LLM-Guided Mining of Performance-Related Commits at Scale

Md Abul Kalam Azad
akazad@umich.edu
University of Michigan-Dearborn
Dearborn, Michigan, USA

Syed Salauddin Mohammad Tariq
ssmtariq@umich.edu
University of Michigan-Dearborn
Dearborn, Michigan, USA

Foyzul Hassan
foyzul@umich.edu
University of Michigan-Dearborn
Dearborn, Michigan, USA

Diego Elias Costa
diego.costa@concordia.ca
REALISE Lab@Concordia University
Montreal, Quebec, Canada

Probir Roy
probirr@umich.edu
University of Michigan-Dearborn
Dearborn, Michigan, USA

ABSTRACT

Performance issues hinder software efficiency and reliability. Mining software repositories can reveal such issues, but existing keyword-, heuristic-, and ML-based methods lack the semantic understanding needed for accurate, large-scale analysis. This paper presents ***PerfMiner***, an LLM-guided framework for identifying and curating performance-related commits at scale. *PerfMiner* employs ***PerfAnnotator-mini***, a lightweight transformer distilled from a large language model to balance semantic reasoning with computational efficiency. Using this framework, we analyzed 170 million commits from 323 700 repositories in C++, Java, and Python, producing a curated dataset of 335 103 real-world performance-related commits. A manual audit of the mined dataset (n=384) confirmed precision of ≈0.90, supporting the reliability of automated labeling; on the curated ground-truth evaluation set (n=1,350), *PerfAnnotator-mini* attains F1=0.89 (precision=0.89). Inference is 381× faster than the 20B-parameter teacher (`gpt-oss:20b`) on an NVIDIA RTX 4090 GPU. These results indicate that our proposed LLM-guided semantic reasoning approach provides a scalable and practical foundation for large-scale studies of performance-related behavior in software repositories.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; Software defect analysis; • **Computing methodologies** → *Supervised learning by classification*; *Natural language processing*.

KEYWORDS

Performance-Related Commits, Mining Software Repositories, Knowledge Distillation, Large Language Models, Commit Classification, Software Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EASE 2026, 10–13 June, 2026, Glasgow, Scotland, United Kingdom
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-XXXX-XXXX-X/26/06
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Md Abul Kalam Azad, Syed Salauddin Mohammad Tariq, Foyzul Hassan, Diego Elias Costa, and Probir Roy. 2026. LLM-Guided Mining of Performance-Related Commits at Scale. In *Proceedings of The 30th International Conference on Evaluation and Assessment in Software Engineering (EASE 2026).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Performance issues degrade software efficiency and reliability, and understanding how developers address them provides valuable insight into software evolution and maintenance practices. As modern projects accumulate extensive revision histories across multiple languages and domains, mining software repositories offers an effective means to study how performance considerations influence code evolution, commit intent, and maintenance behavior [14, 23, 25].

However, automatically identifying *performance-related commits* remains challenging because performance changes are rarely labeled explicitly and are described inconsistently across projects. Existing mining approaches, including keyword-based heuristics and supervised models trained on limited labeled data [3, 32], rely on surface-level linguistic patterns and often require expert validation, limiting their scalability and semantic accuracy for large-scale analysis. This study focuses on static artifacts, commit messages and code diffs, to infer performance-related intent, rather than on dynamic execution measurements or workload profiling.

Building on these limitations, we identify three key challenges that remain in mining performance-related commits. *First, accuracy at scale:* performance-related commits are semantically diverse and often intertwined with non-functional or refactoring changes, making keyword- and heuristic-based methods prone to false positives and negatives [16]. Such filters fail to capture nuanced developer intent and require substantial manual validation to ensure correctness. *Second, scalable quality:* while heuristic pipelines can process large histories efficiently, their precision and recall deteriorate on heterogeneous projects, forcing costly manual validation. The cumulative curation effort grows rapidly with dataset size, creating a trade-off between coverage and reliability. *Third, dataset diversity:* prior studies on performance-related commits often emphasize specific ecosystems, such as Android apps [6], HPC systems [15], or Java-based cloud services [3, 32], leading to narrow and domain-specific datasets [16]. Limited diversity constrains the generalizability and reuse of performance datasets for future empirical studies. Addressing these challenges calls for scalable mining methods that combine

automation with semantic reasoning to reduce dependence on manual validation while maintaining accuracy. Such techniques can serve as semantics-aware first passes, improving the efficiency of large-scale repository analyses without replacing expert curation.

Building on this motivation, we aim to design an LLM-guided mining framework that identifies performance-related commits from code diffs and commit messages. We employ this framework to analyze large-scale software repositories across multiple programming languages and to construct a dataset for studying performance-related development practices. We evaluate its effectiveness in terms of accuracy, scalability, and dataset diversity, and qualitatively analyze model behavior to interpret the semantic cues underlying its predictions.

We investigate these objectives through four research questions:
RQ1 (Accuracy): How accurately can an LLM-guided framework classify performance-related commits across large repositories?
RQ2 (Qualitative Reasoning): What kinds of semantic patterns and contextual cues does *PerfMiner* capture when identifying performance-related commits, and how do these differ from traditional keyword or heuristic methods?
RQ3 (Scalability and Cost Efficiency): How efficiently can the approach process extensive commit histories at ecosystem scale (∼170M commit history) while maintaining practical computational cost and throughput for large-scale deployment?
RQ4 (Dataset Diversity): What is the distribution of mined performance-related commits across languages, repositories, and performance themes?

We present ***PerfMiner***, an LLM-guided framework for mining performance-related commits at *scale*. It employs ***PerfAnnotator-mini***, a lightweight transformer classifier trained via knowledge distillation from a larger teacher model. Through this process, the teacher's classification knowledge is distilled into a compact student model (125M parameters) via confidence-filtered hard labels and is designed to balance accuracy and inference efficiency, as evaluated in Section 4. Using this framework, we analyze 170 million commits from 323,700 repositories across C++, Java, and Python, producing a curated dataset of $\approx 335K$ performance-related commits suitable for large-scale empirical studies.

Empirical evaluation suggests that *PerfAnnotator-mini* attains an F1-score of **0.89** and precision of **0.89**, while providing a 381× reduction in GPU inference time compared with the 20B-parameter teacher. The framework scales to hundreds of millions of commits and yields a semantically diverse, multi-language dataset covering a broad range of performance themes. These findings suggest that LLM-guided semantic reasoning can significantly enhance the efficiency and scope of performance-related repository mining.

This work makes the following contributions:

- ***PerfMiner* framework:** an LLM-guided, knowledge-distilled methodology for scalable and semantics-aware mining of performance-related commits.
- ***PerfAnnotator-mini* model:** a compact transformer classifier that distills LLM-level semantic understanding into a model deployable on commodity CPU hardware.
- **Large-scale dataset:** a curated, precision-audited dataset of ≈335K performance-related commits spanning three major languages and hundreds of thousands of repositories, publicly released to support empirical studies of performance-driven software evolution.
- **Comprehensive evaluation:** a multi-faceted evaluation combining benchmark accuracy assessment, a statistically representative precision audit of the mined corpus, and qualitative open-coding analysis of model reasoning across three languages.

