

DepRadar: Agentic Coordination for Context-Aware Defect Impact Analysis in Deep Learning Libraries

Yi Gao

The State Key Laboratory of
Blockchain and Data Security,
Zhejiang University
Hangzhou, China
gaoyi01@zju.edu.cn

Xing Hu*

The State Key Laboratory of
Blockchain and Data Security,
Zhejiang University
Hangzhou, China
xinghu@zju.edu.cn

Tongtong Xu

Huawei
Hangzhou, China
xutongtong9@huawei.com

Jiali Zhao

Huawei
Hangzhou, China
zhaojiali3@huawei.com

Xiaohu Yang

The State Key Laboratory of
Blockchain and Data Security,
Zhejiang University
Hangzhou, China
yangxh@zju.edu.cn

Xin Xia

The State Key Laboratory of
Blockchain and Data Security,
Zhejiang University
Hangzhou, China
xin.xia@acm.org

Abstract

Deep learning (DL) libraries like Transformers and Megatron are now widely adopted in modern AI programs. However, when these libraries introduce defects—ranging from silent computation errors to subtle performance regressions—it is often challenging for downstream users to assess whether their own programs are affected. Such impact analysis requires not only understanding the defect semantics but also checking whether the client code satisfies complex triggering conditions involving configuration flags, runtime environments, and indirect API usage. We present DEPRADAR, an agent coordination framework for fine-grained defect and impact analysis in DL library updates. DEPRADAR coordinates four specialized agents across three steps: (1) the *PR Miner* and *Code Diff Analyzer* extract structured defect semantics from commits or pull requests, (2) the *Orchestrator Agent* synthesizes these signals into a unified defect pattern with trigger conditions, and (3) the *Impact Analyzer* checks downstream programs to determine whether the defect can be triggered. To improve accuracy and explainability, DEPRADAR integrates static analysis with DL-specific domain rules for defect reasoning and client-side tracing. We evaluate DEPRADAR on 157 PRs and 70 commits across two representative DL libraries. It achieves 90% precision in defect identification and generates high-quality structured fields (average field score 1.6/2). On 122 client programs, DEPRADAR identifies affected cases with 90% recall and 80% precision, substantially outperforming other baselines.

CCS Concepts

- Software and its engineering → Software libraries and repositories.

*Corresponding Author



This work is licensed under a Creative Commons Attribution 4.0 International License.

ICSE '26, Rio de Janeiro, Brazil

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2025-3/2026/04

<https://doi.org/10.1145/3744916.3787763>

Keywords

Multi-Agent Framework, Pull Request, Defect Impact Analysis

ACM Reference Format:

Yi Gao, Xing Hu, Tongtong Xu, Jiali Zhao, Xiaohu Yang, and Xin Xia. 2026. DepRadar: Agentic Coordination for Context-Aware Defect Impact Analysis in Deep Learning Libraries. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3744916.3787763>

1 Introduction

Modern deep learning (DL) programs increasingly depend on libraries such as Transformers [18] and Megatron [5], which streamline model development, training, and deployment [21, 23, 36, 38, 39, 45, 50, 52, 54, 57, 60]. These libraries encapsulate critical functionalities such as tokenization, optimizer scheduling, distributed communication, and memory management, enabling developers to build large-scale models efficiently.

However, as these libraries grow in complexity and scale, they unavoidably introduce software defects. Bugs in DL libraries may not always manifest as program crashes; instead, they often degrade performance silently, introduce numerical inconsistencies, or corrupt training dynamics under certain conditions [23, 59]. For example, a subtle error in tensor parallelism coordination or configuration parsing may lead to incorrect gradient updates or inefficient resource usage, ultimately impairing model convergence. In practice, such defects are often discussed in GitHub pull requests (PRs) [16] or commits [4], where developers propose fixes based on user feedback or internal audits [33, 58]. Thus, we can analyze PRs and commits to assess whether a defect impacts downstream client programs. However, this is highly non-trivial in practice: PRs may reference internal APIs, abstract trigger conditions, or rely on complex runtime configurations. Meanwhile, downstream users may not closely monitor upstream libraries and are frequently unaware of whether such defects could impact their own programs.

Although keeping dependencies up-to-date is a common practice for maintaining software reliability, frequent dependency upgrades are impractical in DL environments, as each update may trigger

costly retraining and break compatibility [24, 37, 41, 48]. Moreover, many defects remain silent across releases or are fixed with minimal documentation. Release notes are typically coarse (e.g., fix stability) and omit critical details such as affected modules, configurations, or triggering conditions. For instance, commit 1674ce3 [7] in *Megatron* silently skewed multi-user training for months before detection, illustrating how such fixes can propagate unnoticed.

Recent efforts have leveraged large language models (LLMs) to assist with code summarization and PR triage [44]. However, these approaches typically treat commits or patches in isolation, without modeling the downstream impact on external client programs. Determining whether a DL library defect affects a specific client program involves several unique challenges:

Challenge 1: Noisy and informal defect representations. Defect-related PRs or commits often contain unstructured descriptions, partial fixes, or discussion noise. The root cause and triggering conditions are rarely formalized, making the automated extraction of defect patterns difficult.

Challenge 2: Deep semantic gap between library changes and user-visible behavior. Many patches fix low-level internal logic (e.g., CUDA graphs, communication buffers), while user programs interact only with high-level APIs. Mapping these low-level changes to user-facing risk factors (e.g., configuration parameters, method calls) requires semantic reasoning.

Challenge 3: Semantic alignment between defect conditions and client usage. Determining whether a client program satisfies the defect’s triggering conditions requires understanding not only the patched library logic, but also the usage context in the user code—including configurations and function calls.

To address these challenges, we propose DEPRADAR, an agent coordination framework for automated defect and impact analysis in deep learning libraries. DEPRADAR assists developers in determining whether a defect in an upstream DL library (e.g., Transformers) has concrete effects on their downstream programs. Unlike prior approaches that rely on static dependency resolution or flat summarization, DEPRADAR leverages a modular agent architecture to reason across both the library side and the client side, enabling context-aware, condition-sensitive impact assessment. By integrating domain-specific coordination rules, agent self-adaptation, and AST-based structural validation, DEPRADAR turns generic multi-agent reasoning into a practical and verifiable workflow for DL defect analysis. DEPRADAR consists of four agents: (1) **Miner Agent** identifies whether a commit or pull request addresses a real defect and extracts structured defect metadata, including bug background, trigger conditions, and affected components. (2) **Code Diff Agent** analyzes the patch semantics, locating modified methods and summarizing the underlying fix logic. (3) **Orchestrator Agent** synthesizes the results from upstream agents into a defect pattern, capturing risk-relevant method calls and parameter combinations from the perspective of external users. (4) **Impact Analyzer Agent** performs contextual analysis on the client program code to determine whether it satisfies the triggering conditions, guided by static analysis and domain-specific validation rules.

To mitigate the long-context limitations of LLMs, DEPRADAR incorporates a progressive context augmentation mechanism that dynamically expands or compresses the input prompt to ensure

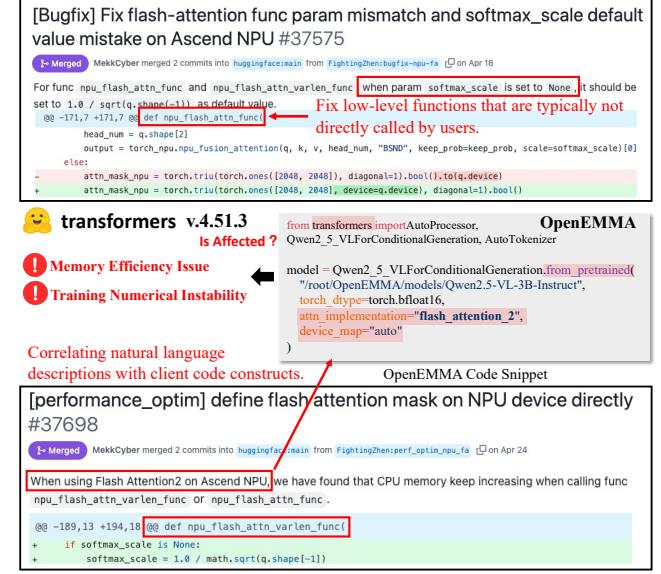


Figure 1: Motivation Example.

information completeness without exceeding token limits. Furthermore, to mitigate hallucinations in LLM-generated outputs, we design a post-processing validation module that uses AST-based static analysis to verify the existence of referenced parameters and methods within the actual client program.

We evaluate DEPRADAR on 157 PRs and 70 commits collected from two widely used DL libraries: Transformers and Megatron. Our study focuses on two core tasks: (1) whether DEPRADAR can accurately generate structured *defect patterns*, and (2) whether it can determine if a downstream client program is semantically affected by a given defect. On the PR dataset, DEPRADAR achieves a defect classification F1-score of 95%, and over 70% of extracted defect pattern fields—such as bug background and trigger conditions—are rated fully accurate by domain experts. In downstream impact analysis across 122 real-world client programs, DEPRADAR achieves an F1-score of 85%, substantially outperforming three strong baselines. We further evaluate DEPRADAR on internal commits of Megatron, where it successfully identifies 12 real-world client programs affected by silent defects, as confirmed by developers.

