Towards LLMinating Performance Anti-Patterns in Android Apps

Rui Rua
NYU Abu Dhabi
rui.rua@nyu.edu

Karim Ali
NYU Abu Dhabi
karim.ali@nyu.edu

***Abstract*—Application performance strongly affects user satisfaction and retention, especially on mobile platforms such as Android. To identify performance inefficiencies in their apps, developers often resort to static analysis tools. However, these tools usually suffer from high false-positive rates since they typically rely on rigid, syntactic pattern-matching rules that lack the semantic understanding of data flow and execution context (e.g. variable states or feasible execution paths) required to distinguish actual inefficiencies from benign code. Large Language Models (LLMs) offer a promising semantic alternative, but their ability to reason about non-functional properties like performance remains under-explored.**

In this paper, we present an empirical evaluation of the targeted detection of established Android performance anti-patterns. We benchmark top-tier LLMs against state-of-the-art Static Analysis Tools (SATs) to determine if this constrained scope allows LLMs to succeed where broader performance analysis fails. By analyzing statically-detectable anti-pattern instances detected in 1048 open-source projects, we conclude that top-tier models can reliably distinguish efficient from inefficient implementations and frequently outperform traditional static analyzers. In particular, Gemini-2.5-pro achieves a precision of 90.8% on real-world projects, significantly outperforming the industry-standard Android Lint (63.8%).

Repl. Package: https://github.com/sanadlab/llmsats_repl_pack

***Index Terms*—Software Maintenance, AI for Software Engineering, Android, Empirical Software Engineering**

I. Introduction

Mobile devices are ubiquitous and operate under strict hardware constraints. Consequently, performance inefficiencies that might be negligible on other platforms manifest as significant lag, unresponsiveness, or rapid battery drain, severely degrading User eXperience (UX) and often leading to app uninstallation [1]. While often functionally correct, performance inefficiencies may be viewed as a type of code smell and an indication of deeper system problems [2]. Though the code may be functionally correct, these issues often point to specific Performance Anti-Patterns (PAPs): recurring implementation choices that unnecessarily compromise latency, memory usage, bandwidth, or energy efficiency.

On Android, diagnosing PAPs is complex due to extreme device/platform fragmentation. Consequently, developers frequently use dynamic analysis through manual or automated execution frameworks [3]. However, dynamic analyses struggle to uncover subtle, context-specific inefficiencies [4]–[6] such as connectivity issues or realistic execution scenarios. Moreover, they are often expensive to run, yielding environment-specific results that may be unfeasible to reproduce [7].

To overcome the challenges that arise from using dynamic analyses, developers have started to rely on SATs as a practical first line of defense against PAPs. Unlike dynamic analyses, SATs are faster to execute and can effectively capture broad semantic patterns. To get precise results, a SAT must analyze the whole program (i.e., application code and its library dependencies). Since running a deep, whole-program analysis requires a lot of computational resources, most SATs default to simple pattern-matching [8]–[10] that does not account for variable states or complex control flows (i.e., context). This limited ability leads to high false-positive rates [11] and significant maintenance overhead [12].

Meanwhile, LLMs have recently demonstrated remarkable capabilities in code comprehension [13], generation [14], and transformation [15]. Their ability to process natural language prompts, learn complex patterns from large code corpora, and grasp broader contextual nuances suggest that they could complement, or even replace, traditional SATs. In particular, LLMs might detect context-sensitive PAPs [16], [17] more accurately. However, current LLMs also face known challenges, including hallucinations [18], limited context windows [19], and difficulties in reasoning holistically about performance [20].

While LLMs still fail at broad performance reasoning, their semantic capabilities can make them uniquely reliable for identifying previously widely studied and established PAPs where context could even determine the validity of a warning. To validate this hypothesis, we present an empirical evaluation in terms of PAPs detection comparing LLMs against state-of-the-art SATs, guided by the following research questions:

- **RQ1:** Can LLMs effectively detect previously established PAPs in Android apps?
- **RQ2:** How do LLMs compare to SATs for PAPs detection?

To answer these questions, we employ various prompt engineering techniques across datasets of verified PAPs in open-source Android projects. Our findings provide quantitative evidence that LLMs are a superior alternative to SATs for this specific scope. We show that top-tier models like `Gemini-2.5-pro (Gemini-2.5-pro)` achieve a precision of 90.8%, significantly outperforming the industry-standard Android Lint (63.8%). We also show that while SATs perform well only on simple syntactic patterns (e.g., *Obsolete Solution*), LLMs sustain high precision across complex categories

TABLE I: Evaluated SATs. PM/DF/G: pattern-matching, data-flow, and graph-based analyses.

SAT	Approach	#PAPs	Supported Languages
Lint [8]	PM / DF	58	Java, Kotlin, Groovy, XML
PMD [9]	PM / DF	24	Java
ADoctor [22]	PM	16	Java, Groovy, XML
EcoAndroid [10]	PM	10	Java
EcoAndroid RL [23]	PM / DF	3	Java, Kotlin
DAAP [24]	PM	22	Java
Chimera [25]	PM	8	Java
xAL [26]	PM	16	Java
DroidLens [27]	PM / G	17	Java, Kotlin

TABLE II: PAPs' Categories and respective examples

Category	Description	Example PAP
API Misuse	Incorrect/inefficient use of APIs	Recycle [29]
Build Optimization	Build-time configuration hurting performance	KaptUsage [8]
Code Smell	Known structural design flaws in code	BLOBClass [27]
Concurrency	Inefficient/incorrect handling of threads/tasks	HeavyAsyncTask [27]
Data Access	Inefficient methods for accessing data sources	AvoidFileStream [22]
Data Manipulation	Inefficient handling of data structures	AvoidArrayLoops [9]
Obsolete Solution	Use of deprecated patterns or APIs	HashmapUsage [22]
RPC/IPC	Inefficient communication patterns	DynamicWaitTime [10]
Resource Management	Improper handling of system resources	WakeLock [10]
Suboptimal Algorithm	Perform a task using a suboptimal method	SlowForLoop [10]
Unnecessary Computation	Perform redundant or avoidable work	StringToString [9]

(e.g., *Concurrency* and *Suboptimal Algorithm*) that require lifecycle and usage understanding. This result demonstrates that the context-aware reasoning enables LLMs to succeed where SATs currently fail.