Replication. A replication package containing all data and artifacts is publicly available on Figshare.[1]

2 RELATED WORK

Mining software repositories for performance bugs is essential due to their severe impact on software reliability; however, existing approaches face critical limitations in scalability and semantic understanding. Early empirical studies characterized performance bugs through recurring root causes such as redundant computations and inefficient I/O operations [14], prolonged resolution times compared to functional bugs [31], frequent reliance on optimizations rather than simple patches [23], and common anti-patterns such as inefficient synchronization and unnecessary computation [4]. Despite these characterizations, identifying performance-related commits systematically and at scale remains challenging.

Traditionally, keyword-based methods filtered commit messages and issue trackers for manual categorization [6, 19], while heuristic linguistic approaches automatically flag performance issue reports but risk biases from predefined patterns and limited cross-ecosystem generalizability [32]. Radu and Nadi's study of non-functional bugs further highlights constraints in dataset scale and representativeness [25]. These surface-level approaches inherently miss commits whose messages describe performance improvements without using explicit keywords, and require substantial manual effort to adapt across project vocabularies and ecosystems.

More recently, transformer-based models have advanced repository mining through BERT-based commit intent classification [8], CodeBERT co-training for limited-label settings [17], bimodal BERT for Just-In-Time defect prediction [13], and binary or multi-label commit classification [30]. However, none of these approaches target large-scale mining of performance improvement commits or leverage knowledge distillation from LLMs for semantic classification.

3 METHODOLOGY

Figure 1 illustrates the *PerfMiner* pipeline, which operates in two phases. In the *model-development* phase, we (i) select and crawl repositories to build a candidate commit pool, (ii) deduplicate and apply single-file granularity filtering, and (iii) train a compact classifier (*PerfAnnotator-mini*) via response-based (hard-label) knowledge distillation from a 20B-parameter open-weight LLM teacher. In the *dataset-curation* phase, we (iv) deploy *PerfAnnotator-mini* to mine performance-related commits from GitHub at scale (≈170 M commits), producing a precision-audited dataset of ≈335 K commits. The following subsections detail each stage.

[1] https://figshare.com/s/48bc6fcc2eb1f94c41d9

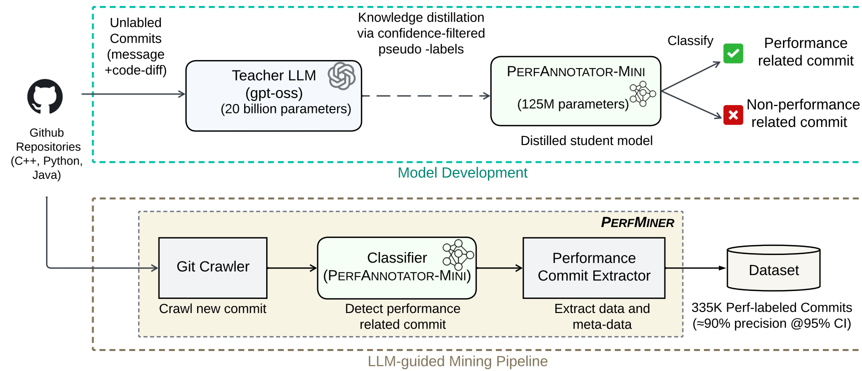


Figure 1: Overview of the *PerfMiner* pipeline. *Model-development phase*: repository selection and commit extraction, deduplication and filtering, teacher labeling with a large LLM, and response-based (hard-label) knowledge distillation into a compact classifier (*PerfAnnotator-mini*). *Dataset-curation phase*: large-scale classification of ≈170 M commits and precision-audited corpus.

3.1 Data Preparation

Repository selection: Following established mining software repository practice [2], we mined public GitHub repositories via the GraphQL API [9] and PyDriller [26]. Consistent with prior performance-related studies [7], repositories with fewer than five stars were excluded to filter out trivial or personal projects. Forked, inaccessible, and private repositories were skipped; only main-branch commits were crawled. The crawl (December 2024) yielded 323,700 repositories and ≈170 M commits from an initial 2 M candidates. This introduces a bias toward popular projects; we discuss this trade-off in Section 6. We scope the study to C++, Java, and Python as a practical choice aligned with prior performance-commit research [4, 15]; the framework itself is not specific to these three languages and can in principle be extended to additional languages in future work.

Commit representation: Each commit is represented as *message* || *code diff*, combining developer intent with implementation evidence [20]. For long inputs, we concatenate title, full message, and diff hunks (ordered by edit size), truncating the tail at the context limit. A comparison of *message-only* versus *message+diff* inputs is reported in Section 4.

Deduplication and granularity: We deduplicated identical diffs using content hashes to remove repeated changes within repositories. We excluded commits that modify multiple files or subsystems, following prior work on tangled changes [12], to focus on localized, single-file edits where performance-related intent can be unambiguously identified.

3.2 LLM-Guided Classification Framework

Large language models (LLMs) offer strong semantic reasoning for interpreting commit intent, but their computational cost makes them impractical for large-scale mining [5]. Because no large-scale *human-annotated* corpus of performance-related commits exists, we employ knowledge distillation (KD): a large LLM teacher generates pseudo-labels with associated confidence scores for an unlabeled commit pool, and a compact student model is trained on the confidence-filtered subset of these pseudo-labels [18, 28]. This hard-label distillation strategy balances semantic interpretive capability with scalable deployment on commodity CPU hardware.

Model selection: We selected `gpt-oss:20b`, an open-weight 20B reasoning LLM that matches or exceeds o3-mini on standard benchmarks despite its compact size,[2] as the *teacher* for its strong instruction-following ability, open licensing, and reproducibility. For the *student*, we adopt GraphCodeBERT [11], a 125M-parameter transformer model as the base encoder and distill it into a task-specific classifier, *PerfAnnotator-mini*. GraphCodeBERT is chosen for its joint pretraining on multiple programming-language and natural-language corpora, explicit data-flow modeling, and ability to run efficiently on CPU hardware. This combination is intended to leverage the teacher's semantic reasoning capability to create a lightweight, domain-adapted model for large-scale repository mining. We compare *PerfAnnotator-mini* against the teacher itself, scored on the same ground truth evaluation set as a reference (Table 1, F1 = 0.90), and multiple baselines spanning heuristic, supervised, and weakly-supervised approaches.

Teacher-labeled corpus and leakage control: For distillation training, we sampled commits from the 170 M-commit pool, stratified by language (C++, Java, Python) and repository popularity. The teacher labeled each sample using the prompt in Figure 2, producing a discrete label and a verbalized confidence score. We retained only predictions whose confidence satisfied $C_i \geq \tau$ (with $\tau = 85$), yielding ≈100K high-confidence pseudo-labeled examples (≈50% positive). This threshold was empirically selected via the ablation study reported in Section 4. We enforce repository-level disjointness between the KD corpus and the development and ground-truth evaluation sets to prevent data leakage.