In summary, the main contributions of this paper include:

- We present DEPRADAR, a novel agent coordination framework for fine-grained defect and impact analysis in deep learning libraries.
- We implement a practical system that integrates LLM-based agents with static analysis and domain-specific inference rules. The replication package is available at [20].
- We conduct an extensive evaluation of 227 library updates (157 PRs and 70 commits) from Transformers and Megatron. Results show that DEPRADAR outperforms three strong baselines in both defect modeling (F1 = 95%) and client impact analysis (F1 = 85%), and identifies real affected clients.

2 Motivation Example

Given the high cost of silent correctness or performance regressions during model training, automatically identifying library defects and

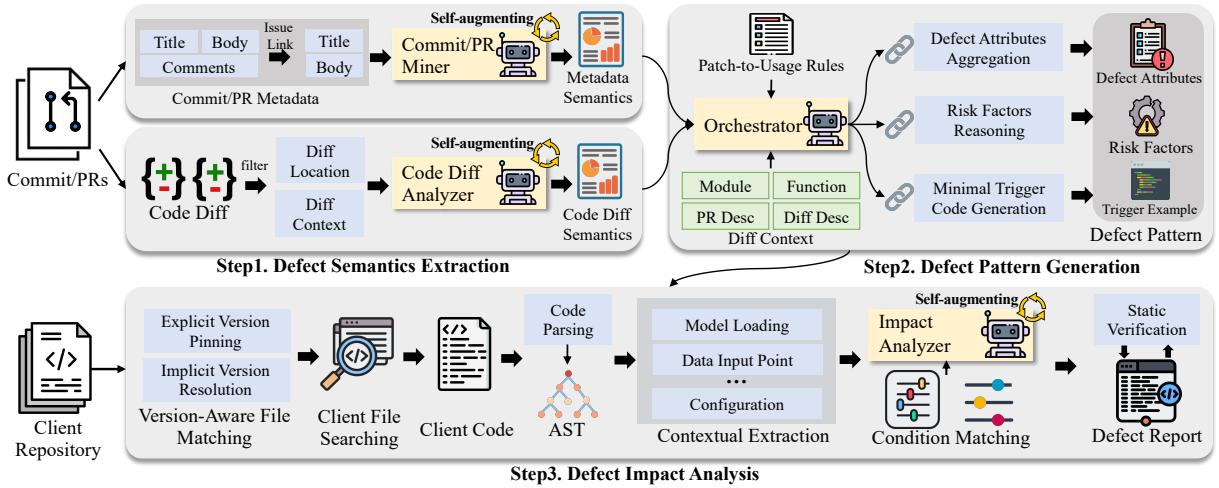


Figure 2: Overview of our approach.

assessing their potential impact on downstream client programs has become an urgent and practical need. Real-world deep learning libraries like Transformers frequently fix silent defects—bugs that do not crash programs but silently degrade numerical accuracy or runtime efficiency under specific conditions. However, client developers often lack the visibility and expertise to determine whether such internal fixes are relevant to their own configurations.

Figure 1 shows this challenge through two representative PRs from Transformers, both affecting Flash Attention on Ascend NPUs: PR #37575 fixes a subtle mistake in the default value of the softmax_scale parameter. If a user enables Flash Attention v2 (`attn_implementation="flash_attention_2"`) without explicitly setting softmax_scale, the resulting attention output may become numerically unstable, harming model quality. PR #37698 resolves a memory inefficiency where the attention mask is first initialized on the CPU and then moved to the NPU. This silent performance defect causes unnecessary CPU memory allocation and long-term accumulation in certain deployment settings.

The middle part of Figure 1 shows a simplified snippet from a downstream client project (*OpenEMMA, an open-source autonomous driving framework*) that uses Transformers with Flash Attention v2 and NPU hardware—satisfying the trigger conditions of both PRs. However, neither bug manifests through obvious exceptions or logs, and the developers were unaware of their presence before manual inspection. Such usage patterns, which involve using pre-trained configurations and enabling hardware-optimized features, are commonly seen in downstream DL programs and can make silent defects difficult to detect.

This example highlights a fundamental gap in current DL library usage: developers cannot feasibly monitor every upstream change or PR, nor can they easily determine if a specific patch is relevant to their setup. Furthermore, existing tools either focus on syntax-level patch analysis or general-purpose summarization, lacking the semantic reasoning and domain knowledge needed to bridge the gap between internal bug fixes and client-side manifestations. These observations motivate our work. We propose a novel approach

named DEPRADAR, analyzes library-level bug-fix PRs and automatically traces their potential runtime impact on client programs. By combining defect pattern modeling, trigger condition extraction, and client-side reasoning, our framework aims to assist users in identifying whether they are silently affected.

3 Approach

We propose DEPRADAR, an agent coordination framework for automated defect impact analysis in DL libraries. As shown in Figure 2, DEPRADAR consists of three steps: (1) extracting structured defect semantics from PRs/commits via dual agents, (2) synthesizing complete defect patterns, and (3) verifying whether clients satisfy the triggering conditions. The framework is built on AutoGen [1] to enable coordinated, multi-turn reasoning across agents.

3.1 Defect Semantics Extraction

This step involves two specialized agents that collaboratively extract defect semantics from different perspectives: the *Commit/PR Miner Agent* processes natural language descriptions (e.g., commit messages, PR discussions), while the *Code Diff Analyzer Agent* focuses on structural code changes. To enable multi-agent coordination, we introduce an *Orchestrator Agent* that validates intermediate outputs and synthesizes final defect patterns by combining upstream results. In this step, the **Orchestrator Agent** monitors the completeness of extracted semantics.

3.1.1 Commit/PR Miner Agent. The Miner Agent is designed to extract structured defect representations from noisy and loosely formatted PR or commit metadata. Given a PR, DEPRADAR processes the title, body, associated issue references, and developer comments to generate a structured defect semantic representation containing three fields: `bug_background`, `impact_scope`, and `trigger_conditions`.

Considering the substantial number of PRs—most of which involve routine updates or feature additions rather than defect fixes, comprehensive analysis of all PRs proves inefficient. To address this, we implement a two-step filtering mechanism. The Miner Agent

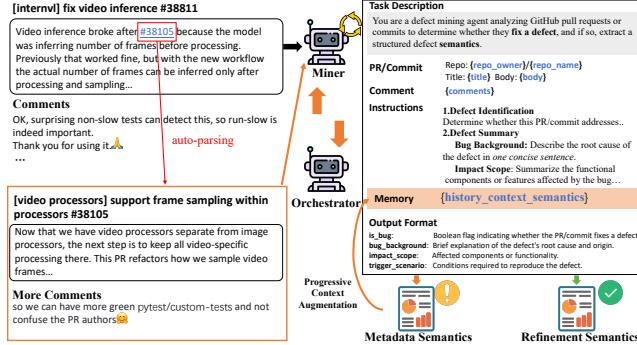


Figure 3: Prompt design for the Miner Agent: mining defect patterns from unstructured PR metadata for PR #38811 and associated PR #38105.

integrates the GitHub API as a tool to retrieve full metadata for each PR. It then applies keyword-based heuristics to retain candidates containing defect-related cues (e.g., *fix*, *bug*, *error*) while excluding clearly benign updates (e.g., *doc*, *refactor*, *feat*). To ensure reliability, we manually inspected all excluded PRs in this period and did not find missed defect-fixing PRs. The full keyword list and filtering script are released in the reproduction package for transparency [20].

Progressive Context Augmentation. To handle noisy, lengthy, or fragmented PR metadata, DEPRADAR adopts a progressive context augmentation strategy coordinated by the Orchestrator Agent. As illustrated in Figure 3, the Miner Agent processes PR content in fixed-size chunks (default $k = 3$ items per round) and generates intermediate summaries capturing partial semantics such as bug background or triggering conditions. After each round, the Orchestrator Agent validates the content completeness of the extracted fields (e.g., *trigger_conditions*, *bug_background*). If the output is insufficient, the Orchestrator Agent will provide feedback to the Miner Agent. The previous semantic representation is preserved as historical context and augmented with the next chunk of PR content. This iterative refinement continues until all required fields are confidently populated or the full PR metadata is exhausted. This mechanism ensures that the agent remains within prompt length limits while gradually expanding coverage, enabling robust reasoning over scattered or delayed defect signals.

For example, in PR #38811 as shown in Figure 3, although the title reported a *video inference failure* without disclosing root causes or triggering conditions, the Miner Agent coordinated by the Orchestrator—automatically followed linked PR #38105 and developer comments to retrieve additional context. Through progressive context augmentation, DEPRADAR accurately reconstructed the defect pattern, identifying the underlying cause as a frame sampling misalignment in multi-frame video inference pipelines and extracting the specific conditions under which the bug is triggered.

