literature. This work complements prior research that has primarily focused on genuine compiler bugs or broader non-bug categories, providing a more nuanced understanding of developer challenges in handling user-reported issues.

9.3 DL Compiler Testing

Recent research has increasingly emphasized the detection of bugs in DL compilers. Shen et al. [56] developed OPERA, a migration-based test generation framework aimed at testing the model loading phase of DL compilers. To test the high-level IR transformation stage, Liu et al. [43] developed NNSmith, a grammar-based model generator, to produce diverse DL models, while Ma et al. [45] introduced a diversity-guided computational graph generator. Yang et al. designed Whitefox [66] focusing on generating optimization-aware tests. It leverages an analysis LLM to extract optimization knowledge from source code and uses generation LLMs to create tests. Additionally, Liu et al. [44] created Tzer, a tool that generates low-level IR through mutation to test the low-level IR transformation stage.

Unlike previous studies focused on bug detection in DL compilers, our work specifically addresses the critical challenge of classifying false-positive bug reports among reported bugs. This approach complements existing research while addressing a practical need: as fuzzers and users increasingly report potential bugs in large volumes, developers face growing time pressure to investigate these reports. To help alleviate this burden, we systematically evaluate four LLM-based techniques for automatically classifying false-positive bug reports and diagnosing their root causes. Our solution offers developers significant time savings while maintaining the rigor of manual investigation.

10 CONCLUSION

False-positive bug reports, though often overlooked, can incur substantial developer effort and delay genuine bug resolution in DL compiler maintenance. In this work, we present the first comprehensive empirical study on false-positive bug reports in DL compilers, covering a total of 1,075 bug reports from TVM and OpenVINO. Among these, 627 were identified as genuine bug reports, while 448 (41.67%) were labeled as false-positive bug reports based on manual inspection and resolution traces. Through systematic analysis, we identify their distribution across compilation stages, uncover root causes. Our findings reveal that most false-positive bug reports stem from Incorrect Environment Configuration, Incorrect Usage, and Misunderstanding of Features or Limitations. We further demonstrate that LLMs can effectively mitigate false-positive bug reports, especially under few-shot prompting strategies, with a precision of 86% and a recall of 42%. This study sheds light on the underexplored problem of false-positive bug reports in DL compilers, summarizes 13 findings, and provides actionable insights for building more robust diagnostic tools and support systems in DL compiler ecosystems.

ACKNOWLEDGMENT

This work was supported by the National Key Research and Development Program of China (Grant No. 2024YFB4506300), the National Natural Science Foundation of China (Grant Nos. 62322208, 12411530122, 62232001).

REFERENCES

[1] Accessed: 2025. Accelerated Linear Algebra (XLA). https://openxla.org/xla.
[2] Accessed: 2025. DLC False-Positive Bug Reports. https://github.com/Thrsu/DLC_False-positive_Bug_Reports.
[3] Accessed: 2025. Glow. https://ai.facebook.com/tools/glow/.
[4] Accessed: 2025. GPT-4o Benchmark – Detailed Comparison with Claude & Gemini. https://wielded.com/blog/gpt-4o-benchmark-detailed-comparison-with-claude-and-gemini.
[5] Accessed: 2025. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/.
[6] Accessed: 2025. nGraph. https://www.intel.com/content/www/us/en/artificial-intelligence/ngraph.html.

[7] Accessed: 2025. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt.

[8] Accessed: 2025. Open Neural Network Exchange (ONNX). https://onnx.ai/.

[9] Accessed: 2025. PyTorch. https://pytorch.org/.

[10] Accessed: 2025. RAGFlow. https://github.com/infiniflow/ragflow.

[11] Accessed: 2025. TensorFlow. https://www.tensorflow.org/.

[12] Accessed: 2025. TensorRT Open Source Software. https://github.com/NVIDIA/TensorRT.

[13] John Anvik, Lyndon Hiew, and Gail C Murphy. 2005. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*. 35–39.

[14] Gabriel Aracena, Kyle Luster, Fabio Santos, Igor Steinmacher, and Marco Aurelio Gerosa. 2024. Applying large language models to issue classification. In *Proceedings of the Third ACM/IEEE International Workshop on NL-based Software Engineering*. 57–60.

[15] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward understanding deep learning framework bugs. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–31.

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[17] Zhijun Chen, Jingzheng Li, Pengpeng Chen, Zhuoran Li, Kai Sun, Yuankai Luo, Qianren Mao, Dingqi Yang, Hailong Sun, and Philip S Yu. 2025. Harnessing multiple large language models: A survey on llm ensemble. *arXiv preprint arXiv:2502.18036* (2025).

[18] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin* 114, 3 (1993), 494.

[19] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[20] Juliet M Corbin and Anselm Strauss. 1990. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative sociology* 13, 1 (1990), 3–21.

[21] Xueying Du, Geng Zheng, Kaixin Wang, Jiayi Feng, Wentai Deng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2024. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *arXiv preprint arXiv:2406.11147* (2024).

[22] Xiaoting Du, Zheng Zheng, Lei Ma, and Jianjun Zhao. 2021. An empirical study on common bugs in deep learning compilers. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 184–195.

[23] Yuanrui Fan, Xin Xia, David Lo, and Ahmed E Hassan. 2018. Chaff from the wheat: Characterizing and determining valid bug reports. *IEEE transactions on software engineering* 46, 5 (2018), 495–525.