Prompting and inference (teacher): The teacher operates in a deterministic zero-shot setting (temperature = 0) with a structured prompt containing (i) task description, (ii) target labels {`Perf`, `Non-perf`}, and (iii) concise label definitions (Figure 2). The prompt

[2]https://openai.com/index/introducing-gpt-oss/

Prompt template with label definition

Task Description. You are an expert software developer specializing in performance engineering. Analyze the commit message and code diff to determine whether the change is *performance-related*. You have to assign a label. Possible labels are: {`Perf`, `Non-perf`}.

Label Definitions.
- **Perf:** Commits that explicitly or implicitly aim to improve execution efficiency or resource utilization. This includes optimizations targeting runtime, latency, CPU, memory, I/O, network throughput, algorithmic or data-structure efficiency, and concurrency/parallelism. The intent may be evident from either the commit message or the code change itself.
- **Non-perf:** Commits that focus on other software quality aspects—such as correctness, maintainability, documentation, logging, readability, testing, or refactoring—without a clear intent or effect on performance.

Use lower confidence values when the developer's intent is ambiguous.

Figure 2: Prompt template used *only* during teacher labeling (zero-shot; task + label descriptions; temperature = 0). The student model (*PerfAnnotator-mini*) does not use this prompt at inference time.

also instructs the teacher to lower its confidence when the developer's intent is *ambiguous*, i.e., when the model cannot confidently infer the class from the message and diff alone. We report discard rates and class balance in Section 4.

Knowledge distillation procedure: Inspired by recent advances in transferring LLM-derived supervision into smaller, task-specific models [18], we distill the teacher's labels into *PerfAnnotator-mini* via a hard-label knowledge distillation framework. Following findings that instruction-tuned LLMs can produce well-calibrated verbalized confidence scores [28], which are better suited to capture uncertainty from reasoning models (`gpt-oss:20b` teacher) than single-token logprobs, we pair each discrete pseudo-label with a confidence score. Consistent with the empirical finding of [18] that teacher confidence correlates with pseudo-label quality, we observe the same monotonic pattern on a randomly sampled subset of our ground-truth data (n=300): teacher accuracy rises from 40% at C_i<80 to 93% at C_i≥95 (Pearson r=0.22, p<0.001; ECE = 0.032). For each unlabeled commit $x_i \in \mathcal{X}$, we prompted the teacher to output both a discrete semantic label $y(x_i)_{pl} \in \{0, 1\}$ (representing `Non-perf` and `Perf` respectively) and a verbalized confidence score $C_i \in [0, 100]$.

To mitigate the label noise inherent to pseudo-labeling and ensure high-quality knowledge transfer, we instituted a strict thresholding strategy, discarding any predictions where the teacher's confidence fell below a predefined threshold τ. The student model G_s (*PerfAnnotator-mini*) is trained to minimize the standard cross-entropy loss $\mathcal{H}$ exclusively on the subset of highly confident examples. Formally, for a batch of size B, the distillation loss $\mathcal{L}$ is defined as:

$$\mathcal{L} = \frac{1}{\sum_{i=1}^{B} \mathbf{1}(C_i \geq \tau)} \sum_{i=1}^{B} \mathbf{1}(C_i \geq \tau) \cdot \mathcal{H}\big(y(x_i)_{pl}, p_{G_s}(y \mid x_i)\big) \quad (1)$$

where $p_{G_s}(y \mid x_i)$ is the student's predicted probability over labels.

We set the confidence threshold to τ = 85 (selected via an ablation study in Section 4). We optimized the student model using AdamW (learning rate 2×10^{-5}, batch size 32) with early stopping based on validation metrics. After distillation, *PerfAnnotator-mini* classifies commits directly from the raw *message* || *diff* input, producing a binary prediction without requiring any prompt or teacher involvement at inference time.

Teacher input window: To avoid exposure mismatch during knowledge distillation, we restrict the teacher's inputs to the student's maximum context (512 tokens), ensuring that labels reflect only information the student can access at inference time.

At deployment, however, 241,832 commits exceeded the student's 512-token window; their diffs were truncated from the end during online annotation. These commits remain in the released dataset, as the message-only ablation (Section 4.1) shows that commit messages alone sustain F1 = 0.87. The precision audit and diversity analysis (RQ4) are conservatively scoped to within-context commits.

Baseline definitions: To contextualize *PerfAnnotator-mini*'s accuracy, we evaluate against four baseline categories: (i) a *keyword baseline* using 95 consolidated regex terms and bigram rules from prior work [14, 15, 32]; (ii) *supervised ML* (TF–IDF + Linear SVM and Logistic Regression), commonly used in SE literature for classification tasks [8, 32], trained on clean, manually annotated labels; (iii) *fine-tuned GraphCodeBERT* [11], sharing *PerfAnnotator-mini*'s encoder but trained on manual labels to isolate the supervision signal; and (iv) *weakly-supervised ML* (TF–IDF + SVM/LR) trained on the same 100K teacher pseudo-labels as *PerfAnnotator-mini*, isolating model architecture.

Large-scale classification: After training, *PerfAnnotator-mini* was deployed for large-scale inference across the 170 M-commit corpus (Section 3.4). Each commit, represented as the concatenation of its message and unified diff, was processed independently to assign a binary label (`Perf` or `Non-perf`). The resulting predictions produced a mined dataset of ≈ 335 K commits automatically classified as performance-related across ≈ 323 K repositories—to our knowledge, the first publicly available performance-commit dataset curated at ecosystem scale with audited precision of 0.90 (95% CI, ±5%). This dataset underlies the large-scale audit (Section 4.1) and the scalability and diversity analyses in Sections 4.3–4.4.

3.3 Dataset Construction and Validation

We use five non-overlapping datasets serving distinct purposes: (i) a *teacher-labeled KD corpus* (≈100K high-confidence examples) for training the distilled student; (ii) a *development set* (300 commits; balanced; manually annotated) for hyperparameter tuning and early stopping across all models; (iii) a *ground-truth evaluation set* (1,350 commits; balanced; manually annotated) for final accuracy evaluation—fully supervised baselines are evaluated via 5-fold out-of-fold prediction on this set, while weakly-supervised and distilled models are trained externally and evaluated directly; (iv) a *sample-based audit* (384 commits; 95% CI, ±5%) for estimating real-world precision at *scale*; and (v) an *RQ2 qualitative subset* (400 commits) drawn from the ground-truth dataset, stratified across TP/FP/FN outcomes and languages, for open-coding analysis of model reasoning patterns.