Finally, to address the critical scarcity of comprehensive datasets of PAPs in the wild (identified as a primary barrier for 67% of performance anti-patterns [21]), we make our datasets publicly available. We release a systematic taxonomy of PAPs and datasets of detected and manually labeled PAP instances, providing the community with a standardized benchmark for future performance analysis research.

II. Empirical Evaluation Setup

A. Selected Static Analysis Tools

Table I lists the ten SATs that we consider in our empirical evaluation, including industry-grade tools (e.g., Android Lint [8], PMD [9]) and academic tools (e.g., EcoAndroid [10], aDoctor [22]). To select these SATs, we first identify potential candidates from the academic literature on Android performance analysis. To be included, a SAT must be publicly available and focus specifically on detecting PAPs in native Android apps or languages (i.e., Java/Kotlin). To ensure technical feasibility and the relevance of their findings, we also require an included SAT to be compatible with recent Android platform versions. To automate their execution within a unified evaluation pipeline, we extend PyAnaDroid [28] to support the selected SATs, since this standardized pipeline already supports automated analysis, building repair capabilities and enables fair comparison across tools.

B. Curated Performance Anti-Patterns

We curate **PFCat**, a dataset of 123 distinct, statically detectable Android PAPs that have been previously identified and validated in academic literature and Android [8] and SATs [9] documentation. We annotate each PAP with a description, list of SATs that detect it, plausible occurrence scenarios, potential fix patterns and/or automatic repair rules, as well as illustrative code examples, also extracted from the tools and academic literature [10], [22], [27].

Each PAP in PFCat is detectable by at least one evaluated SAT. To avoid redundancy, we filter out duplicates reported under different names by various SATs. We then classify the resulting PAPs into 11 distinct categories. The main author categorized them using a deductive thematic analysis, by deriving the categorization from the explicit technical impact described in the academic literature (e.g., mapping a PAP that causes memory leaks to Memory Management). This approach ensures that the categorization is based on established technical definitions rather than subjective interpretation. Table II lists these categories and respective examples, providing the foundation for our comparative analysis of LLMs and SATs.

C. Collected Datasets of Android Apps

In order to systematically and fairly evaluate the effectiveness of both LLMs and SATs in detecting PAPs, we need to collect realistic sources of instances of the PAPs under analysis, present in real world applications. Furthermore, we construct two datasets of Android apps from two distinct sources: PFReal from the open-source F-Droid repository [30] and PFLLM from a repository of LLM-generated Android projects [31]. To ensure that PFReal and PFLLM are representative and diverse, we perform a detailed overview of the collection process and a characterization of the Android projects of both datasets.

1) Constructing **PFReal***:* From F-Droid, we randomly select 1,200 open-source Android projects, verifying that each project exists as an app on the Google Play Store. Using PyAnaDroid [28], we select 1,063 analyzable projects from the initial set: 899 native and 164 cross-platform. PyAnaDroid could not process the discarded projects due to using unsupported cross-platform frameworks (e.g., Xamarin) or incompatible build systems (e.g., Maven) that are not commonly used to build Android apps. We retain cross-platform projects that contain Java and/or Kotlin code, because we can still partially analyze them.

To run our set of SATs on PFReal, the selected apps must be implemented in languages supported by these tools. Table I shows that most SATs detect PAPs in XML, Groovy, Java, or Kotlin. Within PFReal, $\approx$ 64% contain Java code and 52% contain Kotlin code. For 46.4% of the apps, we partially analyze them because portions of their code is written in unsupported languages (e.g., JavaScript and C). In terms of app categories, PFReal span all 17 categories in F-Droid: Internet (14.2%), System (13.6%), and Multimedia (10%).

2) Constructing **PFLLM***:* This dataset contains smaller, self-contained Android projects [31]. To construct PFLLM, we use two LLMs: `gemini-2.5-flash` and `GPT-4.5` to construct 6 projects each. The projects in PFLLM span app categories such as games (e.g., puzzle and arcade games testing logic and graphics rendering), tools (e.g., utilities such as a weather app for Application Programming Interface (API)

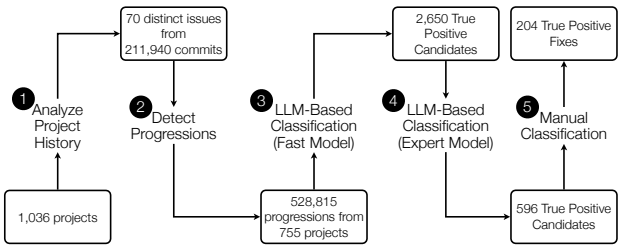

Fig. 1: Our semi-automated process (SAMU) for obtaining the dataset of validated performance fixes for PFREAL.

and location handling), productivity (e.g., a notes app for data persistence and text processing), photography (e.g., photo gallery for storage and image display), and education (e.g., scientific calculator for complex input parsing and UI design). We have executed all projects of PFLLM on a physical Android device (Xiaomi Mi5, Android 28) to verify that they were fully functional.