3.1.2 Code Diff Analyzer Agent. While PR metadata offers surface-level explanations, the actual bug fix logic is encoded within the code changes. The Code Diff Agent analyzes these changes to identify repair intent and recover trigger conditions from the changes. Given a modified patch, we first apply a denoising phase using AST-level parsing to eliminate non-functional edits such as comment

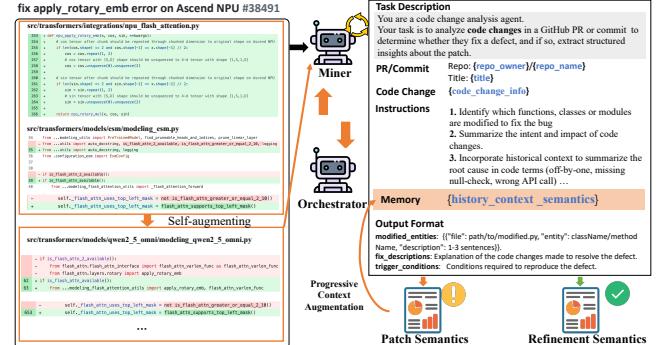


Figure 4: Prompt design for the Code Diff Agent: summarizing patch semantics for PR #38491.

updates, formatting, or import reordering. We retain only structural code changes (e.g., control-flow updates, function/method additions, and validation logic) for downstream analysis.

As shown in Figure 4, the CodeDiff Agent analyzes the code-level patch to extract structured defect semantics from the perspective of program edits. Specifically, it captures both the surface fix operations and their underlying rationale, producing a dual-perspective output that complements textual representations from the Miner Agent. The extracted semantics consist of three key components: (1) modified code entities annotated with semantic change types (e.g., *added null check*, *modified tensor shape validation*), (2) a synthesized explanation of the root cause underlying the defect (e.g., *missing boundary check*), and (3) a set of key program elements—such as parameters and method names—that are relevant to the defect’s trigger conditions. These elements are subsequently used as search keys in the downstream client-side code matching.

Progressive Context Augmentation. To process large or multi-file patches where code diffs exceed the input limit of a single LLM prompt, DEPRADAR adopts a progressive augmentation strategy similar to that of the Miner Agent. As shown in Figure 4, the CodeDiff Agent segments the patch into logical units (e.g., grouped by file) and analyzes $k = 3$ edits per round. At each step, the agent generates intermediate summaries that describe fix actions (e.g., API adjustments, parameter constraints) and defect-related semantics.

The Orchestrator Agent then validates whether key semantic fields—such as *fix_description*, *trigger_conditions* are sufficiently captured. If not, the next round’s input includes both new code chunks and the prior semantics. This iterative feedback loop continues until either all semantic slots are confidently filled or the full patch is exhausted. This feedback loop continues until all required semantics are covered or the patch is fully processed.

3.2 Defect Pattern Generation

At this step, the Orchestrator combines outputs from prior agents and applies domain rules to generate a structured defect pattern for impact analysis. Prior work on bug summarization often stops at generating human-readable descriptions or identifying changed APIs [44, 58]. However, this is insufficient for determining downstream impact. Many bugs involve internal implementation details or latent behaviors (e.g., device-specific logic) whose consequences are only visible when mapped to concrete usage patterns in clients.

Table 1: Domain-Specific Rules for Defect Pattern Inference.

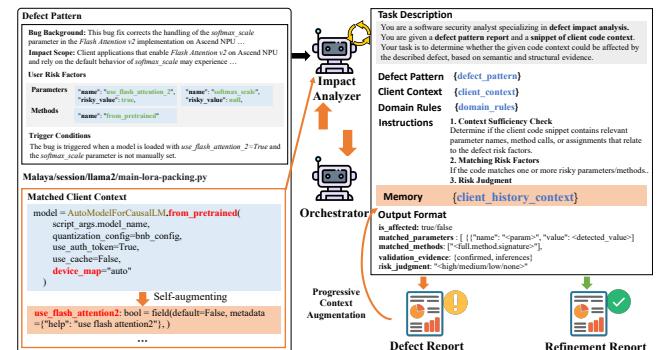
Rule Type	Description and Example
Entity Lift-ing	Promote internal symbols (e.g., <code>_flash_kernel</code>) to nearest user-facing API (e.g., <code>generate()</code>) based on call graph or naming patterns.
Parameter Exposure	Include only parameters visible in public constructors or config (e.g., <code>attention_impl</code>).
Trigger Synthesis	Combine condition fragments (e.g., hardware flags, dtype checks) into a minimal executable client snippet. Example: <code>map device_mesh != None to device_map="auto"</code> .
Semantic Merge	When multiple triggers appear (e.g., backend flag + precision), enforce conjunction to avoid partial matches.

The output follows a structured schema that captures bug-fix semantics from a user-facing perspective. It contains: (1) Defect Attributes: Whether the patch fixes a bug, its background, and impact. (2) Risk Factors: Externally visible method names and parameters that can trigger the bug. (3) Minimal Trigger Example: A synthesized code snippet illustrating the minimal condition required to activate the bug using client-facing APIs. The Orchestrator Agent integrates upstream outputs and applies rule-based reasoning to transform internal fixes into client-relevant risk factors.

To bridge the semantic gap between low-level fixes in patches and user-facing patterns, we design a domain-informed inference layer guided by a configurable set of rules. Defect fixes in DL libraries often modify internal methods that users rarely invoke directly; therefore, mapping internal changes to the minimal user-visible trigger scenario is essential for impact analysis. Our rules steer the Agent to reason beyond raw diffs, identify exposed parameters or configuration keys, and synthesize the smallest usage example that can reproduce the defect in a client context. These rules are implemented as declarative templates stored in an external configuration file, making them easy to extend for additional libraries or edge cases. Table 1 illustrates representative rule types and their objectives; the full specification and domain-specific variations (e.g., for distributed training flags) are included in our artifact [20].

Figure 5 shows the defect pattern generated for PR #37575. This patch fixes an internal bug in Ascend NPU’s Flash Attention implementation, where setting `softmax_scale=None` in functions such as `npu_flash_attn_func` and `npu_flash_attn_varlen_func` leads to incorrect attention scaling. Although these changes modify low-level device-specific logic, the Orchestrator Agent infers that users who enable Flash Attention v2 without explicitly setting `softmax_scale` are silently affected. Moreover, the generated `minimal_client_example` captures a representative code snippet that satisfies the trigger conditions. In the subsequent analysis step, this example serves as a search anchor: static analyzers extract parameter names and API calls from the snippet to match against client code contexts. Overall, the structured defect pattern enables downstream tasks including parameter matching, condition validation, and precise retrieval of affected user code segments.

Bug Background	Is Bug: True
This bug fix corrects the handling of the <code>softmax_scale</code> parameter in the <code>Flash Attention v2</code> implementation on Ascend NPU. When users do not explicitly set <code>softmax_scale</code> , the internal logic incorrectly calculates its default value, potentially leading to inaccurate attention scores.	
Impact Scope	
Client applications that enable <code>Flash Attention v2</code> on Ascend NPU and rely on the default behavior of <code>softmax_scale</code> may experience degraded model performance due to incorrect attention score scaling.	
User Risk Factors	
Parameters	<code>"name": "use_flash_attention_2", "risky_value": true, "match_type": "confirmed"</code>
Methods	<code>"name": "from_pretrained", "match_type": "confirmed"</code>
Trigger Conditions	
The bug is triggered when a model is loaded with <code>use_flash_attention_2=True</code> and the <code>softmax_scale</code> parameter is not manually set.	
Minimal Client Example	
<pre>from transformers import AutoModelForCausalLM, AutoTokenizer model = AutoModelForCausalLM.from_pretrained("model-id", device_map="auto", use_flash_attention_2=True # softmax_scale not set)</pre>	

Figure 5: Defect Pattern example synthesized for PR #37575.**Figure 6: Prompt design for the Impact Analyzer Agent: verifying trigger condition satisfaction from matched client code.**

3.3 Client-Side Defect Impact Analysis

In the final step, DEPRADAR employs an Impact Analyzer Agent to perform semantic alignment and context-aware reasoning, determining whether the client is affected by the defect.

3.3.1 Version-Aware File Matching. The first prerequisite for reliable impact analysis is verifying whether the client depends on a version of the DL library that contains the defect. In Python, this is non-trivial due to varied dependency declarations across pip requirements or manually installed snapshots. When a version is explicitly pinned (e.g., `transformers==4.30.2`), DEPRADAR aligns the PR/commit-level file changes with the target version via conservative AST-based subtree matching.

