Magnifying Inefficiency: How LLMs Amplify Performance Anti-Patterns in Mobile Development

Rui Rua
New York University
Abu Dhabi, UAE
rui.rua@nyu.edu

Karim Ali
New York University
Abu Dhabi, UAE
karim.ali@nyu.edu

Abstract

Application performance critically impacts user satisfaction and retention, yet developing efficient applications remains challenging. The emergence of Large Language Models (LLMs) as coding assistants presents a new paradigm in software development. However, the efficiency of the code they generate has not been extensively scrutinized.

This paper aims to evaluate LLM-generated code efficiency by analyzing the presence of Performance Anti-Patterns (PAPs) (i.e., common coding practices with suboptimal performance) on generated Android source code. To draw conclusions regarding LLM-generated code efficiency, we analyze and compare datasets of both real-world and LLM-generated Android projects using state-of-the-art static analysis tools that detect the presence of PAPs.

Our findings reveal a significant prevalence of performance anti-patterns in LLM-generated code. We observe that functionally equivalent human-written code contains only 0.32–0.84 PAPs per thousand lines of code (KLOC) and general open-source baselines average 4.7, while LLM-generated projects reach a median density of 8.36–11.5. Moreover, 15% of all generated files contain at least one PAP instance, driven by the models' repeated failure to correctly manage resource lifecycles or select efficient algorithms.

Suppl. Material:https://github.com/sanadlab/llm_code_analysis_repl_package

1 Introduction

As Android applications become increasingly critical for service-intensive computation in daily life, the need to optimize memory and energy utilization has never been greater. Mobile devices remain significantly more resource-constrained than computers, yet certain common inefficient programming practices [3] continue to contribute to the suboptimal use of limited resources such as battery power and memory capacity.

Diagnosing and addressing performance issues in Android applications is inherently complex [40] due to extreme device fragmentation and variable runtime contexts. Prior research has identified that performance optimization opportunities exist at multiple levels, such as programming languages/frameworks choice [37, 41], specific libraries usage [48] or coding patterns[10]. These optimization opportunities are important because common performance anti-patterns in Android are widespread [13] and tend to spread across developers regardless of their experience [21].

Meanwhile, LLMs have recently demonstrated remarkable capabilities in comprehending [15], generating [60], and transforming source code [28]. This proficiency has catalyzed an emerging development paradigm colloquially termed *vibe coding*, where developers rely on high-level natural language prompts and iterative intuition to generate apps rather than manual design, implementation and verification[17]. While this approach significantly lowers barriers and accelerates prototyping, it introduces inherent risks to software quality [61]. Consequently, code has been increasingly generated by individuals who lack fundamental expertise required to develop or secure the underlying logic, resulting in inefficient, brittle, unstructured code [61].

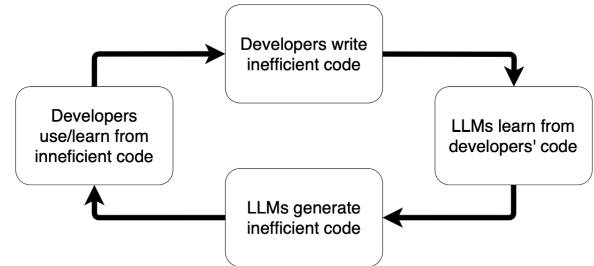


Figure 1: Performance Loop of Hell

On the other hand, LLMs exhibit recognized limitations such as hallucinations [24] and inherent difficulties in reasoning about performance and resource utilization [8, 53]. A critical concern is whether this reliance on AI-generated code contributes to a systemic degradation of app efficiency. Since LLMs learn from vast corpora of public source code where performance anti-patterns are endemic [13, 23], by training on such corpus, models may inadvertently internalize these inefficient patterns as valid solutions and reproduce them when generating code. Such constant reproduction of inefficient code potentially creates a self-reinforcing feedback cycle, which we coin *Performance Loop of Hell* (as illustrated in Figure 1). In this cycle, sub-optimal patterns are continuously produced and amplified through successive generations of both LLM training data and applications. In vibe-coding settings, this cycle could be even more critical. Inefficient LLM-generated code, if accepted and integrated into production systems without optimization, becomes part of the corpus training future LLMs, further skewing the distribution toward inefficiency.

This paper aims to address the current lack of scrutiny regarding AI-generated code efficiency [53]. To systematically evaluate code efficiency, we analyze statically detectable PAPs, recurring code patterns known to cause suboptimal performance regarding latency, memory usage, bandwidth, or energy consumption [16]. PAPs represent detectable manifestations of performance inefficiency that remain functionally correct but carry performance implications. These patterns are particularly relevant to Android, because many are statically detectable through rule-based and data-flow analysis approaches. Their detection is also usually integrated

```java
public class CustomView extends View {
    // Optimization: Pre-allocate objects used in drawing upfront
    // private Paint backgroundPaint = new Paint();

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);

        // ANTI-PATTERN: Allocating a new object inside a high-frequency
        // method call like onDraw(). replacing the following call with a
        // pre-allocated object will fix this issue
        Paint backgroundPaint = new Paint();
        backgroundPaint.setColor(Color.BLACK);
        canvas.drawRect(0, 0, getWidth(), getHeight(), backgroundPaint);
    }
}
```

Figure 2: An example of the *DrawAllocation* anti-pattern.

into default development environments (e.g., Android Lint [19], which is part of Android Studio [20]). Figure 2 shows a known example of such patterns: `DrawAllocation` [18], which allocates memory inside frequently-invoked UI rendering methods (e.g., `onDraw()`). In these methods, repeated allocations trigger garbage collection pauses that interrupt animations and produce perceptible UI lag.

To draw robust conclusions regarding performance, we benchmark 12 LLM-generated applications against two human baselines: a matched control group of 12 apps and a larger historical dataset of 307 projects. We leverage specialized Static Analysis Tools (SATs) to scan this corpus, detecting unique PAPs spanning 11 categories such as Resource Management and Suboptimal Algorithms. This comparison guides our investigation into the following research question:

- **RQ1**: How does LLM-generated code compare to human-developed code in terms of performance?

Our results show that LLM-generated Android code exhibits statistically significant performance inefficiencies compared to human-written equivalents. In particular, LLMs generate code with at least approximately 10× more anti-patterns per KLines of Code (LoC), a pervasive issue affecting 75% of source code files. This inefficiency is not model-specific but reflects fundamental architectural limitations, with particularly severe degradation in several PAP categories such as resource lifecycle management (at least 7.13× higher). These results provide show evidence to support the *Performance Loop of Hell* hypothesis.