Ground-truth dataset: To evaluate classification accuracy, we manually curated a balanced dataset of 1,650 commits, strictly partitioned into a 300-commit development set for hyperparameter tuning and a 1,350-commit ground-truth evaluation set for final accuracy reporting. Performance-related examples were drawn from and extended beyond prior corpora [4, 15, 25]; all commits were re-annotated under a unified guideline regardless of their

source. Non-performance examples were matched by repository, time window, and file type to control context. Two annotators labeled independently, with 3.5 and 8 years of experience in software and performance engineering, respectively; disagreements were resolved by discussion (Cohen's $\kappa = 0.86$). During annotation, both the commit message and the corresponding code diff were inspected to ensure that textual intent aligned with concrete performance-related code changes. Labels were assigned based on semantic evidence rather than runtime benchmarks or performance tests, reflecting developers' optimization intent and implementation actions. The dataset covers ten broad performance categories (algorithmic, memory, I/O, library, concurrency, etc.) and is stratified equally by language (C++: 450, Java: 450, Python: 450 in the evaluation set; 100 each in the development set). The development set was used for hyperparameter tuning of all models, including grid search for the supervised baselines and early stopping for *PerfAnnotator-mini*. No development-set commits appear in the evaluation set or KD corpus. Commit hashes, annotations, and the labeling guidelines are publicly released in the artifact.

Evaluation protocol: Fully supervised baselines are evaluated on the 1,350-commit ground-truth evaluation set using 5-fold cross-validation: we report metrics over aggregated out-of-fold predictions, so each commit is scored by a model that did not train on it. Hyperparameters are selected on the disjoint 300-commit development set and then fixed. Weakly-supervised models and *PerfAnnotator-mini* are trained entirely on the teacher-labeled corpus and evaluated once on the same 1,350 commits. We report precision, recall, and F1 averaged over three random seeds.

Large-scale audit: Because our primary goal is to curate a high-quality dataset of performance-related commits, we conduct a separate precision audit on the mined corpus. We randomly sample 384 commits (95% CI, ±5%) from the output of *PerfAnnotator-mini*, stratified by language and repository popularity. Two annotators verified each label independently using the same message-diff inspection protocol as the ground-truth dataset; disagreements were resolved by consensus (Cohen's $\kappa = 0.86$). We report only precision, as the audit is sampled exclusively from *PerfAnnotator-mini*'s predicted positives—recall and F1 are undefined on this set. Since the audit is conditioned on *PerfAnnotator-mini*'s predictions, it estimates the reliability of the mined corpus rather than comparative model performance; the latter is evaluated separately on the ground-truth evaluation set.

Qualitative analysis protocol: For RQ2, we conducted an open-coding analysis to interpret the semantic reasoning underlying PerfMiner's classifications. We analyzed model behavior on a stratified set of 400 commits, including true positives, false positives, and false negatives, sampled proportionally across C++, Java, and Python projects to ensure linguistic and domain diversity. Two authors independently reviewed each commit's message-diff pair, coding for (i) performance intent in commit messages (e.g., latency, throughput, resource use, or optimization rationale) and (ii) implementation signals in code diffs (e.g., memory management, algorithmic changes, I/O reduction). The coders iteratively grouped semantically similar cues into higher-level categories until thematic saturation was achieved. Inter-rater reliability, measured on the independent coding phase using Cohen's κ (0.87), indicated strong agreement; remaining disagreements were reconciled through discussion to produce the final consensus categories. The resulting six reasoning patterns form the basis of the qualitative findings presented in Section 4.2.

3.4 Implementation and Infrastructure

PerfMiner is deployed on a 50-node CloudLab CPU cluster (Intel Xeon D-1548, 64 GB RAM per node). The system adopts a distributed inference architecture designed to efficiently process large volumes of commits using commodity hardware. Each node operates a lightweight worker that performs three stages in parallel: (i) *crawling* repository data through rate-limit-aware API queries, (ii) *preprocessing and feature extraction* to construct commit-level representations (message and diff), and (iii) *classification* using the distilled student model (*PerfAnnotator-mini*). A central controller coordinates task scheduling, load balancing, and checkpointing through a shared queue backed by an on-disk database. Intermediate artifacts (raw commits, parsed diffs, model outputs, and logs) are written to distributed storage and periodically aggregated for progress monitoring. Workers communicate asynchronously to minimize idle time and recover gracefully from transient API or network failures.

4 EVALUATION

This section empirically evaluates *PerfMiner* and its distilled classifier *PerfAnnotator-mini* with respect to four research questions: accuracy (RQ1), qualitative reasoning (RQ2), scalability and cost efficiency (RQ3), and dataset diversity (RQ4).

4.1 RQ1 (Accuracy): Evaluating *PerfAnnotator-mini* for Commit-Level Classification

Motivation: Current approaches to mining performance-related commits rely predominantly on keyword filters and heuristic rules, which capture only surface-level lexical patterns and miss semantically diverse changes (Section 1). Even supervised models trained on our 1,350-commit ground-truth set (Table 1) plateau at modest accuracy, as we show below. The core difficulty is that performance intent is often implicit: developers may omit performance cues from the commit message, while the performance impact may be subtle in the diff alone. We hypothesize that combining commit messages and code diffs provides complementary signals for more accurate LLM-guided detection: the message captures developer intent while the diff supplies implementation evidence. To test this, we developed *PerfAnnotator-mini*, a compact 125 M-parameter classifier trained via hard-label knowledge distillation from the 20 B teacher (Section 3.2), and evaluate whether it can surpass baselines trained on human-labeled data.

Evaluation setup: Using the datasets and baselines defined in Section 3.3, we evaluate *PerfAnnotator-mini* under two scenarios: (1) classification accuracy on the 1,350-commit ground-truth evaluation set, and (2) deployment precision on the 384-commit audit sample. All baselines are defined in Section 3.2 and evaluated under identical conditions (Section 3.3).

Results. Table 1 summarizes all results. On the 1,350-commit ground-truth evaluation set, *PerfAnnotator-mini* achieves **F1 =**

Table 1: Classification results on the 1,350-commit ground-truth evaluation set. Supervised baselines use 5-fold CV with aggregated out-of-fold predictions (hyperparameters fixed on the 300-commit development set); weakly-supervised and distilled models are trained on LLM pseudo-labels and evaluated directly. The final row reports a separate dataset-quality audit on the mined corpus (precision only; recall/F1 undefined).

Method	Prec.	Rec.	F1
Supervised baselines (hard labels, 5-fold CV)			
TF–IDF + Linear SVM	0.63	0.75	0.69
TF–IDF + Logistic Regression	0.63	0.76	0.69
GraphCodeBERT (fine-tuned)	0.66	0.74	0.70
Weakly supervised (100K LLM pseudo-labels)			
TF–IDF + Linear SVM	0.77	0.84	0.80
TF–IDF + Logistic Regression	0.77	0.84	0.80
Distilled model (100K LLM pseudo-labels)			
PerfAnnotator-mini	**0.89**	**0.89**	**0.89**
Reference LLMs (zero-shot, no training)			
gpt-oss:20b (teacher)	0.91	0.90	0.90
GPT-4o-mini	0.87	0.84	0.86
Heuristic baseline (rule-based)			
Keyword baseline	0.65	0.58	0.61
Dataset-quality audit (384 mined commits, precision only)			
PerfAnnotator-mini	0.90	–	–

0.89 and **precision = 0.89**, outperforming the keyword-based baseline (F1 = 0.61) and all supervised and weakly-supervised baselines—notably surpassing GPT-4o-mini (F1 = 0.86) despite being orders of magnitude smaller. Confirming the label-budget limitation noted in the motivation, SVM, logistic regression, and fine-tuned GraphCodeBERT all plateau at F1 ≈ 0.69–0.70, well below the teacher's F1 = 0.90; this ceiling persists regardless of model capacity, indicating that the principal bottleneck is *data scale*. Rather than scaling up costly human annotation, *PerfAnnotator-mini* trains on 100 K confidence-filtered pseudo-labels via hard-label distillation. To isolate whether this gain stems from the pseudo-label corpus or the encoder architecture, we trained the same TF–IDF classifiers on the identical pseudo-labels; their F1 improved to 0.80 but remained ≈9 points below *PerfAnnotator-mini*, confirming that both large-scale LLM-derived supervision *and* a code-aware architecture are necessary.