However, many projects omit strict versioning (e.g., `transformers>=4.0`), making resolution uncertain. In such cases, DEPRADAR conservatively assumes potential exposure if the defect existed prior to the client’s runtime or install date—reflecting how pip typically resolves to the latest version. This prevents false negatives in real deployments, where latent defects may propagate silently through

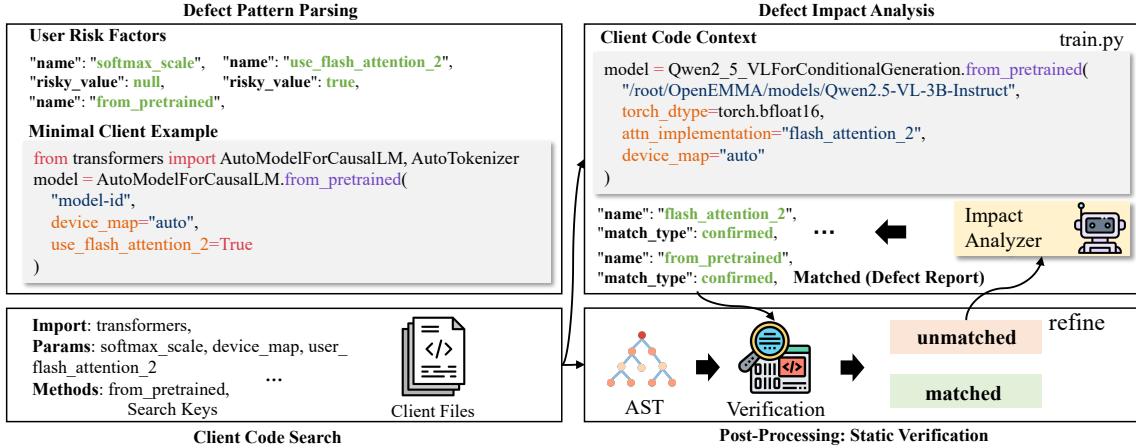


Figure 7: Post-processing and verification: combining impact reports with AST analysis to validate methods and parameters.

frequent updates. By this strategy, DEPRADAR automatically determines whether the client is likely to include the defect and continues to the impact analysis only when risk cannot be ruled out.

3.3.2 Contextual Searching via Syntax-Aware Extraction. Given the defect pattern extracted in Step 2, DEPRADAR traverses the entire client code and constructs ASTs for each Python file. It then searches for occurrences of risk factors, such as specific parameter names, method calls, or triggering constants defined in the pattern.

Instead of analyzing the entire file, DEPRADAR extracts the minimal AST subtree that encloses each matched element (e.g., a model constructor or loop-level invocation). This localized context is then converted back to source code and passed to the Impact Analyzer Agent. This syntax-aware extraction serves two purposes: (1) it preserves semantic locality by capturing only the immediate usage scenario, and (2) it reduces prompt size and noise, enabling focused reasoning. All extracted code regions are aggregated into a candidate context list for downstream defect impact analysis.

Assessing Trigger Conditions. For each matched client context, the Impact Analyzer Agent determines whether the conditions required to activate the defect are satisfied. Figure 6 illustrates the prompt used for this analysis. This includes checking the presence and values of risk-bearing parameters (e.g., `attnImplementation = "flash_attention_2"`) and verifying that corresponding configurations are active in scope.

To handle variation in code style and naming, we incorporate domain-adapted prompting strategies that normalize synonyms (e.g., `enable_flash_attn_v2` vs `use_flash_attention2`) and allow flexible argument ordering. If explicit conditions (e.g., use of GPU) are not contradicted in code, the agent conservatively assumes they hold. This reasoning enables precise filtering: contexts that satisfy all trigger conditions proceed to reporting, while ambiguous or missing evidence leads to conservative rejection or re-analysis.

Progressive Context Augmentation. Not all matched client contexts are sufficient for reliable impact assessment. Similar to earlier agents, we adopt an adaptive multi-round inference loop coordinated by the Orchestrator Agent. Each round starts with the minimal AST subtree containing the matched parameter or call. If the agent's reasoning yields incomplete or uncertain results—due to

ambiguous control flow, shadowed variables, or overridden parameters—the Orchestrator expands the context upward along the AST, from the current node to enclosing blocks, functions, or classes. All intermediate outputs are retained in memory for iterative refinement. This process continues until either a confident conclusion is reached or no broader context remains. It ensures both token efficiency and semantic completeness across diverse code structures.

3.3.3 Post-processing and Verification. To mitigate the hallucinations commonly observed in LLM-generated predictions, we perform static validation by extracting predicted parameters and method calls (e.g., `softmax_scale=None`) from the LLM output and verifying their presence in the client code. These predicted indicators are existence-checked against the client's source code using ASTs constructed via Python's built-in `ast` module. As illustrated in Figure 7, the DEPRADAR checks whether each referenced element in the report—such as `use_flash_attention_2` or specific API calls—can be confirmed by traversing the AST of the matched client context. If any required evidence is missing, the orchestrator triggers a corrective re-analysis. If the evidence remains insufficient after all retries, DEPRADAR conservatively marks the client as unaffected to avoid over-reporting. This conservative strategy prioritizes precision by reducing false positives, ensuring that downstream impact is reported only when strongly supported by code evidence.

3.3.4 Impact Report Generation. The final output of this step is a structured *Impact Report*, which includes: (1) a binary `is_affected` flag, (2) the set of matched parameters and method calls contributing to the judgment, and (3) a justification summary with both reasoning trace and static evidence. These reports are not only valuable for informing users of downstream risks, but also serve as input to potential mitigation strategies, such as patch recommendation or warning generation during CI workflows.

4 Evaluation

4.1 Experimental Settings

Target Libraries. We evaluate DEPRADAR on two representative DL libraries: *Transformers* [18] and *Megatron* [5]. *Transformers*

is a widely used, community-maintained framework for general-purpose NLP and multimodal modeling, with over 20,000 pull requests and frequent feature updates. *Megatron*, by contrast, is an internally maintained NVIDIA [15] library optimized for large-scale distributed training; it has limited PR activity but continuously updated commits on its main branch. These two libraries represent complementary scenarios: community-driven open development and enterprise-scale infrastructure evolution. While DEPRADAR is generalizable to other scientific computing libraries, we focus on DL frameworks because their defects are often configuration-sensitive, tensor-dependent, and hardware-specific (e.g., GPU precision modes), making them representative targets for evaluating defect and impact analysis.

PR/Commit Sampling and Annotation. For *Transformers*, we collect all PRs merged during the Q2 2025 (April–June), obtaining 718 candidates. We then perform a two-step procedure: (1) *Filtering*: PRs clearly unrelated to defect fixing (e.g., documentation, testing) are removed based on title and diff. (2) *Manual annotation*: the remaining 157 PRs are independently reviewed by two DL-domain experts, who decide whether the change is a *real defect fix* or a benign change. Disagreements are later reconciled by discussion. For *Megatron*, since it does not actively maintain public PRs, we collect 535 recent commits from the main branch, exclude 465 non-defect cases, and retain 70 candidates. These are annotated by the same two experts using identical criteria to ensure consistency across PR- and commit-based workflows.

Baselines and Metrics. We compare DEPRADAR against three representative baselines: FlatLLM (Base), FlatLLM (Reasoning), and PyCG [53]. FlatLLM (Base) serves as a zero-shot summarization baseline [44, 58], reflecting the common practice of using a single general-purpose LLM to interpret PRs or commits without orchestration. Although these models are primarily designed for commit or pull request summarization, they provide a realistic and conservative comparison for uncoordinated LLM reasoning over code changes. We also compare DEPRADAR with FlatLLM (Reasoning), a self-reasoning baseline that uses the DeepSeek-R1 [3]. Unlike FlatLLM (Base), FlatLLM (Reasoning) is capable of reasoning internally and handling both defect pattern generation and impact analysis within the same model. Finally, PyCG provides a non-LLM static call-graph baseline. Unlike prior work on commit summarization using ROUGE [17, 44], our task produces structured representations (defect patterns) intended for downstream reasoning. We therefore adopt expert-driven field-level scoring for summary evaluation and binary classification metrics (precision, recall, F1) for defect impact analysis.

LLM Configuration. All agents in DEPRADAR use DeepSeek-V3 [2] as the backbone model. We select DeepSeek-V3 for its ability to efficiently process heterogeneous inputs (e.g., PR discussions, code diffs, and AST fragments) while maintaining stable reasoning quality at a manageable token cost. To ensure reproducibility and avoid mixed-model effects, all agents share the same base model under deterministic decoding settings (temperature = 0, top-p = 1.0).

Client Collection. For PR analysis, we start from the 122 true-positive defect patterns and collect open-source downstream clients on GitHub that depend on *Transformers*. For each defect pattern, we extract key identifiers (e.g., model names, parameters, function calls such as `from_pretrained()` and `attn_implementation`) as

query anchors to locate potential clients. We retain only clients that can be successfully executed, pass integrity checks (e.g., dependency installation, model loading), and are not trivial forks or toy examples. The resulting clients cover diverse downstream usage scenarios, including NLP fine-tuning pipelines, multimodal training frameworks, and inference services, representative of the broader *Transformers* ecosystem. For commit-level evaluation, we use MindSpeed [14], a production-grade open-source project built on *Megatron* for large-scale distributed training and deployment. Its industrial workflow and extensive reuse of *Megatron* modules make it an ideal environment for validating downstream impacts of commit-level fixes.