2 Related Work

2.1 Performance Anti-Patterns (PAPs)

The impact of performance issues on User eXperience (UX) is well-documented, directly influencing user retention and app ratings. Early foundational research establish catalogs of PAPs and inefficient Application Programming Interfaces (APIs) to provide structured guidance for developers [11, 30, 45]. For instance, Linares-Vásquez et al. [30] empirically measure energy consumption across 55 Android apps, identifying 131 *energy-greedy* API methods and highlighting how design principles (e.g., Information Hiding) may inadvertently increase resource consumption.

As the mobile ecosystem has evolved, the taxonomy of PAPs has expanded to cover resource utilization, data usage, and complex memory management. To address these type of issues, researchers have employed static analysis to quantify the prevalence of PAPs in large-scale codebases [13, 29, 35], since dynamic testing is time-consuming, hard to conduct in scale and frequently yields environment specific results. Furthermore, several studies in the literature focus on detecting, defining and analyzing PAPs using static analysis aproaches. Malavolta et al. [13] found that Java-specific PAPs were present in 43% of 724 open-source projects, with nearly half remaining unfixed over time. While Hachia et al. [21] observed that performance anti-patterns are introduced by developers regardless of experience level, SATs often struggle with runtime-dependent root causes.

Building on this foundation, our work utilizes established taxonomies and static analysis techniques not merely to profile human-written software, but to systematically and quantitatively evaluate the performance code produced in *vibe-coding* environments.

2.2 LLMs and Code Efficiency

With the integration of LLMs into the development workflow, a growing body of research evaluates whether AI-generated code introduces new forms of technical debt [31]. This hypothesis has led to the *generative AI paradox*: while LLMs can generate syntactically correct code, they often lack the reasoning to generate efficient and secure code [53]. Abbassi et al. [4] provide a taxonomy of five code inefficiencies (Performance, Readability, Maintainability, General Logic, and Errors), finding that 33.54% of LLM-generated samples exhibited multiple flaws such as performance issues.

Specific implementation quality has also come under scrutiny. Siddiq et al. [52] show that code generated through ChatGPT frequently contains security issues , unused variables and unclosed streams. Additionally, benchmarks such as NoFunEval [53] were developed to evaluate the knowledge of LLMs regarding non-functional requirements such as latency and resource utilization. Their results indicate that across 27 different LLMs, models consistently prioritize functional correctness over performance optimization.

Efforts to optimize LLM output for efficiency have yielded varying results. Simple prompt optimization for Python energy consumption has proven inconsistent [8], but more sophisticated feedback-driven frameworks such as SpeedGen [43] show promise for runtime optimization. By integrating LLMs into an iterative pipeline featuring automated profiling and refinement, it proved that is possible to leverage LLMs to significantly reduce execution time while maintaining functional integrity. This evidence that while LLMs may initially generate suboptimal code, they can be leveraged as powerful tools for performance refactoring when provided with sufficient contextual or dynamic feedback.

Our study extends this line of research by providing a direct empirical comparison between LLM-generated Android applications and human-developed open-source projects in terms of performance.

233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290

Table 1: Categories and respective examples of anti-patterns

Category	Description	Examples
API Misuse	Incorrect/inefficient use of APIs.	Recycle [12]
Build Optimization	Config. hurting build/startup time or app size.	KaptUsage [19]
Code Smell	Known structural design flaws in code	BLOBClass [33]
Concurrency	Inefficient/incorrect handling of threads/tasks.	HeavyAsyncTask [33]
Data Access	Inefficient methods for accessing data sources.	InefficientSQLQuery [39]
Data Manipulation	Inefficient handling of data structures.	AvoidArrayLoops [58]
Obsolete Solution	Use of deprecated patterns or APIs.	HashmapUsage [39]
RPC/IPC	Inefficient communication patterns.	DynamicWaitTime [46]
Resource Management	Improper handling of system resources.	WakeLock [46]
Suboptimal Algorithm	Perform a task using a suboptimal method.	SlowForLoop [46]
Unnecess. Computation	Perform redundant or avoidable work.	MemoizationChance [9]

3 Empirical Evaluation Setup

3.1 Performance Anti-Patterns (PAPs)

To gather an extensive dataset of statically detectable Android PAPs that we could detect on Android projects, we resort to a comprehensive online repository [47] synthesized from academic literature, official Android documentation, and industry-standard SAT rule sets. For each PAP, the repository provides a technical description, compatible SATs, occurrence scenarios, fix patterns, invalidation/void patterns (conditions that can make the PAP not applicable or valid) and illustrative code examples. Each included PAP is detectable by at least one state-of-the-art SAT. Furthermore, the repository provides a taxonomy that classifies these patterns into 11 distinct categories, listed in Table 1. This taxonomy was derived from the explicit technical impact described in the academic literature and SATs documentation (e.g., mapping a PAP that uses a deprecated method to the Obsolute Solution category). This approach ensures that the taxonomy is based on established technical definitions rather than subjective interpretation.

To detect these patterns within our project corpus, we leveraged 13 state-of-the-art SATs identified in the same repository. As shown in Table 2, these include a mix of industry-grade tools, such as Android Lint [19], SpotBugs [54] or PMD [58], and specialized academic tools, such as EcoAndroid [46] and aDoctor [39]. To automate the SATs execution within a unified evaluation pipeline, we reuse PyAnaDroid [49] since it already supports the selected SATs. PyAnaDroid consists of an automated analysis pipeline for Android, combining state-of-the-art static and dynamic analysis tools and build repair capabilities to automatically analyze Android projects.

3.2 Human-Developed Projects (DOpenProjs)

Our process for collecting human-developed open-source projects began by leveraging the F-Droid [1] repository. Using its API, we initially extract 2,000 projects. Then, with the help of PyAnaDroid [49], we select 1,000 projects identified as native Android Projects. This selection criterion ensures that we can fully analyze the source code of the projects by considering only native instances and excluding unsupported cross-platform frameworks (e.g., Kivy [27], Xamarin [36]) and incompatible build systems (e.g., Maven [57], Ant [56]) with the PyAnaDroid pipeline.

Table 2: SATs used in our analysis. SC stands for source code, while BC stands for bytecode

Tool	Input	PAPs	Languages
Lint [19]	SC	58	Java, Kotlin, Groovy, XML
PMD [58]	SC	24	Java
ADoctor [39]	SC	16	Java, Groovy, XML
EcoAndroid [46]	SC	10	Java
EcoAndroid RL [42]	BC	3	Java, Kotlin
DAAP [44]	SC	22	Java
Chimera [9]	BC	8	Java
xAL [14]	BC	16	Java
Infer [7]	BC	7	Java
Spotbugs [54]	BC	26	Java
Spotbugsfbcontrib [6]	BC	39	Java
Detekt [5]	SC	7	Kotlin
DroidLens [33]	BC	17	Java, Kotlin

In order to assure that the projects were solely developed by humans and prevent data contamination from LLM-assisted coding, we inspect the release history and exclude any projects updated after November 2022, the release date of ChatGPT [38]. This process yielded a final baseline of 307 native projects (DOpenProjs).