D. Validated Performance Fixes

To study whether LLMs can distinguish efficient from inefficient code, we have designed Static Analysis Mining Utility (SAMU) to mine version-control histories for confirmed PAP-fix pairs. SAMU combines static analysis and LLMs to retrieve performance fixes at scale through a three-stage validation procedure: (1) SAT confirmation, which identifies *progression* events where a SAT detects a PAP in one commit and its removal in the next; (2) LLM semantic diff analysis, which filters false positives (e.g., file deletions) by evaluating if the issue was genuinely fixed; and (3) manual verification to ensure the fix adheres to PAP specifications. This approach allows us to efficiently isolate rare performance fixes without relying on inconsistent commit message heuristics [32].

Figure 1 shows that SAMU begins by scanning the full history of projects in PFREAL (❶), yielding 528,815 candidate progressions (❷). Since manually processing this volume is infeasible, we employ a cost-effective LLM, `Llama-3.3-70B (Llama-3.3)` (hereafter, *Fast Model*), for initial prioritization (❸). We supply the model with PAP specifications, fix patterns, and relevant code diffs to filter non-fix scenarios (e.g., file deletions). This stage reduces the search space down to 2,650 potential fixes (9.57%). To further refine these candidates, we employ a more robust LLM, `Gemini-2.5-pro` (hereafter, *Expert Model*), to re-evaluate the filtered set (❹). Finally, we manually verify the resulting 596 candidates (❺), labeling each code difference against the PAP definitions. This process yields a final curated dataset **PFPROG** of 204 validated true positive fixes spanning 33 distinct PAPs.

E. Evaluation Metrics

We evaluate detection performance using the following metrics: Precision ($\frac{TP}{TP+FP}$), Recall ($\frac{TP}{TP+FN}$), and F1-Score ($2 \times \frac{Precision \times Recall}{Precision+Recall}$).

TABLE III: LLMs under study. CT: Context Window (Tokens).

Model	Type	CT
`Mistral`	Instruction-Tuned	58,000
`GPT-4.1-mini`	Distilled	128,000
`GPT-4o`	Instruction-Tuned Multimodal	128,000
`Gemini-2.0-flash (Gemini-2.0)`	Instruction-Tuned Multimodal	1,000,000
`Gemini-2.5-pro`	Instruction-Tuned Multimodal	1,000,000

III. CAN LLMS DETECT PAPS? (**RQ1**)

To answer **RQ1**, we evaluate whether state-of-the-art LLMs can distinguish efficient from inefficient code, without explicit rule-based guidance. To achieve that, we design a blind prompt-engineering approach inspired by the NoFunClassify experiment [20], using more recent models and our specific PAP catalog. Our hypothesis is that if models cannot distinguish performance in this controlled setting, they cannot be expected to succeed at detecting PAP in real-world scenarios. For this experiment, we construct pairs of inefficient code instances and corresponding fixes using two sources: small, focused examples from PFCAT (Section II-B), and real-world, verified fixes from PFPROG (Section II-D).

We execute our blind test prompt across PFCAT and PFPROG using seven state-of-the-art LLMs. Table III lists the evaluated models, and Table IV reports their resulting precision. The results reveal a clear performance hierarchy, with top-tier cloud-based models showing a strong ability to discern performant code. In particular, `Gemini-2.5-pro` leads with the highest overall precision on PFCAT (87.29%) and PFPROG (82.56%). The precision of `GPT-4.1-mini` also exceeds 84% on PFCAT, while that of the local model `mistral-nemo-instruct-2407 (Mistral)` is 72.03%, demonstrating its potential viability for less demanding tasks. In this setting, Recall is synonymous with Accuracy, because there are no false negatives (i.e., only wrong choices).

On the contrary, we observe that LLM capability is challenged by the complexity of real-world code in PFPROG. Across all models, the median precision drops from 81.2% to 61.72%. This trend suggests that while leading LLMs are effective, their accuracy is highly sensitive to context length and/or model size. Furthermore, local models such as `Mistral` seem to require further specialization to handle complex, real-world scenarios.

Furthermore, our observations differ from the NoFunClassify [20] experiment, where the best performant model in that experiment presents an accuracy of 36.2%, 35.3%, and 54.9%, for distinguishing snippets regarding Latency, Resource Utilization and RunTime Efficiency, respectively. In contrast, our blind-test experiment shows that LLMs can effectively reason about source code performance, which is justified by our choice of specifically targeting previously cataloged PAPs.

RQ1: Unlike prior studies [20], we observe that LLMs precisely detect PAPs on small, targeted examples (87.29%) and complex, real-world code (82.56%).

TABLE IV: LLM results in the blind test experiment (**RQ1**).

Model	PFCAT	PFPROG	
	Precision	Precision	Failures
`GPT-4o`	77.9%	62.5%	1.74%
`GPT-4.1-mini`	84.95%	57.06%	0
`Gemini-2.0`	79.2%	51.78%	0
`Gemini-2.5-pro`	**87.29%**	**82.56%**	0
Mistral	72.03%	58.43%	3.6%
Median	79.2%	62.5%	2.67%

IV. LLMs vs SATs (**RQ2**)

To compare the effectiveness of LLMs against specialized SATs in detecting PAPs on native Android apps, we rely on two manually verified datasets, PFREALVER and PFLLMVER, of PAP instances detected by SATs, that we derive by analyzing our datasets of projects using SATs. We curate PFREALVER by stratifying an initial random sample of 1,000 PAP instances from the first stage of the SAMU pipeline, limiting each unique PAP type to 50 instances for balance. We then refine this sample by excluding four PAPs that are incompatible with our snippet-based evaluation: two that are obsolete (`InternalGetterSetter` and `ViewTag`) and two (`UnusedResources` and `UnusedIds`) that manual full-project analysis makes unfeasible to perform at scale. To curate PFLLMVER, which contains 171 PAP instances, we execute our full suite of SATs across its 12 projects. Finally, we manually validate and label all identified issues.