4.2 RQ1: Can DEPRADAR effectively generate defect patterns?

Methodology. A key goal of DEPRADAR is to distinguish real bug-fix commits from benign changes and to extract precise, structured defect patterns from natural language and code changes. To assess this capability, we evaluate DEPRADAR on 157 recent PRs from the *Transformers*. These PRs include a wide variety of silent defects—bugs that do not crash the program but can cause incorrect behavior or inefficiency under certain conditions.

Results and Analysis. As shown in Table 2, DEPRADAR achieves high performance in defect identification, with a precision of 90%, recall of 99%, and F1-score of 95%. The specificity (i.e., true negative rate) is 62%, indicating that DEPRADAR can also effectively distinguish non-defect changes such as documentation updates or test refactoring. Compared to FlatLLM (base), which uses a single LLM summarization prompt without agentic coordination, DEPRADAR exhibits superior accuracy across all metrics, demonstrating the advantage of its structured reasoning and patch-aware analysis. DEPRADAR correctly flags non-defect PRs such as #38212 (fixing logging links) and #38260 (adding decorators and docstrings) by recognizing their lack of semantic impact on runtime behavior. In contrast, the FlatLLM(base) misclassifies these cases due to the presence of keywords like *fix* and superficial lexical changes.

Table 2: Defect Identification Performance Comparison.

Tool	TP	FP	FN	TN	Prec.	Rec.	F1	Spec.
DEPRADAR	122	13	1	21	90%	99%	95%	62%
FlatLLM (Base)	112	27	10	7	81%	92%	86%	21%
FlatLLM (Reasoning)	116	19	6	15	86%	95%	90%	44%

While FlatLLM(Reasoning) improves recall over FlatLLM(Base) by generating intermediate reasoning chains, it often produces redundant or speculative interpretations when the patch semantics are ambiguous. For example, in PR #37905 fixing unintended quantization of tied embeddings, FlatLLM(Reasoning) misclassified the change as a performance optimization due to keyword overlap. In contrast, DEPRADAR’s multi-agent coordination enables explicit semantic verification, correctly linking the quantization configuration to the tied-weight condition and identifying it as a real defect.

Error Analysis. We manually inspect all false positives. We find that 13 PRs are flagged as defects despite having no effect on core functionality. These cases primarily involve bug fixes in test components. For example, PR #38471 adjusts FlexAttn test logic to

handle edge-case tensors in Zamba2 and Deepseek3 models, fixing a bug where Gemma3IntegrationTest was never executed. While technically correcting a bug in test behavior, such patches do not influence production usage. DEPRADAR currently lacks the ability to distinguish between test scaffolding and runtime code. Incorporating learned classifiers or test directory heuristics could further reduce false positives in future iterations.

Table 3: Field-wise accuracy and inter-rater agreement of defect patterns ($N = 157$).

Field	Miss (0)	Part. (1)	Acc. (2)	Avg.	Cohen's κ
Bug Background	31 (10%)	61 (19%)	222 (71%)	1.61	0.71
Impact Scope	11 (4%)	38 (12%)	265 (84%)	1.81	0.74
Trigger Conditions	34 (11%)	122 (39%)	158 (50%)	1.39	0.71

Structured Field Evaluation. Beyond binary classification, we further assess the quality of structured defect reports generated by DEPRADAR. Each report includes three key fields—bug_background, impact_scope, and trigger_conditions—that are essential for downstream impact analysis. Two domain experts with over five years of DL development experience independently rated each field on a 3-point scale (0 = missing, 1 = partially correct, 2 = fully accurate). The 3-point design balances granularity and reliability: binary labels overlook partially correct cases, whereas finer scales reduce consistency. Each expert annotated all samples independently to prevent mutual influence. Cohen’s Kappa averaged 0.72 (range 0.71–0.74), indicating substantial inter-rater agreement. Disagreements (9%) were discussed and resolved through reference to PR diffs and commit metadata until consensus was reached. As shown in Table 3, DEPRADAR achieves high field-level accuracy: over 70% of the bug_background and 84% of the impact_scope entries are rated fully accurate. The trigger_conditions field is comparatively harder, with partial scores in cases involving multiple parameters or environment dependencies.

Defect Type Distribution. Modeling trigger conditions proves to be the most challenging. In many PRs, preconditions are implicit—distributed across config defaults, internal logic, and usage assumptions. For example, PR #37575’s behavior depends on both a device setting (NPU) and an unset scale parameter. Capturing such cross-cutting logic requires synthesizing patch semantics, device context, and optional configurations. Future directions include integrating execution traces or dependency graphs to further enhance condition extraction.

To assess whether DEPRADAR generalizes beyond narrow bug categories, we categorize all 122 true positive PRs into seven types (Table 4). DEPRADAR achieves consistent performance across all categories, reflecting its robustness in modeling diverse defect semantics. Notably, only four PRs belong to Type G (crash), while over 97% fall into non-crash categories. These silent defects are difficult to detect via traditional testing, often requiring semantic patch interpretation and usage context reasoning. The dominance of Type C and D defects underscores the need for tools like DEPRADAR that move beyond crash detection and proactively reason about latent faults in modern DL systems.

4.3 RQ2: How effective is DEPRADAR in analyzing whether a client is affected by a defect?

Methodology. After identifying bug-fix PRs and constructing structured defect patterns, a key challenge is to determine whether a downstream client program is genuinely *affected*—i.e., whether it satisfies the trigger conditions of the defect and may exhibit incorrect behavior at runtime. We evaluate DEPRADAR on 122 real-world (defect, client) pairs collected from the benchmark described in Sec. 4.1, and compare it against the three baselines.

Results and Comparison. As shown in Table 5, DEPRADAR achieves the best overall balance between precision and recall ($F1 = 85\%$), substantially outperforming all baselines. Compared with FlatLLM(Base), which tends to over-report due to superficial lexical or contextual cues, DEPRADAR improves precision by 17 points (80% vs. 63%) and specificity by 28 points (72% vs. 44%). FlatLLM(Reasoning) performs moderately better than Base ($F1 = 74\%$ vs. 68%) owing to its internal reasoning ability, but still suffers from false positives and missed configuration-dependent cases due to the lack of structural verification. PyCG achieves the lowest overall performance ($F1 = 47\%$, Prec. = 44%, Rec. = 50%), as its static call-graph analysis misses dynamic dependencies and merges high-level APIs, causing many false positives and low recall.

Qualitative Analysis. We find that DEPRADAR’s advantage stems from its ability to (1) *match actual usage contexts in client code* and (2) *verify their relevance through static AST checks*. For example, in PR #38238, where the fix targets the configuration decoder_bbox_embed_share=False, DEPRADAR correctly infers that a client using GroundingDinoForObjectDetection is **not affected** because the trigger condition is unmet—whereas both LLM baselines misclassify it as positive. Similarly, for PR #38285, where the bug occurs inside a conditional macro, FlatLLM(Reasoning) fails to detect the impact due to implicit control flow, while DEPRADAR succeeds by structurally matching the AST predicates.

To further verify correctness, we construct test scripts for 50 randomly sampled TP cases and confirm that each exhibits the expected faulty behavior when executed under vulnerable library versions, such as memory bloat, incorrect attention scaling, or silent numerical drift. These results show that DEPRADAR can go beyond surface-level call matching to semantically reason about trigger satisfaction in real-world usage. Its structured impact reports and context-aware analysis lead to both higher accuracy and more actionable diagnostics compared to prompt-only LLM summarization.

4.4 RQ3: How does each component contribute to defect modeling and impact analysis?

4.4.1 Defect Pattern Generation Module Ablation. We perform an ablation study to evaluate the contribution of two key design dimensions in our approach: the use of *coordinated framework* and the *adaptive context expansion* mechanism for PRs. Our goal is to quantify how each design choice affects the accuracy of defect pattern extraction. To this end, we construct the dataset from Transformers PRs annotated in RQ1, and compare the following variants:

- **DEPRADAR (Full):** Uses both the coordinated framework and adaptive context expansion.
- **w/o Adaptive Context:** Disables context expansion; each agent processes a fixed-size content window.
- **w/o Coordination:** Replaces multi-agent coordination with a single

Table 4: Defect Taxonomy and Examples in Transformers PRs.

Type	Name	Count	Description	Example PRs
A	Configuration Misuse	16	Incorrect parameter settings, misuse of optional arguments	#38223, #38288
B	Missing Preconditions	13	Lack of boundary checks or unsafe fallback logic	#38264, #38945
C	Semantic Logic Errors	56	Flawed control flow, tensor misalignment, wrong fallback logic	#37704, #38290
D	Compatibility Issues	23	Incompatibilities due to API changes or module relocations	#38841, #39083
E	Concurrency Bugs	4	Synchronization issues or data races across devices/threads	#38557, #38876
F	Performance/Resource Bugs	6	Memory inefficiencies, unbounded CPU usage	#38901, #39043
G	Crash or Fatal Errors	4	Crashes due to runtime exceptions or unsupported behavior	#37690, #38668

Table 5: Impact Analysis Performance (N=122). The client set corresponds to TP defect patterns identified in RQ1.