This final set of projects originate apps from 17 F-Droid categories (from a total of 55), with the most represented categories being System, Multimedia, Internet and Games, with 16.6%, 10.6%, 10.1% and 9.5% representation, respectively. The least represented categories are Graphics, Phone & SMS, and Money, with representation between 0.8% and 2.2%.

3.3 LLM-Generated Projects (DVibeProjs)

To simulate pure *vibe coding* workflows [25], where developers rely primarily on natural language prompts to generate functionality, we prompt `Gemini-2.5-pro (Gemini)` and `GPT-4.5 (GPT)` with their respective default parameters to generate the complete fully-functional source code 12 apps representing different app categories: *Games* (2048, Flappy Bird), *Tools/Weather*, *Task/Productivity* (To-Do), *Gallery* (Gallery), and *Education* (Scientific Calculator). We select these categories to test various technical requirements, including API integration, data persistence, and UI complexity. Through this process, we generated 12 distinct projects (DVibeProjs) and functional apps whose prompts used and generated code we made openly-available [50]. Both models produced fully-functional apps in a single shot, apart from minor platform-specific building issues that were easily manually addressed without modifying the generated source code. We further inspected all the generated projects in order to evaluate if the generated app was fully functional.

3.4 Similar Projects (DSimProjs)

To address the fact that the type and category of an application can influence the kind and number of PAPs found in the source code, we also decided to isolate a dataset of 12 open-source projects similar to those generated with LLMs.

To facilitate a *head-to-head* comparison between LLM and human developers, we identify a subset of human-developed and maintained apps from F-Droid that most closely match the functionality of our 12 LLM-generated projects. We perform a similarity search

291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348

by transforming the natural language prompts used for the LLMs and the textual descriptions of the F-Droid apps into vector embeddings using a pre-trained embedding model (`gemini-embedding-001`). We then calculate the *cosine similarity* between each LLM prompt and the F-Droid metadata. For each project, we then selectp the top 2 projects that matched at least one of the vibe code project (F-Droid) categories. This process resulted in a smaller dataset of 12 apps (DSimProjs), providing a controlled environment to more fairly reason about performance by comparing apps that solve identical problems.

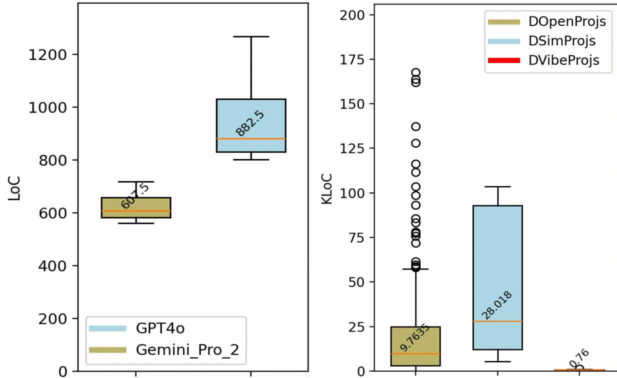


Figure 3: Comparison in terms of LoC for different models and datasets, respectively

Figure 3 presents a box-plot that allows to compare the two used models in terms of variation in verbosity (LoC) between the models. Gemini-generated projects exhibit a consistently higher volume of code, with a median size of approximately 882.5 LoC for `Gemini` and 607.5 LoC for `GPT`. In the rightmost box plot of Figure 3, it is possible to see that there are significant differences in terms of LoC between datasets. DSimProjs exhibits higher median and top quartile than the pre-LLM projects in DOpenProjs and the DVibeProjs projects. It is also possible to conclude that the LLM generated projects of DVibeProjs are significantly less verbose than the remaining projects of the other datasets, with a median size of 760 LoC, against 9.76 KLoC of DOpenProjs and 28.02 KLoC of DSimProjs. The magnitude of the difference between medians between datasets emphasis the need of using density-based metrics in order to draw fair comparisons.

3.5 Evaluation Metrics

To fairly compare the efficiency of human-written and LLM-generated code in terms of occurrence of PAPs, we define a set of metrics based on their frequency and distribution. The following density metrics account for fundamental scale differences across datasets, providing fairer cross-project comparisons than raw counts. Let S_i denote the total number of PAPs detected in project i, and let KLOC_i be its size in thousands of lines of code.

Anti-Pattern Density (APD): We first define the overall PAP density of a project i as $\text{APD}_i = S_i/\text{KLOC}_i$. This metric normalizes the number of detected PAPs by project size, enabling a fair comparison between projects of different scales and between human-written and LLM-generated code.

Table 3: Characterization of the projects' datasets in terms of detected PAP instances and density metrics for both Holistic (H) and Annotated (L) evaluations.

Dataset	Occurrences		Instances		Categs.		APD		SFP	
	H	A	H	A	H	A	H	A	H	A
DOpenProjs	20876	-	70	-	11	-	4.7	-	0.88	-
DSimProjs	877	607	49	49	10	10	0.32	0.84	0.97	0.91
DVibeProjs	244	171	24	24	8	8	11.5	8.36	0.85	0.85

Anti-Pattern Categories Density (CAPD): Given a set of PAP categories C (e.g., API Misuse, Resource Management, Code Smells), let $S_{i,c}$ be the number of PAPs of category $c \in C$ in project i. The category-level density is $\text{CAPD}_{i,c} = S_{i,c}/\text{KLOC}_i$. These densities allow us to analyze whether specific categories of PAPs are more prevalent in LLM-generated or human-written projects.

Static PAPs-Free Proportion (SFP): To capture code *cleanliness*, we compute the proportion of entities that are free of PAPs. Let E_i be the set of code entities in project i (e.g., classes, methods, or files) and let $E_i^{\text{clean}} \subseteq E_i$ be the subset with zero detected PAPs. The smell-free proportion is defined as $\text{SFP}_i = |E_i^{\text{clean}}|/|E_i|$. This metric reflects how much of a project is entirely free of the considered anti-patterns.

In our execution setup, we evaluate LoC and code complexity using SCC [2]. We determine the number of issues and respective categories by executing the SATs presented in Table 2 over the 3 projects datasets previously presented in this section.

4 Analyzing Code Performance

To answer **RQ1**, we employ 13 SATs to analyze our three datasets: DOpenProjs, DVibeProjs, and DSimProjs. We structure the comparison into two phases:

- Holistic Analysis: We compare the raw volume of flagged PAPs in LLM-generated apps against the open-source human baselines.
- Manual Validated Analysis: Since SATs can produce false positives that distort debt estimates [31, 34], we manually validate the detected instances of PAPs reported by SATs in DVibeProjs and DSimProjs. This allows us to determine if the performance gaps persist when analyzing only confirmed inefficiencies.