A. Prompting Strategies

To answer **RQ2**, we employ three prompting strategies:

- *Zero-shot*: we prompt an LLM to identify PAPs in a code snippet given a list of specifications. To approximate a RAG-like setting under context limits, we use a sliding window over the specification list. We have conducted a preliminary experiment with list sizes of 25, 50, and 100 PAPs, showing that the median F1-score peaks at 30.42% with 25 PAPs, but drops to 19.39% and 17.84% for 50 and 100 PAPs, respectively, due to increased false negatives. We therefore fix the list size to 25. For each ground-truth instance, the prompt contains the correct PAP definition and 24 *distractors* (its 12 immediate neighbors), testing the model's ability to identify the correct issue under semantic noise using minimal textual descriptions.
- *Few-shot*: we prompt an LLM by including a PAP specification, along with one (1-shot) or two (2-shot) corresponding code example(s) and respective fix(es), before presenting the code snippet for analysis. This technique mimics a more elaborate scenario that provides additional context to a model to potentially stimulate accurate pattern-matching (a mechanism that several SATs employ) and stimulate In-Context Learning (ICL). We extract the examples provided in the prompt from PFCAT.
- *Few-shot-annotated*: This approach is an extended version of 2-shot prompt engineering by manually annotating the provided code drawn from PFCAT with comments to help LLMs reason about the potential PAPs and fixes, in a similar fashion to a Chain-of-Thought (CoT) [33] approach. These annotations guide the LLM's reasoning regarding the nature of the PAP and its fix. By providing the specification alongside two annotated examples, this technique aims to combine the pattern-matching benefits of few-shot learning with deeper semantic understanding, aiming to help the model to identify subtle PAPs by pinpointing the underlying root causes of the PAPs instances provided as examples.

To optimize our prompts and potentially reduce computational and financial expenditure, we conduct a preliminary experiment with five LLMs on PFREALVER. The results indicate that the inclusion of fixes yields statistically significant improvements in the F1-score. The appropriateness of the T-Test [34] ($p < 0.05$) for this comparison is validated by the failure to reject the null hypothesis of normality, as determined by the Shapiro-Wilk test [35] ($\alpha = 0.05$) when applied to the F1-score distributions. Therefore, for the few-shot prompting strategies, we provide the LLMs with code samples that exhibit performance issues alongside their respective fixes.

B. Results

Table V presents the performance of five distinct models across the PFREALVER and PFLLMVER datasets using different prompting strategies. On the real-world dataset (PFREALVER), the 0-shot strategy proves most effective, with `Gemini-2.5-pro` achieving a peak F1-score of 86.92%. In contrast, few-shot strategies excel on the synthetic dataset (PFLLMVER), where 1-shot prompting with `Gemini-2.5-pro` yields the highest F1-score (86.45%). This divergence suggests that the optimal prompting strategy is highly context-dependent. Upon closer inspection, we observe that source files in PFLLMVER are significantly smaller (in Lines of Code) than those in PFREALVER. This size difference explains why increasing prompt complexity (e.g., adding few-shot examples) benefits the smaller contexts of PFLLMVER. Conversely, on the larger files of PFREALVER, the fact that 0-shot outperforms all example-based strategies suggests that providing code examples, whether annotated or not, may inadvertently introduce noise and confuse the models rather than aid them. Furthermore, a T-Test ($\alpha = 0.05$) reveals no statistically significant difference in F1-scores between the standard 2-shot and 2-shot-annotated approaches on PFREALVER, confirming that manual code annotations do not significantly enhance effectiveness in this context.

Table VI establishes the performance baseline for the SATs that identify PAPs in datasets PFREALVER and PFLLMVER. Each row corresponds to a specific SAT, while the columns report the number of detected PAP instances, the number of unique PAP types identified, and the resulting precision for each dataset. The last column provides an aggregated average precision score. From the data, we observe that there is a significant performance gap between different types of SATs. Industry-grade tools (e.g., Lint and PMD) demonstrate high precision (94.4% and 80.1%, respectively), particularly on the dataset PFLLMVER. In stark contrast, academic prototypes

TABLE V: Comparison of LLM prompting strategies across PFREALVER and PFLLMVER in terms of Precision (P) and F1-Score (F1). The highest value in a column is highlighted in bold.