As a separate dataset-quality assessment, the 384-commit precision audit (Section 3.3) confirmed a precision of **0.90** on the corpus mined by *PerfAnnotator-mini*; because this audit is randomly sampled exclusively from predicted positives, recall and F1 are undefined on this set.

Ablation studies. We conduct two ablations to isolate the contributions of key design choices in our model development.

(i) Contribution of code diffs. To quantify the impact of including code diffs alongside commit messages, we trained a message-only variant of *PerfAnnotator-mini* using identical settings and evaluated it on the same ground-truth dataset. Removing diff context led to a noticeable decline in classification quality: precision dropped from 0.89 to 0.86, and F1 from 0.89 to 0.87. This confirms that code-diff information provides essential evidence for distinguishing performance-related commits from semantically similar but non-performance changes.

(ii) Teacher-confidence threshold. We varied the teacher-label acceptance threshold τ to study the trade-off between label coverage and precision during knowledge distillation. We evaluated model performance across three teacher-confidence thresholds ($\tau \in \{75, 85, 95\}$) and found that $\tau = 85$ achieved the best overall performance (precision = 0.89, recall = 0.89, F1 = 0.89). This configuration provides an optimal balance between data coverage and label reliability, outperforming the more conservative $\tau = 95$ setting (−47% samples, Precision = 0.86, F1 = 0.87) and the more permissive $\tau = 75$ setting (+23% samples, Precision = 0.87, F1 = 0.88). The superior performance at $\tau = 85$ likely stems from its balanced supervision quality, leveraging sufficient high-confidence teacher signals without incorporating the noisier labels introduced at lower thresholds.

Error analysis: Manual inspection reveals two primary sources of misclassification:

- **False positives:** commits whose messages include performance-related terms but whose diffs correspond to neutral refactorings without any runtime impact.
- **False negatives:** commits where performance improvements are semantically evident in the code diff, such as memory optimizations, reduced algorithmic complexity, or I/O batching, but not explicitly stated in the message.

We consider such cases false negatives because classification uses both message and diff; when the diff clearly implements a performance-improving action, the model is expected to detect it even without explicit textual mention.

Across programming languages, *PerfAnnotator-mini* maintained stable accuracy: F1 varied between 0.87–0.91 (C++, Java, Python). Accuracy was highest for algorithmic and memory optimizations, slightly lower for concurrency-related commits, which often involve subtle synchronization patterns. Results were consistent across random seeds (standard deviation < 0.01 for F1), indicating low variance and model stability.

Finding (RQ1). On the 1,350-commit ground-truth evaluation set, *PerfAnnotator-mini* achieves **F1 = 0.89** and **precision = 0.89**, outperforming all supervised (F1≤0.70) and weakly-supervised (F1≈0.80) baselines while approaching the 20B-parameter teacher (F1 = 0.90). Deployment precision on the 384-commit audit is **0.90**. Ablations confirm the critical role of diff context and balanced teacher-label confidence ($\tau = 85$).

4.2 RQ2: Interpreting *PerfAnnotator-mini*'s Semantic Reasoning on Performance Commits

Motivation: While quantitative evaluation establishes *PerfAnnotator-mini*'s precision and scalability, it does not reveal the *semantic basis* of its predictions. To examine this, we qualitatively analyze the semantic and contextual cues captured by *PerfAnnotator-mini*

across diverse repositories, identifying six recurring reasoning patterns (S1–S6).

Method. We derived the six patterns via open coding of a stratified sample of 400 commits (true positives, false positives, false negatives across C++, Java, and Python); the full protocol is described in Section 3.3. The codebook and coded data are available in the replication package.

S1: Lexical and Functional Reasoning. *PerfAnnotator-mini* effectively identifies performance-related commits when developers explicitly articulate optimization intent through terms such as "optimize," "reduce time," or "improve efficiency." These commits often involve substitutions of APIs or function calls with known performance trade-offs, such as replacing standard I/O operations with buffered equivalents or using compiled numerical libraries in place of interpreted routines. *PerfAnnotator-mini*'s ability to align explicit linguistic cues with functional transformations in the diff demonstrates robust lexical grounding. For instance, in the *openSUSE/salt* repository,[3] the commit message *"make the redis-returner use a pipeline … 4 times faster"* was correctly identified: the code replaces individual Redis calls with a batched `pipeline()` and executes them together, yielding substantial runtime savings (Listing 1).

Listing 1: Redis Pipeline Optimization (4× speedup)

```
1 def returner(ret):
2   serv = _get_serv(ret)
3 - serv.set('{0}:{1}'.format(ret['id'], ret['jid']), json.dumps(ret))
4 + pipe = serv.pipeline()
5 + pipe.set('{0}:{1}'.format(ret['id'], ret['jid']), json.dumps(ret))
6 + pipe.execute()
```

S5: Cross-Modal Code-Text Reasoning. Some commits exhibit performance implications that become apparent only when message and code are interpreted together. In the *Circleguard* project,[4] the commit message *"Cache replays if enabled in settings"* sounds functional, describing a configuration change rather than an optimization. However, the code diff (Listing 2) introduces a caching mechanism that avoids redundant computation and repeated file access. *PerfAnnotator-mini* correctly classifies this commit as performance-related despite the absence of explicit optimization language, indicating that it captures cues jointly from code and text to detect performance-oriented behavior.

Listing 2: Adding caching logic controlled by settings

```
1 +   set_options(cache=bool(get_setting("caching")))
2     cg = Circleguard(get_setting("api_key"), os.path.join(
        get_setting("cache_dir"), "cache.db"))
```

Additional patterns (S2–S4, S6): Beyond these two patterns, we observed four additional reasoning patterns. **S2: Structural/Algorithmic.** *PerfAnnotator-mini* infers optimization intent from structural modifications that improve data flow or reduce computation, such as eliminating a redundant XenAPI lookup in *OpenStack Nova*[5] by reusing an existing VM reference. **S3: Latent Optimization through Correctness.** some correctness fixes incidentally improve runtime; for example, releasing a read lock after transaction completion in *eXist-db*[6] resolved a deadlock while also reducing lock contention. **S4: Domain-Specific Semantic Transfer.** *PerfAnnotator-mini* generalizes to domain-specific idioms, such as replacing LLVM's `dyn_cast<>` with the cheaper `isa<>` in *TensorFlow*[7], a subtle compiler-level optimization lacking explicit textual hints. **S6: Subtle/Implicit.** optimization intent is sometimes intertwined with domain conventions; in *Clang*[8], passing the `inline` keyword as an `InlineHint` attribute is invisible at runtime but crucial for code generation efficiency. Full code listings and annotated exemplars for all six patterns are available in the replication package.