[24] Fan Fang, John Wu, Yanyan Li, Xin Ye, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. 2021. On the classification of bug reports to improve bug localization. *Soft Computing* 25 (2021), 7307–7323.

[25] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* 2, 1 (2023).

[26] Yury Gorbachev, Mikhail Fedorov, Iliya Slavutin, Artyom Tugarev, Marat Fatekhov, and Yaroslav Tarkan. 2019. Openvino deep learning workbench: Comprehensive analysis and tuning of neural networks inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*. 0–0.

[27] Jianjun He, Ling Xu, Yuanrui Fan, Zhou Xu, Meng Yan, and Yan Lei. 2020. Deep learning based valid bug reports determination and explanation. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 184–194.

[28] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 392–401.

[29] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 1110–1121.

[30] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[31] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 510–520.

[32] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An empirical study on bugs inside tensorflow. In *Database Systems for Advanced Applications: 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24–27, 2020, Proceedings, Part I 25*. Springer, 604–620.

[33] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2021. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software* 177 (2021), 110935.

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

[35] Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, et al. 2022. Language models (mostly) know what they know. *arXiv preprint arXiv:2207.05221* (2022).

[36] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[37] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

[38] Hiroki Kuramoto, Dong Wang, Masanari Kondo, Yutaro Kashiwa, Yasutaka Kamei, and Naoyasu Ubayashi. 2024. Understanding the characteristics and the role of visual issue reports. *Empirical Software Engineering* 29, 4 (2024), 89.

[39] Muhammad Laiq, Nauman bin Ali, Jürgen Börstler, and Emelie Engström. 2024. Industrial adoption of machine learning techniques for early identification of invalid bug reports. *Empirical Software Engineering* 29, 5 (2024), 130.

[40] Muhammad Laiq, Nauman bin Ali, Jürgen Böstler, and Emelie Engström. 2022. Early identification of invalid bug reports in industrial settings–a case study. In *International Conference on Product-Focused Software Process Improvement*. Springer, 497–507.

[41] Muhammad Laiq, Nauman bin Ali, Jürgen Börstler, and Emelie Engström. 2023. A data-driven approach for understanding invalid bug reports: An industrial case study. *Information and Software Technology* 164 (2023), 107305.

[42] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

[43] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 530–543. https://doi.org/10.1145/3575693.3575707

[44] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–26.

[45] Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2023. Fuzzing deep learning compilers with hirgen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 248–260.

[46] Haoyang Ma, Wuqi Zhang, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2024. Towards Understanding the Bugs in Solidity Compiler. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1312–1324.

[47] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[48] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010), 1–1.

[49] Nitish Pandey, Debarshi Kumar Sanyal, Abir Hudait, and Amitava Sen. 2017. Automated classification of software issue reports using machine learning techniques: an empirical study. *Innovations in Systems and Software Engineering* 13 (2017), 279–297.

[50] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[51] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys. In *annual meeting of the Florida Association of Institutional Research*, Vol. 177.

[52] Hendrig Sellik, Onno Van Paridon, Georgios Gousios, and Maurício Aniche. 2021. Learning off-by-one mistakes: An empirical study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 58–67.

[53] Sakib Shahriar, Brady D Lund, Nishith Reddy Mannuru, Muhammad Arbab Arshad, Kadhim Hayawi, Ravi Varma Kumar Bevara, Aashrith Mannuru, and Laiba Batool. 2024. Putting gpt-4o to the sword: A comprehensive evaluation of language, vision, speech, and multimodal proficiency. *Applied Sciences* 14, 17 (2024), 7782.

[54] Qingchao Shen, Junjie Chen, Jie M Zhang, Haoyu Wang, Shuang Liu, and Menghan Tian. 2022. Natural test generation for precise testing of question answering software. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[55] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 968–980.

[56] Qingchao Shen, Yongqiang Tian, Haoyang Ma, Junjie Chen, Lili Huang, Ruifeng Fu, Shing-Chi Cheung, and Zan Wang. 2025. A Tale of Two DL Cities: When Library Tests Meet Compiler. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2201–2212.

[57] Qingchao Shen, Zan Wang, Haoyang Ma, Yongqiang Tian, Lili Huang, Zibo Xiao, Junjie Chen, and Shing-Chi Cheung. 2026. Optimization-Aware Test Generation for Deep Learning Compilers. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE)*. ACM, to appear.

[58] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[59] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th international symposium on software testing and analysis*. 294–305.

[60] Jian Sun. 2011. Why are bug reports invalid?. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 407–410.

[61] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[62] Dong Wang, Masanari Kondo, Yasutaka Kamei, Raula Gaikovina Kula, and Naoyasu Ubayashi. 2023. When conversations turn into work: a taxonomy of converted discussions and issues in GitHub. *Empirical Software Engineering* 28, 6 (2023), 138.

[63] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 788–799.

[64] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[65] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing numerical bugs in deep learning via gradient back-propagation. In *Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 627–638.

[66] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2023. White-box compiler fuzzing empowered by large language models. *arXiv preprint arXiv:2310.15991* (2023).

[67] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[68] Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. 2013. Categorizing bugs with social networks: a case study on four open source software communities. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1032–1041.

[69] Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E Hassan. 2012. An empirical study on factors impacting bug fixing time. In *2012 19th Working conference on reverse engineering*. IEEE, 225–234.

[70] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 129–140.

[71] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. 2016. Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process* 28, 3 (2016), 150–176.

[72] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021), 110884.