	0-shot				1-shot				2-shot				2-shot-annotated			
	PFREALVER		PFLLMVER		PFREALVER		PFLLMVER		PFREALVER		PFLLMVER		PFREALVER		PFLLMVER	
Model	**P**	**F1**	**P**	**F1**	**P**	**F1**	**P**	**F1**	**P**	**F1**	**P**	**F1**	**P**	**F1**	**P**	**F1**
`GPT-4o`	84.68%	60.9%	63.01%	47.42%	63.89%	**72.53%**	76.98%	82.31%	61.71%	**70.46%**	76.3%	85.12%	**64.68%**	**73.68%**	79.37%	80.97%
`GPT-4.1-mini`	87.05%	61.86%	70.73%	40.22%	49.1%	45.23%	78.26%	57.14%	46.55%	39.96%	81.97%	55.25%	49.07%	44.42%	90.62%	62.7%
`Gemini-2.0`	82.03%	60.2%	66.06%	76.22%	57.32%	63.64%	66.18%	68.97%	57.43%	63.33%	70.99%	73.81%	57.29%	65.96%	74.07%	82.64%
`Gemini-2.5-pro`	**90.81%**	**86.92%**	**87.1%**	**88.89%**	**64.06%**	65.73%	**88.7%**	**86.45%**	**62.77%**	64.44%	**89.91%**	**85.22%**	51.42%	54.68%	**91.09%**	82.88%
`Mistral`	86.67%	71.34%	71.43%	76.19%	50.05%	58.88%	72.29%	83.92%	50.0%	58.53%	72.29%	83.92%	53.2%	65.36%	77.3%	**87.2%**
Median	86.67%	61.86%	70.73%	76.19%	57.32%	63.64%	78.26%	82.31%	57.43%	63.33%	81.97%	83.92%	53.2%	65.36%	79.37%	82.64%

TABLE VI: Distribution of detected PAPs instances (PAPi) by SATs. The highest value in a column is highlighted in bold.

	PFREALVER			PFLLMVER			
Tool	**#PAPi**	**# PAPs**	**P**	**#PAPi**	**# PAPs**	**P**	**Avg. P**
DAAP	139	13	30.9%	16	2	87.5%	59.2%
EcoAndroid	4	2	75%	0	0	-	-
Lint	58	**24**	63.8%	**72**	**8**	**94.4%**	**79.1%**
PMD	**564**	23	**68.4%**	31	5	80.1%	74.25%
ADoctor	235	12	24.3%	45	7	42.2%	32.25%
Total Distinct	1,000	56	-	164	22	-	-

(e.g., ADoctor and DAAP) have considerably lower precision (42.2% and 87.5% on PFREALVER, plummeting to 24.3% and 30.9% on PFLLMVER). This observation suggests that academic prototypes (e.g., ADoctor and DAAP) rely on broad and ineffective detection approaches and require extensive evaluation. This is because the results differ significantly from those reported on their respective papers [22], [24]. Furthermore, industry-grade tools are more actively maintained and updated, which can also help to consistently improve the detection approaches.

Figure 2 compares LLMs against various SATs with respect to precision. The figure shows that top-tier LLMs (e.g., `Gemini-2.5-pro` and `GPT-4o`) have equal or higher precision than the best industry SATs such as Lint and PMD. The results highlight (by observing the values of several models frequently surpassing the dotted line) an overwhelming superiority of many LLMs over academic SATs on both datasets. From Table V, `Gemini-2.5-pro` achieves a precision of 90.81% on PFREALVER and 87.1% on PFLLMVER. In comparison, Table VI shows that industry-standard SATs (e.g., PMD and Lint) achieve significantly lower precision on PFREALVER (68.4% and 63.8%, respectively). These results demonstrate an overwhelming superiority of top-tier LLMs over academic SATs (e.g., ADoctor and DAAP) and a capability to outperform industry-grade tools on real-world data. While all tools seem to find PFREALVER more challenging, LLMs prove to be a highly accurate alternative.

RQ2: Not only are top-tier LLMs such as `Gemini-2.5-pro` (90.8%) and GPT-4o (84.7%) more precise than academic tools such as ADoctor (24.3%) on PFREALVER, but they also surpass industry-standard tools such as Lint (63.8%).

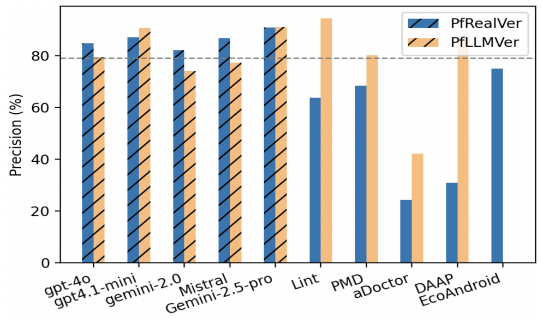

Fig. 2: LLMs' (dashed) and SATs' Precision. The dotted line is the median of best performant SAT (Lint).

C. Impact of Prompt Strategies

Figure 3 visualizes the impact of our prompt-engineering strategies on the top perform LLM precision across our proposed PAP taxonomy performed on PFCAT. The figure shows that not all prompting techniques are equally effective. The 0-shot approach has the largest and most uniform shape, demonstrating its consistent high precision across most PAP categories and achieving a peak precision of 100% on all tasks with `Gemini-2.5-pro` on Data Access. In contrast, few-shot techniques show visibly smaller and irregular shapes, signifying lower overall precision and greater performance variability across categories, except for the *Concurrency* category where it achieved 100% in all strategies except 0-shot (94.83%).

Figure 4 compares the performance profiles of LLMs against industry-standard SATs (Lint, PMD). While Lint achieves 63.8% precision, its detection is narrowly confined to syntactic patterns like Data Manipulation and Obsolete Solution. Conversely, `Gemini-2.5-pro` achieves a superior 90.81% precision with a uniform coverage shape, indicating consistent reasoning across all 11 PAP categories. Notably, the smaller model `Mistral` demonstrates specialized utility, outperforming rule-based tools in complex logic categories such as *RPC/IPC* and *Suboptimal Algorithm*.