Implications: The six reasoning patterns span a spectrum from explicit lexical cues (S1) to latent code-level inference (S6), demonstrating that *PerfAnnotator-mini*'s reasoning extends well beyond keyword matching. This diversity of reasoning strategies provides evidence that the model's classifications—and by extension, the mined dataset's labels—are grounded in substantive semantic signals, not surface heuristics. While RQ4 characterizes *what* types of performance issues the dataset contains (via topic modeling), the patterns here characterize *how* the model identifies them, providing complementary evidence of classifier trustworthiness.

> **Finding (RQ2):** Qualitative analysis of 400 stratified commits identifies six recurring reasoning patterns—from explicit lexical cues (S1) to latent code-level optimizations (S6)—demonstrating that *PerfAnnotator-mini*'s classifications are grounded in diverse semantic signals beyond keyword matching, across C++, Java, and Python.

4.3 RQ3: Scalability and Cost Characteristics

Motivation: Although recent models demonstrate strong semantic reasoning capabilities, their computational requirements pose challenges when applied to corpora comprising tens or hundreds of millions of commits. To investigate these trade-offs, we analyze the inference throughput and deployment cost of *PerfAnnotator-mini* relative to its teacher (`gpt-oss:20b`); while traditional ML/DL alternatives may be cheaper, their lower accuracy (F1≈0.70, Table 1) introduces more label noise, undermining our goal of curating a high-quality performance-commit dataset.

Experimental setup: Throughput (*requests per second, req/s*) was measured end-to-end with batch size 1, including tokenization and I/O overhead. Benchmarks were executed on a GPU workstation (Nvidia RTX 4090) for both the student and teacher models. Per-node CPU throughput was measured on CloudLab nodes (Intel Xeon D-1548, 64 GB RAM). Each request corresponds to the complete processing of one commit, where the input combines the commit message and diff (Section 3.1). The 20B-parameter teacher could not be executed on these CPU nodes due to memory constraints; therefore, CPU measurements are reported only for the distilled model.

[3]https://github.com/openSUSE/salt/commit/903e6f7559135d1f38b01ca0782cb3dc7f91a526
[4]https://github.com/circleguard/circleguard/commit/be3cbc057fd430d9eb3492f8cec39d00415afb50
[5]https://github.com/openstack/nova/commit/d879dc9e0d28aa203426729c6cfea48bdbfa3cfd
[6]https://github.com/eXist-db/exist/commit/42b43b91c453d6921c3284ddacb7581c1b072319
[7]https://github.com/alpa-projects/tensorflow-alpa/commit/95cd28aa7aa6ff97ac12494d26246764f923012b
[8]https://github.com/cpi-llvm/clang/commit/a3fe2842e0cf953241ccc05809afdf84f13798e9

Throughput and cost analysis: Table 2 reports the observed inference rates. On GPU, *PerfAnnotator-mini* achieves 174.9 req/s compared to 0.458 req/s for the teacher, representing a 381× speed-up under identical hardware conditions. On CPU, a single node sustains 2.1 req/s.

Table 2: Measured end-to-end inference throughput (batch=1). GPU rows compare student and teacher on identical hardware; the CPU row reports per-node throughput.

Model / Device	Throughput (req/s)
PerfAnnotator-mini (Nvidia RTX 4090 GPU)	175
`gpt-oss:20b` (Nvidia RTX 4090 GPU)	0.46
PerfAnnotator-mini (Intel Xeon D-1548 CPU, 1 node)	2.1

Using these throughputs and current cloud pricing (AWS `us-east-1`, October 2025), Table 3 reports projected wall-time and dollar cost normalized to the 170M raw-commit scale; end-to-end mining cost depends on pre-filtering. Values assume linear scaling and full utilization; training and storage costs are excluded.

Table 3: Estimated inference cost normalized to 170M raw commits (AWS on-demand pricing, October 2025). We assume parity with RTX 4090 for throughput; real A10G throughput is typically lower.

Deployment	Wall-time	Cost (USD)	$/1M
PerfAnnotator-mini (CPU, 50×c6a.large)	≈18 days	$1.7 K	$10.1
PerfAnnotator-mini (GPU, 1×A10G)	≈11 days	$0.27 K	$1.6
`gpt-oss:20b` (GPU, 1×A10G)	≈11.8 yrs	$104 K	$610
GPT-4o-mini (API)[9]	–	$2.5 M	$14.8 K

Beyond bulk mining, *PerfAnnotator-mini*'s CPU footprint also positions it for integration into CI/CD pipelines and code-review tooling, which we leave to future work.

> **Finding (RQ3):** Knowledge distillation reduces inference cost by over two orders of magnitude ($104K⟶$500 GPU, $1.7K CPU cluster) while retaining F1≈0.89, making ecosystem-scale performance-commit mining practical on commodity hardware—in contrast, proprietary API labeling (≈$2.5M) remains prohibitively costly.

4.4 RQ4: Diversity and Representativeness

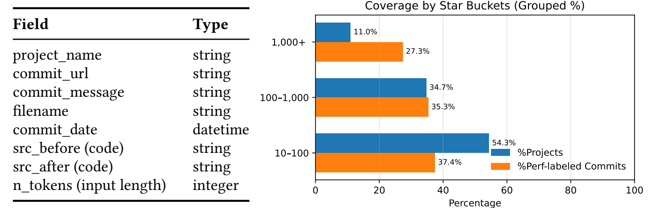
Motivation. RQ4 examines the mined corpus itself, characterizing its language coverage, repository popularity tiers, and performance themes for downstream empirical reuse.

Language and category coverage. Table 4 summarizes the dataset statistics. The *PerfMiner* pipeline produced approximately 335K perf-labeled commits, including 144K in C++, 87K in Java, and 103K in Python. The total uncompressed size is approximately 48.8 GB, with 33.5 GB for C++, 6.0 GB for Java, and 9.3 GB for Python. This distribution follows our filtering criteria (Section 3.1), which retain single-file, localized changes; this matches the audited precision of ≈0.90 reported in RQ1. A total of 241,832 commits exceeded the student model's context window and were truncated during annotation (Section 3.2). To our knowledge, this is the first large-scale, multi-language corpus of real-world performance-related commits.

Table 4: Perf-labeled commits (after deduplication) by language.

Language	Repositories	Perf-labeled commits
C++	125,000	144,498
Java	98,700	87,180
Python	100,000	103,425

Repository popularity and dataset schema. Figure 3 shows the proportion of projects and perf-labeled commits across GitHub popularity tiers (right); the corpus spans repositories from niche to widely-used projects. The left panel shows the dataset record fields (JSONL format).