4.1 Holistic Evaluation

The most straightforward way to compare human- and LLM-generated code in terms of PAPs is by examining the issues detected across all SATs executed on the source code from the three projects datasets. However, this approach has some limitations. First, the different SATs employed are complex to run at scale (given the number and complexity of projects) [22, 48]. Some require a full project build, which often requires significant configuration effort at the project level [26].

Consequently, when executing the full suite of SATs on the DOpenProjs dataset, several tools achieved relatively low execution success rates. For example, even with the ability to repair some typical Android project build errors through the PyAnaDroid pipeline, the success rate of Android Lint (the SAT that detects the widest

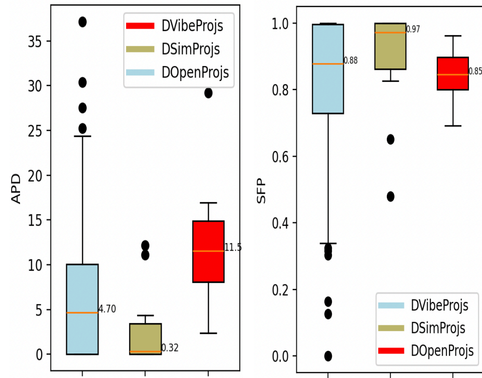


Figure 4: APD values across datasets

Figure 5: SFP values across datasets

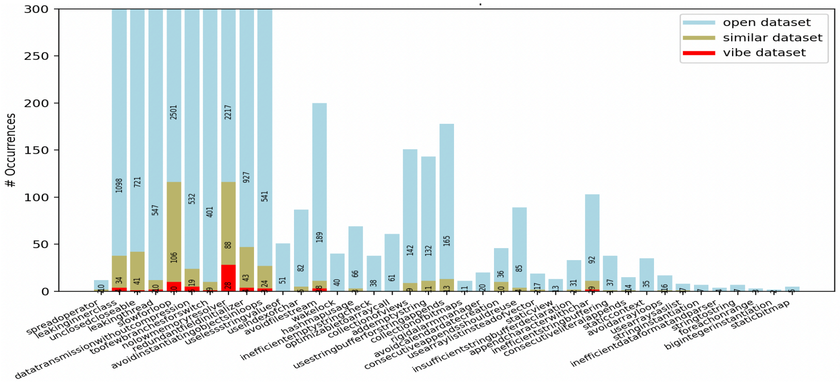


Figure 6: Occurrences of PAPs across datasets (*lightweight* SATs only)

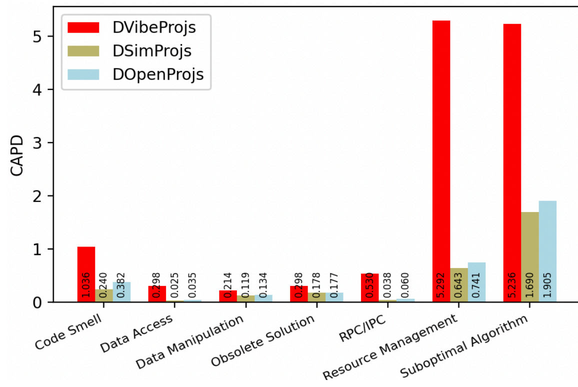


Figure 7: CAPD values across categories for all datasets

range of PAPs) was only 35.5%. To ensure a fair comparison, we therefore restricted our analysis to the issues detected by SATs with execution success rates above 90% in all datasets. The tools that meet this threshold, which we named *lightweight* SATs are aDoctor[39], PMD[58], Detekt[5], and DAAP[44], since these do not require full project compilation and building.

Because multiple SATs can flag the same PAP, the aggregated results contained significant duplication. Additionally, tools like DAAP [44], aDoctor [39], and EcoAndroid [46] often lack precision, reporting issues only at the file or project level. We removed duplicates by grouping instances by PAP, repository, and filename. For each group, we preserved the entry with the highest granularity, prioritizing specific line numbers over file-level flags, or selecting the longest location string when line numbers were unavailable.

Figure 6 shows the unique instances of PAPs identified by the SATs with execution success rates above 90% in each dataset (*lightweight* SATs). It is possible to observe that some PAPs commonly detected in the wild (DOpenProjs) are also frequently observed in the other datasets (e.g., *NoLowMemoryResolver*, *SlowForLoop*), suggesting that LLMs successfully replicate the inefficiencies present in their training data.

Table 3 presents a comprehensive characterization of the datasets based on detected PAP instances and density metrics. The disparity between human and AI-generated code is evident in the Anti-Pattern Density (APD). While the human-developed baseline (DOpenProjs) and the functionally similar projects (DSimProjs) maintain low APD of 4.7 and 0.32 respectively in the holistic evaluation, the LLM-generated code (DVibeProjs) exhibits a significantly higher APD of 11.5. This suggest that for every thousand lines of code, LLMs introduce considerably more performance inefficiencies than human developers.

To provide statistical support for the conclusions regarding the magnitude of the differences observed between the datasets, we conduct a series of statistical tests to assess the significance of the results for the APD and SFP metrics, whose distribution can be seen in Figure 9. Regarding the APD metric, the Shapiro-Wilk [51] test rejects the null hypothesis of normality for the DVibeProjs and DOpenProjs datasets. Consequently, we apply the non-parametric Mann-Whitney U test [32] ($\alpha = 0.05$) to compare the distribution of the DVibeProjs dataset against the other two datasets. The obtained values ($p = 0.0011$ and $p = 0.0019$, respectively) allow for the rejection of the null hypothesis, indicating a statistically significant difference between the samples. Furthermore, the Vargha-Delaney [59] effect size measure determined large effect sizes ($A_{12} = 0.90$ and $A_{12} = 0.76$, respectively), reinforcing the substantial magnitude of the difference between the datasets.

Regarding the SFP metric, the Shapiro-Wilk test again indicates non-normality for the datasets. Therefore, we utilize the Mann-Whitney U test ($\alpha = 0.05$) to compare the distributions of the DVibeProjs dataset against the other two. However, in this instance, the test failed to reject the null hypothesis ($p = 0.0525$ for the comparison against DSimProjs and $p = 0.52$ against DOpenProjs), leading to the conclusion that there are no statistically significant differences between the distributions of these pairs of samples.