Overall, the results presented in Figure 4 suggest that rule-based tools are limited to syntactic categories such as Data Manipulation and Obsolete Solution, top-tier LLMs maintain high precision across categories. Specifically, `Gemini-2.5-pro` consistently detects PAPs in Code Smell (98.5%) and Resource Management (91.35%) categories, where SATs such as Lint effectively register significantly lower precision.

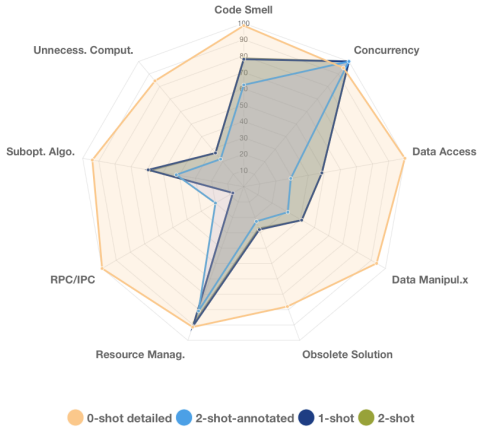

Fig. 3: Prompt strategies' accuracy across PAPs.

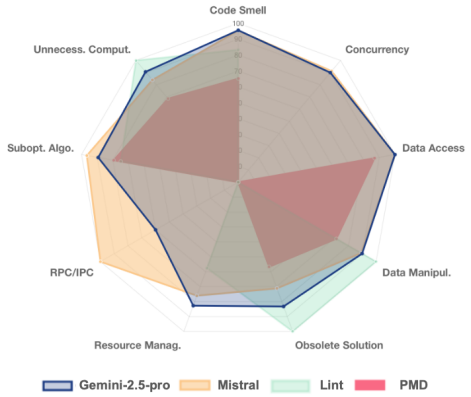

Fig. 4: LLMs vs SATs' precision across PAPs.

V. Threats to Validity and Limitations

*a) **SAMU Limitations**:* We designed SAMU as a high-throughput filter, not a high-precision classifier. Its first stage (Figure 1) deliberately employs SATs known for producing a high volume of false positives (Table VI), a trade-off made to maximize candidate extraction. Similarly, its third stage uses a cost-effective *Fast Model* chosen for cost-effectiveness.

The 9.57% precision achieved by these preliminary stages is, therefore, an expected outcome. This low rate reflects the task's inherent difficulty, where most warning resolutions are simply byproducts of unrelated code changes, a challenge amplified by the model's limited context window on complex inputs. The true success of SAMU, however, is not its precision but its scale: it successfully reduced an intractable pool of over $528,000$ raw candidates to a manageable set of a few hundred, which was the critical step that made the manual inspection and creation of PFPROG feasible.

*b) **Construct Validity**:* While our set of PAPs contain common anti-patterns identifiable by SATs, they serve as a proxy for actual performance indicators, which can only be accurately confirmed and estimated through empirical dynamic analysis. In our execution setup, the computational cost of instrumenting and executing the apps contained in our datasets, combined with the complexity of recreating valid build environments for each [28], renders dynamic analysis unfeasible. Consequently, we define our ground truth based on the SAMU procedure that manually confirmed that the PAPs existed in the codebase and were addressed according to their specification and recommended fix patterns.

We intentionally utilize default parameters to capture a realistic, out-of-the-box developer experience. However, achieving strict determinism is often unfeasible with modern Mixture-of-Experts (MoE) models [36], even at a temperature of 0 [37], due to non-deterministic expert routing and server-side load balancing. To quantify the impact of this volatility on replicability, we conducted a controlled experiment using the PFREALVER dataset, executing two independent 0-shot runs across three models (`GPT-4.1-mini`, `Gemini-2.0`, and `Gemini-2.5-pro`) with the temperature fixed at 0. The results reveal significant inconsistency: `GPT-4.1-mini` produced divergent outputs in 157 instances and `Gemini-2.5-pro` in 243 instances, whereas `Gemini-2.0` remained consistent. This demonstrates that parameter tuning alone cannot guarantee deterministic behavior across all architectures. However, we provide the full inference logs (prompts and raw responses) of our experiments to ensure the study remains auditable.

To assess potential bias in our zero-shot sliding window mechanism (which selects 24 neighboring distractors), we compared it against a randomized selection strategy using three models (`GPT-4.1-mini`, `Gemini-2.0`, `Gemini-2.5-pro`). McNemar's Test [38] confirmed a statistically significant difference between the approaches, with the randomized selection consistently yielding higher precision. This suggests that the sliding window possibly introduces a negative bias, likely because the catalog groups semantically similar PAPs (e.g., data manipulation clusters) together, making differentiation more difficult for the models than a heterogeneous random set.

Furthermore, we focused our comparison of LLMs against SATs in terms of the precision metric over a comprehensive recall analysis. While precision can be reliably calculated by manually validating the finite set of alerts generated by each tool, a true recall measurement would require identifying all false negatives over the projects sources (that is, every instance of a PAP that was missed by every detection method). Given the scale of our corpus, which spans hundreds of projects and millions of lines of code, conducting an exhaustive manual audit to establish a complete ground truth of all existing PAPs was computationally and manually intractable.

*c) **Internal Validity**:* The performance of LLMs is highly sensitive to the specific models chosen, their configurations (e.g., temperature, parameter size, context window), and prompt engineering strategies employed. To mitigate the risk of our findings being tied to a single model type and to enhance the robustness of our conclusions, we select a diverse set of contemporary models. This selection spans different cloud-based model families, varied scales, and diverse training optimizations (instruct-tuned, distilled, efficiency-focused). While this diversity aims to provide a broader view, our findings still reflect the capabilities of these specific models and evaluated prompting techniques. Similarly, the selection of

specific SATs influences the baseline prevalence data and the initial set of candidate fixes identified for PFLABEL. In terms of SATs selection, we have initially considered other tools that can detect Android performance issues such as Paprika [39] and Leafactor [29]. However, we do not use them in our experiments because they were subsumed by other tools. In particular, DroidLens is a superset of Paprika and Leafactor detects the same issues that tools such as Lint already cover.