Field	Type
project_name	string
commit_url	string
commit_message	string
filename	string
commit_date	datetime
src_before (code)	string
src_after (code)	string
n_tokens (input length)	integer

Figure 3: (Left) Dataset record fields (JSONL format); full details in replication package. (Right) Distribution of projects and perf-labeled commits by popularity tier (GitHub stars).

Performance themes. To examine thematic variation, we applied BERTopic [10] to high-confidence commits (≥0.90), yielding 47 fine-grained topics; two annotators consolidated these into ten broad categories.[10]

Following prior work on clustering analysis [21, 24], two annotators collaboratively reviewed the top 20 representative commits per cluster and consolidated semantically overlapping ones into ten broad categories, leveraging established performance taxonomies [15, 31]. Figure 4 shows the ten themes and their relative prevalence across C++, Java, and Python. Commonly recurring themes include memory management, concurrency, API efficiency, and resource optimization. While these themes have been identified in prior manual-scale studies [14, 15, 31], our large-scale analysis confirms their prevalence, demonstrating that the dataset captures diverse performance patterns across languages.

Cross-language themes. Top-10 topic overlap is moderate (Jaccard: C++–Java = 0.70, Java–Python = 0.58, C++–Python = 0.64), indicating shared core concerns (memory management, concurrency) alongside expected language-specific themes (JVM tuning, Python I/O).

> **Finding (RQ4).** The corpus contains ≈**335K** performance-related commits across C++, Java, and Python, organized into ten recurring topic-model themes; cross-language Jaccard overlap (0.58–0.70) reflects both shared and language-specific concerns.

[9] API cost from observed mean input length 78.45 tokens and ≈5 output tokens at GPT-4o-mini pricing ($0.15 / 1 K input, $0.60 / 1 K output) ≈ $0.0148 per commit.

[10] Configuration: `all-mpnet-base-v2` message embeddings, `min_cluster_size=50`; mean coherence $c_v = 0.372$ ($\sigma = 0.01$) and inter-topic cosine separation 0.341.

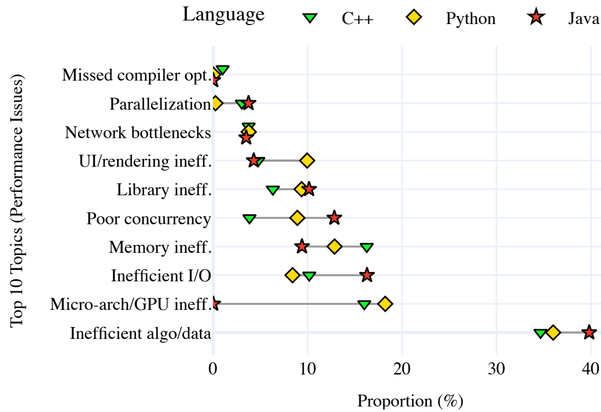


Figure 4: Top performance themes discovered by BERTopic (relative prevalence per language).

5 OBSERVATIONS AND POTENTIAL RESEARCH QUESTIONS

Our analysis of performance-related commits reveals consistent patterns in how developers describe and manage optimization work. These observations (O1–O5) motivate a set of potential research questions (PRQs 1–8) summarized in Table 5, illustrating how the dataset can support empirical studies, tool design, and practitioner-focused research. More broadly, the dataset supports (i) characterizing optimization behavior and communication in commit histories, (ii) benchmarking and training commit-level models for intent detection and triage, and (iii) retrieval-based resources for tooling such as data-driven automated program repair (APR) or repair recommendation for performance issues and LLM-agent benchmarks (e.g., intent understanding, change retrieval, patch suggestion).

O1. Performance work often accompanies regular maintenance. Performance improvements frequently appear with bug fixes or refactorings rather than as standalone optimizations. This observation motivates PRQs 1 and 4, focusing on when performance work arises during project evolution and how tools can detect optimization within maintenance changes.

O2. Developers express performance intent implicitly. Efficiency changes are described through neutral terms like "reduce overhead" or "improve stability," rarely using explicit performance language. This motivates PRQ 2 on identifying semantic and contextual cues that reveal latent optimization intent.

O3. Language ecosystems differ in optimization practices. C++ commits emphasize algorithmic or memory efficiency, while Python and Java focus on throughput or scalability. This motivates PRQ 3 on cross-language and domain differences and how classifiers can adapt to them.

O4. Performance intent overlaps with other quality concerns. Many commits improving efficiency also target maintainability or reliability. This connects to PRQs 4, 5, and 7 on multi-intent classification, regression triage, and the role of performance signals in code review.

O5. The rationale for performance changes is seldom explicit. Commit messages often describe what was changed but not why or how it affects efficiency. This relates to PRQs 6 and 8 on recovering rationale and on using mined examples as reusable evidence for documentation and example-based assistance.

Table 5: Potential research questions (PRQs) illustrating how different *communities* can explore multiple *research dimensions* using the dataset.

Potential Research Questions (PRQs) and Related Work
Software Engineering Researchers
PRQ-1: How frequently does performance-related activity occur alongside maintenance or bug-fix changes, and how does this vary over time?
PRQ-2: What linguistic or contextual cues can distinguish preventive optimizations from reactive ones in commit histories? [29]
PRQ-3: How do optimization practices and the expression of efficiency intent differ across programming languages and application domains?
Tool Builders
PRQ-4: How can mining tools automatically detect commits that address multiple non-functional goals, such as performance and maintainability (e.g., multi-label intent detection from message+diff)? [8]
PRQ-5: How can commit-level data be leveraged to support profiling, regression triage, test-selection, or data-driven repair recommendation (APR) for performance issues in CI (e.g., agentic triage/patch suggestion)?
Developers and Practitioners
PRQ-6: How do developers perceive and trust automatically generated labels of performance intent in their workflows? [1]
PRQ-7: To what extent do performance cues influence code review priorities or testing decisions during maintenance? [27]
PRQ-8: How can mined performance examples be transformed into educational resources or guidelines for performance-aware development (e.g., curated exemplars or retrieval-based assistance), and into benchmark tasks for evaluating LLM agents in performance-aware maintenance?

6 THREATS TO VALIDITY

We acknowledge several threats to the validity of our findings, categorized into construct, internal, and external threats.

Construct Validity: Construct threats concern how accurately our datasets capture performance-related commits. Ground-truth labels combine newly annotated commits with samples from prior corpora (Section 3.3) under a unified guideline (Cohen's $\kappa = 0.86$). Our construct is *static developer intent expressed in commit messages and diffs*, not measured runtime performance; dynamic profiling evidence is a complementary signal for future work.

Internal Validity: Internal threats stem from teacher dependency: misclassifications can propagate to the student. The 384-commit audit (Section 4.1) confirmed stable precision (≈0.9) but cannot quantify systematic LLM-labeler biases (e.g., toward explicit performance keywords or English-language messages); we mitigate via confidence filtering and three-seed averaging.