4.2 Manually Annotated Evaluation

This section presents a more fine-grained and systematic comparative analysis between human- and LLM-developed code regarding statically detectable PAPs, aiming to address the main limitations

of the holistic experiment. Given the low execution success rate of most SATs in our pipeline and the high proportion of false positives know to be inherent to SATs [34], we conduct a manually analysis that considers all SATs and manually verifies the detected PAPs instances.

Due to the magnitude of the datasets of detected PAPs across the three projects datasets and the size of their respective codebases, it is impractical to verify al of them with a manual approach. This would require extensive manual verification and validation. Furthermore, this analysis focuses exclusively on the DSimProjs and DVibeProjs datasets and a selected subset of detected PAP instances. Because the magnitude of these datasets is considerably reduced (when compared to the larger dimension of the DOpenProjs dataset), the full set of 13 tools could be manually executed on each project of the two datasets.

In order to select the instances of PAPs to analyze, we apply a multi-step filtering process. First, the ViewTag, InternalGetterSetter PAPs were excluded since they are longer relevant in newer platform versions. Afterwards, since several SATs fail to accurately report the precise location of a PAP, we considered only one instance per source file. Subsequently, we removed duplicate issues, i.e., the same instances detected by different SATs (Following the same approach described in Section 4.1). Finally, for high-frequency PAPs, we manually verified a random sample of up to 50 instances per PAP and estimated the total true positives as $N_{\text{true}} = N_{\text{det}} \times r_{\text{tp}}$, where N_{det} is the total detections and r_{tp} is the true positive rate derived from the sample.

To further evaluate whether the PAPs instances detected were indeed true positives and not simply false positives or more context-sensitive cases identified by these tools, a manual validation of these issues was performed. The most experienced author in Android development meticulously analyzes the code the reported PAPs instances on all of the DVibeProjs and DSimProjs apps to classify the occurrences of these issues as true (true positive) or not (false positive). This manual verification aims to confirm that the PAP exists in the codebase and in the source file reported by the SAT that detected the instance, and the PAP detected is applicable in that specific context or the recommended fix should/can't be applied. An example of the latter is the `SlowForLoop` PAP, that recommends the usage of `Fore-each` loops instead of bounded loops. However, a standard `for` loop might be necessary if the index of the element is required for logic beyond simply accessing the element, or if the collection needs to be modified during iteration (e.g., removing elements), which is often unsafe with an enhanced `for-each` loop.

The manual validation step resulted in a curated set of a total 815 annotated PAPs, with 208 annotated PAPs for the LLM-generated and 607 for the DVibeProjs dataset. Figure 8 highlights these totals by individual project, visualizing the relationship between all flagged issues, the subset that was manually annotated, and the confirmed true positives.

By comparing the distributions presented in Figure 8, is it possible to observe a fundamental difference in how PAPs manifest. The histogram on the left demonstrates that PAPs are systemic across the LLM-generated corpus: nearly every generated application contains confirmed PAPs instances. In contrast, the chart on the right highlights the high variability inherent in human development. While some human projects exhibit significant spikes, others remain comparatively clean.

Table 1 presents the quantitative breakdown of the manual validation. The contrast in efficiency is evident: the DVibeProjs dataset exhibits an APD of 8.36 in the labeled evaluation, compared to just 0.84 for the human-written DSimProjs. This implies that LLM-generated code is roughly 10 times more dense with performance defects than equivalent human code.

Figure 9 illustrates at left the APD distribution in a box-plot. The DVibeProjs box-plot is positioned significantly higher in terms of median and upper and lower quartiles, evidencing that this inefficiency is not limited to a few outliers but is a systemic characteristic of the generated code. **??** reinforces this via the distribution of the SFP metric presented in the right box-plot. While human projects frequently contain files entirely free of PAPs (indicated by medians approaching 91%), the DVibeProjs distribution is lower and wider, with a median of 0.85%. This indicates that it is rarer for an LLM than to a human to generate a file that is completely free of known performance defects.

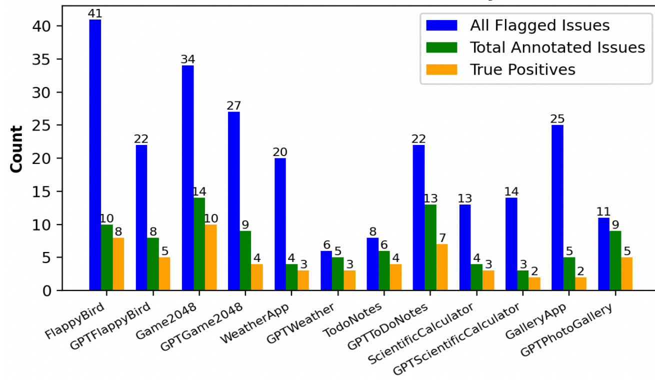
Figure 10 dissects these densities by category (CAPD), revealing specific LLMs blind-spots in efficient source code generation. (DVibeProjs consistently presents higher values than the DVibeProjs dataset, exhibiting high discrepancy over these same human baselines in Resource Management and Suboptimal Algorithms. While humans and LLMs seem to evidencing struggles with Code Smells, LLMs show a catastrophic lack of awareness regarding resource lifecycles (e.g., managing memory, releasing sensors, closing streams), exhibiting densities nearly 10x higher than human counterparts in this critical area.

In terms of statistical evaluation, the results observed for APD and SFP diverge slightly from the holistic evaluation described in Section 4.1. Regarding theAPD metric, the Shapiro-Wilk test confirmed that the DVibeProjs and DOpenProjs datasets follow a normal distribution. Consequently, the T-test allows for the rejection of the null hypothesis, confirming the existence of a statistically significant difference in APD between the two datasets ($p < 0.0001$). Subsequently, to assess the magnitude of this difference, Cohen's d was employed. This evaluation confirmed a large effect size ($d = 2.11$).

Regarding the SFP metric, the Shapiro-Wilk test again confirmed the normality of the DVibeProjs and DOpenProjs datasets. However, the T-test—similar to the Mann-Whitney test in the holistic evaluation—failed to reject the null hypothesis ($p = 0.0525$ for the comparison against DSimProjs and $p = 0.3101$ against DOpenProjs). This allows us to conclude that there are no statistically significant differences between the means of these pairs of samples.