*d) **External Validity**:* Our empirical analysis relies on a large corpus of open-source projects primarily sourced from F-Droid. While diverse, this sample may not fully represent the characteristics of proprietary applications, which constitutes a potential threat to external validity. Furthermore, our findings are scoped by the capabilities of the selected SATs, which in turn dictates our focus on native Android development. The chosen SATs offer robust support for Java and Kotlin but lack mature coverage for modern cross-platform frameworks such as Flutter or React Native, where performance-oriented SATs are still emerging. Consequently, our analysis does not cover PAPs specific to these technologies, and our findings regarding issue prevalence and LLM effectiveness may not generalize to applications built using these frameworks.

VI. Related Work

The negative impact of performance issues on UX is well-established, leading to extensive research on their identification [40] and mitigation [22], [29]. Foundational work in this area has focused on creating catalogs of PAPs and inefficient APIs to guide developers and tool builders [7], [40]. For example, [7] empirically estimated the energy consumption of method calls across 55 Android apps. Their work produces a catalog of 131 energy-greedy API methods and offers guidelines for developers, such as reconsidering design principles (e.g., Information Hiding) that may inadvertently increase energy usage. Building on these catalogs, other studies have leveraged SATs to quantify the prevalence and evolution of PAPs in large codebases [21], [32].

[32] analyze the version history 724 open-source Android projects, finding that nine common Java-specific PAPs are present in 43% of the projects, with 45% of these instances remaining unfixed over time. Prior work has also explored human factors behind such issues. By analyzing 255k commits from 324 Android apps, [41] conclude that the introduction of code smells is spread across developers, regardless of their experience. Our work primarily focuses on statically-detectable PAPs, building upon this body of knowledge.

SATs examine code without executing it to identify potential issues using techniques such as rule-based systems, Abstract Syntax Tree (AST) pattern-matching, and data-flow analysis [23]. Commonly adopted tools include Integrated Development Environment (IDE) plugins (e.g., Android Lint [8] and EcoAndroid [10]), academic prototypes for performance diagnosis (e.g., Paprika [39]), and tools that detect and automatically refactor identified anti-patterns [29]. The usage of SATs such as Android Lint [8] is widespread for detecting known PAPs at development time. However, a recent study [21] concludes that 57% of already cataloged performance anti-patterns have not been addressed in the literature. While 76% of the issues are not yet detectable and/or repairable by existing tools, 67% lack datasets that allow them to be effectively understood, detected, and corrected. Our study is the first, to the best of our knowledge, to conduct a large-scale empirical evaluation of LLMs for the static detection of PAPs on Android. Furthermore, we contribute with novel datasets of statically-detected PAPs over hundreds of real-world projects.

Despite known LLMs' limitations, prior work have demonstrated that their accuracy may be enhanced, even at inference time, through techniques such as context augmentation and prompt engineering techniques [33]. Recent studies have also shown promising results on using LLMs to complement traditional static analysis approaches, namely for security vulnerabilities [16], [17] or other program analysis domains [17], [42]. For example, LLift [42] leveraged LLMs for path analysis and finding specific bugs such as Use-After-Free. IRIS [17] employed LLMs to infer taint specifications.

Beyond vulnerability detection, LLMs have been also effectively used for extracting static features from APK files and automatically adjudicate static analysis alerts [16], [17] to reduce developer overhead. NoFunEval [20] evaluate LLMs for several non-functional requirements by leveraging prompt-engineering approaches. Such work highlight that current code LLMs may falter on the evaluated requirements, including performance. Our work extends this line of research by specifically investigating the application of state-of-the-art LLMs to the static detection of PAPs.

VII. Conclusion

This paper investigates the potential of LLMs for statically detecting PAPs in native Android apps, comparing them against traditional SATs. Our empirical evaluation demonstrates that LLMs can effectively distinguish between performant and non-performant code (**RQ1**) and can detect specific PAPs categories with high accuracy (**RQ2**).

With appropriate prompting, not only do LLMs compete with specialized SATs in real-world scenarios, but they often surpass them. The 0-shot prompting strategy is the most effective, with `Gemini-2.5-pro` achieving a precision of 90.8% and an F1-score of 86.92% on a dataset of real-world PAPs. This performance significantly exceeds that of academic SATs such as aDoctor (24.3% precision) and even surpasses the industry-standard Android Lint (63.8% precision). While rule-based tools such as Lint excel in narrow, well-defined categories, top-tier LLMs demonstrate a more generalized and context-aware reasoning ability, providing consistently high performance across a broader spectrum of PAPs. These results strongly support the conclusion that LLMs are already a viable and often superior alternative for the static detection of performance anti-patterns in Android applications.

References

[1] MoldStud, "The importance of app performance monitoring and optimization in android development," https://bit.ly/439AZtd, February 2023, accessed: 2025-04-04.

[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[3] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017.

[4] C. Escobar-Velásquez, A. Mazuera-Rozo, C. Bedoya, M. Osorio-Riaño, M. Linares-Vásquez, and G. Bavota, "Studying eventual connectivity issues in android apps," *Empirical Software Engineering*, vol. 27, 2022.