External Validity: The corpus covers public GitHub repositories (≥5 stars) in C++, Java, and Python; behavior in industrial, private, or other-language codebases may differ. We exclude tangled multi-file commits [12] to scope label semantics, so cross-file performance refactors are under-sampled. Manual labeling at scale is infeasible: our 1,650-commit balanced ground truth (1,350 eval + 300 dev) is, to our knowledge, the largest manually-labeled performance-commit corpus across three languages, while a 384-sample audit (95% CI, ±5%) validates mined-corpus precision at population scale. The released artifact supports downstream reuse for performance-bug detection and patch characterization [22].

7 CONCLUSIONS

This paper introduced *PerfMiner*, an LLM-guided framework for mining performance-related commits at scale. Through knowledge distillation from a large teacher model (`gpt-oss:20b`), we developed *PerfAnnotator-mini*, a compact transformer classifier that preserves semantic accuracy while providing practical scalability. Empirical results show an F1-score of 0.89 and precision of 0.89, enabling analysis of 170 M commits and the creation of an ≈335K-commit, multi-language dataset. *PerfMiner* captures diverse semantic cues underlying performance-related changes, demonstrating that combining semantic reasoning with efficient modeling supports reproducible, large-scale studies of performance-driven software evolution.

ACKNOWLEDGMENTS

This material is based in part upon work supported by National Science Foundation awards NSF#2417471 and NSF#2152819.

REFERENCES

[1] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 291–300.

[2] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Predicting the popularity of github repositories. In *Proceedings of the The 12th international conference on predictive models and data analytics in software engineering*. 1–10.

[3] Jinfu Chen, Weiyi Shang, and Emad Shihab. 2022. PerfJIT: Test-Level Just-in-Time Prediction for Performance Regression Introducing Commits. *IEEE Trans. Softw. Eng.* 48, 5 (May 2022), 1529–1544. doi:10.1109/TSE.2020.3023955

[4] Yiqun Chen, Stefan Winter, and Neeraj Suri. 2019. Inferring Performance Bug Patterns from Developer Commits. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Berlin, Germany, 70–81. doi:10.1109/ISSRE.2019.00017

[5] Giuseppe Colavito, Filippo Lanubile, Nicole Novielli, and Luigi Quaranta. 2024. Leveraging gpt-like llms to automate issue labeling. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 469–480.

[6] Teerath Das, Massimiliano Di Penta, and Ivano Malavolta. 2016. A quantitative and qualitative investigation of performance-related commits in android apps. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 443–447.

[7] Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. 2022. DeepDev-PERF: a deep learning-based approach for improving software performance. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 948–958. doi:10.1145/3540250.3549096

[8] Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud. 2021. Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Information and Software Technology* 135 (2021), 106566. doi:10.1016/j.infsof.2021.106566

[9] GitHub, Inc. 2023. *GitHub GraphQL API Documentation*. https://docs.github.com/en/graphql Accessed: 2024-11-11.

[10] Maarten Grootendorst. 2022. BERTopic: Neural topic modeling with a class-based TF-IDF procedure. arXiv:2203.05794 [cs.CL] https://arxiv.org/abs/2203.05794

[11] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. http://arxiv.org/abs/2009.08366 arXiv:2009.08366 [cs].

[12] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 121–130.

[13] Yuze Jiang, Beijun Shen, and Xiaodong Gu. 2025. Just-in-time software defect prediction via bi-modal change representation learning. *Journal of Systems and Software* 219 (2025), 112253.

[14] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88.

[15] Md Abul Kalam Azad, Nafees Iqbal, Foyzul Hassan, and Probir Roy. 2023. An Empirical Study of High Performance Computing (HPC) Performance Bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, Melbourne, Australia, 194–206. doi:10.1109/MSR59073.2023.00037

[16] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071.

[17] Jian Yi David Lee and Hai Leong Chieu. 2021. Co-training for commit classification. In *Proceedings of the Seventh Workshop on Noisy User-generated Text (W-NUT 2021)*. 389–395.

[18] Juanhui Li, Sreyashi Nag, Hui Liu, Xianfeng Tang, Sheikh Muhammad Sarwar, Limeng Cui, Hansu Gu, Suhang Wang, Qi He, and Jiliang Tang. 2025. Learning with Less: Knowledge Distillation from Large Language Models via Unlabeled Data. In *Findings of the Association for Computational Linguistics: NAACL 2025*, Luis Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 2627–2641. doi:10.18653/v1/2025.findings-naacl.142

[19] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1013–1024. doi:10.1145/2568225.2568229

[20] Abhinav Reddy Mandli, Saurabhsingh Rajput, and Tushar Sharma. 2025. COMET: Generating commit messages using delta graph context representation. *Journal of Systems and Software* 222 (2025), 112307.

[21] Preksha Nema, Pauline Anthonysamy, Nina Taft, and Sai Teja Peddinti. 2022. Analyzing user perspectives on mobile app privacy at scale. In *Proceedings of the 44th International Conference on Software Engineering*. 112–124.

[22] Daniel Nichols, Aniruddha Marathe, Harshitha Menon, Todd Gamblin, and Abhinav Bhatele. 2024. HPC-Coder: Modeling Parallel Programs using Large Language Models. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*. 1–12. doi:10.23919/ISC.2024.10528929

[23] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, reporting, and fixing performance bugs. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 237–246. doi:10.1109/MSR.2013.6624035

[24] Md Nahidul Islam Opu, Shahidul Islam, Sara Rouhani, and Shaiful Chowdhury. 2025. Understanding the Issue Types in Open Source Blockchain-based Software Projects with the Transformer-based BERTopic. arXiv:2506.11451 [cs.SE] https://arxiv.org/abs/2506.11451

[25] Aida Radu and Sarah Nadi. 2019. A Dataset of Non-Functional Bugs. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR)*. IEEE/ACM, 399–403.

[26] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 908–911.

[27] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2017. Review participation in modern code review: An empirical study of the android, Qt, and OpenStack projects. *Empirical Software Engineering* 22, 2 (2017), 768–817.

[28] Katherine Tian, Eric Mitchell, Allan Zhou, Archit Sharma, Rafael Rafailov, Huaxiu Yao, Chelsea Finn, and Christopher Manning. 2023. Just Ask for Calibration: Strategies for Eliciting Calibrated Confidence Scores from Language Models Fine-Tuned with Human Feedback. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 5433–5442. doi:10.18653/v1/2023.emnlp-main.330

[29] Jun Yang, Yuehan Wang, Yiling Lou, Ming Wen, and Lingming Zhang. 2023. A Large-Scale Empirical Review of Patch Correctness Checking Approaches. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco CA USA, 1203–1215. doi:10.1145/3611643.3616331

[30] Sarim Zafar, Muhammad Zubair Malik, and Gursimran Singh Walia. 2019. Towards standardizing and improving classification of bug-fix commits. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–6.

[31] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2012. A Qualitative Study on Performance Bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 199–208.

[32] Yutong Zhao, Lu Xiao, and Sunny Wong. 2024. A Platform-Agnostic Framework for Automatically Identifying Performance Issue Reports With Heuristic Linguistic Patterns. *IEEE Trans. Softw. Eng.* 50, 7 (July 2024), 1704–1725. doi:10.1109/TSE.2024.3390623