4.3 Synthesis

Comparing the holistic and manually annotated evaluations reveals a striking consistency in the performance gap between LLM-generated and human-written code. While the holistic evaluation established a broad automated baseline using the raw output of only four SATs, the annotated evaluation incorporated all 13 SATs



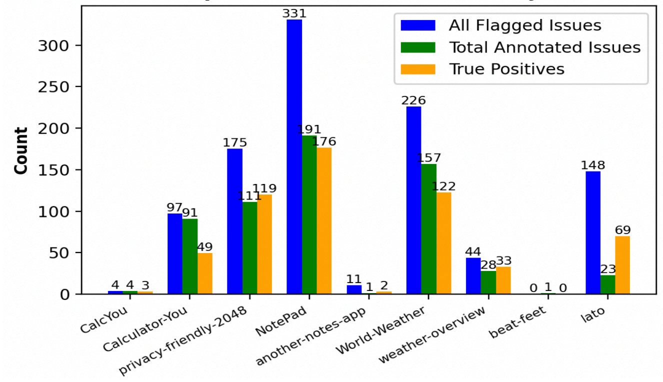


Figure 8: Comparison of PAPs across Projects of the DVibeProjs and DSimProjs dataset, respectively

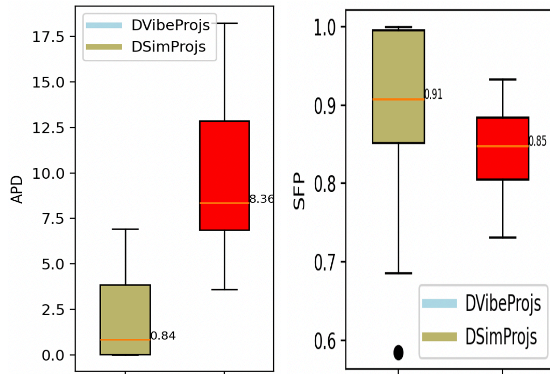


Figure 9: APD and SFP values for the manually-annotated datasets, respectively

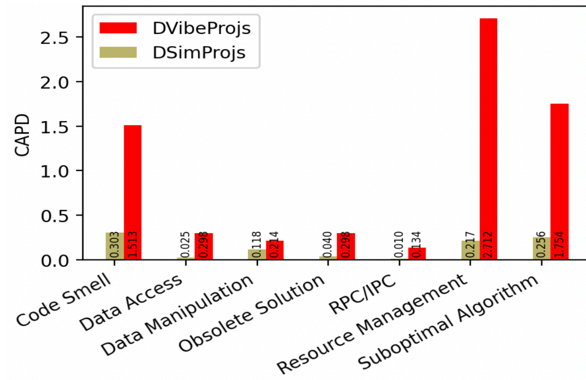


Figure 10: CAPD values for the manually-annotated datasets

and applied rigorous manual verification to eliminate false positives. This expansion of the toolset explains the discrepancy in APD values (e.g., the increase from 0.32 to 8.4 for DSimProjs). Although manual verification removes false positives, the inclusion of additional detection tools significantly broadened the scope of identified issues. Despite these methodological differences, the statistical tests confirm a robust conclusion: the high volume of PAPs detected in the holistic phase was not an artifact of tool over-sensitivity, but a reflection of genuine code defects.

We establish the statistical reliability of these findings through the statistical tests performed in Section 4.1. In the holistic evaluation, we confirmed statistically significant differences ($p = 0.0011$ and $p = 0.0019$) between LLM-generated code and human baselines. This confirms that the holistic APD of 7.49 is not a random artifact.

Crucially, the magnitude of this disparity intensifies under manual scrutiny, a finding quantified by our effect size measurements. In the holistic view, the Vargha-Delaney statistic yielded *Large* effect sizes ($\hat{A}_{12} = 0.90$ and $\hat{A}_{12} = 0.76$, respectively), indicating a substantial probability that an LLM-generated project will have a higher APD than a human one. This gap widened in the annotated dataset, where the Student T-tests [55] and Cohen's d statistically validated the 10× *gap* observed in the raw medians (8.36 vs 0.84 APD). This indicates that for every unit of code generated, LLMs introduce an order of magnitude more performance debt than human developers working on identical requirements.

Interestingly, our statistical analysis of the SFP adds nuance to the *Performance Loop of Hell*. Both the Mann-Whitney U test (holistic) and the T-test (annotated, $p > 0.05$) failed to reject the null hypothesis regarding SFP. This lack of statistical significance suggests that LLMs do not necessarily produce "dirty" files more *frequently* than humans (in terms of percentage of clean files). However, combined with the massive effect size in APD, this implies that when LLMs *do* introduce defects, they do so with extreme density.

> **RQ1**: Evidence confirms that LLM-generated code exhibits statistically significant degradation, with Vibe Coded projects displaying a 10× higher density of performance anti-patterns than human-written equivalents. This systemic generation of suboptimal code threatens to fuel the self-reinforcing *Performance Loop of Hell*.

Furthermore, LLMs demonstrate severe difficulties with resource lifecycle management and suboptimal algorithm solutions. The finding that on median, 19% of LLM code entities contain at least one PAP indicates that developers cannot simply refactor a few critical

modules. Instead, they must conduct comprehensive code review and refactoring across the entire codebase.

Finding 1: LLMs do not merely reproduce human error: they amplify it. . Validated by large statistical effect sizes, our results show that LLMs significantly augment inefficiencies in critical areas like Resource Management and Suboptimal Algorithms.

5 Threats to Validity

Internal Validity. The performance of LLMs is sensitive to model selection, configuration, and prompting strategies. To mitigate bias, we evaluate two distinct state-of-the-art model families (OpenAI and Gemini). However, findings may not fully generalize to other architectures (e.g., Llama, Claude) or highly specific configurations. Furthermore, we utilize simple, high-level specification prompts in natural language to reflect an out-of-the-box developer experience. Sophisticated prompt engineering or iterative refinement could yield different results. Finally, the inherent non-determinism of Mixture-of-Experts (MoE) models means that re-generation may produce slight variations in code quality, a factor intrinsic to current generative AI that complicates strict reproducibility.

Construct Validity. We rely on statically detectable anti-patterns as proxies for performance efficiency. While these patterns correlate with resource degradation, static analysis cannot measure actual runtime impact (e.g., latency, energy consumption) or distinguish between high-frequency execution contexts (e.g., rendering loops) and harmless initialization code. Consequently, our findings quantify the prevalence of potential defects rather than their confirmed runtime penalty. Future work should focus on dynamic analysis to validate the significancy of the impact of these detected PAPs.

External Validity. Our human-written baselines are derived exclusively from F-Droid [1]. While representative of open-source development, these projects may differ in quality and architecture from large-scale commercial applications backed by dedicated QA teams. Additionally, our analysis is constrained by the capabilities of available SATs, limiting our scope to native Android (Java/Kotlin). Our findings do not extend to cross-platform frameworks (e.g., Flutter, React Native), which represent a significant portion of modern mobile development but lack comparable static analysis tooling.