Tool	TP	FP	FN	TN	Prec.	Rec.	F1	Spec.
DEPRADAR	61	15	7	39	80%	90%	85%	72%
FlatLLM (Base)	50	30	18	24	63%	74%	68%	44%
FlatLLM (Reasoning)	54	25	14	29	68%	79%	74%	54%
PyCG	34	44	34	10	44%	50%	47%	19%

Table 6: Ablation Study on Defect Pattern Extraction (N=157).

System Variant	TP	FP	FN	TN	Prec.	Rec.	F1	Spec.
DEPRADAR (Full)	122	13	1	21	90%	99%	95%	62%
w/o Coordination	78	27	45	7	74%	63%	68%	21%
w/o Adaptive Context	118	17	5	17	87%	96%	91%	50%

LLM prompt that ingests all content (PR title, body, patch) at once, with no agent collaboration or role differentiation.

Table 6 reports precision, recall, and F1 scores for defect classification. The full DEPRADAR achieves the highest performance ($F_1 = 95\%$), while removing context adaptation causes a moderate decline ($F_1 = 91\%$), primarily due to missed trigger conditions dispersed across PR metadata. The single-prompt variant exhibits the largest drop ($F_1 = 68\%$), as it lacks structured decomposition and often fails to disambiguate configuration details or code-specific semantics. We find that this variant frequently hallucinates parameter names or overlooks patch semantics when processing longer inputs.

These results highlight that: (1) **adaptive context expansion** is crucial for capturing scattered defect signals across multi-round PR discussions, and (2) **multi-agent specialization** offers better grounding and modular control compared to flat summarization. Together, these components enable DEPRADAR to robustly extract structured, actionable defect patterns across diverse bug types.

4.4.2 Impact Analysis Module Ablation. We evaluate the contribution of two critical modules in our pipeline: (1) the *verification layer* that checks structural consistency between predicted risk factors and client code, and (2) the *domain mapping layer* that maps internal library changes to user-facing constructs.

To prevent false positives from LLM hallucinations, DEPRADAR performs AST-based validation to confirm that predicted risk parameters and method calls are actually present in the matched client context. This layer also re-invokes the Impact Analyzer with feedback when inconsistencies arise. Library patches often involve internal APIs (e.g., `scale_mask`, `prepare_inputs`) that users do not call directly. DEPRADAR encodes a set of DL-specific heuristics

Table 7: Ablation Study on Client Impact Analysis (N = 122).

System Variant	TP	FP	FN	TN	Prec.	Rec.	F1	Spec.
DEPRADAR (Full)	61	15	7	39	80%	90%	85%	72%
w/o Verification Layer	59	30	9	24	66%	87%	75%	44%
w/o Domain Mapping	52	22	16	32	70%	76%	73%	59%

to project such internal signals to observable constructs—e.g., translating a buffer allocation bug to `AutoModel.from_pretrained()` in user code. This mapping is crucial for matching abstract trigger conditions with real-world usage.

We construct two system variants: - **w/o Verification Layer**: Skips consistency checks. The model’s first prediction is accepted without AST validation or retry logic. - **w/o Domain Mapping**: Disables mapping from internal changes to user-facing methods and parameters. Table 7 shows the performance impact on the Transformers client dataset. Removing the verification layer significantly increases false positives, reducing precision from 80% to 66%. Manual inspection reveals typical failure cases where the model hallucinates parameters such as `flash_mask_type`, which do not exist in the actual client code. Without domain mapping, recall drops ($90\% \rightarrow 76\%$) as the model fails to align internal logic with surface-level usage. For instance, in PR #38257 (import error handling), the risk method `_get_module()` is never directly called by users, but its failure propagates through `AutoModel.from_pretrained()`. Without mapping, this link is missed.

These results highlight the importance of grounded reasoning and domain abstraction in client impact analysis. Naively prompting LLMs—even with defect-aware context—is insufficient unless predictions are anchored in concrete code structure and adapted to real-world usage patterns.

4.5 RQ4: Can DEPRADAR Effectively Handle Defect and Impact Analysis from Commits?

Methodology. We evaluate DEPRADAR on *Megatron*, a commit-driven project without PRs, to examine its generalizability beyond PR-based workflows (Table 8). Unlike PR-based analysis, this setting evaluates whether DEPRADAR can maintain accuracy when only minimal commit information is available.

Results. For defect pattern generation, it achieves a precision of 96% and a recall of 90%, despite the absence of high-level PR context. The few false negatives mainly correspond to commits involving test or documentation fixes that contain partial but non-functional code edits. This result indicates that DEPRADAR’s architecture—combining AST-based filtering, patch reasoning, and

contextual alignment—retains high accuracy even when contextual cues are minimal.

Table 8: Defect and Impact Analysis from Commits.

Task	TP	FP	FN	TN	Prec.	Rec.	F1	Spec.
Defect Pattern	54	2	6	8	96%	90%	93%	80%
Impact Analysis	12	8	5	29	60%	71%	65%	78%

Impact Analysis. For impact analysis, the task becomes considerably more challenging. Unlike PRs, commits lack discussion and rationale context. As a result, while recall remains high (the system still identifies most real impacts), precision declines due to increased false associations from superficial diffs. Despite this sparsity, DEPRADAR still achieves a recall of 71% and a precision of 60%, successfully identifying 12 developer-confirmed client impacts among 54 true defect patterns.

Developer-Confirmed Impacts. In the downstream client repository MindSpeed [14], the development team validated DEPRADAR’s predictions, confirming 12 affected commits with real runtime impacts. Among these, four were categorized as **high-risk** defects that directly degraded model accuracy or stability (e.g., gradient scaling), while the remaining eight were **low-risk** fixes such as logging inconsistencies, checkpoint I/O errors, or boundary-type corrections. Table 9 summarizes representative developer-confirmed cases, illustrating how DEPRADAR distinguishes critical defects from benign propagation in industrial DL systems.

To further illustrate DEPRADAR’s effectiveness, we highlight two developer-confirmed cases from *Megatron*. In commit *3bdccb Fix DDP scaling factor with Context Parallel* [8], *Megatron* developers fixed a bug in gradient scaling logic by adding a missing flag:

```
1# Original:
2target_gradient_scaling_factor = 1.0 / parallel_state.
3    get_data_parallel_world_size()
4#
5# Fixed:
5target_gradient_scaling_factor = 1.0 / parallel_state.
6    get_data_parallel_world_size(
6with_context_parallel=True)
```

DEPRADAR successfully identified this as a silent numerical defect and generated a defect pattern with trigger conditions based on the usage of context parallelism. In the downstream client MindSpeed [14], which integrates *Megatron* modules for large-scale model pretraining and deployment, a corresponding usage case was automatically matched through static analysis and flagged as *is_affected: true*. The developers later confirmed the bug’s relevance and acknowledged its active impact on MindSpeed’s performance. Instead of performing a full *Megatron* upgrade (which would introduce compatibility issues), the developers implemented a localized patch incorporating the upstream fix - demonstrating the practical value of our impact analysis.

Another case refers to *Megatron Commit 1674ce3* [7]. This commit addresses a training defect in *Megatron* where vision encoder weights were not correctly marked as non-tensor-parallelizable, leading to incorrect gradient reductions in TP-enabled training. DEPRADAR automatically identified this commit as a real defect and extracted its root cause and trigger conditions. It further confirmed that MindSpeed is affected under TP4 configurations. Empirical

Table 9: Developer-confirmed affected commits in *Megatron*.

Risk	Commit	Primary Impact
High	06e9f28 [6]	LR scaling bug under FSDP+TP mode
High	1674ce3 [7]	Vision weight inconsistency under TP4
High	3bdccb [8]	Gradient scaling instability (context parallel)
High	dc4c5e7 [13]	Pipeline gradient accumulation mismatch
Low	8e2de56 [9]	Loss logging accumulation error
Low	9a45638 [10]	Optimizer checkpoint save/load fix
Low	a77a883 [11]	PP layer uneven split bugfix
Low	db07e3f [12]	Rotary-pos cosine check correction

validation revealed that the bug caused gradient inconsistencies for unsharded vision weights during training and severe visual artifacts (*screen tearing*) during inference. To mitigate the issue, the MindSpeed team applied deterministic training mode to ensure consistent updates and is actively evaluating a fix by adding a new function *_allreduce_duplicate_grads()* and modifying *finalize_model_grads()*. Together, these results demonstrate that DEPRADAR can effectively uncover and contextualize silent defects that propagate into large-scale production systems, underscoring its robustness in real-world DL infrastructure.

4.6 RQ5: What is the cost of using DEPRADAR?

Methodology. We analyze the computational efficiency of DEPRADAR when processing defect and impact analysis tasks, covering 157 PRs and 70 commits.