[5] X. Li, J. Chen, Y. Liu, K. Wu, and J. Gallagher, "Combatting energy issues for mobile applications," *ACM Transactions on Software Engineering and Methodology*, 04 2022.

[6] R. Rua and J. Saraiva, "A large-scale empirical study on mobile performance: energy, run-time and memory," *Empirical Software Engineering*, p. 67, 12 2023.

[7] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: An empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. ACM, 2014, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597085

[8] Google. (2021) Android lint checks. Last visit: 2025-02-05. [Online]. Available: https://googlesamples.github.io/android-custom-lint-rules/

[9] The PMD Project, "PMD: An extensible cross-language static code analyzer," https://github.com/pmd/pmd, accessed: 2025-15.

[10] A. Ribeiro, J. F. Ferreira, and A. Mendes, "Ecoandroid: An android studio plugin for developing energy-efficient java mobile applications," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 62–69.

[11] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at facebook," *Commun. ACM*, vol. 62, no. 8, p. 62–70, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3338112

[12] J. He, R. MacQueen, N. Bombardieri, K. Ali, J. R. Wright, and C. Cifuentes, "Finding an optimal set of static analyzers to detect software vulnerabilities," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023, pp. 463–473.

[13] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[14] Y. Wang, W. Wang, S. Joty, and S. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *ACM Transactions on Information Systems*, 01 2021.

[15] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 6004–6016, focuses specifically on code transformation, specifically language translation, using unsupervised methods.

[16] C. Wen, Y. Cai, B. Zhang, J. Su, Z. Xu, D. Liu, S. Qin, Z. Ming, and T. Cong, "Automatically inspecting thousands of static bug warnings with large language model: How far are we?" *ACM Trans. Knowl. Discov. Data*, vol. 18, no. 7, Jun. 2024.

[17] Z. Li, S. Dutta, and M. Naik, "IRIS: LLM-assisted static analysis for detecting security vulnerabilities," in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=9LdJDU7E91

[18] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Trans. Inf. Syst.*, vol. 43, no. 2, Jan. 2025. [Online]. Available: https://doi.org/10.1145/3703155

[19] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024. [Online]. Available: https://aclanthology.org/2024.tacl-1.9/

[20] M. Singhal, T. Aggarwal, A. Awasthi, N. Natarajan, and A. Kanade, "Nofuneval: Funny how code LMs falter on requirements beyond functional correctness," in *First Conference on Language Modeling*, 2024.

[21] D. Liao, S. Pan, S. Yang, Y. Zhao, Z. Xing, and X. Sun, "Automatically analyzing performance issues in android apps: How far are we?" 2024. [Online]. Available: https://arxiv.org/abs/2407.05090

[22] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of android-specific code smells: The adoctor project," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 487–491.

[23] R. B. Pereira, J. F. Ferreira, A. Mendes, and R. Abreu, "Extending ecoandroid with automated detection of resource leaks," in *2022 IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, 2022, pp. 17–27.

[24] G. Rasool and A. Ali, "Recovering android bad smells from android applications," *Arabian Journal for Science and Engineering*, vol. 45, 2020.

[25] M. Couto, J. Saraiva, and J. P. Fernandes, "Energy refactorings for android in the large and in the wild," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 217–228.

[26] I. Fatima, H. Anwar, D. Pfahl, and U. Qamar, "Detection and correction of android-specific code smells and energy bugs: An android lint extension," in *QuASoQ@APSEC*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:228631374

[27] C. Mao, H. Wang, G. Han, and X. Zhang, "Droidlens: Robust and fine-grained detection for android code smells," in *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2020, pp. 161–168.

[28] R. Rua and J. Saraiva, "Pyanadroid: A fully-customizable execution pipeline for benchmarking android applications," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023, pp. 586–591.

[29] L. Cruz, R. Abreu, and J. Rouvignac, "Leafactor: Improving energy efficiency of android apps via automatic refactoring," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 05 2017, pp. 205–206.

[30] F-Droid - Free and Open Source Android App Repository. Accessed: April 20, 2026. [Online]. Available: https://f-droid.org/

[31] Anonymous, "A dataset of vibe-coded android apps," https://anonymous.4open.science/r/vibe_coded_android_apps-7D3E/, accessed: 2025-10.

[32] T. Das, M. D. Penta, and I. Malavolta, "Characterizing the evolution of statically-detectable performance issues of android apps," *Empir. Softw. Eng.*, vol. 25, no. 4, pp. 2748–2808, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09798-3

[33] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," *Proceedings of the 36th International Conference on Neural Information Processing Systems*, 2022.

[34] B. L. Welch, "The generalization of 'student's' problem when several different population variances are involved," *Biometrika*, vol. 34, no. 1-2, pp. 28–35, 01 1947. [Online]. Available: https://doi.org/10.1093/biomet/34.1-2.28

[35] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)†," *Biometrika*, vol. 52, no. 3-4, pp. 591–611, 12 1965. [Online]. Available: https://doi.org/10.1093/biomet/52.3-4.591

[36] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: scaling to trillion parameter models with simple and efficient sparsity," *J. Mach. Learn. Res.*, vol. 23, no. 1, Jan. 2022.

[37] S. Ouyang, J. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of chatgpt in code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, pp. 1 – 28, 2023.

[38] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, 1947.

[39] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the software quality of android applications along their evolution (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 236–247.

[40] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications," *Empirical Softw. Engg.*, vol. 24, no. 4, p. 2209–2235, Aug. 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09682-0

[41] S. Habchi, N. Moha, and R. Rouvoy, "The rise of android code smells: Who is to blame?" in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 445–456.

[42] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3649828