6 Conclusions

This paper presents the first comprehensive empirical evaluation of LLM-generated Android code efficiency through the lens of performance anti-patterns. We analyze 12 LLM-generated applications and compared them against two baselines of 307 open-source projects (DOpenProjs) and 12 functionally-equivalent human-written projects (DSimProjs). Using a comprehensive catalog of 187 statically-detectable PAPs across 11 categories and 13 state-of-the-art SATs, we conduct both holistic and manually-verified analysis to ensure result robustness.

Our results demonstrate that LLM-generated Android code exhibits statistically significant performance inefficiencies compared to human-written equivalents. Specifically, LLM-generated applications exhibit a median Anti-Pattern Density (APD) ranging from 11.5 (holistic) to 8.36 (annotated) per KLoC. This stands in stark contrast to the 0.32–0.84 APD observed in functionally similar human projects, representing an approximate 10x increase. This inefficiency is systemic rather than sporadic, affecting 15% of all source files in LLM projects compared to only 9% in human projects. Furthermore, our results highlight critical blind spots in Resource Management and Suboptimal Algorithms, where LLMs failed to manage lifecycles (e.g., closing streams, recycling objects) at rates nearly an order of magnitude higher than human developers.

Our findings confirm a systemic efficiency problem in LLM-generated code and support the *Performance Loop of Hell* hypothesis, where models trained on uncurated code perpetuate and amplify existing inefficiencies. To mitigate the risks of deploying unoptimized, energy-draining applications, we propose a multi-tiered approach: near-term mandatory static analysis and quality gates for AI-generated code, and medium-term initiatives to curate performance-aware training corpora that prioritize execution efficiency alongside functional correctness.

References

[1] [n. d.]. *F-Droid - Free and Open Source Android App Repository.* https://f-droid.org/ Accessed: January 24, 2026.

[2] 2022. *Sloc Cloc and Code (scc).* https://github.com/boyter/scc#sloc-cloc-and-code-scc

[3] NetForemost 2025. *Top Mobile App Performance Issues.* NetForemost. https://netforemost.com/top-mobile-app-performance-issues/

[4] Altaf Abbassi, Leuson Silva, Amin Nikanjam, and Foutse Khomh. 2025. A Taxonomy of Inefficiencies in LLM-Generated Python Code. 393–404. doi:10.1109/ICSME64153.2025.00043

[5] Artur Bosch and detekt Contributors. 2024. detekt: Static code analysis for Kotlin. https://github.com/detekt/detekt Accessed: 2026-01-07.

[6] Dave Brosius and fb-contrib Contributors. 2024. fb-contrib: A FindBugs/SpotBugs auxiliary detector plugin. http://fb-contrib.sourceforge.net Accessed: 2026-01-07.

[7] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods Symposium*. Springer, 459–465.

[8] Tom Cappendijk, Pepijn de Reus, and Ana Oprescu. 2025. Generating Energy-efficient code with LLMs. In *2025 IEEE/ACM 9th International Workshop on Green and Sustainable Software (GREENS)*. arXiv:2411.10599 [cs.SE] https://arxiv.org/abs/2411.10599

[9] M. Couto, J. Saraiva, and J. P. Fernandes. 2020. Energy Refactorings for Android in the Large and in the Wild. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 217–228. doi:10.1109/SANER48275.2020.9054858

[10] L. Cruz and R. Abreu. 2017. Performance-Based Guidelines for Energy Efficient Mobile Applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 46–57. doi:10.1109/MOBILESoft.2017.19

[11] Luis Cruz and Rui Abreu. 2019. Catalog of energy patterns for mobile applications. *Empirical Softw. Engg.* 24, 4 (Aug. 2019), 2209–2235. doi:10.1007/s10664-019-09682-0

[12] L. Cruz, R. Abreu, and J. Rouvignac. 2017. Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 205–206. doi:10.1109/MOBILESoft.2017.21

[13] Teerath Das, Massimiliano Di Penta, and Ivano Malavolta. 2020. Characterizing the evolution of statically-detectable performance issues of Android apps. *Empir. Softw. Eng.* 25, 4 (2020), 2748–2808. doi:10.1007/s10664-019-09798-3

[14] Iffat Fatima, Hina Anwar, Dietmar Pfahl, and Usman Qamar. 2020. Detection and Correction of Android-specific Code Smells and Energy Bugs: An Android Lint Extension. In *QuASoQ@APSEC*. https://api.semanticscholar.org/CorpusID:228631374

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Lin Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the*