Results. As shown in Table 10, the *Defect Pattern (PR)* task consumes about 5K tokens and 0.5 minutes per case, mainly due to the larger PR context. *Defect Pattern (Commit)* is lighter (<2K tokens, <0.3 min), while *Impact Analysis (Client)* costs roughly 3K tokens and 1 minute per case due to multi-source reasoning. These results show that token consumption and runtime scale with task complexity—PRs require richer contextual reasoning, while commits are lighter; client-side analysis incurs higher overhead due to multi-source dependency tracing. Across all evaluated tasks, the total token consumption reaches approximately 1.29 million tokens, corresponding to a cost of about \$0.5 under DeepSeek-V3’ pricing. This demonstrates that even under conservative assumptions, the entire evaluation can be reproduced at negligible cost (under \$1).

Table 10: Computation cost and runtime of DEPRADAR across different task types. Tokens include both input and output.

Task Type	Avg. Tokens	Total Tokens	Avg. Time (min)	Total Time (min)
Defect Pattern (PR)	5K	785K	0.5	78.5
Defect Pattern (Commit)	<2K	140K	<0.3	21
Impact Analysis (Client)	3K	366K	1.0	122
Total	-	1.29M	-	221.5

Comparison with Baselines. Compared with baseline models, the cost of DEPRADAR remains competitive. *FlatLLM (Base)* consumes a similar number of tokens, since its input context (PR body and code diffs) is comparable. *FlatLLM (Reasoning)* processes the same input but incurs higher expenses due to internal reasoning chains and

longer inference times. In contrast, DEPRADAR introduces only minor coordination overhead while achieving faster overall execution, as each agent operates within a focused sub-context. The overall computational cost (in terms of tokens, runtime, and monetary expense) remains within a reasonable range. Considering its analytical depth and coordination capability, DEPRADAR achieves an effective balance between efficiency and accuracy, demonstrating strong practicality for real-world deployment.

5 Discussion

5.1 Generalizability, Adoption, and Extension

Generalizability. While our evaluation focuses on two popular DL libraries—Transformers and *Megatron*, the modular architecture of DEPRADAR is broadly applicable to other Python-based libraries that expose public APIs and have structured commits or PRs. We observe that DEPRADAR performs consistently across a range of defect types (e.g., configuration misuse, semantic errors, performance issues), suggesting good generalization. Extending the system to non-Python libraries (e.g., TensorFlow C++) would require adapting AST parsing and static analysis routines.

Developer Adoption. Case studies in *Megatron* show that users may prefer to locally backport patches instead of upgrading the entire library version, due to compatibility or dependency constraints. In such scenarios, DEPRADAR’s ability to surface concrete trigger conditions and risk factors can assist developers in writing minimal fixes. This highlights its potential utility as a defect-awareness tool in long-lived or production-grade codebases.

Extension Directions. One promising extension is to integrate dynamic tracing or execution logs to better infer runtime-specific defects (e.g., those triggered by hardware mismatch or environment settings). Another direction is model-agnostic tuning, where different LLMs (e.g., GPT-4, DeepSeek, Claude) are compared in terms of consistency, robustness, and hallucination rates under the same agent roles. We also plan to explore IDE integration to offer actionable defect awareness during development or CI workflows.

5.2 Threats to Validity

Threats to Internal Validity. Some of our annotations—such as defect presence or client impact—rely on expert judgment, though we observe substantial inter-annotator agreement. Certain trigger conditions may also depend on runtime paths not visible in static code. Although DEPRADAR performs consistency checks and retry rounds, occasional hallucinations from LLMs cannot be fully ruled out. **Threats to External Validity.** Our evaluation is limited to two actively maintained DL libraries. Libraries with less structured development workflows or sparse metadata may degrade the effectiveness of agent-based reasoning. In addition, the current system is designed for Python-based ecosystems; applying it to other languages requires further adaptation.

6 Related Work

Commit and Pull Request Understanding Understanding and analyzing PRs/commits is a fundamental task in software engineering research, aiming to support tasks such as code review, bug triage, and change impact analysis [22, 25, 30, 33, 35, 40, 44, 46, 47, 49, 51, 55, 58]. Liu et al. [44] proposed a sequence-to-sequence model

with a pointer-generator mechanism to automatically generate pull request (PR) descriptions by leveraging commit messages and code comments. They further applied reinforcement learning to directly optimize ROUGE scores and mitigate the mismatch between training objectives and evaluation metrics. Bai et al. [22] introduced the CLG framework, which generates PR checklists automatically from contribution guidelines by combining multi-label classification and text generation techniques. Their method significantly outperformed baselines on summarization tasks and demonstrated practical value in open-source community management.

Security and Maintenance Challenges in Third-Party Library Dependencies. Prior work has extensively studied the risks posed by third-party libraries, including vulnerability propagation, delayed patch adoption, and ecosystem-wide dependency management challenges [21, 26, 28, 29, 31, 32, 34, 42, 43, 56, 61–67]. Zeng et al. [60] present a comprehensive survey on the security of third-party dependencies, highlighting trends in detection, mitigation, and secure integration practices. Wu et al. [57] perform a large-scale empirical study over 44,450 triplets in the Maven ecosystem, revealing that 73.3% of downstream projects depending on vulnerable libraries remain practically unaffected—underscoring the importance of precise impact assessment. Chen et al. [27] propose VESTA a vulnerability-oriented test generation framework that uses exploit code migration to evaluate real-world vulnerability reachability, improving detection success by 53.4% over baselines. Unlike prior work, DEPRADAR targets subtle, non-crashing defects in DL libraries by bridging defect modeling and downstream impact analysis through extracting fine-grained trigger conditions from code changes and semantically matching them to real client usage.

7 Conclusion

In this paper, we presented DEPRADAR, an agent coordination framework for automating defect impact analysis in DL libraries. DEPRADAR tackles the key challenge of determining whether bug fixes in libraries like *Transformers* and *Megatron* affect downstream client programs. By coordinating Miner, Diff Analyzer, and Impact Analyzer agents—DEPRADAR bridges the semantic gap between low-level library patches and high-level user-facing behavior, enabling context-aware impact assessment. Our evaluation on real-world PRs and commits demonstrates its effectiveness, achieving up to 95% F1 in defect classification and 80% precision in impact analysis. In the future, we plan to extend its capabilities in several directions: (1) broadening support for additional DL libraries (e.g., vllm [19]) and defect types, (2) scaling the evaluation to larger datasets of client programs. While the current workflow focuses on PRs and commits containing explicit fixes, future extensions may also consider report-only or reproduction-case scenarios. We further plan to extend DEPRADAR to security vulnerability analysis, enabling it to interpret CVE reports and perform structured impact reasoning similar to defect cases.

Acknowledgement

This research is sponsored by the National Key R&D Program of China (No. 2024YFB4506400) and CCF-Huawei Populus Grove Fund. We also thank the anonymous reviewers for their insightful comments and suggestions.