Association for Computational Linguistics: EMNLP 2020. 1536–1547. A foundational model for code comprehension and representation, often used for tasks like code search and summarization..
[16] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.
[17] Yuyao Ge, Lingrui Mei, Zenghao Duan, Tianhao Li, Yujia Zheng, Yiwei Wang, Lexin Wang, Jiayu Yao, Tianyu Liu, Yujun Cai, Baolong Bi, Fangda Guo, Jiafeng Guo, Shenghua Liu, and Xueqi Cheng. 2025. A Survey of Vibe Coding with Large Language Models. arXiv:2510.12399 [cs.AI] https://arxiv.org/abs/2510.12399
[18] Google. 2012. Memory allocations within drawing code. https://googlesamples.github.io/android-custom-lint-rules/checks/DrawAllocation.md.html. Accessed: 2025-05-11.
[19] Google. 2021. *Android Lint Checks*. https://googlesamples.github.io/android-custom-lint-rules/ Last visit: 2025-02-05.
[20] Google. 2025. Android Studio. https://developer.android.com/studio. https://developer.android.com/studio Accessed: 2025-12-29.
[21] Sarra Habchi, Naouel Moha, and Romain Rouvoy. 2019. The Rise of Android Code Smells: Who is to Blame?. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 445–456. doi:10.1109/MSR.2019.00071
[22] Jiaqi He, Revan MacQueen, Natalie Bombardieri, Karim Ali, James R. Wright, and Cristina Cifuentes. 2023. Finding an Optimal Set of Static Analyzers To Detect Software Vulnerabilities. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 463–473. doi:10.1109/ICSME58846.2023.00060
[23] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. 2015. Tracking the Software Quality of Android Applications Along Their Evolution (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 236–247. doi:10.1109/ASE.2015.46
[24] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Trans. Inf. Syst.* 43, 2, Article 42 (Jan. 2025), 55 pages. doi:10.1145/3703155
[25] IBM. 2025. Vibe coding. https://www.ibm.com/think/topics/vibe-coding. Accessed: 2025-04-04.
[26] Jaehyeon Kim, Rui Rua, and Karim Ali. 2025. BuilDroid: A Self-Correcting LLM Agent for Automated Android Builds. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE '25)*. IEEE/ACM, Seoul, South Korea. Tool Demo Track.
[27] Kivy Organization. 2024. Kivy: Open source Python framework for rapid development of applications that make use of innovative user interfaces. https://kivy.org/. Accessed: October 26, 2024.
[28] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, Vol. 33. 6004–6016. Focuses specifically on code transformation, specifically language translation, using unsupervised methods..
[29] Dianshu Liao, Shidong Pan, Siyuan Yang, Yanjie Zhao, Zhenchang Xing, and Xiaoyu Sun. 2024. Automatically Analyzing Performance Issues in Android Apps: How Far Are We? arXiv:2407.05090 [cs.SE] https://arxiv.org/abs/2407.05090
[30] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. ACM, 2–11. doi:10.1145/2597073.2597085
[31] Daniel Maia, Marco Couto, Jooão Saraiva, and Rui Pereira. 2020. E-Debitum: Managing Software Energy Debt. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. 170–177. doi:10.1145/3417113.3422999
[32] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Ann. Math. Statist.* 18, 1 (03 1947), 50–60. doi:10.1214/aoms/1177730491
[33] Chenguang Mao, Hao Wang, Gaojie Han, and Xiaofang Zhang. 2020. Droidlens: Robust and Fine-Grained Detection for Android Code Smells. In *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 161–168. doi:10.1109/TASE49443.2020.00030
[34] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. 2019. Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 209–219. doi:10.1109/ICPC.2019.00040
[35] Alejandro Mazuera-Rozo, Catia Trubiani, Mario Linares-Vásquez, and Gabriele Bavota. 2020. Investigating types and survivability of performance bugs in mobile apps. *Empirical Software Engineering* 25 (05 2020), 1–43. doi:10.1007/s10664-019-09795-6
[36] Microsoft. 2024. Xamarin | Open-source mobile app platform for .NET. https://dotnet.microsoft.com/en-us/apps/xamarin. Accessed: October 26, 2024. Note: Xamarin is now evolved into .NET MAUI..
[37] Wellington Oliveira, Renato Oliveira, and Fernando Castor. 2017. A Study on the Energy Consumption of Android App Development Approaches. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 42–52. doi:10.1109/MSR.2017.66
[38] OpenAI. 2022. ChatGPT [Large language model]. https://chat.openai.com/chat Accessed: 2026-01-15.
[39] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2017. Lightweight detection of Android-specific code smells: The aDoctor project. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 487–491. doi:10.1109/SANER.2017.7884659
[40] J. Park, Y. B. Park, and H. K. Ham. 2013. Fragmentation Problem in Android. In *2013 International Conference on Information Science and Applications (ICISA)*. 1–2. doi:10.1109/ICISA.2013.6579465
[41] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2021. Ranking programming languages by energy efficiency. *Science of Computer Programming* 205 (2021), 102609. doi:10.1016/j.scico.2021.102609
[42] Ricardo B. Pereira, João F. Ferreira, Alexandra Mendes, and Rui Abreu. 2022. Extending EcoAndroid with Automated Detection of Resource Leaks. In *2022 IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MobileSoft)*. 17–27. doi:10.1145/3524613.3527815
[43] Nils Purschke, Sven Kirchner, and Alois Knoll. 2025. SpeedGen: Enhancing Code Efficiency through Large Language Model-Based Performance Optimization. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1–12. doi:10.1109/SANER64311.2025.00045
[44] Ghulam Rasool and Azhar Ali. 2020. Recovering Android Bad Smells from Android Applications. *Arabian Journal for Science and Engineering* 45 (02 2020). doi:10.1007/s13369-020-04365-1
[45] Jan Reimann, Martin Brylski, and Uwe Assmann. 2014. A Tool-Supported Quality Smell Catalogue For Android Developers. *Softwaretechnik-Trends* 34 (2014). https://api.semanticscholar.org/CorpusID:653529
[46] Ana Ribeiro, João F. Ferreira, and Alexandra Mendes. 2021. 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)*. 62–69. doi:10.1109/QRS54544.2021.00017
[47] Rui Rua. 2019. Performance and energy issues in Android apps. https://rrua.github.io/android-performance-issues/. Accessed: 2026-01-07.
[48] Rui Rua and João Saraiva. 2023. A large-scale empirical study on mobile performance: energy, run-time and memory. *Empirical Software Engineering* (12 2023), 67. doi:10.1007/s10664-023-10391-y
[49] Rui Rua and João Saraiva. 2023. PyAnaDroid: A fully-customizable execution pipeline for benchmarking Android Applications. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 586–591. doi:10.1109/ICSME58846.2023.00077
[50] SANAD Lab. 2025. A collection of Android applications generated by LLMs. https://github.com/sanadlab/vibe_coded_android_apps. Accessed: 2026-01-18.
[51] S. S. Shapiro and M. B. Wilk. 1965. An analysis of variance test for normality (complete samples)†. *Biometrika* 52, 3-4 (12 1965), 591–611. arXiv:https://academic.oup.com/biomet/article-pdf/52/3-4/591/962907/52-3-4-591.pdf doi:10.1093/biomet/52.3-4.591
[52] Mohammed Latif Siddiq, Lindsay Roney, Jiahao Zhang, and Joanna C. S. Santos. 2024. Quality Assessment of ChatGPT Generated Code and their Use by Developers. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. 152–156.
[53] Manav Singhal, Tushar Aggarwal, Abhijeet Awasthi, Nagarajan Natarajan, and Aditya Kanade. 2024. NoFunEval: Funny How Code LMs Falter on Requirements Beyond Functional Correctness. In *First Conference on Language Modeling*. https://openreview.net/forum?id=h5umhm6mzj
[54] SpotBugs Team. 2024. SpotBugs: Find bugs in Java programs. https://spotbugs.github.io/ Accessed: 2026-01-07.
[55] Student. 1908. The Probable Error of a Mean. *Biometrika* 6, 1 (1908), 1–25.
[56] The Apache Software Foundation. 2025. Apache Ant. https://ant.apache.org/. Accessed: January 24, 2026.
[57] The Apache Software Foundation. 2025. Apache Maven Project. https://maven.apache.org/. Accessed: January 24, 2026.
[58] The PMD Project. [n. d.]. PMD: An extensible cross-language static code analyzer. https://github.com/pmd/pmd. Accessed: 2025-05-01.
[59] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. doi:10.3102/10769986025002101
[60] Yue Wang, Weishi Wang, Shafiq Joty, and Steven Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *ACM Transactions on Information Systems*. 8696–8708. doi:10.18653/v1/2021.emnlp-main.685

[61] Muhammad Waseem, Aakash Ahmad, Kai-Kristian Kemell, Jussi Rasku, Sami Lahti, Kalle Mäkelä, and Pekka Abrahamsson. 2025. Vibe Coding in Practice: Flow, Technical Debt, and Guidelines for Sustainable Use. arXiv:2512.11922 [cs.SE] https://arxiv.org/abs/2512.11922