References

- [1] 2025. AutoGen. <https://microsoft.github.io/autogen/stable/index.html>.
- [2] 2025. DeepSeek. <https://www.deepseek.com/>.
- [3] 2025. DeepSeekR1. <https://github.com/deepseek-ai/DeepSeek-R1>.
- [4] 2025. GitCommit. <https://github.com/git-guides/git-commit>.
- [5] 2025. Megatron-LM. <https://github.com/NVIDIA/Megatron-LM>.
- [6] 2025. MegatronCommit06e9f28. <https://github.com/NVIDIA/Megatron-LM/commit/06e9f28b30e55a1c857784696626f9a703450d97>.
- [7] 2025. MegatronCommit1674ce3. <https://github.com/NVIDIA/Megatron-LM/commit/1674ce30b82a90a729cc7bf2f93b41da47c0983e>.
- [8] 2025. MegatronCommit3bdccb. <https://github.com/NVIDIA/Megatron-LM/commit/3bdccbbe5d2a455a75e28969be7250cd4bd27bae>.
- [9] 2025. MegatronCommit8e2de56. <https://github.com/NVIDIA/Megatron-LM/commit/8e2de5622b8df22e7c71e04c85670e8f3cf457c>.
- [10] 2025. MegatronCommit9a45638. <https://github.com/NVIDIA/Megatron-LM/commit/9a45638758de6bb61a2951185a2d5999e1dad92>.
- [11] 2025. MegatronCommitta77a883. <https://github.com/NVIDIA/Megatron-LM/commit/a77a883e248e68df1912df4ef2fc05b712947fce>.
- [12] 2025. MegatronCommitdb07e3f. <https://github.com/NVIDIA/Megatron-LM/commit/db07e3fe810fe7e0f12bab374830823fd2333b5>.
- [13] 2025. MegatronCommitdc45e7. <https://github.com/NVIDIA/Megatron-LM/commit/d4c5e71c734e2e72a02d6f7b9851ba9b4c80706>.
- [14] 2025. MindSpeed. <https://gitee.com/ascend/MindSpeed>.
- [15] 2025. NVIDIA. <https://www.nvidia.com/en-sg/>.
- [16] 2025. PullRequests. <https://docs.github.com/en/pull-requests>.
- [17] 2025. ROUGE. [https://en.wikipedia.org/wiki/ROUGE_\(metric\)](https://en.wikipedia.org/wiki/ROUGE_(metric)).
- [18] 2025. Transformers. <https://github.com/huggingface/transformers>.
- [19] 2025. vLLM. <https://github.com/vllm-project/vllm>.
- [20] 2026. DepRadar. <https://github.com/testmigrator/DepRadar>.
- [21] Mahmoud Alfafel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering* 28, 3 (2023), 59.
- [22] Shuotong Bai, Chenkun Meng, Guodong Li, Huaxiao Liu, and Lei Liu. 2025. CLG: Automated checklist generation for improved pull request quality. *Expert Systems with Applications* 277 (2025), 127178.
- [23] 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.
- [24] 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), 31.
- [25] Siyu Chen, Shengbin Xu, Yuan Yao, and Feng Xu. 2022. Untangling composite commits by attributed graph clustering. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware*. 117–126.
- [26] Tianyu Chen, Lin Li, Bingjie Shan, Guangtai Liang, Ding Li, Qianxiang Wang, and Tao Xie. 2023. Identifying Vulnerable Third-Party Libraries from Textual Descriptions of Vulnerabilities and Libraries. arXiv:2307.08206 [cs.CR]
- [27] Zirui Chen, Xing Hu, Xin Xia, Yi Gao, Tongtong Xu, David Lo, and Xiaohu Yang. 2024. Exploiting library vulnerability via migration based automating test generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.
- [28] Zirui Chen, Zhipeng Xue, Jiayuan Zhou, Xing Hu, Xin Xia, and Xiaohu Yang. 2025. Diffploit: Facilitating Cross-Version Exploit Migration for Open Source Library Vulnerabilities. arXiv:2511.12950 [cs.SE] <https://arxiv.org/abs/2511.12950>
- [29] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24 (2019), 381–416.
- [30] Martin Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 341–350.
- [31] Ruiyan Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2020. Towards measuring supply chain attacks on package managers for interpreted languages. *arXiv preprint arXiv:2002.01139* (2020).
- [32] Yi Gao, Xing Hu, Zirui Chen, Tongtong Xu, and Xiaohu Yang. 2025. Vulnerability-Triggering Test Case Generation from Third-Party Libraries. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 125–135.
- [33] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher A Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagarla, Philip Makedonski, Matin Nili Ahmadabadi, et al. 2022. A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Software Engineering* 27, 6 (2022), 125.
- [34] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. 2022. Characterizing usages, updates and risks of third-party libraries in Java projects. *Empirical Software Engineering* 27, 4 (2022), 90.
- [35] Ali Tunahan Isik, Haticce Kübra Çağlar, and Eray Tüzün. 2025. Enhancing Pull Request Reviews: Leveraging Large Language Models to Detect Inconsistencies Between Issues and Pull Requests. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 168–178.
- [36] Li Jia, Hao Zhong, Xiaoyin Wang, Lipeng 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.
- [37] Yuxuan Jiang, Ziming Zhou, Boyu Xu, Beijie Liu, Runhui Xu, and Peng Huang. 2025. Training with confidence: catching silent errors in deep learning training with automated proactive checks. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation* (Boston, MA, USA) (OSDI '25). USENIX Association, USA, Article 18, 17 pages.
- [38] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23 (2018), 384–417.
- [39] Chenglong Li, Zheng Zheng, Xiaoting Du, Xiangyue Ma, Zhengqi Wang, and Xinheng Li. 2024. KnowBug: Enhancing Large language models with bug report knowledge for deep learning framework bug prediction. *Knowledge-Based Systems* 305 (2024), 112588.
- [40] Yi Li, Shaohua Wang, and Tien N Nguyen. 2022. UTANGO: untangling commits with context-aware, graph-based, code change clustering learning model. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 221–232.
- [41] C. Liu, R. Cai, and Zhou Y.Chen X.Hu H.Yan M. 2024. Understanding the implementation issues when using deep learning frameworks. *Information and software technology* 166, Feb. (2024), 1.1–1.16.
- [42] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*. 672–684.
- [43] Shuhuan Liu, Xing Hu, Xin Xia, David Lo, and Xiaohu Yang. 2025. An Empirical Study of Vulnerable Package Dependencies in LLM Repositories. *arXiv preprint arXiv:2508.21417* (2025).
- [44] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shaping Li. 2019. Automatic generation of pull request descriptions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 176–188.
- [45] Yunbo Lyu, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widayarsi, Zhipeng Zhao, Xuan-Bach D Le, Ming Li, and David Lo. 2023. Chronos: Time-aware zero-shot identification of libraries from vulnerability reports. *arXiv preprint arXiv:2301.03944* (2023).
- [46] Willem Meijer, Mirela Riveni, and Ayushi Rastogi. 2024. Ecosystem-wide influences on pull request decisions: insights from NPM. *arXiv preprint arXiv:2410.14695* (2024).
- [47] Qianshuang Meng, Weiqin Zou, Biyu Cai, and Jingxuan Zhang. 2024. KeyTitle: towards better bug report title generation by keywords planning. *Software Quality Journal* 32, 4 (2024), 1655–1682.
- [48] Mohammad Mehdi Morovati, Amin Nikanjam, and Khomh Zhen Ming Jack Jiang. 2024. Bug characterization in machine learning-based systems. *Empirical software engineering* 29, 1 (2024), 1.1–1.29.
- [49] Ward Muylaert and Coen De Roover. 2018. Untangling composite commits using program slicing. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 193–202.
- [50] Ahmed Fawzi Otoo, Sara Al-Jdaeh, and Maen Hammad. 2019. Automated classification of software bug reports. In *Proceedings of the 9th international conference on information communication and management*. 17–21.
- [51] Shengyi Pan, Jiayuan Zhou, Filipe Roseiro Cogo, Xin Xia, Lingfeng Bao, Xing Hu, Shaping Li, and Ahmed E Hassan. 2022. Automated unearthing of dangerous issue reports. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 834–846.
- [52] Federica Pepe, Fiorella Zampetti, Antonio Mastropaoletti, Gabriele Bavota, and Massimiliano Di Penta. 2024. A taxonomy of self-admitted technical debt in deep learning systems. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSM)*. IEEE, 388–399.
- [53] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. Pycg: Practical call graph generation in python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1646–1657.
- [54] Nikita Samarin and Primal Wijesekera. [n. d.]. Understanding How Third-Party Libraries in Mobile Apps Affect Responses to Subject Access Requests. ([n. d.]).
- [55] Akhila Sri Manasa Venigalla and Sridhar Chimalakonda. 2024. An exploratory study of software artifacts on GitHub from the lens of documentation. *Information and Software Technology* 169 (2024), 107425.
- [56] Lauri Williams, Giacomo Benedetti, Sivana Hamer, Ranjinda Paramitha, Imranur Rahman, Mahzabin Tamanna, Greg Tystahl, Nusrat Zahan, Patrick Morrison, Yasemin Acar, et al. 2025. Research directions in software supply chain security. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–38.

- [57] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. 2023. Understanding the Threats of Upstream Vulnerabilities to Downstream Projects in the Maven Ecosystem. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1046–1058.
- [58] Tao Xiao, Hideaki Hata, Christoph Treude, and Kenichi Matsumoto. 2024. Generative AI for pull request descriptions: Adoption, impact, and developer interventions. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1043–1065.
- [59] Xiao Yu, Haoxuan Chen, Feifei Niu, Xing Hu, Jacky Wai Keung, and Xin Xia. 2025. Towards Understanding Bugs in Distributed Training and Inference Frameworks for Large Language Models. *arXiv preprint arXiv:2506.10426* (2025).
- [60] Jia Zeng, Dan Han, Yaling Zhu, Yangzhong Wang, and Fangchen Weng. 2024. A Survey of Third-Party Library Security Research in Application Software. *arXiv preprint arXiv:2404.17955* (2024).
- [61] Fangyuan Zhang, Lingling Fan, Sen Chen, Miaoying Cai, Sihan Xu, and Lida Zhao. 2024. Does the vulnerability threaten our projects? automated vulnerable api detection for third-party libraries. *IEEE Transactions on Software Engineering* (2024).
- [62] Lyuye Zhang. 2024. Vulnerability root cause function locating for java vulnerabilities. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 444–446.
- [63] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. 2023. Mitigating persistence of open-source vulnerabilities in maven ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 191–203.
- [64] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Lida Zhao, Jiahui Wu, and Yang Liu. 2023. Compatible Remediation on Vulnerabilities from Third-Party Libraries for Java Projects. *arXiv:2301.08434* [cs.SE]
- [65] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. 2023. Software composition analysis for vulnerability detection: An empirical study on Java projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 960–972.
- [66] Zhuotong Zhou, Yongzhuo Yang, Susheng Wu, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Magneto: A Step-Wise Approach to Exploit Vulnerabilities in Dependent Libraries via LLM-Empowered Directed Fuzzing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1633–1644.
- [67] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. 995–1010.